Though the architecture doesn't possess a physically centralized common memory, a concept of global distributed storage has been implemented (namely, a virtual shared memory is obtained by partitioning a single global address space into local blocks of memory). The prototype currently operating at our labs is shown in the following figure, comprising a 16-Transputer Array (TR0,...,TR15), with 2Mbytes of RAM each, connected through a MSI implementation of the network, and a host processor, used for I/O purposes (leading programs, collecting solutions, monitoring significant parameters of parallel execution, ...). The host processor is an ordinary PC-BIM, supplied with a BDM Transputer board; this Root Transputer is connected via links 1 and 2 to TR0 and TR15 respectively.

Our distributed implementation has been based on a new parallel version of C language, specifically tailored for Transputer systems, namely "Parallel C", by S. L. and S. SL,8. Difference from Occam, which only supports a CSP-like (Communicating Sequential Processes) communication model, based on synchronous message passing, Parallel C also allows a shared-memory communication paradigm. This is obtained by making use of the so-called "threads" (dynamically startable flows of execution, sharing code and static data). Of course, care must be taken to avoid undesired conflicts on shared locations; appropriate "semaphore" primitives are provided by the run-time library for the purpose.

Node organization

To reduce inefficiencies due to remote accesses, program code is replicated on each node of our architecture; precisely, on each element of the Transputer array a copy of the task depicted in the next figure is loaded.

![Diagram](image)

Such an organization, compromising a total amount of four different threads, two semaphores and one shared buffer, has been the result of a number of constraints and choices, for instance:

(a) The low-priority `email' thread, comprising not only the strict emulation of divided `WAM' instructions but also some system "modules" such as the scheduler, the termination handler, etc., has been implemented as a large monolithic block, including as many functions as possible (in our opinion this was the most rapid way of porting the pre-existent VAX implementation).

(b) It is a crucial point to ensure that the high-priority input `drive' thread, receiving messages from other processors via the Delta network, must never be blocked; this implies that some form of non-blocking buffering for non-urgent messages (see later) must be provided. Therefore, communication between `drive' and `email' occurs through buffer `buf' and semaphore `buf_lock'.

Notice that the hardwired internal Transputer process scheduling is organized into two priority levels; while low-priority processes can be interrupted, either by re-activation of a high-priority process or at the end of a time-slice, an high-priority process must suspend itself.

(c) The absence of the ALT construct obliged us to introduce a semaphore `out_lock' to protect the shared `data_out' channel.

(d) A restriction of Parallel C forbidding threads at different priorities from sharing semaphores obliged us to add two intermedium semaphores: `buf_in_mark' (low-priority) and `make_output' (high-priority).

(e) Special care was taken to avoid potential deadlocks.

In the following, besides some considerations about scheduling and messages, the basic components of the process structure will be discussed:

A number of different messages are discriminated by `drive', mainly: (1) `Non-urgent' messages, to be delivered to `email' thread and not requiring immediate response (among these for instance, messages for the creation of a new process, for memory deallocation upon process termination, for sending a solution to the parent/pupil process). (2) Acknowledgements in response to `email' requests (e.g. when remote data are returned, or the result of a search on a remote Binding List). (3) Requests from the outside for remote access (handled by the component `remac.h'). (4) Requests from the outside for searching the local Binding List (handled by the component `search.BL'). (5) An `idle' mark message (see later), enabling a distributed load-balancing function. Lastly, other messages have been provided which are responsible for loading a new object file, resetting the system with new parameters, extracting statistical information.

At the end of the execution of each K-WAM instruction, before fetching next one, `WAM_loop' tests (consulting `mail_flag') if some non-urgent message has arrived in the meantime. If so, the message is delivered to the appropriate system module and immediately handled.

System modules (actually, collection of procedures) can be activated not only by external requests (i.e. when a non-urgent message is received from another processor), but also by internal requests (i.e. issued by the process currently running on that processor). For each module, different procedures are provided to handle both situations. (The only exception is `mem_han', implementing the demand fixed-length block memory handling policy, which responds to internal requests only.)

Messages issued by `email' can be synchronized or asynchronous. In the former case, `email' is suspended, waiting for an acknowledge (a request for remote access is an example); in the latter case, `email' continues execution, either going on with the same process (for instance, when a newly spawned process is distributed to a different processor), or suspending current process and restarting the first process of the scheduling queue (this occurs after notifying a solution to the parent/pupil process).

