JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 15, 859-884 (1999)

Reducing Memory Traffic and Accelerating Prolog Execution in a Superscalar Prolog System

RUEY-LIANG MA AND CHUNG-PING CHUNG*
Computer & Communications Research Laboratories, ITRI
Hsinchu County, Taiwan 310, R.O.C.
E-mail: rlma@ozu.ccl.itri.org.tw
**Institute of Computer Science and Information Engineering*
National Chiao Tung University
Hsinchu, Taiwan 300, R.O.C.

Memory access operations constitute about 32.7% of all the operations executed in a typical Prolog program. Among these memory accesses, 75% are to the program control structures (environments and choice points). These memory accesses plus possible data cache misses greatly impair system performance, and the problem is even more severe in a VLIW, superscalar, or superpipelined Prolog system. This paper describes an innovative windowed register file management technique called SORWT (splittable overlapping register window technique). With SORWT, environments, choice points, and arguments can be stored in a windowed register file. SORWT reduces the number of memory accesses to only 25% of the number made when a conventional stack scheme is used. This paper describes in detail how Warren and PLM instructions can be implemented using SORWT, and it presents a register file overflow/underflow handling mechanism called the memory window matrix (MWM) and a mapping function for use between register windows and the MWM. Thirty benchmark programs are used to study performance issues, the overhead of SORWT, optimal register file and window sizes, and the argument overflow rate.

Keywords: fine-grained parallelism, memory traffic reduction, modified windowed register file, prolog system design, simulation, window overflow handling

1. INTRODUCTION

In Prolog program execution, a large percentage of time is spent on data memory accesses. In the research of [4], memory access instructions constitute about 32.7% of all the instructions executed in a RISC-type Prolog system. These memory accesses plus possible data cache misses greatly impair system performance. If the data cache miss ratio in a Prolog machine is 5% [20] and the overhead of a cache miss is 20 cycles (in contrast to the one-cycle operations in a RISC system), the performance degradation is about 32.7% in a sequential RISC Prolog system.

This problem is even more serious for a VLIW [7] or superscalar [10] Prolog system. In [4], Amdahl's law was applied to evaluate fine-grained speedup. Suppose that all memory accesses are sequentialized (the memory is single-ported) for a VLSI shared memory system implementation. In this architecture, fine-grained parallelism is limited by the number of memory operations, and the maximum speedup is 1/0.327 =3.06. Hence in such a VLIW or superscalar Prolog system, the speedup is bounded by 3.06.

Received May 8, 1998; accepted August 20, 1998.
Communicated by Jieh Hsiang.

The large number of memory accesses is due to the abstract Prolog execution model of Warren [21]. The Warren model has had a great influence on the following Prolog system designs: PLM [6], PLUM [16], BAM [9], and Low RISC [13]. This model employs several distinct memory stack areas to store control structures (the control stack), complex data (the heap), and other data structures for Prolog execution (the trail and the push down list).

In the Warren model, for forward execution, a dynamic data structure called an *environment* is implemented for procedure calls to store parameters and return addresses. The environment is similar to an activation record for conventional languages. This data structure is of particular importance in recursive operations.

In each procedure call, only one clause is tried at a time. The control returns on a successful try, or it backtracks to execute the next clause. Thus the entry point to a procedure is not fixed, and a procedure may begin execution on different clauses. To implement its backtracking feature, Prolog uses a fixed-length dynamic data structure called a *choice point* to record the status of the current procedure. The choice point can be used to restore the system states and provide an entry point to the next executable clause.

Resolution of a subgoal in Prolog is carried out by means of forward executions and backtrackings. A great deal of bookkeeping is needed, and the control structures (environments and choice points) must be stored in the main memory. Previous studies on WAM in [19] and on PLUM demonstrated that about 70% to 75% of memory data accesses are made to the control stack. The large number of memory loads/stores introduces a large amount of overhead and sequentiality, making Prolog program execution very inefficient. How to implement forward executions and backtrackings efficiently and correctly is thus an important issue in the design of a Prolog machine.

Using a windowed, overlapping register file [11] facilitates the implementation of procedure calls/returns for conventional languages such as Fortran and C [3]. A windowed register file divides registers into groups of nonoverlapping or overlapping sections called windows. With overlapping register windows, parameters can be passed in the register file directly on subroutine calls and returns, thereby reducing the number of memory accesses and speeding program execution. A pointer is necessary to implement control transfers so that the register windows are activated as in a stack. Window overflow/underflow will occur because of the nature of the stack, and the stack must be handled properly to prevent possible data loss.

The RISC II register file [11] stores only local data in register windows. This is because conventional programming languages use only a dynamic data structure called an activation record to implement procedure calls. This data structure is equivalent to an environment in Prolog. Direct implantation of RISC II register windows in a Prolog machine merely converts accesses to the environments from memory operations into register accesses. Such an implementation eliminates 25% of all memory access operations for a Prolog system design. Yet it is likely that further improvement is possible

This paper introduces an innovative windowed register file management technique called SORWT (splittable overlapping register window technique). SORWT is designed to match the execution characteristics of Prolog programs. It includes Prolog program parameters (arguments), choice points, and environments in a windowed register file. With SORWT, 75% of all memory access operations (in effect, all memory accesses to the environments and choice points) are eliminated, and the memory access operation ratio of typi-

cal Prolog programs can be reduced from 32.7% to 8.18%. This represents a potential speedup of 1/0.0818=12.2 over the ordinary single-ported memory Prolog system. In this design, speedup is no longer bounded by 3.

SORWT can be applied to register files of different sizes and configurations; only the format of the register window pointers needs to be redefined. The management methods of SORWT can be implemented easily in a Prolog compiler and a modified windowed register file architecture. And the gain in performance that results from SORWT is significant, particularly in a parallel system. Several Prolog systems [4,18] have implemented BAM [9] into a VLIW or superscalar design. The large number of memory accesses in these designs limits fine-grained parallelism, and the incurred cache misses may degrade the system performance severely. By reducing the memory operation ratio in Prolog programs, SORWT can help exploit more fine-grained parallelism to enhance the performance of a system that employs instruction-level parallelism.

We have compiled and simulated thirty benchmark programs for use in evaluating SORWT. This benchmark set covers the programs used in PLM, BAM, and SPUR [2] to evaluate their performance, plus some other frequently-used programs collected from applications overlooked in the PLM and SPUR benchmark sets. The results of the simulator, the program execution traces, were used as the input to the SORWT simulator to acquire the data and performance measurements presented in this paper.

This paper is organized as follows. Section 2 describes the organization, the principles, the design considerations, and the implementation of SORWT in detail. In section 3, we present a memory window matrix (MWM) used to save and restore register window information upon register window overflow/underflow. The backtracking characteristic of Prolog causes the order of procedure (and hence window) activation to be disruptive, and use of SORWT and the MWM can remedy the inefficiency of the traditional stack method in handling register window overflow/underflow. Section 3 also describes the mapping function between the register file and memory window matrix, and between the overflow/underflow detection method and handling procedure. In section 4, thirty Prolog benchmarks are used to evaluate the register file performance and the many design issues discussed in this paper. The implementations of argument overflow and register translation in the compiler are also discussed. Section 5 compares the execution cycle counts of control instructions in several related machines and discusses the effects of our design on instruction-level parallelism. Section 6 presents conclusions.

2. DESIGN OF SORWT AND THE WINDOWED REGISTER FILE

2.1 The Design Principles and Management of SORWT

This section describes the principles, implementation, and operations of SORWT. It also includes a description of how the register file is organized, based on certain features of Prolog. Furthermore, the implementation of SORWT in Warren and PLM instructions is discussed in detail.

