A PARALLEL UNIFICATION COPROCESSOR

F. N. Sibai K. L. Watson Mi Lu

Department of Electrical Engineering
Texas A&M University
College Station, Texas 77843.

ABSTRACT

Unification is a pattern matching operation very frequently used in Artificial Intelligence. Logic and PROLOG interpreters and theorem provers heavily rely on unification. Since its beginning, unification has suffered from a slow execution. The execution time of logic programs and other programs based on unification can be significantly reduced by improving the performance of the unification operation. In this paper, we introduce a parallel unification coprocessor for speeding up the unification algorithm. The machine is simulated at the register transfer level and the simulation results as well as performance comparisons with two serial unification coprocessors are presented.

1. INTRODUCTION

Currently, the field of Artificial Intelligence (AI) is gaining in popularity and actively expanding into many areas of application. AI has served as the basis for the fifth generation computer project with the primary goal of replacing the traditional Von Neumann computers by smarter ones capable of reasoning, learning, making inferences and decisions, and understanding speech and pictures [1]. The logic programming language PROLOG and the functional programming language LISP are widely considered as the two primary programming languages available to AI programmers and knowledge base developers. PROLOG's statements in the form of Horn clauses, its argument match capability, and nondeterministic and database management features make it very suitable for AI and expert system applications.

However, in its present form, PROLOG is time-consuming and very inefficient when run on sequential general purpose machines. For instance, Abe [2] notes that PROLOG's performance drops to one tenth of the performance of procedural languages (*e.g.*, C, FORTRAN) when executed by a general purpose computer. This is clearly why today's expert systems and other AI application programs implemented in PROLOG appear to run very slowly on traditional general purpose computers. In order to improve the performance of PROLOG, several implementations of PROLOG incorporate a number of parallel logic programming schemes such as Conery's AND/OR model [3]. Ponder and Patt [4] described each of these PROLOG implementations and compared them. Another effective way to eliminate PROLOG's inefficiency and slow execution on these machines is to improve the performance of the unification algorithm on which PROLOG interpreters and other logic programming interpreters are built. In addition to being frequently used in logic programming, unification has applications in databases, theorem proving, expert and knowledge-based systems, and natural language and image processing.

Unification is an operation which attempts to make two terms equal and often generates conditions for this equality to hold. These conditions appear under the form of variable substitutions or bindings. For instance, the unification of the two terms $f(X,a)$ and $f(b,a)$, where a and b are constants and X is a free variable, succeeds (and the two terms become equal) if the first argument of the first term, X, is bound to the first argument of the second term, b. In this example, the substitution set generated by the unification operation contains the single binding X/b and is referred to as the *unifier*. In general, to successfully unify two functions, the heads(or functors) of the two terms, f, must be identical and the ith argument of term 1 must unify(match) with the ith argument of term 2, for all arguments of the functions. A free variable can unify with any term and, as a result of unification, generates a binding. Two constants can only successfully unify if identical. Two terms are said to be *unifiable* if the attempt to unify them is successful.

It is not sufficient that the functors be identical and the arguments be matched for the unification operation to succeed. The variable bindings must be consistent with each other. The unification of $f(X,a,b)$ and $f(c,Y,X)$, where X and Y are variables and a,b and c are constants, illustrates this point. To successfully unify these two terms, the heads and arguments of the terms are matched together yielding : f/f, X/c, a/Y, b/X. The functors match is successful since both functors are identical and the argument match produces three variable substitutions: X/c, Y/a and X/b. The first and second substitutions bind X to c and Y to a, however, the third substitution binds X to b, which is clearly a conflict with the first binding. Thus, in this example, unification fails.

The unification operation was shown by Woo [5-6] to consume on the average 55-70% of the execution time of PROLOG interpreters, far more than any other operation. To improve the performance of unification, several attempts are being made to design faster algorithms and to design serial and parallel unification machines solely responsible for executing the unification operation.

We introduce in this paper a parallel machine to speed up the unification algorithm. First, in the next section, we present the serial and parallel unification machines which have been proposed or developed up to the present. Secondly, we describe the architecture of the parallel unification machine and the algorithm designed to run on it. Next, the organizations of the two different processors in the machine are briefly discussed. Finally, the results of the machine's simulation are presented.

2. HARDWARE FOR UNIFICATION

Unification was originally developed by Robinson [7] as the heart of the Resolution principle around 1965. The original algorithm had an exponential time complexity. Several attempts have been made later in the 1970s to devise a faster algorithm. Perhaps the best sequential unification algorithms are Paterson and Wegman's algorithm [8] with linear time complexity and Martelli and Montanari's algorithm [9]. However, a linear time complexity for the unification algorithm fell short of making PROLOG's performance acceptable and attempts to create a parallel algorithm had to be made.