Distributed Scheduling

One of the main issues in a distributed multiprocessor is the implementation of an efficient parallel scheduling policy, able to achieve a fair load balancing of processes among processors. Two principles have been followed: a) in a physical architecture like ours, lacking a true common memory, a distributed policy is in any case to be preferred to a centralized one; b) in order not to compromise locality of reference, processes can be transferred only once; after a process has been inserted into the Scheduling Queue of a processing node, further migrations are forbidden (a process in the course of execution may have generated a lot of data structures in the local memory of a Transputer, which we don't want to move).

Our load-balancing mechanism operates on a demand basis, so being able to distribute
processes only when really needed. It works as follows: (a) there is a Scheduling Queue for each processor; (b) when new processes are spawned in a processor, they are inserted into the Scheduling Queue of that node, ordered on the grounds of priority information; (c) all the time in idle, mask messages circulate through a virtual ring network connecting all processors (realized via the Delta network); a bit set in a given position of the mask indicates that the corresponding processor is idle (i.e., the number of processes in that node is below a programmable threshold); (d) when a processor T_i receives the idle, mask message, if there are idle processors and if T_i has sendable processors (a process is sendable only if it hasn't been started yet; this measure is intended to reduce the cost of transmission and minimize non-local accesses), T_i sends one process to each idle processor, as long as there are sendable processors; at the end, idle, mask is propagated to T_{i+1} (note that if T_i is idle, before the propagation the corresponding bit in the mask is switched on). The above scheduling policy, which is able to favor locality of reference by preferably allocating a new process in the same node of the parent process, with minor adjustments and appropriate tuning has shown a very satisfactory load-balancing performance in a wide range of situations, and has been essential to achieve the speed-up figures reported in the next section.

Remote Access and Lazy Copying

The remote, has component of drive is in charge of responding to three different requests coming from the outside: (1) read a memory location (it should be stressed that, due to the nature of Prolog and our implementation of multiple environments, remote write operations are not required); (2) read a Stack Frame (a child process residing in another processor will eventually need a remote ancestor stack frame; before going on with the continuation, a new local copy is created); (3) read the top level of a substructure (this is the so-called copying mechanism: a remote structure is incrementally copied, each level at a time; note that the copy operation is performed only when actually essential, i.e., when unifying a remote structure with a local one), including the special producer structure. Note that remote, has is programmed so that it can ensure a very fast response, for this reason, it doesn't do any dereferencing operations and just limits itself to read and pass one or a number of locations (therefore, the control of dereferencing is left to the processor requesting the remote access).

Searching Remote Binding Lists

During the dereferencing algorithm the need for a search on BL may arise. If a binding is not found on the local BL, a remote search is initiated, starting from the processor reference in the bottom BL element. The search BL component of drive returns one of the following three results: (1) the desired binding, in case of success; (2) continuation processor (search must be continued in a different processor); (3) failure (if the end of BL is reached without finding a binding).

Solution Copying by parameters

The computational model of the language prescribes that solutions produced by descendant processes (originated by a parameter) must be wholly copied and collected in a list in the Heap of the parent process. A distributed implementation has to ensure some form of mutual exclusion in the copy operation, since different processes might compete in adding an element to the same shared list. In our system this is obtained by centralizing the function, in each processor, in the copyself module of emit thread, which handles with no interference both internal and external requests.

4. Performance of K-LEAF implementations

The performance of the sequential K-LEAF implementation on a variety of commercial machines and of the parallel implementation on Pipes is analyzed throughout this section.

The (averaged) figures in table[1] were obtained on slightly instrumented versions of K-LEAF executors running the following all-solutions benchmarks:

- quee1: the e-queue problem formulated as a pure generate (permutations) and test problem. It is the most "abstract" version of the problem. It exploits K-LEAF lazy evaluation.
- quee2: the standard a-queue Prolog benchmark, where the positions on the chessboard are generated by a predicate that non-deterministically outputs an integer from 1 to n.
- quee3: a more efficient Prolog version of e-queue where the positions on the chessboard are selected from a previously constructed list of integers from 1 to n.
- fib: the standard functional definition of Fibonacci numbers (AND-parallelism).
- s&n: the standard salt and mustard Prolog benchmark with meta-calls.
- hamilton: all the hamiltonian paths of length 7 in a fully connected graph of 10 nodes.
- matmul: multiplication of a matrix 400x400 by a vector of 400 elements (AND-parallelism).
- primes: a Prolog formulation of the Eratosthenes's sieve for computing the prime numbers.
- logig: an event driven logic simulator executed in "inverted" mode to act as a fault-finder in a two-bit adder. It exploits K-LEAF lazy evaluation [Bock88].
- image: the high level phase of an image understanding system described in [Maio89].

