SNAP Simulator Result

Technical Report No. CENG 89-11

Changhwa Lin and Dan Moldovan

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-0781

March 6, 1990

[0]This research has been funded by the National Science Foundation Grant No. MIP-89/02426

Contents

Abstract

When developing a computer architecture, theory alone is not suffcient for proving capability and ensuring optimal performance. In this vein, a hardware simulation program has been developed to verify the design and analyze the performance of the Semantic Network Array Processor (SNAP). Implemented in Common LISP, the simulator provides a complete executable model of the SNAP architecture. It monitors each simulated operation with respect to computation and communication time through the use of a timestamp, while a background process maintains the effective global time. It also contains a repitiore of user configurable parameters for the SNAP architecture such as the distribution of nodes in the array, memory space per node, and internal queue sizes. The simulator has been used to obtain performance estimates in several application areas. It provides an effective means by which to assess and refine an architecture before the hardware is available.

1 Introduction

The primary objective of the SNAP project has been to design and implement a specialized, highly parallel architecture for knowledge representation and reasoning. The approach used was to first identify key issues and operations for the knowledge processing and then to map these requirements directly into hardware. When carefully executed, this method can provide performance advantages over the contrasting approaches of designing parallel AI languages or using commercially available multiprocessors. However, the success of this approach depends upon both the selection of the proper operations and their effective hardware implementation. The SNAP Simulator provides assistance with both of these aspects of the design process.

Work on SNAP began in 1983 at USC [Moldovan 1983] and a preliminary version of a simulator was completed in 1985 [Tung 1985]. Since then considerable experience has been gained with the architecture and the SNAP design has been enhanced. A new simulation program has also been written to reflect the new design. In particular, the simulation program presented in this report describes an executable model of SNAP that performs knowledge processing using the marker propagation paradigm.

1.1 Objectives of Simulation

Although a basic simulation can provide some insight into the expected performance of a theoretical machine, more information is needed before building the real hardware. A detailed simulation at the instruction level is essential for making a wide range of design trade-off decisions. For the purposes of optimizing the SNAP architecture, the following objectives were pursued in the SNAP simulator:

- provide a testbed to evaluate the completeness and correctness of the SNAP instruction set,
- allow experimentation with a set of propagation rules for facilitating the marker propagation paradigm,
- obtain accurate performance measurements, including overhead, of both hardware instructions and communication times
- allow investigation of trade-offs associated with the number of processing cells contained in a chip, which forms a critical balance point between communication and processing speed in the architecture,

- gather interconnection network bandwidth and traffic measurements, including effects of varying queue sizes and message lengths,

- provide a development platform for SNAP application programs where routines can be written, executed, and debugged for final installation in the real hardware when it becomes available.

1.2 Overview of the SNAP Simulator

In its simplest implementation, a simulator provides only a set of passive functions which mimic the functionality of the real machine. However, to facilitate realistic performance measurement of a computer architecture, it is necessary for a simulator to execute a background process which continuously measures time and activity. The SNAP Simulator supervises all operations with respect to computation and communication time through the use of a timestamp, while a background process maintains the effective global time. This approach has the desirable effect of providing a user-transparent performance monitor when developing an application program.

2 SNAP Simulator Commands

The commands available in the simulator are a superset of the SNAP primitive instructions previously described in the SNAP hardware overview [Moldovan 1989]. As in the hardware implementation, SNAP primitives are broadcast by the central controller to all chips in the array for on-chip execution. The simulator fully supports the SNAP primitive instruction set and also provides some of the controller functions such as instruction broadcast, node allocation, and selected housekeeping tasks. The remainder of the controller functions will be implemented in the interface software driver in the host computer.

To provide a good application program development environment, various status inquiry and debugging directives are also included in the set of commands supported by the simulator:

- directives for initializing and restarting the simulator,
- directives for reporting the simulator status after execution,
- switches for display of the marker activities during execution for tracing and debugging,
- switches for display of the communication time for marker activities,
- instructions for the necessary controller functions.

2.1 Basic SNAP Primitives

The syntax for the basic primitives described below are the actual instruction formats of the basic SNAP primitives. They are slightly different from the formats described in the earlier SNAP hardware overview [Moldovan 1989]. Most of the instructions supported by the simulator are preceded by a "s-" prefix. The prefix is to distinguish the basic SNAP primitives from Lisp primitives such as *search* which coincide with the SNAP instruction set. Note that the supplemental directives provided by the simulator are not preceded by the "s-" prefix.

There are several kinds of arguments which have to be supplied for the instructions. The marker argument is supplied to indicate the marker name then the controller will allocate a marker for the given name or a number can be given for the marker argument to invoke the marker directly. The same rule is applicable to the relation arguments and node arguments. For the register argument, only the number can be used.

Each primitive function and its syntax is described below:

- S-SEARCH *(s-search node-name marker)*

  The s-search function will set the content of *marker* to 1 in the selected node which described by the node-name

  For example, (s-search 'clyde 'elephant) means to set marker(elephant) in node(clyde) to 1.

- S-SEARCH-COLOR *(s-search-color node-color relation marker)*

  S-SEARCH-COLOR provides two pattern strings to the array; the first is the color of the cell memory, and the second is a relation pointer. This instruction causes every SNAP chip to check these two conditions and in the cells where the match is successful, the content of the marker specified in the third argument is set to 1. For example, (S-SEARCH-COLOR 'test 'isa 'found) means to search cells with color TEST and relation ISA, and set marker FOUND in those cells.