The extensive use of call/return and backtracking in Prolog makes a windowed register file particularly suitable for a Prolog machine design. In addition, there are other features of Prolog that greatly favor this approach. These include:

(1) According to a static analysis of fifteen benchmarks in [12], the average number of input argument arities of a procedure is 2.8. Of all the procedures studied, 73.4% had an arity count of less than 4, and 94.5% a count of less than 7. Prolog clauses were found to have an average of 2.64 permanent variables. [14] studied 39 benchmarks and found that the average number of procedure input arguments was 3.2. The small numbers indicate that it may be feasible to store these arguments using register windows.

(2) Permanent variables [21] that are stored in the environment are variables needed to fulfill more than one goal in the body of a clause. These variables must be saved so that succeeding goals can access them. Note that, since there are multiple, overlapped register windows in a windowed register file design, variables are saved in input registers and transferred to those subgoals that need them without running the risk of being overwritten. Therefore saving permanent variables in an environment is no longer necessary. This approach not only reduces the number of memory accesses, but also simplifies implementation of the environment stack. Since the permanent variables are now no longer needed, the length of the environment becomes fixed, as is that for the choice point. This is one very desirable result of using register windows. Without the permanent variables and the N value (the number of permanent variables), the environment now consists of only three registers (CP, B, and E), which we will discuss later in this section. A fixed-sized environment can be stored in a register window conveniently. Further more, this implementation can keep both the arguments and permanent variables in the input register area, thereby reducing the number of memory accesses to permanent variables compared to the number required in WAM and PLM.

Fig. 1(a) shows the organization of a conventional windowed overlapping register file. The key design issue is that the output registers of window W_i overlap with the input registers of window W_{i+1}, where i and i+1 represent the procedure level count (the call depth). Parameter passing between procedures is, hence, automatic and does not incur memory data transfer overhead. Note that in this scheme, only one window pointer, the CWP(Current Window Pointer), is used.

Fig. 1(b) incorporates the conventional windowed register file into a Prolog machine. The register window can be used to store the environment and arguments. The input and output areas of the register window store the arguments and can be used for unification and parameter passing. The special registers of the environment (shown in Fig. 2) are those in the local area of the register window.

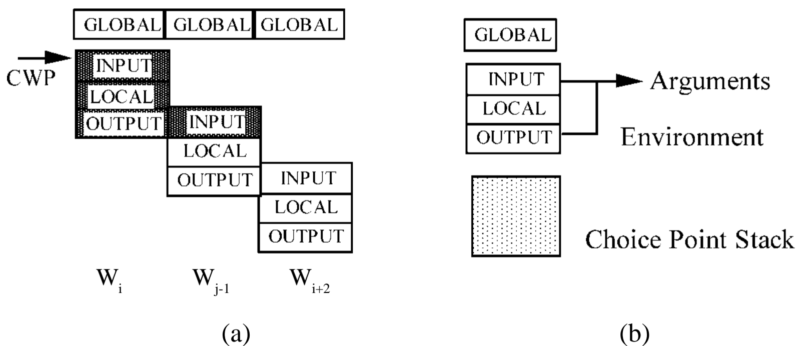

Fig. 1. (a) A windowed overlapping register file; (b) conventional windowed register file implementation in a prolog machine.

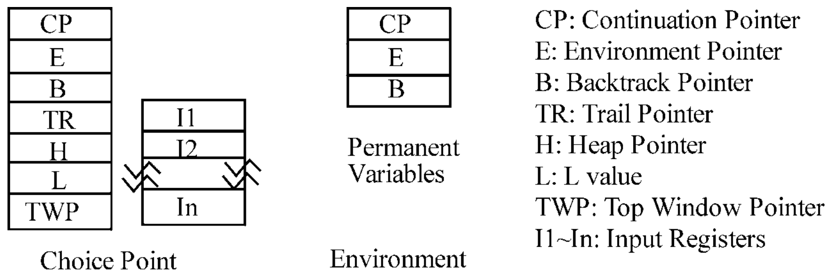

Fig. 2. Data in choice point and environment.

The drawback of this design is that although environments can be stored in registers, the choice points are still kept in the data memory, and memory accesses to the choice points constitute about half of the total data memory accesses. To eliminate this drawback, we need a design that stores both environments and choice points in registers.

As shown in Fig. 2, the choice point has seven register pointer values and the contents of the arguments for restoring the system status [6], and the environment is composed of CP, B, and E pointer values and permanent variables. The pointer values (CP, E, B, TR, H, L, TWP) of the choice point and environment are stored in local registers, and the arguments of the choice pointer and permanent variables can be stored in input and output registers. In the following paragraphs, we will describe how to store a choice point and an environment in a register window.

As we have discussed, since there are multiple, overlapping register windows in a windowed register file design, arguments are stored in input registers and transferred to the subgoals without running the risk of being overwritten, so the permanent variables of an environment are no longer needed, and the environment now consists of only three registers (CP,E, and B).

During Prolog program execution, a choice point is created if the procedure has more than one clause, and an environment is created if this clause has more than one subgoal. If either only a choice point or only an environment is created by a clause, the created data can be stored in the current register window; however, if both are created by a clause, both of them should be stored in the same register window. In this case, a clause creates a choice point first and then an environment. We notice that the values of the continuation pointer (CP), the environment pointer (E), and the backtrack pointer (B) of the environment are the same as those of the choice point. So the program can detect whether a choice point has been created in the current register window; if so, the environment does not need to be created, and the local register values are valid.

A problem concerning the arguments may arise with the use of register windows. According to a characteristic of the multiple register window design, the arguments of a choice point and the arguments of a clause being executed can share the same input registers at different times. The arguments of a choice point are used when backtracking, and the clause arguments are used in forward execution. Normal program execution may bind the variables in forward execution. However, should the clause fail and backtrack, the bound variables must be restored to their original status and used to retry in the other clauses. To solve the restoration problem of the bound variables, standard Prolog code can be used to

reset the values back to their original unbound variables. The failure handling routine of a Prolog system may unbind the bound variables in backtracking, so that the variables in the choice point are restored and the program can continue to execute correctly.

Hence the choice point, environment, and arguments can be stored together in a register window. In this way, memory accesses to the control structures are no longer needed, and the total number of memory accesses in typical Prolog programs is reduced by 75%.

The environment used for forward execution can be discarded on return. But the choice point created will be needed if the clause fails and retries, so it should be preserved in the register window when the clause is executed successfully and returns. SORWT uses two window pointers, as shown in Fig. 3. A CWP points at window W_i, in which the input and local variables are accessed. Another pointer, the TWP (top window pointer), points at window W_{i+n} (where n may be any positive integer), in which the input registers are used as the output registers of W_i. This split window structure facilitates saving of and access to the environment without disturbing the information stored in windows W_{i+1} to W_{i+n-1} (which is W_{i+1} in this example). The register windows preserved between CWP and TWP contain the choice point(s) that may still be useful.

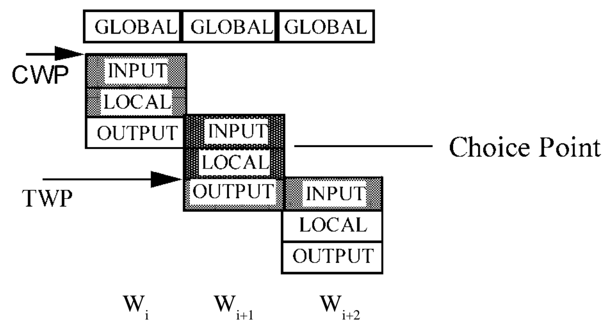

Fig. 3. Organization of the splittable overlapping register window.