According to our principle of a user controllable parallelism, the parallel versions of some programs (quees1, fib, image, logig, hamilton, s&n) have been quite straightforwardly derived from the sequential ones, by identifying the sources of useful parallelism and putting the right annotations on OR-parallel definitions and AND-parallel goals. In particular in quees1 and s&n the test phase has been kept sequential. Moreover, in quee1, quee2 and fib an extra parameter has been added in order to play with the granularity of parallel processes. This parameter represents the threshold below which the computation must proceed sequentially, executing the sequential (non-annotated) program. So for example fib(22) means that recursion will generate parallel processes only down to fib(0) [12].

4.1 Performance of the sequential implementation

The emulated version (C-SEM on Sun 3/280) is in 3-4 times slower than a compiled (into machine language, as Quintum 2.2) Prolog.

<table>
<thead>
<tr>
<th>Program</th>
<th>C-exp</th>
<th>Sun 3/280</th>
<th>C-Sem</th>
<th>C-exp T100</th>
<th>Sun 3/280 T100</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(27)</td>
<td>21900</td>
<td>63000</td>
<td>4800</td>
<td>105644</td>
<td>9600</td>
</tr>
<tr>
<td>quee1(5)</td>
<td>5150</td>
<td>12000</td>
<td>1000</td>
<td>17446</td>
<td>1810</td>
</tr>
<tr>
<td>primes(1000)</td>
<td>6100</td>
<td>21700</td>
<td>2000</td>
<td>21700</td>
<td>2130</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td></td>
<td></td>
<td>79860</td>
<td>753949</td>
</tr>
</tbody>
</table>

The most interesting aspect is the viability of a portable compilation route from K-LEAF and Prolog directly to C (C-exp). The efficiency of this experimental compiled code is comparable with Quintum 2.2. This is mainly due to the highly sophisticated C compilers available for RISC.
machines which make less and less economic the compilation from high level languages into machine code. Better results come out when the compilation also includes some simple optimizations like choice-point elimination in case of if-then-else constructs and mapping of arithmetic expressions directly into the corresponding C expressions, after denesting variables.

Comparison of C-cmp with the C column (with a rate from 1 to 5), where the execution times of efficiently implemented C expressions versions of the same algorithms are reported, suggests the applicability of well compiled and optimized logic/functions to wider fields of programming than the traditional rapid prototyping or AI applications.

A look to the Dec300 column is in order. Its absolute figures (600RIPS on a 10-11 MIPS machine; a 2 MIPS version is already available) should be carefully considered before undertaking the adventure of designing a brand new sequential specialized processor.

4.2 Performance of the parallel implementation

Along this section we will concentrate on the parallel implementation of K-LEAF on the 16-nodes PIPES prototype. Most of the measurements were conducted on the system of distributed Concurrent K-WAM's, each being an OR-parallel extension of the sequential K-WAM.

<table>
<thead>
<tr>
<th>Program</th>
<th>Tseq</th>
<th>TI</th>
<th>T16</th>
<th>Tseq</th>
<th>T16/Tseq</th>
</tr>
</thead>
<tbody>
<tr>
<td>queen58</td>
<td>32987</td>
<td>4322</td>
<td>3080</td>
<td>14.06</td>
<td>1.32</td>
</tr>
<tr>
<td>queen97</td>
<td>154899</td>
<td>20919</td>
<td>13937</td>
<td>14.94</td>
<td>1.34</td>
</tr>
<tr>
<td>queen28</td>
<td>23167</td>
<td>15974</td>
<td>1840</td>
<td>11.57</td>
<td>1.08</td>
</tr>
<tr>
<td>queen38</td>
<td>9635</td>
<td>10982</td>
<td>960</td>
<td>11.43</td>
<td>1.13</td>
</tr>
<tr>
<td>flip(22)</td>
<td>26367</td>
<td>27701</td>
<td>1867</td>
<td>14.85</td>
<td>1.05</td>
</tr>
<tr>
<td>logtim</td>
<td>106873</td>
<td>8106</td>
<td>13.18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>2265</td>
<td>2312</td>
<td>200</td>
<td>11.56</td>
<td>1.02</td>
</tr>
<tr>
<td>image</td>
<td>2404</td>
<td>2404</td>
<td>2400</td>
<td>10.01</td>
<td>1.00</td>
</tr>
<tr>
<td>malloc(400)</td>
<td>83916</td>
<td>6765</td>
<td></td>
<td>12.40</td>
<td>401</td>
</tr>
</tbody>
</table>

Good quality speedsups are exhibited by programs with a great deal of potential parallelism (OR for queen*, AND for f1b) and limited communication. Matrix multiplication (AND parallelism) is in its original version on line hardly achieved a speedsup, due to the high rate of remote accesses to the node holding the matrix. The final version adopts a structured representation for matrix rows, which causes a complete row copying upon the first remote access.