- S-SET-COLOR *(s-set-color node-name node-color)*

  S-SET-COLOR modifies the COLOR of node-name For example, (S-SET-COLOR 'man 'mammal) will modify the color of MAN to MAMMAL.

- S-CREATE *(s-create node-name1 relation node-name2)*

  S-CREATE is function that creates new relations between nodes For example, (S-CREATE A R B) inserts a new relation R from node A to node B. If either A or B is new, the controller assigns them to new nodes.

  Initially, the array is loaded with nodes and links by the controller. This allocation procedure is built upon the function S-CREATE. Conceptually, this procedure employs as many S-CREATEs as the number of links in the semantic network.

- S-DELETE *(s-delete node-name1 relation node2)*

  The S-DELETE function has the opposite effect of S-CREATE. It deletes a pointer relation between a pair of nodes. For example, (S-DELETE A, R, B) deletes the relation R linking node A to B. If either node becomes isolated after the DELETE, then the node can be permanently deleted when the controller invokes garbage collection.

- S-LOAD *(s-load marker register data)*

  The S-LOAD instruction causes all cells with MARKER to load DATA into REGISTER. For example, (S-LOAD 'ACCOUNT 2 9000) means that all nodes with marker ACCOUNT equal to 1 will load 9000 into register 2.

- S-AND *(s-and marker1 marker2 marker3)*

  The S-AND function performs a logical AND of the first two markers. The marker3 is set or reset based on the result of and the first two marker argument. For example, (AND 2 4 7) will set marker 7 in those nodes where both markers 2 and marker 4 are set.

- S-OR *(s-or marker1 marker2 marker3)*

  The S-OR function performs a logical OR of the first two arguments. The marker3 is set or reset based on the result of the or operation. For example, (S-OR 2 4 7) will set marker 7 for those nodes where either marker 2 or marker 4 is set.

- S-NOT *(s-not marker1 marker2)*

  The S-NOT function negates the status of marker1 and places the result in marker2. For example, (S-NOT 5 4) means that if marker 5 is set for a node, then marker 4 becomes reset. Otherwise, marker 4 is set.

- S-MARKER *(s-marker marker1 marker2 propagation-rule-list)*

  The marker instruction introduces a marker into the network. All nodes with marker1 will begin propagating marker2 according to the propagation rule list The originating node does not set marker2. The propagation rule list is a list contains a propagation rule and one or two relations.

  For example, (S-MARKER 1 2 '(SEQ ISA ROLE)) causes all nodes with marker 1 set to propagate marker 2 once along the link relation of the affected nodes, and then once through the Role relation.

- S-COLLECT *(s-collect marker)*

  S-COLLECT is a content addressable read. The controller collects the node-names of the nodes that have marker set and return the result to the calling process. For example, (SETQ X (S-COLLECT 1)) gets all the node-names of nodes which have marker 1 set and binds them to variable X.

- S-COLLECT-RELATION *(s-collect-relation marker)*

The function S-COLLECT-RELATION also performing content addressable read. The controller collects all the information of node relations for those nodes with marker set. The result will be returned as a list of pairs of node-name and relations.

For example, (SETQ X (S-COLLECT-RELATION 1)) gets all the relation names in nodes which have marker 1 set, and binds them to X.

- S-STOP-MARKER *(s-stop-marker marker1 marker2 marker3)*

  The S-STOP-MARKER instruction causes a stop <marker3> command to be placed on all cells with [marker1] and [marker2] set. Thus, these cells will not allow <marker3> to propagate. For example, (STOP-MARKER #1, #2, #3) will cause all cells with #1 and #2 set to not propagate #3.

- S-CLEAR-MARKER *(s-clear-marker marker1 marker2 marker3)*

  With this instruction, the cells with marker1 and marker2 set will clear marker3. For example, (S-CLEAR-MARKER 4 5 6) will cause all cells with markers 4 and 5 set to clear marker 6.

- S-CLEAR-STOP-MARKER *(s-clear-stop-marker marker1 marker2 marker3)*

  This instruction clears the stop-marker at marker3 for all cells with marker1 and marker2 set. For example, (S-CLEAR-STOP-MARKER 4 5 6) will cause all cells with markers 4 and 5 set to clear their stop-marker 6.

- S-EQUATE *(s-equate relation1 relation2)*

  The S-EQUATE instruction causes the SNAP to treat relation1 as if it were relation2 during marker propagation and vice versa. For example, (S-EQUATE 'R1 'R2) will allow any marker that propagates along R2 to also propagate along R1.

- S-CLEAR-EQUATE *(s-clear-equate relation1 relation2)*

  The S-CLEAR-EQUATE instruction causes SNAP to forget about the S-EQUATE between relation1 and relation2.

- S-READ *(s-read marker register)*

  The S-READ instruction causes all cells with marker to output the content of the register addressed by the register argument. For example, (S-READ 1 2) means that all nodes with marker 1 set will output register 2 to the controller. The use of S-READ is similar to S-COLLECT which have to be invoked by other function which will receive the result.

- S-REG-ADD *(s-reg-add marker register1 register2)*

  The S-REG-ADD instruction causes all cells with marker set to ADD the data in register1 and register2 and store the result in register1. For example, (S-REG-ADD 1 2 3) means that all nodes with marker 1 set, will add contents of registers 2 and 3 and store the result in register 2.