In the 1980s, in the midst of strong efforts for parallelizing unification, Yasuura [10] showed that unification contains essentially sequential computation which might not be accelerated by a parallel computation scheme in the worst case. He stated that it is very difficult to design a parallel unification algorithm in time $O(log^k n)$ (n being the number of terms to be unified and k being a constant) even if an infinite number of processors is used. Furthermore, Dwork [11] claimed that parallelism cannot significantly improve the performance of the best sequential solutions for unification which is composed of a term matching step and a binding check step. The subproblem of binding consistency check is believed to be serial in nature unlike the subproblem of term matching, which was shown to offer a potential for parallelism. Since software attempts to improve the performance of unification seemed to be restricted by the serial nature of the consistency check, hardware implementations quickly became the obvious alternative.

In 1981, Chang [12] designed a machine to execute Robinson's unification algorithm. This one-chip machine is composed of a controller, stacks, registers and an equation table — an external RAM where the two terms to be unified and the results of the computation are stored. After executing a current unification task, the machine records the binding information in the equation table if unification is successful, otherwise it indicates the failure of the operation. The chip was never built and no performance evaluation was made.

The SUM unification coprocessor [13] was designed to support an LMI Lambda LISP machine. The SUM coprocessor conducts unification assisted by a content–addressable memory (CAM) for fast access of binding agents.

Another unification coprocessor, consisting of a hardware processing unit and a variable stack, was developed by Woo [5–6], and was shown to improve the performance of unification considerably. The unify function implemented in hardware is composed of eight routines, each dealing with one data type. This unification unit holds the result of unification in a flip-flop. If set, it indicates that the two input terms are unifiable. If nested functions are encountered, the unit invokes itself recursively for each nesting. Woo measured the AT&T unit's performance to be 14-15 times faster than the UNSW interpreter's unify function on a VAX 11/780.

Also, Gollakota [14] designed a 54-pin unification coprocessor equipped with an on-chip binding memory, which is half CAM and half RAM. It is designed to receive the address of two lists of terms and the arity denoting the number of terms in the lists, and unify all the terms in the lists.

Most of these coprocessors were shown to speed up unification's execution. To speed up unification even further, other approaches must be explored. Although the studies by Yasuura and Dwork do not strongly encourage parallelizing unification, parallel processing and hardware techniques must be explored fully due to the high frequency of the unification operation in logic programming interpreters.

Shobatake [15] designed a cellular systolic array to implement unification. The critical characteristic of his design is that for n symbols in the input terms, the required number of cells which is the length of the terms is very large and is in the order of $O(n^{2n})$.

Chen [16] proposed an overlapping algorithm originating from Robinson's algorithm with a one–dimensional systolic–like architecture. No measurements were made.

Shih [17] proposed an array of mesh–connected unification units to speed up AND parallelism in clauses. Shih also proposed four binding algorithms and studied their performance on his proposed architecture. His machine was designed to be used as a coprocessor to be invoked mainly when the number of siblings in a clause body and the size of the logic program are both large.

Inagawa [18] implemented a multiprocessor system for unification and showed that the unification parallelization effect was evident for a small number of processing units. In this system, the consistency check operation is conducted with the shared memory and the shared variables' old bindings are read from memory by the processors and checked for consistency with the new bindings before updating the variables' cells in memory. This requires however a lengthy backtracking step to restore the variables' cells in the main memory if the unification operation ends with failure.

3. PARALLEL UNIFICATION MACHINE

3.1 ARCHITECTURE

We describe in this section the architecture of the parallel unification machine for speeding up the unification operation and present the unification algorithm designed to run on it. We base our algorithm and system architecture on the fact that unification is composed of two steps, the first being the match step which offers a high level of parallelism, and the second being the consistency check step with a low potential for parallelism.

