DESIGN AND PERFORMANCE MEASUREMENTS OF
A PARALLEL MACHINE FOR THE UNIFICATION ALGORITHM

F. N. Sibai            K. L. Watson            Mi Lu
Department of Electrical Engineering
Texas A&M University
College Station, Texas 77843 U.S.A.

ABSTRACT

Unification is known to be the most repeated operation in logic programming and PROLOG interpreters. To speed up the execution of logic programs, the performance of unification must be improved. We propose a parallel unification machine for speeding up the unification algorithm. The machine is simulated at the register transfer level and the simulation results as well as performance comparison with a serial unification coprocessor are presented.

1. INTRODUCTION

In today's knowledge-based world, logic programming and functional programming stand at the top of the programming choices available to Artificial Intelligence (AI) programmers and knowledge base developers. Therefore, it is not surprising to find PROLOG, a logic programming language which has been very popular in Europe and Japan, gaining in popularity world-wide amongst academia and industry.

PROLOG's statements in the form of logic propositions, its argument match capability, and nondeterministic and database management features make it very suitable for AI and expert system applications. These favorable features are probably behind Japan's new generation computer technology (ICOT) department's decision to adopt PROLOG as the official kernel language of the fifth generation computer system (FGCS) project which started in 1981. The primary goal of the FGCS project is to replace the traditional Von Neumann computers by smarter ones capable of reasoning, learning, associating, making inferences and decisions, and understanding speech, written text and pictures [1]. The choice of PROLOG by ICOT shows the important and leading post that PROLOG has achieved in the fifth generation, and marks the beginning of an era in which PROLOG is accepted world-wide.

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 basically explains why today's expert systems and other AI application programs implemented in PROLOG run very slowly on traditional general purpose computers. The sources of PROLOG's inefficiency and slow execution on these machines can be related to two algorithms on which PROLOG interpreters and other logic programming interpreters are built: unification and backtracking. It is only logical to improve the execution of these two algorithms in order to speed up the runtime of PROLOG programs and other logic programs. This paper focuses on unification.

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 (i.e., the two terms become equivalent) if the first argument of the first term, $X$, is bound to the first argument of the second term, $a$. In general, to successfully unify two functions, the heads (or functors) of the two terms, $f$, must be identical and the $i$th argument of term 1 must unify (match) with the $i$th 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.

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 \( b \), clearly a conflict with the first binding. Thus, in this example, unification fails.

The unification operation was shown by Woo [3-4] to consume on the average 55-70% of the execution time of PROLOG programs. Thus the unification operation is one of the most repeated operations in logic programming interpreters. For this reason, researchers have attempted to improve unification's execution and for that purpose have taken the following three directions:

1. Development of parallel unification algorithms.
2. Development of serial unification coprocessors supporting a host processor.
3. Development of parallel unification hardware to be incorporated in logic programming machines such as PROLOG machines.

We take the more promising third direction and propose here a parallel machine to speed up the unification algorithm. In the next section, we outline the available unification algorithms, unification coprocessors and parallel unification machines which have been proposed or developed up to the present. Following that section, we first describe the architecture of the proposed system and the unification algorithm designed to run on it. Next, the data formats and processor organizations are discussed. Finally, the simulation results are presented.

2. UNIFICATION ALGORITHMS AND HARDWARE

2.1 UNIFICATION ALGORITHMS

Unification was originally developed by Robinson [5] as the heart of the Resolution principle in the mid 1960s. 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 [6] with linear time complexity, and Martelli and Montanari's [7]. 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, parallel unification algorithms have been the target of intensive research work until Yasuura [8] 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 [9] 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 checking step. However, for the subproblem of term matching, parallelism should provide some room for performance improvement.

2.2 SERIAL UNIFICATION COPROCESSORS

In 1981, Chang [10] designed a machine to execute Robinson's unification algorithm at Caltech. This one-chip machine is composed of a controller, stacks, registers and an equation table — an external KAM 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.

At Syracuse University, the SUM unification coprocessor [11] equipped with a content-addressable memory (CAM) was designed to support an LMI Lambda LISP machine. The host Lambda machine whenever encountering a unification task assigns it to the SUM coprocessor. The SUM coprocessor conducts unification assisted by a CAM for fast access of binding agents.

Another unification coprocessor, consisting of a hardware processing unit and a variable stack, was developed by Woo [3-4] at AT&T, and was shown to improve the performance of unification considerably. 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 [12] designed a 54-pin unification coprocessor equipped with an on-chip binding memory, which is half CAM and half RAM, and capable of addressing up to 16K words in memory. 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. Essentially, this is a one-bus machine with a microprogrammed control, an on-chip binding memory, a stack in which register contents are saved during recursive invocations, two memory address registers used to hold the addresses of the two lists in memory, and an arity register initially holding the number of terms in the lists.