- S-REG-SUB *(s-reg-sub marker register1 register2)*

  The S-REG-SUB instruction is identical to S-REG-ADD except that a subtraction is done instead of an addition.

- S-REG-MULT *(s-reg-mult marker register1 register2)*

  The S-REG-MULT instruction is identical to REG-ADD except that a multiplication is done instead of an addition.

- S-REG-DIVIDE *(s-reg-divide marker register1 register2)*

  The S-REG-DIVIDE instruction is identical to REG-ADD except that a division is done instead of an addition.

- S-TEST *(s-test marker1 register prediction-operator marker2)*

  The S-TEST instruction causes all cells with marker1 to check the content of the register with the prediction-operator which is either zerop, postivep or negativep. If the condition to be check is TRUE then marker2 is set.

  For example, (S-TEST 1 2 'ZEROP 2) means that all nodes with marker 1 set will check to see if the content of register 2 is a zero. If the condition is true then marker 2 is set.

- S-MARKER-ADD *(s-marker-add marker1 register1 register2 marker2 propagation-rule-list)*

  The MARKER-ADD instruction causes all cells with marker1 to send the data in register1 to other nodes according to the propagation-rule-list. When the data gets to a destination node, the data is added to the data of register2 of the destination node. The result is stored in register1 of the destination node. Also, the marker2 is then set in the destination node. The data to be sent out is the copy of the result of the addition (equal to the content of register1 of the destination node).

  For example, (S-MARKER-ADD 1 1 2 2 '(SPREAD ROLE LINK)) means that all nodes with marker 1 set will send the data in register 1 to all nodes connected to it by ROLE or LINK relations. When the message gets to one of these nodes, the data is added to the data in the nodes register 2 and the result is stored in the register 1. Marker

2 in the destination node is then set, and this node tries to propagate the message to its ROLE and LINK neighbors.

- S-MARKER-SUB *(s-marker-sub marker1 register1 register2 marker2 propagation-rule-list)*

  S-MARKER-SUB is identical to S-MARKER-ADD except that a subtraction is done at the destination node instead of an addition.

- S-MARKER-MULT *(s-marker-mult marker1 register1 register2 marker2 propagation-rule-list)*

  S-MARKER-MULT is identical to S-MARKER-ADD except that a multiplication is done at the destination node instead of an addition.

- S-MARKER-DIVIDE *(s-marker-divide marker1 register1 register2 marker2 propagation-rule-list)*

  S-MARKER-DIVIDE is identical to MARKER-ADD except that a division is done at the destination node instead of an addition.

- S-MARKER-MIN *(s-marker-min marker1 register1 register2 marker2 propagation-rule-list)*

  S-MARKER-MIN is identical to S-MARKER-ADD except that the result is the minimum of the two number instead of performing an addition at the destination node.

- S-MARKER-MAX *(s-marker-max marker1 register1 register2 marker2 propagation-rule-list)*

  S-MARKER-MAX is identical to S-MARKER-MIN except that a maximum of the two numbers is done at the destination node instead of an minimum.

- S-MARKER-MIN+ *(s-marker-min+ marker1 register1 register2 marker2 propagation-rule-list)*

  S-MARKER-MIN+ is identical to S-MARKER-MIN except that a 1 is added to the result of minimum operation.

  For example, (MARKER-MIN+ 1 1 2 2 '(SPREAD ROLE LINK)) means that all nodes with marker 1 set will sent the content of register 1 to all nodes connected by either ROLE or LINK relations. When the message gets to a destination node, a minimum is performed on the data and the content of register 2 at the destination node and the result is then added with 1 and stored in register 1. Then the node sets the marker 2. The result in the register will be sent out according to the propagation-rule-list.

2.2 Controller Commands

The controller commands described here for the simulator supplement the SNAP primitive instructions. Although they do not appear in previous descriptions of the SNAP instruction set, they are necessary for proper operation of the complete SNAP system and will be implemented within the controller in the final SNAP system.

- S-DOWNLOAD *(s-download '((nodename relation nodename) ...))*

  The S-DOWNLOAD is provided for faster cell array loading. The instruction executes a sequence of s-create commands, but with reduced interaction (i.e. communication overhead) between host and controller as compared to separately issued s-create commands.

  For example, (S-DOWNLOAD '((he isa man) (she isa girl))) is same as executing (s-create '(he isa man)) and (s-create '((she isa girl))) separately.

- SET-RELDUAL *(s-set-reldual '(relation1 relation2))*

  S-SET-RELDUAL is to inform the controller of a forwards-backwards type relation pair. This is useful when applying s-create and s-delete.

  For example, (SET-RELDUAL '(isa sub)) will tell the controller that the ISA SUB should include a reverse link.

- RESET-SNAP *(reset-snap)*

  RESET-SNAP is a software reset for the SNAP array and will reset only markers and stop markers. For the simulator, the system clock and communication statistics are also cleared.

- CLEAR-SNAP *(clear-snap)*

  CLEAR-SNAP will clear every data item in the SNAP array.

2.3 Supplemental Commands

The following commands are mainly designed for the user of the simulator. They provide some basic debugging functions such as displaying the SNAP chip status including the queues and the allocation map in the controller. Thus the functions described in this section may or may not provided in the real hardware.

- SHOW-PERF *(show-performance)*

  This instruction will display the communication statistics and the system clock.

- SHOW-ALLOCATION *(show-allocation)*

  SHOW-ALLOCATION will display the node allocation map.