The distributed binding scheme seem to cover quite well both direct K-LEAF (and Prolog) and lazy K-LEAF programs (queen58, logtim). The distributed search in binding list that was found to be a serious bottleneck in highly lazy programs (where continuations and their result variables are built on the host) is acceptably efficient when mixed with OR-parallelism. In many cases logtimes in expanding the search space means locality in binding lists and shorter searches.

In the last column the number of generated processes is included. This is an indication of the amount of parallelism present in the algorithms and can serve to understand the differences in speedup for the different programs. As a matter of fact T1/Tseq gives a low inherent parallelism.

Comparison among sequential times Tseq (sequential programs by sequential executor or a TI00) and parallel times (Tseq/TI) of each speedup which, for the benchmark with stronger application flavour like logtim and image are beyond the critical efficiency threshold of 50% which is the threshold of interest for parallel machines. The column T1/Tseq gives the cost of parallelization. It reaches its maximum (356) when most of the variables have to be searched. The binding list (like in queen58) is not controlled (535 generated processes). In the other cases, where suitable sequential pieces can be guaranteed (like the safe test in queen1) the overhead can be kept less than 10%. The low overhead, 5%, imposed by the parallelization of f1b is an indicator of an effective mapping of AND-parallelism into OR-parallelism by parameter.

The following table reports the parallel performance of queen58 and f1b as functions of the granularity, which is controlled by an extra parameter. This is mirrored by the number of generated processes (Npro) and the average size (gran = Tseq/Npro). The sequential test in queen51 guarantees large granitize. The impact of granularity is more evident in f1b, whose body is only made of three arithmetic operations and two calls. Parallelization efficiency around 50% (best speedup/2) is obtained with granizes of 0.1-1 milliseconds.

<table>
<thead>
<tr>
<th>Program</th>
<th>Nproc</th>
<th>gran</th>
<th>T16</th>
<th>Tseq</th>
<th>Tseq/TI</th>
</tr>
</thead>
<tbody>
<tr>
<td>queen1</td>
<td>3</td>
<td>356</td>
<td>65</td>
<td>1690</td>
<td>12.30</td>
</tr>
<tr>
<td>queen51</td>
<td>27360</td>
<td>8400</td>
<td>1690</td>
<td>23167</td>
<td>12.60</td>
</tr>
</tbody>
</table>

C-expansion on the PIPES architecture was tried in order to be able to show the possibility of getting a better absolute performance. However a quite limited effort has been devoted to it, only on sequential standard WAM implementations, which form the core of both parallel Prolog and K-LEAF systems, while the parallel structure is still generated by emulated instructions.

<table>
<thead>
<tr>
<th>Program</th>
<th>Tquintus/625kW</th>
<th>Tseq</th>
<th>TI</th>
<th>C-exp</th>
<th>Tseq/TI</th>
<th>C-exp</th>
<th>T1/T16</th>
<th>C-exp/T16</th>
<th>C-exp/T16 Imulated</th>
</tr>
</thead>
<tbody>
<tr>
<td>queen1</td>
<td>12000</td>
<td>17446</td>
<td>19482</td>
<td>1637</td>
<td>15.00</td>
<td>11.90</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f1b</td>
<td>47000</td>
<td>106844</td>
<td>113722</td>
<td>7750</td>
<td>15.30</td>
<td>14.37</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As regards the speedsups reported in the last double column, as expected, compiled speedsups are worse than emulated ones. One reason is the finer granularity of compiled sequential parts. This is reasonably compensated by "scaling up" the problems of a factor comparable with the compilation overhead, for example moving from queen1 to queen97. Some ineffectiveness is still present, principally due to the increased rate of remote accesses over sequential execution. This is because we have faster sequential threads, while the communication run-time support (C procedures, high priority threads) have not been changed (the cost of a remote access remains the same, approx. 200usecs). In any case the parallel performance of the 16-node machine is still more than 6 times that of Quintus/2.5 on a 4MIPS machine. An improvement of this part is still possible by cooling critical parts of the concurrence run-time support in a assembly code. A final implementation should introduce context switching to compensate network latency on remote accesses.