The machine performs the unification operation on two terms and outputs failure, or the variable bindings in case of success, with the following guidelines: i) the match and consistency steps must be performed concurrently; ii) there must be a fast backtrack operation in case of failure; iii) the machine must function as a coprocessor supporting a host processor. To parallelize the match step, concurrent execution of term matching is to take place in a number of identical processors to speed up the matching phase of unification. This requires a number of identical processors called Match Processors(MP) which perform separate argument matches in parallel. In this way, any argument match that results in failure is detected quickly and the whole unification operation is stopped with result: FAIL. At the end of the matching phase, the processors, one by one, send the resulting unification substitutions or variable bindings to a special processor(CCP) equipped with a CAM to perform the consistency check step. This allows the consistency check on the current binding and the matching of the next arguments to be performed in parallel. The CAM, which is to hold all bound variables, is designed to speed up the access to the variable bindings. The unification machine is to serve as a coprocessor for a host processor with the purpose of conducting the unification tasks needed by the host. The host supplies the addresses of the two terms to be unified in memory to the coprocessor which generates the variable bindings and sets the succeed (SU) flag if unification succeeds, otherwise it sets the fail (FA) flag. The host-coprocessor configuration is shown in Figure 1.

The unification machine architecture is shown in Figure 2. Here, a shared memory holds initially the two terms to be unified. The control unit(C.U.) controls the read operation of these two terms. Initially, the addresses of the two terms are fed to the C.U. which dynamically schedules the first arguments of the two terms to be matched in the first available MP. While this MP is conducting the match operation on the first two arguments, the next arguments of the two terms are read by another free MP. This last step is repeated until all MPs are busy or until all arguments have been scheduled. If the match operation of any MP results in failure, the C.U. raises its FA flag signaling that the result of the whole unification operation is FAIL, and stops the operation of all the processing units in the system. This happens anytime any of the processing units in the system detects a failure in the unification of the two terms. In case the match operation conducted by the MP succeeds, the MP generates a variable binding(if at least one of the two matched arguments is a variable) as the result of the argument match and requests the transfer of the variable binding generated by it to the CCP processor. It can happen however that, at one time, more than one MP request the transfer of their variable bindings to the CCP. Thus, an arbitration algorithm is needed to select one MP to initiate the transfer and force the other requesting MPs to wait. The dynamic arbitration algorithm that was chosen is the Independent Request/First-Available From Left arbitration algorithm. We chose this dynamic algorithm over other algorithms because of its flexibility, good fault-tolerance, high speed and low cost. In this algorithm, the match job is scheduled to the first MP available from left, i.e., in case more than one MP is free, the job is granted to the requesting MP closest to the C.U.(first from

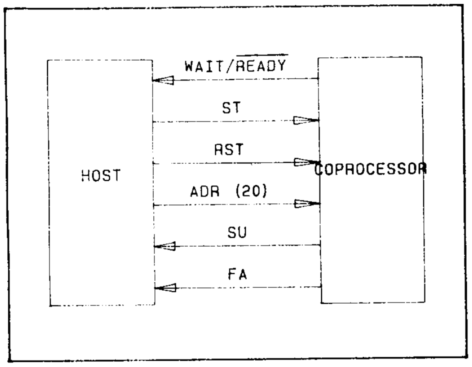

Figure 1. Host-Coprocessor Configuration

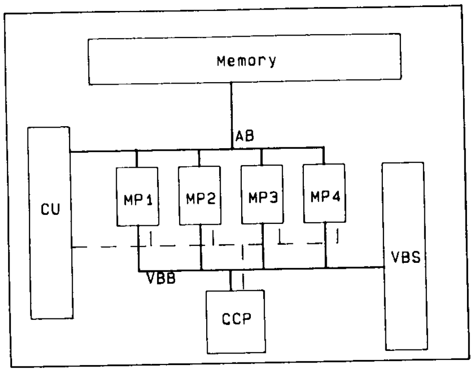

Figure 2. System Architecture

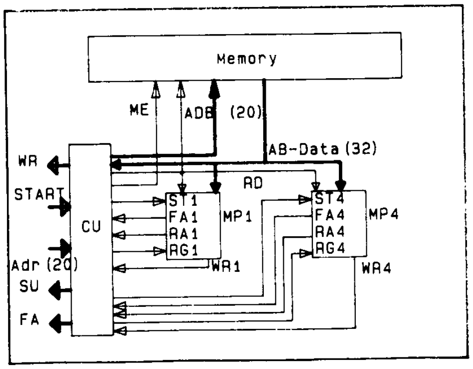

Figure 3. C.U.–MPs Configuration

left). Here, two lines are required for each MP: a REQUEST line(RA) and a GRANT line(RG) as shown in Figure 3. The MPs requesting transfer raise their RA lines, the C.U. then checks whether the CCP processor is not busy and if so, sets the RG line of the requesting MP nearest to it and leaves the other RG lines low.