Fig. 4 illustrates how the environment and choice point can be preserved in a register window for correct program execution. In SORWT, TWP is adjusted to the next available window in response to a successful unification of the clause head with its callee. Fig. 4(c) shows that X is bounded to 5 in f(X). After f(X) returns, TWP remains unaltered while CWP points back at the caller of f(X)--the main procedure in this case. In this way, the choice point of f(X) is preserved in the second window even after g(Y) is activated. In Fig. 4(e), the choice point of g(Y) has also been preserved. In Fig. 4(f), it can be seen that the unification of h(5,1) will fail in procedure h, so the TWP does not need to be adjusted. Finally, in Fig. 4(g), g(Y) is retried to obtain g(2). The detailed implementation and operation of SORWT will be described in the next section.

This approach has another advantage. To implement the cut operation, the PLM discards all the choice points above the B register value saved in the current environment. But a choice point may have been created if the current procedure has more than one clause, and in this case one more choice point must be discarded in the cut operation. The PLM

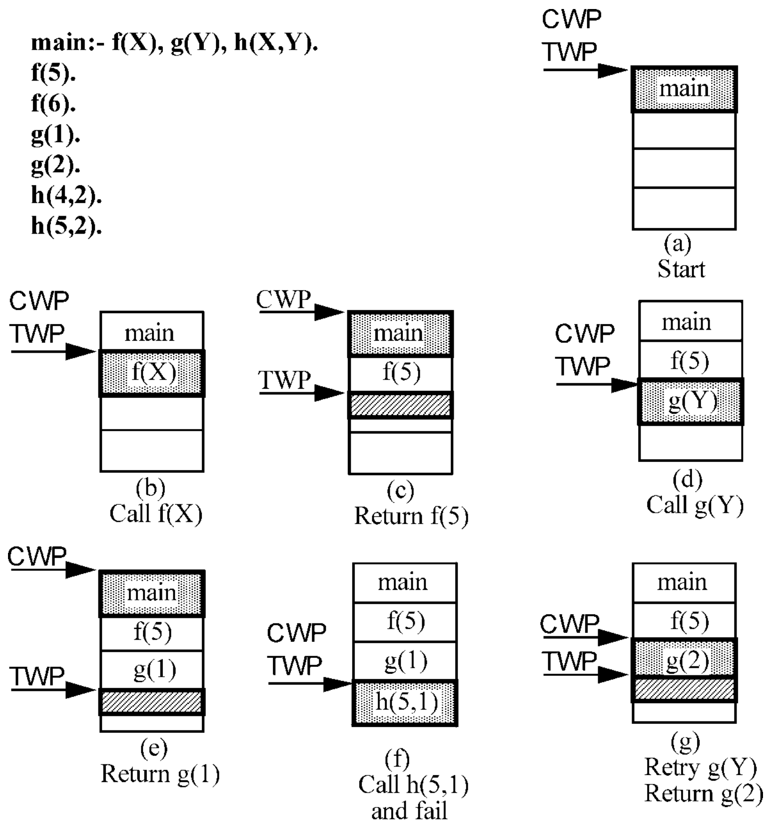

Fig. 4. Prolog program execution snapshots of SORWT.

uses a cut bit in the environment to identify this choice point. In SORWT, on the other hand, the B value is generated by the choice point, so the cut bit and detection are no longer needed.

For a windowed register file with a fixed window size, the following relationship holds:

$$\mathbf{S_{RF} = S_{global} + S_{RW} * N_{RW}}, \tag{1}$$

where S_{RF} represents the size of the register file, S_{global} is the number of registers for global variables, S_{RW} is the size of (the nonoverlapping portion of) a register window, and N_{RW} is the number of windows. Some factors must be considered in selecting the values of S_{RW} and N_{RW}. A larger value of S_{RW} implies that more local data can be stored in a register window. This in turn means that the possibility of argument overflow is reduced, and so is the number of memory loads/stores. On the other hand, the number of windows in the register file thus becomes smaller, resulting in a shallower procedure call depth before register window overflow occurs. Moreover, a larger window size also means more memory traffic between the CPU (central processing unit) and memory in a window overflow/underflow. Selection of register file parameters thus requires careful study and evaluation.

SORWT can be applied to register files of different sizes and configurations. Section 3 will describe how SORWT copes with different sizes and configurations of register files, register windows, and memory window matrices.

The current implementation of our register file contains 148 registers. Twenty of these are global registers, and the rest are window registers. Candidate S_{RW} values are usually a power of two for efficient register identifier decoding. The choice of S_{RW} and N_{RW} also affects the instruction set architecture since the number of bits needed to encode a register identifier and the number of global registers allowed also vary with S_{RW} and N_{RW}. From parameter count statistics obtained from the many available Prolog benchmarks, we selected two pairs of S_{RW} and N_{RW} values, [S_{RW} = 8, N_{RW} = 16] and [S_{RW} = 16, N_{RW} = 8], and compared the performance of the different register system designs using these two pairs of values. The performance evaluation results for the two designs described here are presented in section 4.

2.2 Implementation of SORWT in WAM and PLM

Certain instructions in the Warren and PLM instruction sets may be affected by the use of register windows. This subsection discusses how these instructions can be implemented.

A number of special-purpose registers are needed to handle window scheduling and manipulation. Some of these registers have the same names as those in PLM. **CWP** is a pointer pointing at the current working register window. Accesses to input and local registers are made through CWP as the base register. **TWP** points at the register window containing the output registers. The other special-purpose registers include **PC**, the program counter, **CP**, the continuation pointer, **E**, the environment pointer, **B**, the backtrack pointer, **TR**, the trail pointer, **H**, the heap pointer, **L**, the entry point for the next clause, and **S_{RW}**, the window size. Registers **L1** to **L7** are the local registers.

A procedure return in Prolog is handled in a manner different from that used in conventional languages. In WAM as well as in our design, the procedure return is called *proceed.*

The implementations of procedure call/proceed, environment creation/deletion, choice point handling, and backtracking in SORWT are described in detail below.

(1) Procedure call/return:

```
call:
   CWP  <-  TWP               ;set CWP to the next available window.
   CP   <-  return address    ;store return address in the CP register.
adjust window:
   TWP  <-  CWP + S_RW.       ;set a new TWP pointer after unification succeeds.
proceed:
   if  CWP > B    then        ;CWP <= B means  some useful
      { TWP  <-  CWP }        choice points exist between CWP
   CWP  <-  E                 and TWP, so the TWP will not
   PC   <-  CP                be changed in order to reserve them
                              (the window is splittable).
```

(2) Environment creation/deletion:

allocate:

```
if  CWP <> B    then        ;if CWP is not equal to B, meaning no
{  L1  <-  CP               choice point is created in current window,
   L2  <-  E                then save an environment (as shown in
   L3  <-  B }              Fig. 2).
E   <-  CWP                 set current environment pointer E to CWP.
```

deallocate:

```
CP  <-  L1                  ;restore environment.
E   <-  L2
```

(3) Choice point management:

try_me_else:

```
L1  <-  CP                  ;save choice point (as shown in Fig. 2).
L2  <-  E
L3  <-  B
L4  <-  TR
L5  <-  H
L6  <-  TWP
L7  <-  L
B   <-  CWP                 ;set B pointer.
```

retry_me_else:

```
L7  <-  L                   ;rewrite L value of choice point.
```

trust_me_else:

```
B   <-  L3                  ;restore B.
```

(4) Backtracking:

fail:

```
CWP   <-  B                 ;set CWP to B to the access choice point.
CP    <-  L1                ;restore status in the choice point.
E     <-  L2
B     <-  L3
TR    <-  L4
H     <-  L5
undo trail                  ;unbound variables.
TWP   <-  L6                ;restore TWP.
PC    <-  L                 ;set program counter to the next clause entry.
```

3. HANDLING OF OVERFLOW AND UNDERFLOW IN SORWT

In this section, the behaviors of window operations in Prolog program execution are described, and the effects of these behaviors on overflow and underflow handling in SORWT are discussed. An MWM (memory window matrix) organization is introduced to handle register window overflows/underflows. Mathematical expressions showing the mapping between register windows and MWM are also provided.