The number of accesses to nonlocal memories are reported below for some benchmarks. For sake of simplicity we have conduced to a single number the sum of the various kind of such remote accesses, namely: accesses to read the content of a variable, accesses to get the top level of a structure, accesses to remote binding lists, accesses to a remote frame. Their cost in the present prototype is quite high: it can be uniformly approximated to ~200usecs (with ~40usecs taken by physical communication of 40 bytes, and 160usecs taken by 4 switches' of Parallel-C thread). Since the number of remote accesses is very much program dependent, reflecting the way structures are allocated throughout the program evolution, the induced overheads (Taux + rem-acc+200usecs), calculated w.r.t. T1, are significant only for the particular programs considered. They are rather small for the lazy and strict versions of queen58, while, in the case of logti im, a lazy program largely dealing with streams communicating nested data structures, the
non-local accesses to closures and data make the overhead more significant (15%).

<table>
<thead>
<tr>
<th>Program</th>
<th>Spruce</th>
<th>PE-8</th>
<th>PE-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>queen(0)</td>
<td>5094</td>
<td>44</td>
<td>123</td>
</tr>
<tr>
<td>queen(0)</td>
<td>735</td>
<td>513</td>
<td>80</td>
</tr>
<tr>
<td>fib(22)</td>
<td>2351</td>
<td>287</td>
<td>123</td>
</tr>
<tr>
<td>login</td>
<td>2021</td>
<td>484</td>
<td>79880</td>
</tr>
</tbody>
</table>

The two prone columns report the number of processes transferred, according to the load balancing algorithm whose efficiency is shown by the following diagram. It reports on the lower scale the number of ready processes present in the architecture and in the upper scale the variation of such a number in the range 0-16, coupled with the number of active processors (gray area). The upper black area is a measure of inefficiency in trying to keep processors working on available tasks.

4.3 Lazy vs. eager computations

A quick comparative analysis of benefits and drawbacks of outermost (lazy) resolution of the functional component of K-LEAF is tried in the following.

Three kinds of computations are in order about 1) the "programming" power offered by laziness, 2) the capability of better controlling the exploration of a search space, 3) the overheads induced by lazy computations.

The two selected benchmarks are witnesses of the ability of a lazy L-4F language to clearly and simply express the problem: a pure generate and test problem in the case of queen(s), a cyclic network of stream-connected agents in the case of the logic simulator (inverted to work as a fault finder).

Regarding point 2) comparison between eager and lazy computations shows the usefulness of laziness in performing a lazy expansion of the search space. Analogous results can be obtained, only in the sequential case, by Prolog systems with dynamic strategies, like Nu-Prolog [Nash85], but at the cost of an awkward annotation of the program.

However, coming to point 3), this does not mean that to write a Prolog program with the same search behavior is impossible: queen(s) is such a program where the generate and test phases have been intermixed in order to explore exactly the same search space. But this has been obtained at the cost of a clever program transformation and in general this may happen to be extremely complex. Anyway in this case, and in most of the cases, where a lazy strategy (and thus closure construction on the heap) is adopted, but not necessary, the overhead is in the range 200%, 300%, both in the sequential and the parallel environment. Among such programs we also consider those compiled as fully lazy, in spite of the fact that they are inherently strict. 5. Related works

While a substantial experience is becoming available on the implementation of logic languages on shared memory parallel architectures, very few attempts on distributed structures have been tried and already evaluated.

As for the shared memory approach, an excellent work has been carried on by the PeySyp Project [Bann88] and the FutOpt Project [Szeu89] both based on SRI model.

The implementation of Aurora on shared-memory multiprocessors is in an advanced state [Terz89] showing linear speedups with high efficiency and limited cost of parallelization (<5%). The first rough comparison seems to indicate that (for 11 nodes of a Symmetry) they gain a 4-10% in relative speedup (perhaps due to the advantages of shared memory) while our absolute speedups are better (5-10%) due to a lower cost of parallelization, thanks to our annotations. It is worth to note that Prolog programs can be directly executed in parallel by Aurora (even if no annotations are introduced), while in our case they must be slightly modified (e.g. no cut in a parallel procedure is allowed) because of the stronger nature of our annotations and of the more rigorous interface among sequential and parallel computations, but, on the other hand, such annotations can guarantee a better control on the granularity of computations.