After being granted the transfer, the MP with the set RG starts the simultaneous transfer of the variable binding to the CCP and the variable binding store(VBS). The variable bindings are stored in the VBS simultaneously as they are transferred to the CCP to be checked for consistency with the previous bindings, thus eliminating the overhead time needed to store the bindings at the end of the unification task in case unification succeeds. In case the unification operation fails, no time–consuming backtrack operation is needed to restore the pointers of the variables in memory, for the VBS stack pointer can be reinitialized to its old value in less than three clock cycles.

3.2 UNIFICATION ALGORITHM

The algorithm running on the unification machine is presented in Figure 4. The function "arity(x)" returns the number of arguments in the compound term x. In this algorithm, the addresses of the two terms to be unified are read by the C.U. and the Arity register is set to 1 in the C.U. Afterwards, the unification algorithm can be broken up into two loops running in parallel. The first loop, a WHILE loop, takes place in the MPs. One cycle of this loop starts by the reading of the next two arguments of the two terms to be unified by one free MP which conducts the match operation on them. If the match fails, the MP sets the fail (FA) flag and the algorithm stops. If the match succeeds, the MP initiates a transfer request to transfer the variable binding to the CCP and VBS. Next, Arity is decremented in the C.U. and if not zero, a new loop cycle is started all over again but this time, the read and match operations may take place in another free MP.

The second loop takes place in the CCP. There, after the C.U. grants a transfer request to an MP, the transfer of the variable binding from the MP to the CCP and VBS is initiated. When the transfer of the binding is completed, the CCP conducts a consistency check on that new binding. If that check fails, the FA flag is set. Else, if that binding was the last binding generated, then the succeed (SU) flag is set.

The MATCH procedure accepts two terms x and y, checks their types, and generates the binding x/y if either one is a variable and provided the other one is not the same variable. If neither x nor y is a variable, then x should be exactly identical to y for the match to succeed.

The CONSISTENCY_CHECK procedure accepts the binding x/y where x is always a variable and y is of any type. If the variable x is not bound by previous substitutions then the binding x/y is recorded, else x's previous binding is checked for consistency with y. The procedure is recursively invoked when the consistency check of two compound terms is required.

```
BEGIN_UNIFY
 Read address of terms and set Arity to 1 in C.U.
 PARALLEL_BEGIN
 WHILE Arity not zero DO
  BEGIN_WHILE
   Read arguments(x, y) into a free MP
   IF both x and y are functors
     THEN Arity=Arity + arity(x)
   MATCH(x, y)
   IF match fails THEN set FA flag and STOP
   ELSE IF a binding x/y is generated
    THEN request transfer of binding to CCP/VBS
   Arity = Arity - 1
  END_WHILE
 LOOP FOREVER
  IF there is a granted transfer request THEN
    Transfer the variable binding x/y to CCP and
       to VBS from the requesting MP
  CONSISTENCY_CHECK(x, y)
  IF check fails THEN set FA flag and STOP
  ELSE IF last binding THEN set SU flag and STOP
 END_LOOP
 PARALLEL_END
END_UNIFY
```

Figure 4. Parallel Unification Algorithm

4. THE MP AND CCP PROCESSORS

Data words are 32 bits wide. Four data types are currently allowed: variable, function, list and constant. The MP processor is responsible for executing the match operation on the two terms to be unified. Because compound terms are broken up into functors and arguments which are matched separately, the match operation becomes the simple task of comparing two terms and generating a substitution if at least one of the two terms is a variable. To perform this, the MP reads the first term to be matched into a register and decodes it. If the term occupies more than one word, the rest of the term is read and stored in an internal buffer. The second term is then read into a second register, decoded, and if both terms are of identical types, the contents of the two registers are compared. When both terms occupy more than one word, the next word of term 1 is read from the internal buffer into the first register, while the next word of term 2 is read from the external memory into the second register and the contents of the two registers are again compared.

The MP has a two-bus organization. The first bus, DBUS, is interconnected with the external AB and VBB buses through input and output data registers. DBUS also connects these data registers to the internal RAM and two 32-bit registers DREG1 and DREG2 designed to hold the two term words to be compared. DREG1 and DREG2 are connected to an ALU(comparator/subtractor). The second bus, CBUS, links three 5-bit registers, FL, CS, and SR, to an adder used to increment the pointer to the 32-word deep RAM. Also a comparator is provided to compare the contents of the SR and FL registers.