Basically, stack pointers such as CWP and TWP can be treated as subscripts of the array MWM of memory locations which are used for storing overflowed register window values. The low-order bits of these subscripts are used to choose a specific register window. Each register window is augmented by a tag which stores the high-order bits of its current subscript. For each register window access, if the high-order bits agree, then no memory accesses are required. On the other hand, if the high-order bits of the subscript are larger or small than the actual register window's current high-order bits, this indicates that a register window overflow or underflow, respectively, has occurred, and an appropriate memory access must be used to rectify the situation.

3.1 Problem of Overflow/Underflow With Register Windows

Using a windowed register file to handle call/return operations of a conventional programming language is simple, as shown in Fig. 5. The CWP needs to be adjusted forward and backward by a constant distance only, and a stack is used to handle window overflow and underflow. Because the window adjustment is sequential, the overflow window can simply be pushed onto the stack and then popped back when it is needed again.

In Prolog execution, on the other hand, window adjustment is much more complex. The operations of call, proceed, cut, and backtracking all change the values of the window pointers in different ways. The effects and results of these operations are also shown in Fig. 5. The call operation modifies CWP to TWP and adjusts TWP to the next window. The return instruction, proceed, restores CWP to E register, and when no choice point exists between CWP and TWP, it adjusts TWP backward to CWP. The fail operation restores CWP and TWP to the corresponding values in the choice point; the cut operation adjusts the TWP backward to the next window of CWP if any choice point exists between CWP and TWP.

Conventional Programming Language

	CWP
Call	o $+S_{RW}$
Return	u $-S_{RW}$

o overflow is possible

u underflow is possible

Prolog

	CWP	TWP
Call (Call/Execute)	TWP	o $+S_{RW}$
Return (Proceed)	u E	CWP [1]
Backtracking (Fail)	u B	u L6
Cut		u $CWP+S_{RW}$ [2]

1. If CWP>B (no choice point exists)
2. If CWP<=B (some choice points exist)

Fig. 5. Call/return instructions and register window overflow/underflow behaviors.

These instructions may induce a window overflow or underflow. This fact makes overflow/underflow detection and handling in SORWT much more complex. The return and fail operations restore window pointers from some registers; these restoration operations permit a skip return and adjust the window pointers to point at appropriate windows. Using a stack to handle register window overflow and underflow in Prolog, as in a conventional programming language, is inappropriate. Hence in SORWT, we use a matrix-style data structure, called a memory window matrix (MWM). In the following, we introduce the organization and operation of the MWM.

3.2 Register Window Overflow/Underflow Management

The distinct features of register window overflow/underflow management in SORWT are as follows:

(1) The detection of overflow and underflow is carried out using a tag mechanism; in comparison, in a conventional programming language system, such as SPARC, a WIM register is used [17].

(2) Whenever overflow/underflow occurs, through the use of the CWP and TWP values, a register window block and a memory window block can be identified, and register window save/restoration can take place directly between the CPU and the MWM, without incurring any unnecessary overhead.

(3) The overflow/underflow characteristics of Prolog cause the order of procedure (and hence window) activation to be disruptive; use of the MWM can remedy the extreme inefficiency in register window overflow/underflow handling that would result from using a traditional stack method.

The MWM is a memory space for preserving overflow register windows. The MWM consists of many memory windows, as shown in Fig. 6. Memory windows are identical in size to register windows, and they are used to save register windows when register windows overflow.

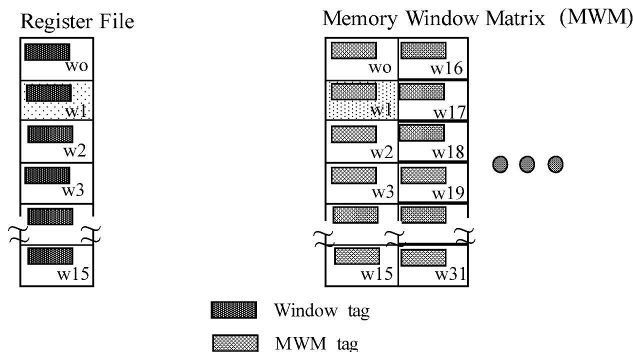

Fig. 6. Organizations of register file and memory window matrix.

At the beginning of the execution of a program, register window w0 is used for the main procedure. If a subgoal is called, then the values of CWP and TWP are changed to point at the next window. The return operation will modify the value of CWP, but if there is a choice point in the current window, then TWP will not be adjusted backward, as mentioned above. The window(s) between the split window pointers is (are) used to store choice point(s).

If the number of register windows used during program execution exceeds the number of windows in the architecture and further windows are required, then a register window overflow occurs. The data in register window w0 are flushed to a corresponding memory window, and the space of register window w0 is then used for the next procedure. The handling of further register window overflows follows the same procedure.

A register window pointer is used to specify a window in the register file. In SORWT, there need to be two window pointers, CWP and TWP, both of which are used for overflow/underflow detection and handling. Besides the usual register window identifier field, one extra field is added in the pointers to detect and handle window overflow and underflow. The format of the window pointers is shown in Fig. 7. The window pointers comprise two fields. Bits 3 to 0 in CWP or TWP are the window identifier field, which specifies one of the register windows (16 windows in our design). We also call it the row address field in MWM access because it also indicates the row address of the corresponding memory window. The number of bits in this field is adjustable according to the architectural specification of the register file. The second field is the window tag field. It contains 8 bits (from bit 11 to bit 4) in our design, and it reflects the value of the window tag, which is used for register window overflow/underflow detection. When overflow occurs, the value of this field can be used to locate the column address of a memory window.

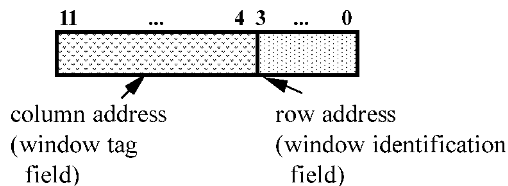

Fig. 7. The format of CWP and TWP.

The SPARC CPU, by Sun Microsystems Inc. [17], uses a register called WIM to detect window overflow and underflow in the register file. In WIM, only one bit is "set," and each bit represents a register window. If the program activates the next window and its corresponding bit in WIM is 1, then a window overflow occurs, the register window is flushed to a memory stack and the "1" bit in WIM is shifted right before the program can proceed. When a register window underflow triggered by a procedure return occurs, the reverse procedure is performed. This method is good enough for a sequential window adjustment system; unfortunately, it is unsatisfactory for a Prolog machine. An alternative must be found that can meet the requirements of Prolog program execution. In our design, we propose to use a tagged windowed register file architecture for overflow/underflow detection.

In our register window design, as shown in Fig. 6, a tag is appended to every window. A tag value can be accessed from the tag field of the specified register window. When a register window is to be activated, if its tag is valid, then we can match it against the window tag field (bits 11 to 4) of CWP or TWP. An overflow or underflow has occurred if these two values do not match. If the register window tag is smaller than the window tag field of CWP/TWP, then a window overflow has occurred; if it is larger, an underflow has occurred.

