TECHNICAL REPORT NO. 297

Architectual Support for Delayed Specialization of Logic Programs

by

Jonathan W. Mills

December 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

Architectural Support for Delayed Specialization of Logic Programs

Jonathan Wayne Mills[*]
Indiana University
Bloomington, Indiana 47405

Abstract

Delayed specialization characterizes dynamic program modification and conditional instruction execution as the transformation of an unspecialized physical program P into a specialized virtual instance P' during the execution of P. The original program P is compiled into sequences of mixed instructions and schemata. Schemata are fetched as part of the instruction stream and used, typically with part of the processor's state, to select operations which are performed in their place. Schemata may be used with microcoded and random logic control units. Complex schemata are implemented by decoding the instruction stream before selecting microcode and/or generating control signals; simple schemata (such as conditional branches) by gating processor status and control signals. Architectural support for delayed specialization of logic programs includes schemata for dereferencing, unification and trailing which typically execute in one clock cycle and allow the WAM's unification instructions to be expanded in-line into sequences of one to five instructions and/or schemata. WAM instructions emulated with schemata are space and time efficient because code for matching, binding and failure is compressed into a single schema, regardless of the number of arguments known at compile-time. Partial unification in the LIBRA exemplifies delayed specialization in a language-specific architecture. To show how an existing computer architecture can implement unification, the SPARC RISC architecture is modified, and a new schema, **unify**, is added to its instruction set.

1. INTRODUCTION

Since the Warren abstract machine (WAM) for Prolog was first described (Warren 1983) two approaches have been taken to provide architectural support for logic programming. The first

[*] 101 Lindley Hall, Department of Computer Science, (812) 855-6486, jwmills@iuvax.cs.indiana.edu

approach, analogous to the design of a CISC architecture, embeds WAM instructions and recursive unification routines in microcode, then adds specialized functional units and buses as necessary to minimize the WAM instruction cycle time. This is done in the PLM (Dobry 1987), PSI (Taki et al. 1987), IPP (Abe et al. 1987) and HPM (Nakazaki et al. 1985). The second approach, analogous to the design of a RISC architecture, provides functional support for WAM instructions but leaves the exact coding of the WAM instruction to the compiler. This is the approach taken in the LIBRA (Mills 1988, 1989), BAM (Van Roy 1989), Pegasus (Seo and Yokota n.d.) and Toshiba's IP704 (Matoba et al. 1989). None of these architectures are general purpose processors, although the LIBRA and the BAM provide support for general arithmetic and operating system functions. For example, the LIBRA provides only integer add and subtract, shift, status manipulation and bus lock instructions to construct general arithmetic and operating system functions. The PLM, a special-purpose coprocessor, provides only **escape** instructions and delegates the majority of arithmetic and operating system support to a host processor.

Such limitations of language-specific architectures are magnified by the recent performance increases of commercial architectures, making the justification of language-specific processors and coprocessors difficult.[1] In such a context language-specific architectures must be measured not only by their raw performance, but by the cost of transferring this performance to commercial architectures. The cost of the transfer will determine how much influence language-specific architectures have on commercial architectures. Compiling techniques are the first enhancements transferred because the hardware cost is zero; no change in the commercial architecture is necessary. Hardware functions are the last enhancements to be transferred, and only those which provide the greatest performance increase for the largest number of programming languages are even considered.

This paper describes a general technique to define hardware support for high-level programming languages in existing architectures, concentrating on Prolog in particular. The hardware that results from the application of this technique has a low cost since it calls for little modification of the existing architecture, and can result in signifcant increases in performance. The approach is based on *delayed specialization*, a general characterization of dynamic program modification and conditional instruction execution. The characterization provides a common definition for conditional instructions, multi-way microcode branching, branch folding and instruction polymorphism, and as yet unexplored improvements to existing instruction sets, such as dynamic

1 Although it is possible that this is only an analogous turn of Myer and Sutherland's (1967) *wheel of reincarnation* which was first used to describe the cyclical increases and decreases in the complexity of graphics processors.

branch compression. Delayed specialization is implementation independent, and may be used with microcoded and random logic control units. It is accomplished by dividing an instruction set into *instructions* and *schemata*, which are bit strings whose function varies with context. A schema is implemented with a specialization function which translates the schema into one or more operations, typically but not necessarily including the processor status as an input to the specialization function. An example of a complex schema which compresses the branching paths of a unifier into a single cycle is the LIBRA's **unify**; an example of a simple schema is the Motorola 68000's conditional branch instruction **Bcc**.