The CCP processor is responsible for efficiently conducting the consistency check on the variable bindings generated by the MPs. This is done by maintaining the variable information in a CAM with parallel search and write capabilities. As soon as a variable binding is read by the CCP, the variable is searched in the CAM from which its binding status is extracted. Depending on the binding information of the variable, three situations may occur: 1) The variable does not match any CAM entry. A new entry for the variable is created in the CAM and the variable's binding is stored in the internal RAM; 2) The variable is temporarily bound to another variable. In this case, the binding is stored in the RAM and the binding status and pointer fields for the temporarily bound variables are updated in the CAM; 3) The variable matches a CAM entry. The variable's old binding is read from the internal RAM and compared with the variable's new binding. If not consistent, the CCP sets its FAIL line and unification ends with failure.

The CCP has a three–bus organization. The main bus, MBUS, which is 32 bits wide, connects the I/O data register to the CAM, binding RAM and DR1 and DR2 registers which hold the two bindings to be compared. The I/O data register serves as a buffer between the external VBB bus and MBUS. The DR1 and DR2 data registers are connected to a comparator capable of comparing words of different types. The first secondary bus, SBUS1, links the CAM's data register, DR, to seven registers. Five of these registers are also connected to the other secondary bus, SBUS2, which links them to a LIFO hardware stack. Both secondary buses are connected to an ALU capable of handling add and subtract operations. The microcontrol units of the MP and CCP processors function on a two-phase clocking scheme. During phase 1, the microinstruction is decoded while during phase 2, it is executed.

5. PERFORMANCE EVALUATION

The machine was simulated at the register transfer level with ISPS [19]. For performance comparison purposes, the microcycle time was taken to be 50 nanoseconds (decode, stack operations: 1 cycle; register transfers, CAM, RAM and ALU operations: 2 cycles; external memory operations: 4 cycles). Unifications of several terms of different types were simulated and the simulation results are given in Table I. PUM refers to the execution time in nanoseconds of the unification of these terms on the parallel machine, while UNIFIC refers to their execution time on the serial coprocessor UNIFIC, also operating with a microcycle time of 50 nanoseconds. Table II shows the unification execution times of functions with increasing arity on the parallel and serial machines. The unification of these terms generates the bindings X_i/a_i. Table III gives the execution time of the unification of two terms with increasing level of nesting. The bindings generated by the unification of these nested functions are X/b and Y/a. The speedup obtained in Tables II and III over UNIFIC falls in the range 1.490-2.791 and is plotted in Figure 5.

As expected, the speedup increases as the arity is increased exploiting the concurrency of the match and consistency check provided by the machine. For the above terms, the machine reached its top performance with only two MPs. To study the effects of the addition of a third and even a fourth MP to the system on the machine's performance, we have simulated the unification of the following terms and measured the execution times of the machine performing unification with one to four MPs and the percent gain in speed contributed by the addition of each MP.

1. $h(f(f(X,X,V,a),a,f(b,a,X,X),c), \quad Z, \quad V, \quad g(g(g(g(U))))) \; / \; h(Y, Y, a, X)$
2. $h(f(f(X,X,V,a),a,f(b,a,X,X),c), \quad Z, \quad V, \quad g(g(g(g(U))))) \; / \; h(Y, Y, a, a)$
3. $h(f(f(X,X,V,a),a,f(b,a,X,X),c), \quad Z, \quad b, \quad g(g(g(g(U))))) \; / \; h(Y, Y, a, a)$
4. $h(g(g(g(g(g(f(X,a,b,c)))))), \quad g(g(Z)), \quad b, \quad g(g(g(g(U))))) \; / \; h(Y, Y, a, Y)$
5. $h1(g(g(g(g(g(f(X,a,b,c)))))), \quad Z, \quad V, \quad g(g(g(g(a))))) \; / \; h1(Y, Y, a, Y)$
6. $h1(f(f(X,X,V,a),a,b,V), \quad f(Z,a,b,g(g(U))), \quad Z, \quad g(g(c)), \quad X) \; / \; h1(Y, Y, f(a,a,V,a), V, a)$
7. $h1(f(f(X,X,V,a),a,b,V), \quad f(Z,a,g(g(U))), \quad Z, \quad g(g(c)), \quad b) \; / \; h1(Y, Y, f(a,a,V,a), V, a)$

In Table IV, the execution times of the unification of the above terms on the machine with 1-4 MPs and the percent gains in speed recorded for the unification machine with 1-4 MPs with respect to the machine with one MP(pipeline) are given. The percent gains in speed are plotted in Figure 6.