Every window in the MWM can be explicitly addressed using the column and row fields in CWP and TWP, or using the register window id and the tag value of a register window. When an overflow occurs, we can use the window tag and window id of a register window to locate a memory window and to then flush the overflow window into the corresponding location in the MWM. The underflow procedure restores a window and its tag from the MWM to the register file; the particular window in the MWM is identified by the column and row address fields of the window pointers. This method provides for efficient direct window accesses in the MWM and greatly reduces register window overflow/underflow overhead.

Register windows are flushed to memory upon window overflow and are restored upon underflow. In our Prolog system, upon underflow, the overwritten window may contain a choice point, which should be saved to the MWM. A cp bit is used to specify whether there is a useful choice point in the window.

Fig. 7 illustrates how the register file and MWM sizes can be adjusted. To adjust the sizes, we need only redefine the CWP and TWP formats. If the size of the register file, the number of windows, or even the MWM parameters are changed, only these two bit fields need to be modified.

For example, if the register file is doubled in size but the window size remains unchanged, then the window identifier field is modified to contain bits 4 to 0, and the field to its left is shifted left by one bit position. In addition, the row dimension of the MWM is equivalent to the number of windows in the register file, and the column dimension is controlled by the window tag field. Adjusting these two fields adjusts the size of the MWM. Hence in SORWT, the register file and the MWM may be adjusted to suit different needs.

3.3 Mapping Between Register Windows and the MWM

Another feature of SORWT is that all register windows are assigned corresponding memory windows, and the memory window addresses can be calculated from CWP and TWP. This feature reflects the characteristics of Prolog and handles Prolog programs well. The mapping between register windows and memory windows is described below.

We first define several terms :

CD: call depth, the depth of procedure calls (the procedure level count);
A_{BASE}: base address of the memory window matrix (MWM);
A_{MW}: starting address of a specific memory window;
N_{RW}: number of windows in the register file;
S_{MW}: the size of a memory window;
W_n: register window id in the register file (field two in Fig. 7);
T_n: tag value of a register window (field three in Fig. 7);
ROW: row address of a window in the MWM;
COL: column address of a window in the MWM.

From these definitions, the following relation holds:

$$\mathbf{A_{MW} = A_{BASE} + CD * S_{MW}}. \tag{2}$$

The starting address of a particular memory window is the base address of the MWM plus the call depth of procedure calls multiplied by the memory window size. From this expression, we can determine the memory window location of a particular procedure level count in the MWM. The row address indicates the register window id of the register file:

$$\mathbf{ROW = W_n = CD \ \ mod \ \ N_{RW}}, \tag{3}$$

and the column address (stored in the window tag) is equal to

$$\mathbf{COL = T_n = CD \ \ div \ \ N_{RW}}. \tag{4}$$

The row and column addresses are stored in CWP and TWP, and we can specify the call depth using them:

$$\mathbf{CD = COL * N_{RW} + ROW} \tag{5}$$
$$\mathbf{= T_n * N_{RW} + W_n}. \tag{5a}$$

From (2), (5), and (5a), the following expressions hold:

$$\mathbf{A_{MW} = A_{BASE} + (COL * N_{RW} + ROW) * S_{MW}} \tag{6}$$

$$\mathbf{A_{MW} = A_{BASE} + (T_n * N_{RW} + W_n) * S_{MW}}. \tag{6a}$$

When register window overflow occurs, by expression (6), the tag value ($\mathbf{T_n}$) and the register window id ($\mathbf{W_n}$) can be used to obtain the location of the corresponding memory window in the MWM. Upon underflow, as shown in (6a), the column and row address fields of the window pointers are used to fetch the memory window in the MWM.

Expression (6) can be obtained directly from CWP and TWP. The binary representation of the window pointers is the direct result of this expression. MWM accesses can, thus, be made very efficiently.

4. PERFORMANCE EVALUATION AND PRACTICAL IMPLEMENTATION OF SORWT

This section compares the performance of the SORWT register file using the two different configurations mentioned in section 2. Thirty benchmark programs were used in the performance evaluation, which was performed through software simulation. The evaluation metrics used are the number of window overflows and underflows, the number of argument overflows, and the number of memory accesses.

We will discuss the characteristics of the benchmarks and the performance of the SORWT register file with respect to different benchmarks. Finally, we shall investigate the implementation and efficiency of the argument overflow stack and register translation.

4.1 Benchmark Programs and Their Characteristics

Table 1 lists the thirty benchmark programs. Twenty-two of the programs are the benchmark programs used in the PLM system [5] and BAM literature. In [2], thirteen out of the twenty-two were used to assess the system performance of SPUR. Eight of the programs were collected during the course of this research. They represent applications originally overlooked in the PLM benchmark set, such as natural languages and expert systems. We believe the enhanced benchmark set is more representative of general Prolog applications.

Table 1. Classification of Benchmarks.

Benchmark Attribute	Database Query	Logic Inference	List and Structure Processing
Benchmark Name	query, dbil, match, movel1, move2	hanoi, mutest, ckt2, nlpx, sheet, queens, qen6c, qen6s, qen6x, color 13g, color13b, mapx	con1, con6, nrev1, divide10, lgo10, ops8, times10, plain25, pri2, qs4, btree, psort1, psort2

In the following discussion, these benchmark programs are categorized into three groups according to their characteristics and applications, as indicated in Table 1. All Prolog programs can be placed into one of these three categories, based on each program's data processing characteristics, although different categorizations may also be valid. The three categories are database query programs, logical inference programs, and list and structure processing programs. Some programs may exhibit all three of these characteristics, and this categorization is not intended to be precise. The purpose of this table is merely to show the properties and distribution of this set of benchmarks.

Database query-type Prolog programs consist mainly of database clauses. In the processing of such programs, register windows are used mainly to store choice points. Procedure calls are rarely recursive, and the call depth is usually shallow. On the other hand, backtracking is very frequent.

Lists and structures are two complex data structures in Prolog. Processing of these data structures is generally recursive, and the depth of recursion, which in turn indicates the number of register windows required, is determined by the length of the data structures. In the processing of such programs, the register windows are used mainly to store environments, and window overflows/underflows occur very frequently.

Table 2. Comparison of the number of overflows and underflows.

	Number of Windows				
	8		16		Maximum
Benchmark	#overflows	#underflows	#overflows	#underflows	call depth
query	0	0	0	0	6
dbil	0	0	0	0	7
match	0	0	0	0	3
move1	0	0	0	0	4
move2	0	0	0	0	5
hanoi	3	3	0	0	9
mutest	38	31	0	0	15
ckt2	329	315	67	67	28
nlpx	406	406	0	0	11
sheet	19	9	8	4	19
queens	15	9	0	0	14
qen6c	110	97	28	22	25
qen6s	556	639	192	189	18
qen6x	2118	2476	731	729	18
color13g	32	10	22	6	32
color13b	35969	36112	32140	32124	32
mapx	197	197	0	0	9
con1	0	0	0	0	1
con6	0	0	0	0	2
nrev1	24	24	16	16	31
divide10	4	4	0	0	11
log10	5	1	0	0	12
ops8	0	0	0	0	5
times10	4	4	0	0	11
plain25	5	5	0	0	12
pri2	147	27	122	36	101
qs4	72	53	43	39	51
btree	0	0	0	0	7
psort1	45	42	0	0	12
psort2	23	20	0	0	12

In logical inference programs, the code is composed of a large number of both inference clauses and database clauses. These programs exhibit both database query and list and structure processing characteristics. Inference clauses are usually recursive, and database clauses constitute boundary conditions for the depth of recursive calls. This restricts the number of register windows needed in inference resolution, so fewer register windows are needed than in list and structure processing.