- SHOW-MARKER *(show-marker chip-address)*

  This instruction will display the color and marker information of the desired chip.

- SHOW-M_QUEUE *(show-m_queue chip-address)*

  SHOW-M_QUEUE will print out the contents of the marker queue in the desired chip.

- SHOW-O_QUEUE *(show-o_queue chip-address)*

  SHOW-O_QUEUE will print out the contents of the output queue in the desired chip.

- SHOW-I_QUEUE *(show-o_queue chip-address)*

  SHOW-I_QUEUE will print out the contents of the input queue in the desired chip.

2.4 Relation Definition

Semantic networks in SNAP are built upon relations between nodes. SNAP supports 64 *primitive relations.* Primitive relations differ from general relations as follows: primitive relations are pointers between semantic network nodes and are stored as registers in the RM, where as other relations called *node relations* are stored as distinct cells in the array.

The primitive relations are user definable. However, the following primitive relation conventions have been adopted:

1. Superconcept: a relation between a supertype and its subsumee.
2. Individual: a relation between a type and a individual.
3. Split: a relation to separate exclusive types.
4. Cancel: the relation to be used when an exception exists in the role of a special type (or individual) under a supertype.

5. Generic: the relation to group individuals with same characteristics.

6. Role: the relation to show the inherit properties of a type (or individual). For example, in sentence "`Animal has hair`", there is a role primitive relation between node `Animal` and node `Has-part`.

7. Link: similar to role except it is for non-inherent properties such as love, hate, etc.

2.5 Marker Propagation Rules

The propagation rules in SNAP govern how markers are passed. SNAP has the capability to allow markers to travel through several relation types at the same time. This feature is supported in the simulator as well as multi-relation propagation rules.

The propagation rules have the format of Rule, Relation1, Relation2, where Relation1 and Relation2 are the relations that rule affects. The following propagation rules are supported by the SNAP simulator:

1. (SEQ R1 R2): the SEQUENCE propagation rule allows the marker to propagate through R1 once, then to R2 once.

2. (SPREAD R1 R2): the SPREAD propagation rule allows the marker to traverse through a chain of R1 links. For each cell in the R1 path, if there exist any R2, the marker switches to R2 link and continues to propagate until the end of the R2 link.

3. (COMB R1 R2): the COMBine propagation rule allows the marker to propagate to all R1 R2 links without limitation.

4. (END-SPREAD R1 R2): This propagation rule is the same as SPREAD except that it marks only the last cells in the paths.

5. (END-COMB R1 R2): This propagation rule is the same as COMB except that it marks only the last cells in the paths.