Most of these coprocessors were shown to speed up unification's execution.

2.3 PARALLEL UNIFICATION HARDWARE

To speed up unification even further other approaches must be explored. Although the studies by Yasuura and Dwork do not 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 [13] used 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 $O(n^2)$. 

Chen [14] proposed an overlapping algorithm originating from Robinson's algorithm with a one-dimensional systolic-like architecture. No measurements were made.

Shih [15] proposed a $k \times n$ mesh-connected unification units to speed up AND parallelism in clauses. He 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 [16] implemented a multiprocessor system for unification and PROLOG processing and showed that the unification parallelization effect was evident for a small number of processing units.

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 proposed machine performs the unification operation on two terms, and outputs failure or the variable bindings in case of success with the following guidelines: 1. the match and consistency steps must be performed concurrently; 2. there must be a fast backtrack operation in case of failure. 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 the design of 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 proposed system architecture is shown in Figure 1. 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 a 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 a 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 to select which MP is to transfer its binding to the CCP 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 left). Here, two lines are required for each MP: a REQUEST line(RA) and a GRANT line(RG) as shown in Figure 2. 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. In this scheme, the C.U. can execute any programmed selection algorithm such as the First-Available From Left algorithm. This is why this scheme is the most flexible. Also, a failed unit cannot cause system failure, which makes this algorithm more fault-tolerant than others.

After being granted the transfer, the MP with the set RG then starts the simultaneous transfer of the variable binding to the CCP and the variable binding store(VBS). The MP initiates the transfer by setting the ITF line shown in Figure 3. 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

We present below the algorithm running on the unification machine. The word “Arity” refers to the number of arguments in each compound term.

BEGIN-UNIFY
Read address of terms and set Arity to 1 in C.U.
PARALLEL-BEGIN
WHILE Arity not zero DO
  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 FAIL flag, reset and EXIT
    ELSE request transfer and wait until granted
    IF a binding x/y is generated
      THEN request transfer of binding to CCP and VBS
      Arity = Arity - 1
  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 FAIL flag and EXIT
  ELSE IF last binding THEN set SU flag and EXIT
END-LOOP
PARALLEL-END
END-UNIFY

In the above 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 match processors(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 its FAIL flag informing the C.U. of the result of the unification operation. The C.U. in its turn raises its FAIL flag to inform the host CPU of the result of the operation. If the match succeeds, the MP initiates a transfer request to transfer the variable binding to the CCP and VBS. Next, the 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. When the C.U. grants a transfer request to a match processor, the transfer of the variable binding from the MP to the CCP and VBS is initiated. When the transfer is completed, the CCP conducts a consistency check on that new binding. If that check fails, the CCP informs the C.U. of this and the C.U. raises its FAIL line. Else, if that binding was the last binding generated, then the CCP informs the C.U. of the success of the whole unification operation.

4. DATA FORMATS

Data words are 32 bits wide. Four data types are currently allowed: variable, function, list and constant. A two-bit data type tag(DT) distinguishes the four types. 00 is assigned to constants, 11 to variables, and 10 to compound terms(functions or lists). Figure 4 shows the data formats of these 4 types. For constants, DT=00 and occupies bits 0-1 as seen in Figure 4-a. Bit 2 is reserved for cdr-coding. Bits 3-4 represent the constant type as follows.

CT CONSTANT TYPE
00 INTEGER  b_5 - b_31 = 2's complement number
01 FLT POINT  b_5 - b_23 = Mantissa
             b_24 - b_31 = Exponent
11 NIL
10 SYMBOL

For a symbol, bits 5-8 indicate the number of characters in the symbol as follows.

<table>
<thead>
<tr>
<th>b_9 - b_{15}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1</td>
</tr>
<tr>
<td>0 0 1 0</td>
</tr>
<tr>
<td>0 1 0 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>

Rest of symbol is in the following word

Figure 4-b shows the format of a variable. Here, DT=11 and a 3-bit B field contains the binding status of the variable. If bits 3-4 are 00 then the variable is not bound(free). If bits 3-4 are 01 then the variable is temporarily bound to another variable and if 10, the variable is bound. Bit 5 indicates the type of the variable's binding when the variable is bound(if 0 the binding is a constant, else it is a compound term). A 20-bit pointer field occupies bits 12-31 of the variable word. This pointer addresses a memory space of 1 Megawords and points to the address of the variable.
symbol in memory and serves as the variable's identifier. For a compound term, shown in Figure 4-c, DT=10, and the number of arguments in the term is held by the 4-bit ARITY field. This restricts the number of arguments in the term to 15. The total number of words occupied by the compound term is located in the FL field.