4.2 Performance of SORWT With Different Register Window Parameters

Table 2 lists the number of register window overflows and underflows for SORWT using the benchmark programs, for [window size (S_{RW}) =8, window number (N_{RW}) =16] and [S_{RW} =16, N_{RW}=8]. The benchmarks were executed by a simulator program, and a window pointer trace file was obtained through the simulation. This trace file was then fed to a register file simulator program to generate the overflow/underflow results.

In database query programs, window overflows and underflows rarely occur. In addition, the number of register windows required is independent of the number of database clauses. Examples are the programs *query* and *dbi1*. The virtues of SORWT can be fully put to use when this type of program is run. Since the choice points for database queries are saved in the register file, many memory stack accesses are eliminated.

Call depths for list and structure type programs are usually deep. Since the number of data elements in a list or structure is usually much larger than the number of register windows available, window overflows and underflows occur frequently. One example is the benchmark nrev1: For N_{RW}=8, the number of overflows was 24; for N_{RW}=16, this number was reduced to 16. This reduction in the number of overflows is just the number of extra windows available.

For logic inference type programs, due to frequent backtracking, the number of register windows has a definite effect on the number of window overflows. An example is the benchmark *qen6c*, for which the number of window overflows for N_{RW}=8 was 110. For N_{RW}=16, on the other hand, it was 28, reflecting a reduction of 74.5%. That a larger number of register windows reduces the number of overflows/underflows is clearly indicated in Table 2.

In summary, in database query type applications, either of the two configurations provides acceptable performance. In list and structure type programs, the deep recursion incurs a large number of overflows and underflows in either configuration although backtrackings are handled very well because of the splittable nature of the windows. In logical inference type programs, as shown in Table 2, the different window configurations result in a significant difference in window overflow and underflow counts. For this reason, we adopted [window size (S_{RW}) =8, window number (N_{RW}) =16] for use in our design.

Overhead due to register window overflow/underflow can be expressed by

$$\mathbf{\#overflow * (T_{SRW} + T_{trap}) + \#underflow * P_{cp} * (T_{SRW} * 2 + T_{trap}) + \#underflow * (1 - P_{cp}) * (T_{SRW} + T_{trap}),} \quad (7)$$

where T_{SRW} is the time needed for register window transfer between the CPU and memory window matrix, Ttrap is the overhead due to interrupt services, and Pcp is the possibility of cp bit =1 indicating an underflow. Ttrap consists of the following time components: pipeline disruption, machine state saving, interrupt service routine processing, and return delay.

4.3 Analysis and Handling of Argument and Control Structure Overflow

Argument overflow occurs if the number of I/O arguments exceeds the number of I/O registers. In our design, twenty global registers are visible at any level of procedure: twelve of them are special-purpose registers, such as the E, B, and HP registers for Prolog execution, and eight of them are temporary registers. Four local, four input, and four output registers are in the register window in our design, making an addressable register space of thirty-two. Input arguments are allocated to i_1 through i_n by the compiler, where n can take on any natural integer value. If n exceeds the allowed number of input registers (four in this case), the code generator will convert the extra input registers to memory addresses in the argument overflow stack. For each procedure call, the compiler also allocates parameters to be passed to output registers o_1 to o_n. If n exceeds four, the extra output parameters are again allocated to the argument overflow stack.

Fig. 8 shows how a register window of [window size =8, window number =16] is used to store the control structure (environment and choice point). Storage of the choice point is similar to that of the environment. Since there are only four local registers in a window, some of the pointers of the choice points must be pushed onto the stack pointed to by CHP. Those pointers used less frequently may be pushed into memory.

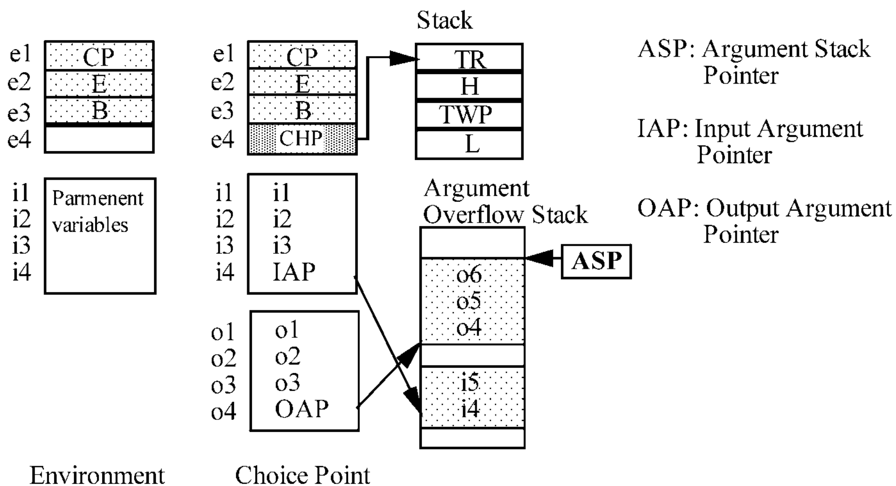

Fig. 8. Implementation of the environment, choice point, and argument overflow stack.

The overflow arguments are stored in the argument overflow stack. Table 3 lists statistical data on the number of argument overflow references. These reads and writes are due to procedure call argument overflows. It can be seen that, on average, the argument overflow reads constitute 4% of all the data memory reads, and the writes constitute 1.6% of all the data memory writes. Among the benchmark programs, *hanoi* is recursive, and all its procedure calls access variables in the argument overflow stack. Hence its number of memory reads is relatively large. The simulation results indicate that most arguments are stored in input and output registers, and that only a few need to be stored in the argument overflow stack.

Table 3. Memory reference to the argument overflow stack.

Benchmark	Number of Memory References to Argument Overflow Stack			
	Reads	R/Rtotal	Writes	W/Wtotal
query	0	0.00	0	0.00
dbil	59	0.04	13	0.01
match	0	0.00	0	0.00
move1	15	0.02	1	0.00
move2	14	0.02	1	0.00
hanoi	765	0.30	255	0.04
mutest	80	0.01	1	0.00
ckt2	858	0.02	981	0.02
nlpx	192	0.00	1536	0.05
sheet	42	0.05	42	0.05
queens	70	0.03	36	0.02
qen6c	519	0.03	317	0.03
qen6s	0	0.00	0	0.00
qen6x	0	0.00	0	0.00
color13g	70	0.07	18	0.04
color13b	118707	0.04	18	0.00
mapx	108	0.01	1	0.00
con1	0	0.00	0	0.00
con6	0	0.00	0	0.00
nrev1	0	0.00	0	0.00
divide10	27	0.18	9	0.02
log10	0	0.00	0	00
ops8	7	0.08	5	0.02
times10	18	0.15	9	0.03
plain25	138	0.06	199	0.11
pri2	0	0.00	0	0.00
qs4	200	0.09	150	0.04
btree	0	0.00	0	0.00
psort1	0	0.00	0	0.00
psort2	0	0.00	0	0.00
Average		0.04		0.016

Rtotal: total number of reads to data memory
Wtotal: total number of writes to data memory

5. THE EFFECT OF SORWT ON CYCLES PER INSTRUCTION AND FINE-GRAINED PARALLELISM

5.1 Comparison of the Number of Cycles on Different Machines

Table 4 shows the number of cycles needed for the instructions concerning the environment (allocate, deallocate), choice point (try_me_else, retry_me_else, and trust_me_else), and window adjustment (call, proceed, and cut) in PLM, SPUR, and our design. These implementations use only register or memory transfer instructions and branch instructions. The difference in performance is due to the implementation: SORWT stores environments and choice points in register windows, so fewer cycles are needed to implement these operations. The two cycles are needed for call and proceed instructions: one for window adjustment, and one for call and return operations. PLM does not need to adjust the window pointer and implements it in one cycle.