The unification of the two terms in example 1 succeeds generating the substitutions $Y/Z/f(f(X,X,V,a),a,f(b,a,X,X),c)$; V/a; $X/g(g(g(g(U))))$. In this example, because the first

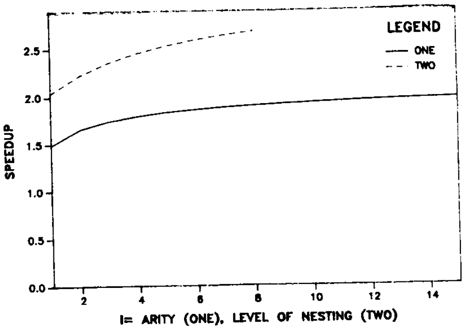

Figure 5. Speedup Vs. Arity/Level of Nesting

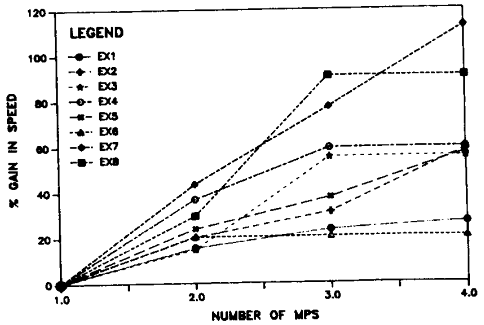

Figure 6. Plots of the Percent Gains in Speed

TABLE I. Execution Time of the Unification of Terms of Different Types

Term 1	Term 2	Binding	UNIFIC(ns)	PUM(ns)	SPEEDUP
X	A	X/A	1900	1700	1.118
X	Y	X/Y	1500	2000	0.750
X	f(a)	X/f(a)	2000	2300	0.870
a	f(X)	FAIL	850	700	1.214
h(X,Y)	h(a,b)	X/a,Y/b	5550	3350	1.657
h(X,X)	h(a,b)	FAIL	5400	3300	1.636
h(X,Y)	h(f(a),b)	X/f(a),Y/b	5650	3800	1.487
h(X,a)	h(b,Y)	X/b,Y/a	5600	3200	1.750
h(X,f(a))	h(f(a),X)	X/f(a)	5800	4100	1.415
h(X,f(a))	h(Y,f(Y))	X/a,Y/a	7650	3800	2.013
h(X,X,Y)	h(Y,a,e)	FAIL	7450	4150	1.795
h(X,Y,Y)	h(Y,e,a)	FAIL	8400	4150	2.024
h(f(Y,X),a)	h(f(a,b)X)	FAIL	9800	4800	2.042
h(f(X,X),a)	h(f(a,b),X)	FAIL	9700	4150	2.337

Table II. Execution Time of
$f(X_1,\ldots,X_i)$ / $f(a_1,\ldots,a_i)$

i	UNIFIC(ns)	PUM(ns)	SPEEDUP
1	3800	2550	1.490
5	10800	5900	1.831
10	19550	10150	1.926
15	28300	14400	1.965

i = ARITY

Table III. Execution Time of
$h(f_1(\ldots(f_i(X)\ldots),a)$ /$h(f_1(\ldots(f_i(b)\ldots),Y)$

i	UNIFIC(ns)	PUM(ns)	SPEEDUP
1	8250	4050	2.037
4	16200	6600	2.455
8	26800	10000	2.680
12	37400	13400	2.791

i = LEVEL OF NESTING

Table IV. Execution Times and
% Gain in Speed for Examples 1-7

EX.	# OF MPs	EXECUTION TIME (ns)	% GAIN IN SPEED
1	1	14450	0.00
1	2	12500	15.60
1	3	11700	23.50
1	4	11400	26.75
2	1	11550	0.00
2	2	9600	20.31
2	3	8800	31.25
2	4	7300	58.22
3	1	10400	0.00
3	2	9050	14.92
3	3	6700	55.22
3	4	6700	55.22
4	1	10350	0.00
4	2	7550	37.09
4	3	6500	59.23
4	4	6500	59.23
5	1	10050	0.00
5	2	8100	24.07
5	3	7300	37.67
5	4	6400	57.03
6	1	23550	0.00
6	2	19550	20.46
6	3	19550	20.46
6	4	19550	20.46
7	1	22850	0.00
7	2	15900	43.71
7	3	12900	77.13
7	4	10750	112.56

substitution is long and therefore requires a lengthy access of the VBB bus, and the second and third substitutions are rather short, the third MP (MP3) and the fourth MP (MP4) perform the match operation on the last arguments and contribute to the performance of the machine. The contribution of the second MP is larger than that of MP3 which is larger than that of MP4. In this example, the contribution of MP4, measured by the gain in speed introduced by the addition of MP4, is rather small.