Regarding AND-parallelism, a shared memory implementation has been developed on the Symmetry at MCC, [Hermengildo89]. The only benchmark reported is matrix multiplication which has a linear speedup with 90% efficiency on a 10-node Symmetry. In our case, where pieces of the matrix have to be transmitted along the network, efficiency is around 80%.

W.r.t. shared memory architectures, the very similar nature of results tells us that the gain in scalability we get by moving from shared to distributed memory does not necessarily involve dramatic falls of performance. This is true if the programmer is conscious of working in a distributed parallel environment and able, through a suitable programming style [Buddell88] and simple annotations [Hirschmnn88], to control the evolution of parallel computations.
Among the approaches to a (almost-)distributed memory implementation of parallel Prolog, the BC-machine [AIL88] seems to have reached a good stage of development. It is based on intensive copying instead of a logically shared cactus stacks. Preliminary results [AIL89] on a Sun/350 array show better (10%) absolute speedups on standard OR-parallel benchmarks, with a smaller (than SRF) parallelism scheme. However, it is not yet clear how much this approach is applicable to a variety of programs, where the OR branching is not always performed at the beginning.

A recent attempt to put Aurora on a switch-based multiprocessor is described in [Mudamb91] where the target machine is IBM Butterfly GP/1000 (up to 40 processors) which is similar in spirit to our machine. Their cost of parallelization seems to be very high (more than 100% overhead w.r.t. SRF/5200) compared with ours (at most 30% on heap-intensive programs); moreover for queen(9), the only benchmark analyzed in detail, relative speedup (TU/T15) is 7.4 and absolute one is only 3.68. The speedup efficiency is thus considerably low.

Our results on the implementation of Prolog and K-LEAF on a distributed architecture show that exploitation of OR and AND parallelism inherent in logic and functional languages is possible on highly-scalable architectures, with efficiencies (w.r.t. sequential compiled code) comparable with those obtained on bus-based machines, provided that: 1) the network efficiently support both block transfer and non-local memory accesses in an any-to-any mode, e.g., by "cut-through" packet switching (note that the current PIPEs prototype does not have hardware support for processor-to-non-local-memory interface and context-switch as WAM level); 2) the programmer has a way to control, in an easy and disciplined way the grain size of sequential computations; 3) in some cases, the programmer is conscious of the distributed nature of the machine, and knows how to accordingly tune the size of data structures transferred by a non-local access (like in matmult benchmark).

Acknowledgment

This work has been partially supported by ESPRIT Project n. 415.

6. References


Verso un Supporto a Tempo di Esecuzione per Linguaggi Logici e Funzionali

Mauro Gaspari
Franco Saracco

DELPHI
Via Della Verna, 11,
Viareggio, Italia.

Net: mgaspari@delphi.usdp
fsaracco@delphi.usdp

Sommario

In questo articolo viene descritta un’implementazione del Prolog in un ambiente COMMON LISP. L’implementazione è basata su una estensione della macchina astratta Lisp con primitive della macchina astratta di Warren. Il concetto di variabile logica è stato introdotto in COMMON LISP. La nuova Macchina Astratta garantisce una piena interoperabilità tra i due linguaggi; il supporto a tempo di esecuzione è costruito.

1. Introduzione

Gli studi nel campo dell’Intelligenza Artificiale hanno dato origine ad un insieme di nuove tecniche di programmazione che possono essere utilizzate per articolare e integrare le tradizionali tecniche e metodologie per lo sviluppo del software. In particolare, per quanto concerne i linguaggi di programmazione, questo tipo di studi ha contribuito alla nascita di linguaggi basati su nuovi paradigmi di programmazione, quali quello funzionale, logico, ad oggetti. Sostanzialmente le tecnologie di IA permettono di elevare il livello di astrazione a cui vengono scritti i programmi, colmando in parte l’abisso che esiste tra il cervello dell’uomo e la macchina di von Neumann. Dal punto di vista implementativo questi linguaggi sono caratterizzati da diversi e specifici supporti a tempo di esecuzione la cui complessità è maggiore se maggiore è la complessità del linguaggio. Da quest’ultimo dipendono inoltre l’ambiente di sviluppo e l’insieme di strumenti di programmazione associati.

D’altro canto applicazioni complesse come la maggiore parte di quelle che si sviluppano in Intelligenza Artificiale, richiedono che parti diverse di codice vengano scritte con linguaggi diversi. Questo sostanzialmente per due motivi: il primo è che un linguaggio di programmazione può risultare più adeguato a un altro per risolvere un certo sotto problema, il secondo è basato