Table 4. Comparison of the number of cycles.

Implementation \ Operation	allocate	deallocate	try	retry	trust	call/proceed	cut
Our System	2-5	3-4	10	2	2	2	1-4
PLM	11	6	20	2	5	1	10
SPUR	5	4	20	6	3	4	5.5

5.2 Effect of SORWT on Fine-grained Parallelism

Memory access operations constitute 32.7% of all the operations executed in typical RISC Prolog programs. In the research on instruction-level parallelism, the models of [4] and [10] both issue only one load or store request per cycle to the data cache. In these models, the serialized memory accesses limit the peak instruction-level parallelism to less than 3, even if the parallelism of ALU operations, register transfers, and control operations can be increased without limit. One way to overcome this limit on parallelism due to the serialized memory accesses is to enhance the design of the CPU I/O, cache, and memory system, and to initiate more than one load or store operation per cycle.

Cache updates must be performed in the order specified by the program, and no cache update is performed until it is absolutely correct to do so. A store operation updates the data in the cache/memory, changing the processor state. Thus a store operation must be performed in program order with respect to other memory access operations. This preserves the in-order state of the program in the data cache and main memory. On the other hand, a store instruction is held until all previous loads and stores operation are completed, and acts as a barrier to the execution flow. A store instruction reduces the fine-grained parallelism of a program and cannot be executed in parallel with other store and load instructions antecedent to it.

Load operations may be executed in parallel in a VLIW or superscalar machine with a parallel memory, such as a multi-bank cache. But the problem of cache bank conflicts arises, and the conflict ratio was about 15% in our simulation for a two-bank cache memory.

Dependency analysis is necessary for memory disambiguation in parallel accesses to separate cache banks. However, the majority of memory accesses in Prolog programs are made to the stack, and they cannot be disambiguated since they are indirect memory accesses made by means of register pointers. Therefore, the parallelism of load operations in Prolog programs is very limited.

Store operations constitute 46% of all memory access operations in typical Prolog programs. The large percentage of write operations is due to the creation of choice points and environments, the construction of lists and structures, and information on bound variables that is pushed onto the tail.

Most store operations are for saving data to the control stack; and the choice points created are used only when program failures and retries occur in the procedure. Therefore, it is possible that some choice points will be created and left unused once the input query is satisfied. The B register value in the environment is used for the cut operation only; it may be redundant if no cut operations exist in the clause. For forward execution in Prolog programs, the register values of E and CP are saved in the environment and are accessed only when they are updated. In conventional programming languages, data are commonly read several times but written only once. In Prolog programs, the control information is stored in the data memory, and the load/store operation counts to the control data are almost equal, as discussed earlier in this section. Most memory accesses are made to the control information, so the memory access ratio of the store operations is much higher than it is in functional programming languages.

The large number of store operations to the control stack creates a write barrier. Thus even if the design implements parallel memory and all the load operations can be executed in parallel with the other instructions, at least about 32.7%*46% = 15% of all operations must still be executed sequentially in a fine-grained parallel machine because of these write barriers.

Another problem concerning the large number of memory access operations is the penalties due to cache misses and write policies. The performance degradation for cache misses in a VLIW and superscalar system is

memory operation ratio * cache miss ratio * cache miss overhead* speedup. (8)

In our simulation, the cache miss ratios of the thirty benchmark programs ranged from 3.5% to 5% when the cache size varied from 16 Kbytes to 64 Kbytes. If the overhead of a cache miss is 20 cycles, by expression (8), the performance degradation is 45.8% to 65. 4% (assuming the speedup is 2). With a write-through policy, all store operations will incur a cache miss, and the result will be a catastrophic decline in performance.

SORWT provides a good solution to all these problems. In SORWT, memory accesses to choice points and environments are eliminated. Since accesses to choice points and environments constitute 75% of all memory accesses in typical Prolog programs in conventional designs, SORWT can reduce the ratio of memory access operations from 32.7% to 8.2%. Besides the immediate benefit of shorter delays and better pipelining, 24.5% of all operations are now converted from memory access operations to register transfer operations, thereby increasing the potential parallelism in a parallel processing system.

Table 5 lists performance data for SORWT and the other techniques for memory systems.

Table 5. Performance measurements of SORWT and other cache design techniques.

Configuration	One-port Cache	Two-bank/port Cache with bank conflicts	N-bank/port Cache no bank conflict	SORWT idealized	SORWT* 16 window argument overflow stack
% Memory Operation	32.7%	26.6%	15%	8.2%	8.44%
Speedup Bound	3.06	3.76	6.67	12.2	11.8
Cache Miss Penalty (1%)	6.54%	6.54%	6.54%	1.64%	1.69%
Cache Miss Penalty (5%)	32.7%	32.7%	32.7%	8.2%	8.44%
Cache Miss Penalty (5%) (Speedup=2)	65.4%	65.4%	65.4%	16.4%	16.9%
Extra Hardware Costs		additional data-cache interface	high hardware cost	additional window pointer and decoder	additional window pointer, MWM and decoder

*Overhead of window overflow/underflow is 1.27% should be added.
Cach misses consume 20 cycles each.

The percentage of load operations is 54% of all memory operations, and the percentage of store operations is 46% in Prolog programs. The memory access percentage is 32.7% of all operations, thus the percentages of load and store operations (L-Ratio and S-Ratio) are 17.7% and 15%. Let the peak parallelism of load operations in a K-bank cache memory be $L_{peak}(K)$, and let the conflict ratio of the K-bank cache be C(K). Then the operation ratio of all serialized memory accesses with a K-bank cache becomes:

$$\textbf{(L-Ratio - L-Ratio * C(K)) / } \mathbf{L_{peak}}\textbf{(K) + L-Ratio * C(K) + S-Ratio.} \quad (9)$$

In our simulation, C(2) was 15% and $L_{peak}(2)$ was 1.69, so the serialized memory access ratio to a two-bank cache was 26.6%.

With an N-bank (unlimited-bank) cache memory, an ideal system with no bank conflict, the write barriers of the store operations (15%) still limit the parallelism of the system, as discussed above.

In SORWT, the memory operation ratio is 8.2%, only one fourth of the original memory access ratio. In our implementation, there are 16 windows in the register file, and the overhead of window overflow/underflow is obtained by the simulation results and expression (7). The overhead is only 1.27% on average in a sequential execution environment. Another overhead of our implementation is due to the argument overflow, the overflowed

arguments stored in memory. As shown in the Table III, the argument overflow ratio is quite small, only 2.9% on average. In our simulation, it increased by 0.24% of memory access operations in program execution. The memory operation ratio of our implementation in SORWT is 8.2% + 0.24% = 8.44%, and an extra overhead of window overflow/ underflow to the MWM is 1.27%.

Table 5 also projects the performance degradation caused by cache misses in various designs. It assumes that the cache miss penalty is 20 cycles. Notice that these measures are obtained in a sequential execution environment. Thus if the speedup is two in a parallel system, the performance degradation due to cache misses is doubled. For example, if the cache miss ratio is 5%, then in a single-port, two-port, or N-port cache system, the performance penalty is 65.4% if a speedup of two is achieved. In comparison, the peak speedups in [18] and [4] were even greater than two. Thus, SORWT is even more effective for high-performance parallel system designs.