Example 2 is a modified version of example 1 in which the last argument of the second term is altered to cause unification to fail. In this example, because of the long first substitution and the short second and third substitutions, all four MPs are utilized and contribute to the performance of the parallel unification machine. However, in this example, since the outcome of the operation is determined in the fourth MP, the addition of the fourth MP boosts the machine's performance to 58.2% faster than the pipeline(1 MP) although the fourth MP runs for only 6.2% of the total execution time. In this case, without the fourth MP, the machine's gain in speed with respect to the pipeline is degraded to 31.25%. Therefore, in example 2, the addition of the fourth MP improves the performance of the machine considerably.

In example 3, the third argument of the first term in example 1 is altered to cause unification to fail earlier than in example 2. Here, the functor match and the first two argument matches are executed in MP1 and MP2, and the third argument, which causes unification to fail in this example, is matched in MP3. This is why MP4 does not affect the execution time in this example. The third MP increases the speedup over the pipeline by 55.22%, a noticeable jump over the 14.92% speedup produced by the machine with two MPs. In example 4, which is similar to example 3, the third argument match fails, causing MP4 to stay idle and to yield no performance improvements. In examples 5 and 7, unification fails and the addition of MP4 raises the speedup over the pipeline by 57.03% and 112.56%, respectively, with large gains in speed over the machine with only three MPs. In example 6, unification succeeds generating the substitutions $Y/f(f(X,X,V,a),a,b,V)$; $Z/f(X,X,V,a)$; $V/g(g(U))$; X/a and U/c. In the examples where unification succeeds, such as examples 1 and 6, the CCP run-time stays constant as the number of MPs used in the system is increased and is therefore independent of the number of MPs used. This results in an increase in the percent utilization time of CCP as the number of MPs used is increased. From the simulation results obtained, few cases (which are rarely encountered in logic programs) arise in which MP4 contributes and produces significant gains in speed, and therefore the cost of the addition of the fourth MP is not justified. Thus the machine with only three MPs provides a performance very near to the top possible performance in most cases.

To compare the machine's performance with that of the Hardware Unification Unit (HUU) [6], the unification operations occurring in the following program are simulated and their execution times on the parallel machine are evaluated.

```
P(X,Y) ← P(X,Z) , P(Z,Y)
P(a,b)
P(b,c1)
P(b,c2)
P(b,c3)
P(b,c4)
P(b,c5)
```

with query: P(a,U)?. On HUU, the average execution time of the term unifications taking place during the execution of the above program was measured by Woo to be 10.5 microseconds, assuming a microcycle time of 100 nanoseconds. On the parallel unification machine, this average execution time is 3.3 microseconds with only one MP allowed in the system, and 3.0 microseconds with two or more MPs allowed. This amounts to a speedup of 3.18 and 3.53 for the machine with one and two MPs respectively. In general, the average term unification time was measured by Woo to take between 10 and 20 microseconds on HUU and 100 microseconds for the UNSW PROLOG interpreter on a VAX 11/780. Considering the unification examples in Table I, the average unification time of these examples on the parallel unification machine (PUM2) is 3250 nanoseconds and therefore, on the average, the parallel machine executes the terms in Table I 30.77 times faster than an average unification on the UNSW interpreter and 3.08-6.15 time faster than an average unification on HUU. This significant gain in speed can be attributed to the novel architecture of the parallel unification machine which partitions unification into concurrent match and consistency check operations, the efficient three-bus CCP organization and, last but not least, the variable CAM which allows fast retrieval and update of variable information.

6. CONCLUSIONS

A parallel unification coprocessor partitioning unification into a match step and a consistency check step and conducting these two steps concurrently in a pipeline fashion was presented. The machine's architecture, algorithm, and processors were described. The machine's performance was simulated and compared with two serial unification coprocessors.