Figure 4-d shows the format of a CAM word. The CAM is located in the CCP processor and holds information on the variables previously bound. Bits 0-1 of a CAM word hold the binding status of the variable. If 0, the variable is temporarily bound, if 1, the variable is bound to a constant and when 11, the variable is bound to a compound term. Bits 2-21 hold the variable identifier which is represented by the 20-bit address of the variable symbol in memory. A pointer to the variable's binding in the binding RAM, also internal to the CCP, is contained in bits 22-31.

5. MP AND CCP ORGANIZATIONS

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, such as symbols of more than three characters, 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 as shown in Figure 5. 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 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 FL, a register which holds the address of the next word to be accessed in the internal RAM to an adder used to increment the pointer to the RAM.

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. Here, the binding is stored in the RAM and the binding status and pointer fields for these 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, to achieve this, incorporates two features for efficient and high performance: (i) a CAM for parallel search of variables and fast retrieval of their binding information, and (ii) a three-bus organization providing efficient internal processor parallelism and allowing as many as eight operations to be performed concurrently.

The organization of the CCP processor is given in Figure 6. The main bus, MBUS, connects the I/O data register (DIN) 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. DR1 and DR2 are connected to a comparator capable of comparing words of different types. The first secondary bus, SBUS1, links the CAM's data register to seven registers: RAM and CAM pointer registers, sum register and arity register. These registers are also connected to the other secondary bus, SBUS2, which links them to a hardware stack. Both secondary buses are connected to an ALU. The microcontrol units of the MP and CCP processors operate on two non-overlapped phase clocks. During phase 1, the microinstruction is decoded while during phase 2, it is executed. The microcontrol units are in charge of interprocessor communications and generating internal control signals.

6. RESULTS

The machine was simulated at the register transfer level with ISPS [17]. For performance comparison purposes, the microcycle time was taken to be 50 nanoseconds (decode, stack operations: 1 cycle; register transfer, 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 processor UNIFIC, also operating with a microcycle time of 50 nanoseconds. Table II shows the execution time of the unification 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 increased level of nesting. The bindings generated for these nested functions are $X/b$
and $Y/a$. The speedup obtained in Tables I and II ranges between 1.490 and 1.965 for the functions with increasing arity, and between 2.037 and 2.791 for the functions with increasing level of nesting.

As expected, the speedup increases as the arity is increased exploiting the match parallelism and fast consistency check provided by the machine. For these terms, the machine reached its top performance with only two MPs and from the simulation results obtained so far, the addition of a 4th MP rarely contributes to the machine's performance. We believe that 3 MPs will be eventually chosen. We are in the process of simulating more examples to study other machine performance features.

7. SUMMARY

A parallel unification machine partitioning unification into a match step and a consistency check step, and conducting these two steps concurrently was presented. The machine's architecture, algorithm, data formats and processor organizations were described. The machine's performance was simulated and compared with a serial unification coprocessor.

REFERENCES

Figure 1. System Architecture

Figure 2. Argument Scheduling

Figure 3. MP Selection and Binding Transfer
Figure 4. Data Formats

d. CAM word

Figure 5. MP Organization
### TABLE I. Execution Time of The Unification of Terms of Different Types

<table>
<thead>
<tr>
<th>Term 1</th>
<th>Term 2</th>
<th>Binding</th>
<th>UNIFIC(ns)</th>
<th>PUM(ns)</th>
<th>SPEEDUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>A</td>
<td>X/A</td>
<td>1900</td>
<td>1700</td>
<td>1.118</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>X/Y</td>
<td>1500</td>
<td>2000</td>
<td>0.750</td>
</tr>
<tr>
<td>X</td>
<td>f(a)</td>
<td>X/f(a)</td>
<td>2000</td>
<td>2300</td>
<td>0.870</td>
</tr>
<tr>
<td>a</td>
<td>f(X)</td>
<td>FAIL</td>
<td>850</td>
<td>700</td>
<td>1.214</td>
</tr>
<tr>
<td>h(X,Y)</td>
<td>h(a,b)</td>
<td>X/a,Y/b</td>
<td>5550</td>
<td>3350</td>
<td>1.657</td>
</tr>
<tr>
<td>h(X,Y)</td>
<td>h(a,b)</td>
<td>FAIL</td>
<td>5400</td>
<td>3300</td>
<td>1.636</td>
</tr>
<tr>
<td>h(X,Y)</td>
<td>h(f(a),b)</td>
<td>X/f(a),Y/b</td>
<td>5650</td>
<td>3800</td>
<td>1.487</td>
</tr>
<tr>
<td>h(X,a)</td>
<td>h(b,Y)</td>
<td>X/b,Y/a</td>
<td>5600</td>
<td>3200</td>
<td>1.750</td>
</tr>
<tr>
<td>h(X,f(a))</td>
<td>h(f(a),X)</td>
<td>X/f(a)</td>
<td>5800</td>
<td>4100</td>
<td>1.415</td>
</tr>
<tr>
<td>h(X,f(a))</td>
<td>h(Y, f(Y))</td>
<td>X/a,Y/a</td>
<td>7650</td>
<td>3800</td>
<td>2.013</td>
</tr>
<tr>
<td>h(X, Y)</td>
<td>h(Y, a,e)</td>
<td>FAIL</td>
<td>7450</td>
<td>4150</td>
<td>1.795</td>
</tr>
<tr>
<td>h(X, Y)</td>
<td>h(Y, e,a)</td>
<td>FAIL</td>
<td>8400</td>
<td>4150</td>
<td>2.024</td>
</tr>
<tr>
<td>h(f(Y, X), a)</td>
<td>h(f(a, Y), X)</td>
<td>FAIL</td>
<td>9800</td>
<td>4800</td>
<td>2.042</td>
</tr>
<tr>
<td>h(f(f(X), a))</td>
<td>h(f(a, b), X)</td>
<td>FAIL</td>
<td>9700</td>
<td>4150</td>
<td>2.357</td>
</tr>
</tbody>
</table>