Compared to the single-port memory, a two-port cache requires an additional CPU interface to permit parallel memory access operations. In the best case, as shown in Table 5, a two-port cache overlaps 32.7% − 26.6%=6.1% of the memory access operations with the other memory operations. An unlimited-bank cache incurs excessively high hardware costs, and even if we assume that there is no cache bank conflict, the serialized memory operation ratio is still worse than that obtained from SORWT. SORWT requires, other than CWP, a window pointer TWP and an extra decoder for the window pointers. The memory window matrix is a memory block allocated for the SORWT window overflows/underflows, replacing the overflow stack in traditional RISC window systems. The hardware cost and time complexity of the MWM are similar to those of a stack, and the MWM is extremely efficient in handling backtracking.

SORWT greatly reduces the number of load/store operations (and hence the number of write barriers) and performance degradation due to cache misses. With SORWT, a single-port memory is sufficient for a system, and complex memory disambiguation is no longer needed. The memory operations used to control structures that are eliminated are changed to register operations, which can be easily parallelized in a VLIW, superscalar, or superpipelined system. Thus SORWT is an extremely effective memory system technique for acquiring higher instruction-level parallelism by eliminating excessive and unnecessary memory traffic in a Prolog machine.

6. CONCLUSIONS

Memory access operations are executed sequentially in common VLIW, superpipelined, and superscalar systems [4, 10]. Even with a multi-ported memory or a multiple-bank cache design, write barriers severely limit fine-grained parallelism. The store operation ratio is 46% of all memory access operations in typical Prolog programs. This is because tedious bookkeeping work is required to guide the Prolog program execution. Consequently, innovative design alternatives are needed to solve this problem in the design of a high-performance Prolog machine.

SORWT implements the Prolog feature that was originally realized by storing environments and choice points in a memory stack. It effectively reduces the memory access overhead incurred due to the heavy bookkeeping workload. Program execution is, thus, greatly accelerated even in a sequential environment. Better still, fine-grained parallelism is greatly enhanced in a parallel Prolog machine.

The algorithms used in SORWT are simple and easy to implement. SORWT is a low-cost but effective technique suitable for RISC-type, superscalar, superpipelined, and VLIW Prolog systems, and it provides a significant gain in performance.

Our design incorporates 148 registers, of which twenty are used as global registers and 128 as window registers. The 128 window registers are divided into 16 eight-register windows. In each window, the registers are further separated into two groups: four input registers and four local registers. This configuration is based on the results of intensive program and hardware simulation/analysis. Four I/O registers are sufficient for argument storage for three quarters of the benchmarks; in addition, with the argument overflow stack, any number of arguments can be handled. The four local registers are for storing environments and choice points, allowing accesses to main memory to be eliminated.

To handle register window overflow/underflow, we propose a direct mapping function between register windows and memory windows. This MWM architecture allows for direct access to any memory window, as opposed to the conventional, sequential stack access. Furthermore, the memory window matrix size is adjustable at compile time, which gives the system more flexibility. This feature of SORWT is particularly useful in handling backtracking, a unique and vital feature in Prolog.

Thirty benchmark programs in three categories were used in system analysis and evaluation. The analysis clearly shows that SORWT outperforms nonoverlapping or non-splittable register windows in Prolog execution, and, for a system with 128 window registers, the register file configuration of sixteen eight-register windows is most effective.

REFERENCES

1. A. Aho, R. Sethi and D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1986.
2. G. Borriello, A. Cherencon, P. Danzing and M. Nelson, "Special or general purpose hardware for prolog," Technical Report, UCB/CSD-87-314, Department of Computer Science, University of California, Berkeley, 1986.
3. C.P. Chung, S. C. Jeng, H. C. Chou and C. Cheng, "Design of the dual-alu CRISC and its concurrent execution," *Journal of Information Science and Engineering*, Vol. 5, No. 3, 1989, pp. 251-274.
4. A DeGloria and P. Faraboschi, "Instruction-level parallelism in prolog: analysis and architecture support," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 224-233.
5. T. Dobry, Y. Patt and A. Despain, "Design decisions influencing the microarchitecture for prolog machine," in *Proceedings of the 17th Annual Workshop and Symposium: MICRO 17*, 1984, pp. 217-231.
6. B. Fagin and T. Dorby, "The berkeley PLM instruction set: an instruction set for prolog," Technical Report, UCB/CSD-80-126, Department of Computer Science, University of California Berkeley, 1985.
7. J. Fisher, "Very long instruction word architectures and the ELI-512," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983, pp. 82-87.
8. D. Halbert and P. Kessler, "Windows of overlapping register frames," CS292R-Course Final Report, Department of Computer Science, University of California, Berkeley,

1980.
9. B. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, W. Bush, A. Despain, J. Pendleton and T. Dobry, "Fast prolog with an extended general purpose architecture," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 282-291.
10. M. Johnson, *Superscalar Micro Processor Design*, Prentice Hall, 1991.
11. M. Katevenis, "Reduced instruction set computer for VLSI," Technical Report, UCB/CSB-83-141, Department of Computer Science, University of California, Berkeley, 1987.
12. H. Matsumoto, "A static analysis of prolog programs," *ACM SIGPLAN Notices*, Vol. 20, No. 10, 1985, pp. 48-59.
13. J. Mills, "A high-performance LOW RISC machine for logic programming," *Journal of Logic Programming*, Vol. 7, No. 2, 1989, pp. 179-212.
14. R. Onai, H. Shimisu and M. Aso, "Analysis of sequential prolog programs," *Journal of Logical Programming*, Vol. 4, No. 3, 1986, pp. 119-141.
15.D. Patterson, "Reduced instruction set computer," *Communication of the ACM*, Vol. 6, No. 8, 1985, pp.8-21.
16. A. Singhal and Y. Patt, "A high performance prolog processor with multiple function units," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pp. 195-202.
17. *SPARC RISC User's Guide*, Ross Technology Inc., 1990.
18. C. Su and A. Despain, "An instruction scheduler and register allocator for prolog parallel microprocessors," in *Proceedings of the International Computer Sympoisum*, 1992, pp. 699-706.
19.E. Tick, *Studies in Prolog Architecture*, Ph. D. thesis, Department of Computer Science, Stanford University of California, 1987.
20. S. U. Tung, *Design Considerations about a Prolog RISC Processor: LISCP-II*, master's thesis, Department of CSIE, NCTU, Taiwan, R.O.C., 1989.
21. D. Warren, *An Abstract Prolog Instruction Set*, Technique Note 309, Artificial Intelligence Center, SRI International, 1983.

Ruey-Liang Ma (馬瑞良) received the B.E. degree in electrical engineering from National Cheng-Kung University, R.O.C., in 1987, and the M.E. and Ph.D. degrees in computer science and information engineering from National Chiao-Tung University in 1989 and 1995. He is currently a manager at the Computer & Communication Research Laboratories, the Industrial Technology Research Institute, Hsinchu, Taiwan. His research interests are in the fields of computer architecture, CPU design, ASIC design and implementation.

Chung-Ping Chung (鍾崇斌) received the B.E. degree from National Cheng-Kung University, Taiwan, Republic of China, in 1976, and the M.E. and Ph.D. degrees from Texas A&M University in 1981 and 1986, respectively, all in electrical engineering. Since 1986 he has been with the Department of Computer Science and Information Engineering at Nation Chiao-Tung University, Hsinchu, Taiwan, Republic of China, where he is a professor. He is currently the director of the Advanced Technology Center at the Computer & Communication Research Laboratories, the Industrial Technology Research Institute, Hsinchu, Taiwan. His research interests include computer architecture, parallel processing, and parallel compiler design.