The parallel unification coprocessor with only two MPs was shown to be significantly faster than the two serial coprocessors for compound terms of arities higher than 2. The study of the unification of long terms indicated that the coprocessor reaches its peak performance with 3 MPs in most cases. This seems to confirm that the cost of the fourth MP is not justified and therefore it seems that only 3 MPs will be needed in the future. The significant speedup of the parallel coprocessor over the serial ones can be attributed to the following parallel machine features:

i. An architecture mapping the structure of unification and partitioning unification into match and consistency check operations performed concurrently in a pipeline fashion.
ii. An efficient allocation policy allowing the functors to be matched first, followed by the respective arguments of the two terms, and treating nested functions as level-0 arguments.
iii. A simultaneous transfer of the variable bindings to the CCP and to the VBS in which they

are stored thus eliminating the overhead time for saving the variable bindings at the end of unification.

iv. A fast backtrack feature.

v. A fast All-Processor-Stop policy conducted by the C.U. which stops the operation of the processors in the system as soon as failure in any processor is detected. This policy has proven its effectiveness in Inagawa's [18] multiprocessor system.

vi. A CAM for maintaining the variable information of previously bound or temporarily bound variables, with parallel search and write capabilities. The CAM provides fast retrieval and update of the contents of its words.

vii. A three-bus organization of the CCP processor allowing as many as eight operations to take place concurrently.

Future work includes the design of the microcontrol units of the MP and CCP processors, and the logic design of the C.U. and of the VBS.

REFERENCES

[1] Miller, R. K., *Fifth Generation Computers.* Lilburn, Georgia: The Fairmont Press, 1987.

[2] Abe, S. *et al*, "High Performance Integrated Prolog Processor IPP," in *The 14th Annual Int. Symp. on Computer Architecture*, pp. 100-107, Pittsburg, PA, June 1987.

[3] Conery, J., *Parallel Execution of Logic programs.* Norwell, MA: Kluwer Academic Publishers, 1987.

[4] Ponder, C., Patt, Y., "Alternative Proposals for Implementing Prolog Concurrently and Implications Regarding Their Respective Microarchitectures," in *The 17th Annual Workshop on Microprogramming*, pp. 192–203, New Orleans, LA, October 1984.

[5] Woo, N. S., "A Hardware Unification Unit: Design and Analysis," in *The 12th Annual Int. Symp. on Computer Architecture*, pp. 198-205, Boston, MA, June 1985.

[6] Woo, N. S., "The Architecture of the Hardware Unification Unit and an Implementation," in *The 18th Annual Workshop on Microprogramming*, pp. 89-98, Pacific Grove, CA, 1985.

[7] Robinson, J. A., "A Machine–Oriented Logic Based on the Resolution Principle," *Journal of the ACM*, vol. 12(1), pp. 23-41, January 1965.

[8] Paterson, M. S., Wegman, M. N., "Linear Unification," *Journal of Computer and System Sciences*, vol. 16(2), pp. 158-167, April 1978.

[9] Martelli, A., Montanari, U., "An Efficient Unification Algorithm," *ACM Transactions on Programming Languages and Systems*, vol. 4(2), pp. 258-282, April 1982.

[10] Yasuura, H., "On Parallel Computational Complexity of Unification," in *Proc. of the Int. Conf. on FGCS*, pp. 235-248, Tokyo, Japan, Nov. 1984.

[11] Dwork, C. *et al*, "On the Sequential Nature of Unification," *J. Logic Programming*, vol. 1(1), pp. 35-50, June 1984.

[12] Chang, S., *Towards a Theorem Proving Architecture*, Dept. of Computer Science, California Inst. of Technology, Pasadena, CA, 1981.

[13] Robinson, P., "The SUM: an AI Coprocessor," *Byte*, vol. 10(6), pp. 169-180, June 1985.

[14] Gollakota, R., *Design and Analysis of UNIFIC: a Coprocessor for the Unification Algorithm*, M. S. Thesis, Dept. of Electrical Engineering, Texas A&M University, College Station, TX, August 1986.

[15] Shobatake, Y., Aiso, H., "The Unification Processor Based on a Uniformly Structure Cellular Hardware," in *The 13th Annual Int. Symp. on Computer Architecture*, pp. 140-148, Tokyo, Japan, June 1986.

[16] Chen, W., Hsieh, K., "An Overlapping Unification Algorithm and Its Hardware Implementation," in *Proc. of the 1987 Int. Conf. on Parallel Processing*, pp. 803-805, University Park, PA, 1987.

[17] Shih, Y., Irani, K., "Large Scale Unification Using a Mesh–Connected Array of Hardware Unifiers," in *Proc. of the 1987 Int. Conf. on Parallel Processing*, pp. 787-794, University Park, PA, 1987.

[18] Inagawa, M. *et al*, "Unification Parallelism for Prolog Processing," *Systems and Computers In Japan*, vol. 19(1), pp. 37-46, January 1988.

[19] Barbacci, M. *et al*, "ISPS Computer Description Language," Dept. of Computer Science and Electrical Engineering, Carnegie-Mellon University, Pittsburg, PA, 1981.