Figure 6. CCP Organization
Table II. Execution Time of The Unification of \( f(X_1, \ldots, X_i) / f(a_1, \ldots, a_i) \)

<table>
<thead>
<tr>
<th>( i = \text{ARITY} )</th>
<th>( \text{UNIFIC(ns)} )</th>
<th>( \text{PUM(ns)} )</th>
<th>( \text{SPEEDUP} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3800</td>
<td>2550</td>
<td>1.490</td>
</tr>
<tr>
<td>2</td>
<td>5550</td>
<td>3350</td>
<td>1.657</td>
</tr>
<tr>
<td>3</td>
<td>7300</td>
<td>4200</td>
<td>1.738</td>
</tr>
<tr>
<td>4</td>
<td>9050</td>
<td>5050</td>
<td>1.792</td>
</tr>
<tr>
<td>5</td>
<td>10800</td>
<td>5900</td>
<td>1.831</td>
</tr>
<tr>
<td>6</td>
<td>12550</td>
<td>6750</td>
<td>1.859</td>
</tr>
<tr>
<td>7</td>
<td>14330</td>
<td>7600</td>
<td>1.882</td>
</tr>
<tr>
<td>8</td>
<td>16050</td>
<td>8450</td>
<td>1.899</td>
</tr>
<tr>
<td>9</td>
<td>17800</td>
<td>9300</td>
<td>1.914</td>
</tr>
<tr>
<td>10</td>
<td>19550</td>
<td>10150</td>
<td>1.926</td>
</tr>
<tr>
<td>11</td>
<td>21300</td>
<td>11000</td>
<td>1.936</td>
</tr>
<tr>
<td>12</td>
<td>23050</td>
<td>11850</td>
<td>1.945</td>
</tr>
<tr>
<td>13</td>
<td>24800</td>
<td>12700</td>
<td>1.953</td>
</tr>
<tr>
<td>14</td>
<td>26550</td>
<td>13550</td>
<td>1.959</td>
</tr>
<tr>
<td>15</td>
<td>28300</td>
<td>14400</td>
<td>1.965</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>( i = \text{LEVEL OF} )</th>
<th>( \text{UNIFIC(ns)} )</th>
<th>( \text{PUM(ns)} )</th>
<th>( \text{SPEEDUP} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8250</td>
<td>4050</td>
<td>2.037</td>
</tr>
<tr>
<td>2</td>
<td>10900</td>
<td>4900</td>
<td>2.224</td>
</tr>
<tr>
<td>3</td>
<td>13550</td>
<td>5750</td>
<td>2.357</td>
</tr>
<tr>
<td>4</td>
<td>16200</td>
<td>6600</td>
<td>2.455</td>
</tr>
<tr>
<td>5</td>
<td>18850</td>
<td>7450</td>
<td>2.530</td>
</tr>
<tr>
<td>6</td>
<td>21500</td>
<td>8300</td>
<td>2.590</td>
</tr>
<tr>
<td>7</td>
<td>24150</td>
<td>9150</td>
<td>2.639</td>
</tr>
<tr>
<td>8</td>
<td>26800</td>
<td>10000</td>
<td>2.680</td>
</tr>
<tr>
<td>12</td>
<td>37400</td>
<td>13400</td>
<td>2.791</td>
</tr>
</tbody>
</table>