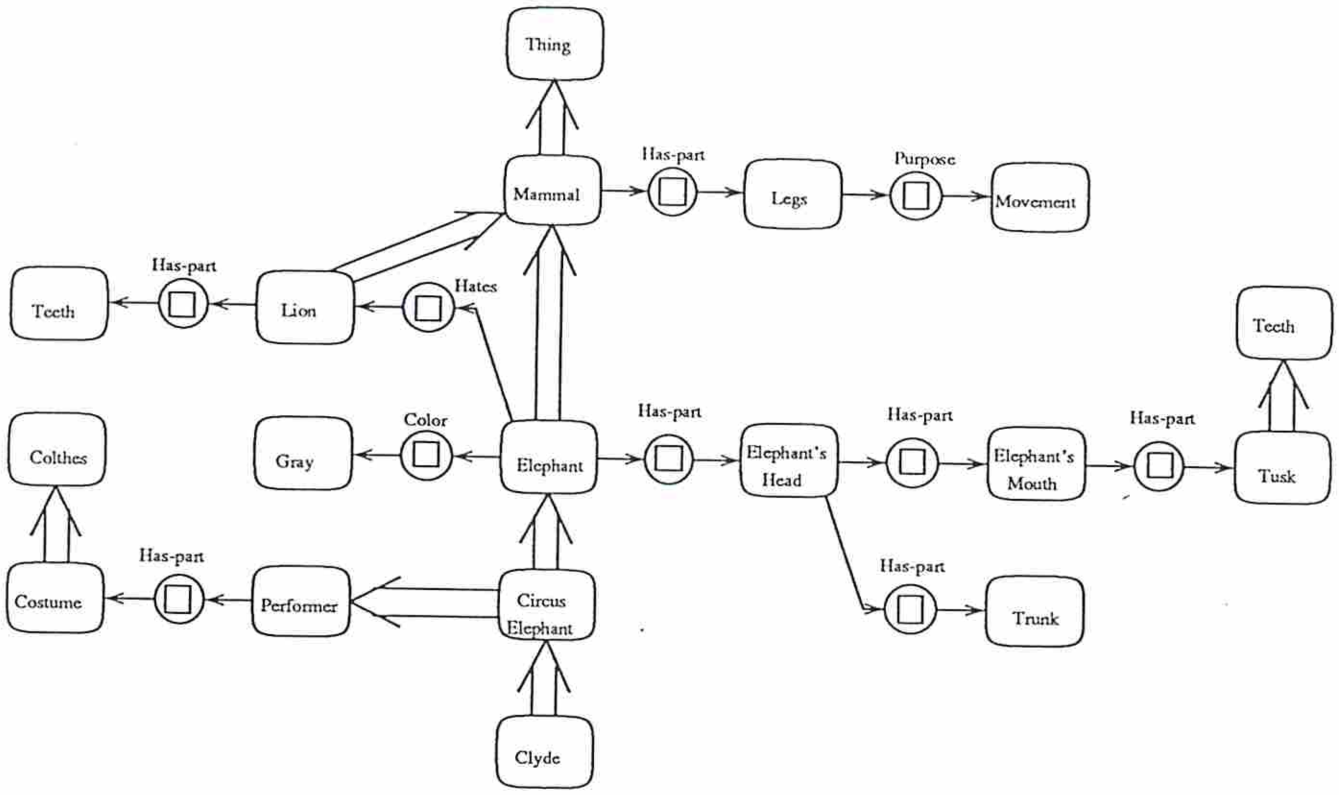

Figure 1: Semantic Network for Clyde

3 Simulation Examples

In this section, several sample programs are presented. Each was coded and executed on the SNAP Simulator. The results for each are described below.

3.1 Clyde the Elephant

The first example program performs inheritance. It deals with Clyde and the semantic network in Figure 1. With this semantic network, we process the following query on SNAP:

"DOES CLYDE HAVE TEETH?"

The SNAP program to answer this query is very simple. It relies on the guiding of markers toward the destination (TEETH). Only valid nodes are allowed to propagate markers. Thus, markers cannot travel from ELEPHANT to HATES to TIGER to HAS-PART to TEETH. The program is shown below. Details of the parts of this algorithm which are not simulator-specific are contained in the report by Moldovan, Lee, and Lin [Moldovan 1989].

```
;declare the relations
(set-rel-assoc '(isa sub))
(set-rel-assoc '(role r-role))

;loading the database
(s-download '((clyde isa circus-elep)
              (circus-elep isa performer)
              (circus-elep isa elep)
              (elep isa mammal)
              (performer role has-part-4)
              (has-part-4 role costume)
              (costume isa clothes)
              (elep role color-1)
              (color-1 role gray)
              (elep role hates-1)
              (hates-1 role lion)
              (lion isa mammal)
              (lion role has-part-2)
              (has-part-2 role teeth)
              (elep role has-part-3)
              (has-part-3 role elep_head)
              (elep_head role has-part-5)
              (has-part-5 role elep_mouth)
              (elep_head role has-part-6)
              (has-part-6 role trunk)
              (elep_mouth role has-part-7)
              (has-part-7 role tusk)
              (tusk isa teeth)
              (mammal isa thing)
              (mammal role has-part-1)
              (has-part-1 role legs)
              (legs role purpose-1)
              (purpose-1 role movement)))

;setup the relation group
(s-set-color 'has-part-1 'has-part)
(s-set-color 'has-part-2 'has-part)
(s-set-color 'has-part-3 'has-part)
(s-set-color 'has-part-4 'has-part)
(s-set-color 'has-part-5 'has-part)
(s-set-color 'has-part-6 'has-part)
(s-set-color 'has-part-7 'has-part)
```

```
(s-set-color 'hates-1 'hates)
(s-set-color 'color-1 'color)
(s-set-color 'purpose-1 'purpose)

;main program (reset-snap)
(s-search-color 'hates '% 0)
(s-search-color 'color '% 0)
(s-search-color 'purpose '% 0)
(s-search-color 'has-part '% 0)
(s-stop-marker 0 '% '%)
(s-search-color 'has-part '% 1)
(s-search 'clyde 2)
(s-search 'teeth 3)
(s-clear-stop-marker 1 '% '%)
(s-marker 2 2 '(comb isa role))
(s-marker 3 3 '(comb sub r-role))
(s-comm-end)
(s-and 2 3 4)
(s-clear-marker 0 '% 4)
(setq x (s-collect 4))
(show-perf)
```

3.2 Classifier Example

Consider the semantic network from Figure 2. This network includes a set of concepts (`person`, `parent`, `grandparent`, `mammal`, `dog` and `date`) and a set of roles (`child`, `birthday`, `pet`). Each concept denotes a set of objects and each role denotes a property that must be true for each concept to which that role is associated. The concepts have a hierarchical order, for example concept `mammal` includes concepts `person` and `dog`. A superconcept link connects a simpler concept to a more general one. We say that a more general concept subsumes a simpler concept. One inference problem is to find new subsumtion relations between the concepts of a given network and then to modify the structure of the network. For instance we want to determine if `parent` subsumes `grandparent`, and if it is true, connect `grandfather` to `parent` with a superconcept link.

A given concept `A` is said to subsume another concept `B` if only if:

1. All primitive concepts that subsume `A` also subsume `B`. (Here `A` is `parent` and `B` is `grandparent`.)

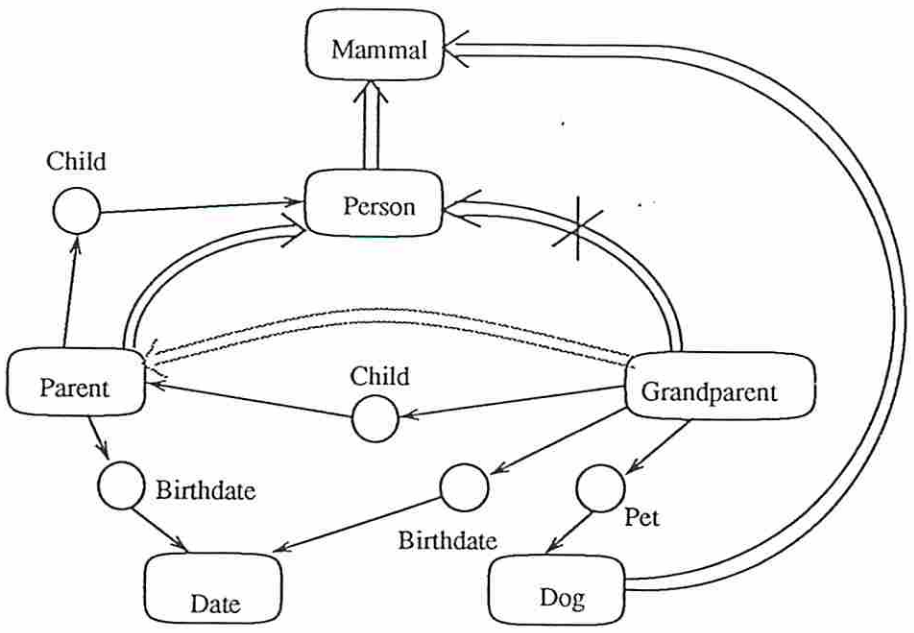

Figure 2: Classification on a semantic network

2. For each roleset of A, some roleset of B denotes the same relation. (Role-set of A includes birthdate and child and is a subset of roleset of B.)

3. The value description of A's roleset subsumes that of B's (For the birthdate roleset, both parent and grandparent have the same value description namely date. For child, parent's value description subsumes grandparent's, namely person subsumes parent.)

The program is listed below for execution on the SNAP Simulator, for a detailed explanation of the operation of the algorithm itself, please refer to Moldovan, Lee, and Lin [Moldovan 1989].

```
;declare relation pairs
(set-reldual '(isa sub))
(set-reldual '(role r-role))
(set-reldual '(isa-link sub-link))

;loadinf the database
(s-download '((person isa mammal)
              (grandparent isa person)
              (parent isa person)
              (grandparent role child-1)
              (child-1 role parent)
              (grandparent role birthdate-1)
              (birthdate-1 role date)
              (grandparent role pet-1)
              (pet-1 role dog)
              (parent role child-2)
              (child-2 role person)
              (parent role birthdate-2)
              (birthdate-2 role date)
              (child-1 isa-link child)
              (child-2 isa-link child)
              (pet-1 isa-link pet)
              (birthdate-1 isa-link birthdate)
              (birthdate-2 isa-link birthdate)
              (birthdate isa-link relation)
              (pet isa-link relation)
              (child isa-link relation)))

;main program
(reset-snap)
```

```
(s-search 'relation 31)
(s-marker 31 0 '(spread sub-link))
(s-comm-end)
;condition I
(s-search 'grandparent 30)
(s-marker 30 1 '(seq isa))
(s-search 'parent 29)
(s-marker 29 2 '(seq isa))
(s-comm-end)
(s-and 1 2 3)
(s-clear-marker 3 '% '%)
(s-or 1 2 4)
;condition II
; roleset of grandparent
(s-marker 30 5 '(seq role))
; roleset of parent
(s-marker 29 6 '(seq role))
;search for second highest level in relation for parent
(s-marker 31 7 '(seq sub-link))
(s-comm-end)
(s-and 5 0 8)
(s-marker 8 10 '(spread isa-link))
(s-and 6 0 9)
(s-marker 9 11 '(spread isa-link))
(s-clear-marker 31 '% '%)
(s-comm-end)
(s-and 10 7 12)
(s-clear-marker 12 '% 11)
(s-and 11 7 14)
; condition III
(s-marker 8 15 '(seq role))
(s-marker 9 16 '(seq role))
(s-comm-end)
(s-and 12 '% 17)
(s-marker 17 18 '(spread isa))
(s-and 11 '% 19)
(s-comm-end)
(s-clear-marker 19 '% 18)

(cond ((and (null (s-collect 4))
(null (s-collect 14))
(null (s-collect 18)))
```

```
(format t " (show-perf)
```

4 The Organization of the SNAP Simulator

The SNAP simulation program consists of approximately 4000 lines of Sun Common LISP code. This language was selected on the basis of SNAP's use for symbolic processing. The simulation program is organized on the basis of the hardware structure of SNAP as shown in Figure 3. The multitasking facility provided in Sun Common LISP is used to create a subprocess to simulate the **functions** of each module in the hardware block diagram. Two primary processes are created in background when the simulator is running: a controller process and a chip-array process.

In addition, some assumptions have been made in the implementation of the SNAP Simulator:

- the marker unit, processing unit, and communication unit are simulated separately and their results are combined, although they operate in parallel in the actual SNAP hardware,
- the marker unit forms messages in a sequential manner,
- if there are conflicts between the marker unit and the processor unit, the marker unit has priority
- if there are conflicts between the marker unit and the communication unit when accessing queues, the communication unit has priority
- each SNAP instruction has its own weight of execution time: when there is no message for the marker unit, the execution time of current step is based only on the instruction execution time.

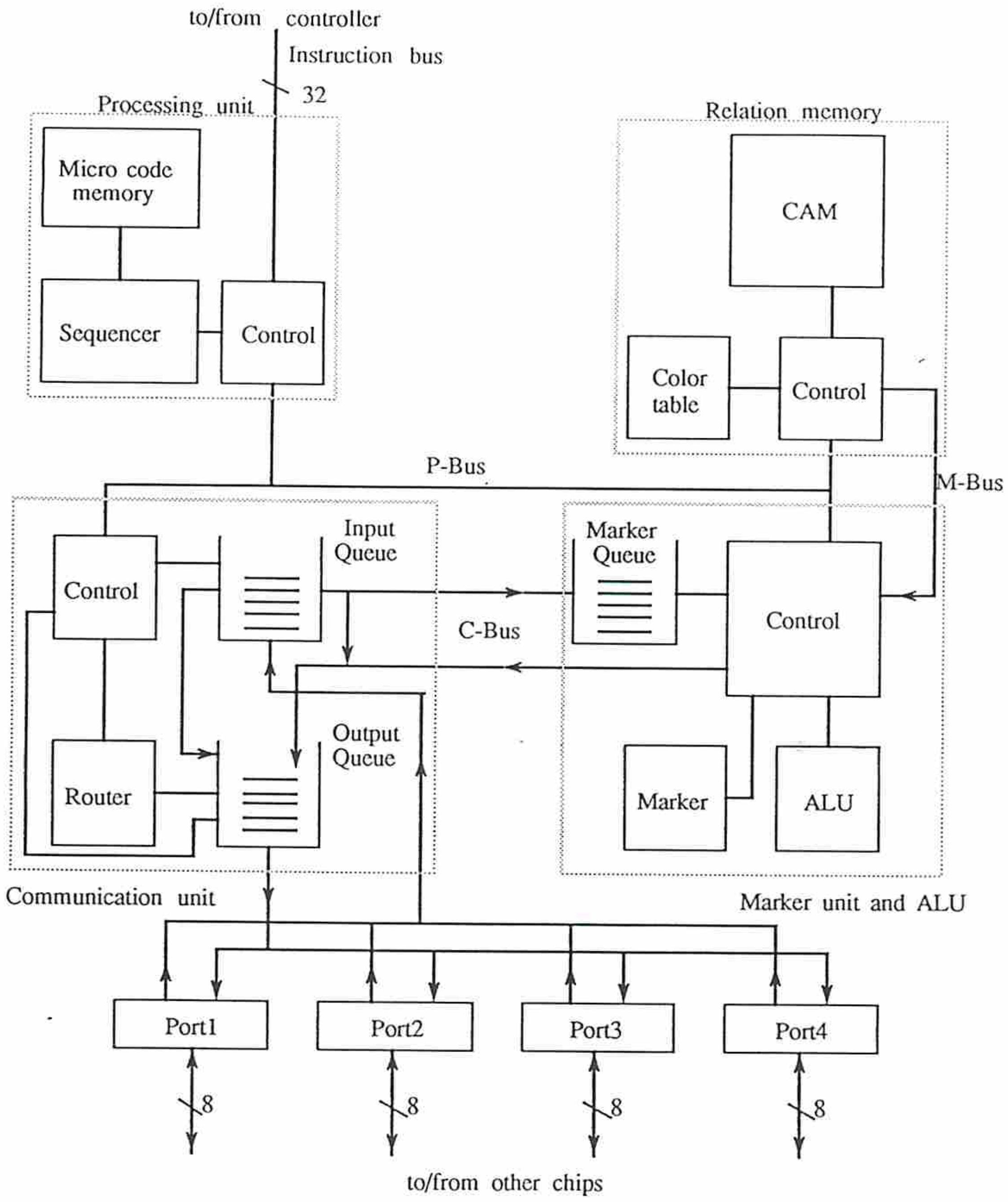

Figure 3: SNAP Chip Hardware Structure

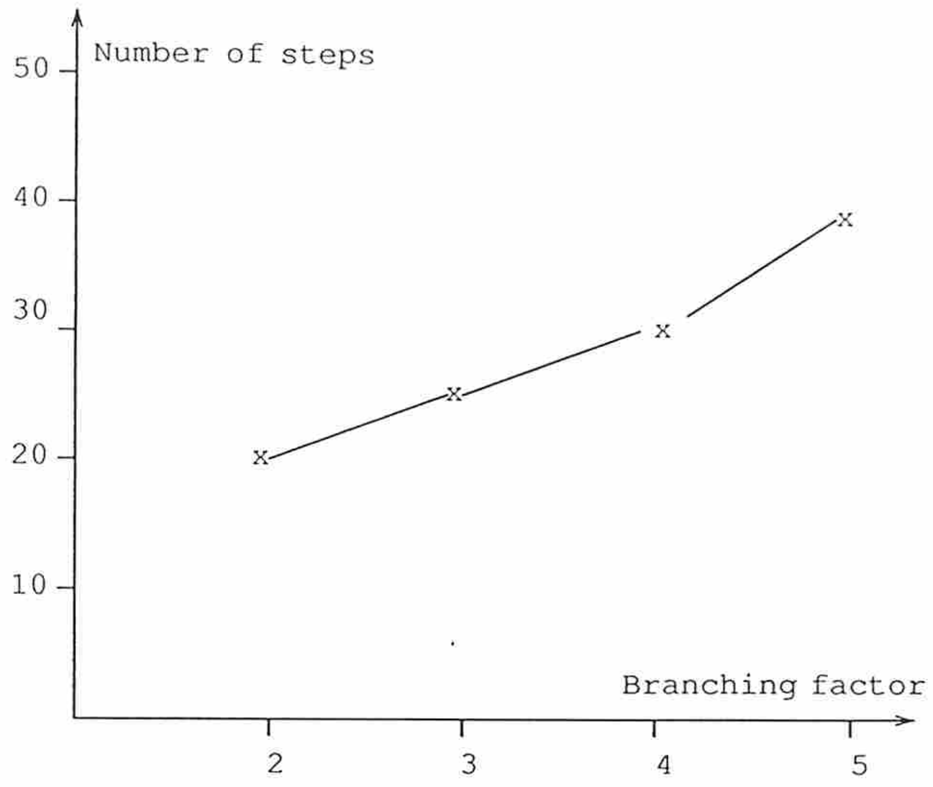

Figure 4: Simulation result for different Branching Factor

5 Impact of the SNAP Simulator

The simulator has made it possible to achieve iterative improvements in the SNAP architecture. This includes enhancements to the propagation rules, the interconnection network, and the instruction set which would not have been evident without concrete simulation results. This section describes some of the results precipitating specific modifications and enhancements to the SNAP hardware and architecture.

5.1 Braching factor

The branching factor was studied through a semantic network which had a tree with height of four. The first problem was to match the query with the network having different number of nodes and branching factors. The output of this operation was the matching time as a function of branching factor. The result is shown in Figure 4.

Number of chips	Execution time
1	325
2	1029
4	421
32	325

Figure 5: Results of Different Allocation Schemes

5.2 Allocation Scheme

There are two different allocation schemes which can be applied on the SNAP (1) sequential allocation, which completely fills chip memory before continuing to allocate nodes in the next chip, (2) spiral allocation scheme, which will perform breadth-first type of allocation. The default mode in the simulator is to allocate the nodes with the spiral allocation scheme, but can the user can modify this parameter if desired. Furthermore, the performance of different allocation schemes may be readily compared.

For example, without reassigning the node allocation a result derived from a simulation run with only one SNAP chip (using sequential allocation), the results were applied for 32 SNAP chips (using spiral allocation) were compared. Other configurations varying the number of chips were also compared. The result is shown in Figure 5. This behavior has some interesting implications: (1) For only one chip, no external communication is needed and therefore the processor utilization is highest. (2) For two chips, the communication overhead is the increased, forming a worst case system throughput. (3) From 4 chips up to 32 chips, the performance increases up to a limit (around 350 cycles) which means the communication overhead is approaching a constant value. From the results above, neither spiral allocation nor sequential allocation is a very good allocation scheme for the SNAP. Thus from this simulation it is evident that the optimization of the allocation scheme, also requires knowledge of the communication pattern used.

6 Conclusion

With the long development time and high cost of a full-scale or prototype hardware implementation, a custom simulation program such as the SNAP Simulator can be of significant assistance in the development of a new computer architecture. Potential benefits include early performance estimates and feedback on instruction set completeness and orthogonality. Furthermore, as the hardware becomes available, several high level modules in the simulator can be used as hardware drivers with little or no modification.

7 References

References

Chung 1989 Chung, S., Moldovan, D. and Tung, Y. [1989] "Reasoning on Connection Machine", Technical Report CENG-89-13. University of Southern California Department of EE Systems.

Lee 1989 Lee, W. C. [1989]. "Bandwidth Analysis of Message-Passing Networks". Technical Report CENG89-24. University of Southern California-Department of EE Systems.

Moldovan 1983 Moldovan, D.I. [1983]. "An Associative Array Architecture Intended for Semantic Network Processing" USC Dept of EE-Systems, Tech Report PPP 83-8.

Moldovan 1985a Moldovan, D.I. and Y.W. Tung [1985]. "SNAP: A VLSI Architecture for Artificial Intelligence Processing" *Journal of Parallel and Distributed Computing*, 2:2, 109-131.

Moldovan 1989 Moldovan, D.I., Lee, W. and Lin, C. [1989]. "SNAP: A Marker Propagation Architecture for Knowledge Processing", USC Dept of EE-system, Tech Report CENG89-10.

8 Appendix: Booting the SNAP Simulator

The SNAP Simulator is installed in a SUN 3/280 system server under the directory:

/home/gringo/changhwa/lisp/snap/

There are several versions available. The simulation program under the v3.2 directory is the most recent and complete version. It also provides several controller functions.

The simulation program can be started by performing the following procedure:

1. Enter the SUN COMMON LISP environment (either full or base version).
2. Load the simulation program by the command

   /home/gringo/changhwa/lisp/snap/v3.2/main.lbin

Note: to avoid keying the tedious path name to reach the simulation program, one can make a path in his own working directory directly to the simulation program.

After loading the simulation program, the menu in Figure 6 will appear. The default configuration for the SNAP hardware can be obtained by simply pressing the default (CAPS key followed by the RETURN key) for each response. The SNAP will be created per the specification. The SNAP Simulator is now booted and you may proceed with loading and execution of the SNAP application program the same as any regular LISP program.

```
****** SNAP Array Processor Simulator Ver 3.0 ******

The SNAP is configured as in the following table
Number of SNAP chips in array (256 maximum) : 64
Number of nodes in a SNAP chip (64 maximum) : 32
Number of markers available (64 maximum) : 32
Average memory space per node (10 maximum) : 5
Size of marker queue per chip for marker unit (16 maximum) : 16
Size of input queue per chip for incoming messages (16 maximum) : 16
Size of output queue per chip for output messages (16 maximum) : 16
Do you want to change any of them? (y or N) :
Do you need only one SNAP Array? (Y or n):
Which level of tracing information do you want (1...5)? : (normal 2)
```

Figure 6: Simulator Start Up Menu