Architectural support for delayed specialization of logic programs includes schemata for dereferencing, unification and trailing which typically execute in one clock cycle and allow the WAM's unification instructions to be expanded into in-line sequences of one to five instructions and/or schemata. A WAM emulated with instructions and schemata is space and time efficient because code for matching, binding and failure is compressed into one schema which executes in a single cycle, regardless of the number of arguments known at compile-time. Partial unification in the LIBRA exemplifies delayed specialization in a language-specific architecture, and can easily be incorporated into existing RISC and CISC architectures. To show how an existing computer architecture can be extended with delayed specialization for unification, the **load** and **store** instructions of the SPARC RISC architecture are modified, and a new schema, **unify**, is defined.

We begin with definitions. The intent of these definitions is to partition an instruction set into *instructions* and *schemata*, without being forced to express schemata in terms of instructions. Instead, schemata and instructions are expressed in terms of *normalized operations* which are functions of identical arity whose arguments are pointers to the data they use. Normalized operations are used to eliminate the possibility of instructions and schemata with varying lengths, a concept which is implementation-dependent and not relevant to the definition of delayed specialization.

Definition 1. *η is a **hardware architecture** composed of functional units (a CPU).*

Definition 2. *W is the set of all strings w over $\{0,1\}$ such that the length of $w \leq n$ where n is the number of bits composing a word for some hardware architecture.*

Definition 3. *F is a set of continuous functions $\{f_0, f_1, f_2, \ldots f_{|F|}\}$ each of arity a_j.*

Definition 4. *A is the maximum arity of F, where $A = maximum(a_0, a_1, a_2, \ldots a_{|F|})$.*

Definition 5. *D is a set of **data addresses** $\{d_0, d_1, d_2, \ldots d_{|D|}\}$.*

Definition 6. *$d_0 \in D$ is designated as the null address; i.e. d_0 specifies no data.*

Definition 7. *H is a set of **operations** $\{h_0, h_1, h_2, ..., h_{|H|}\}$ for the hardware architecture η such that $H \subseteq F \times D^M$.*

Definition 8. *σ is a sequence of operations $(h_0, h_1, h_2, ..., h_{|\sigma|})$ such that $\sigma \in H^*$.*

A data address defines the location of data to be processed by the architecture (integer, list pointer, etc), but is distinct from the data it specifies. An integer in a register is specified by some d_j which is the address of that register. An integer in a memory cell is specified by some d_j which is the address of that memory cell. Using the null specifier d_0 as a "placeholder" normalizes all operations in H (which have a fixed arity A) by mapping them to $f_j \in$ F with $a_j \leq$ A.

Definition 9. *I is a set of **instructions** $\{i_0, i_1, i_2, ..., i_{|I|}\}$ where each $i_j \in W$ and is associated with an operation from H, such that i_j defines some pair $(w_j, h_j) \in W \times H$.*

Definition 10. *e is a sequence of instructions $(i_0, i_1, i_2, ..., i_{|e|})$ such that $e \in I^*$.*

Note that while every *e* defines some sequence of operations σ, σ may exist that are not defined by any *e* because I may not be equivalent to H. The definition of I does not provide for conditional instruction execution. Conditional program execution is possible under I either by computing a branch address or by defining a sequence *e* that modifies itself, but each i_j always performs the same operation. This distinction between *instructions* that do not choose from a set of operations, and "other things" that do, is intentional. We now define the "other things" as *schemata*.

Definition 11. *Σ is an arbitrarily chosen set of sequences of operations $\{\sigma_0, \sigma_1, \sigma_2, ..., \sigma_{|\Sigma|}\}$ such that $\forall \sigma \in \Sigma$, σ is defined by $e \in I^*$.*

Definition 12. *C_τ is the **processor state**, or context, of η at time τ after executing some e or α.*

Definition 13. *S is a set of **schema** $\{s_0, s_1, s_2, ..., s_{|S|}\}$ where each $s_j \in W$, and where each s_j has a corresponding specialization function g_j which maps $(s_j, C_\tau) \rightarrow \Sigma$*

Any relation that can be expressed as a set of operations arising from one or more choices can be implemented as a schema, as long as each operation can be selected based on the the processor state C_τ (Figure 1).

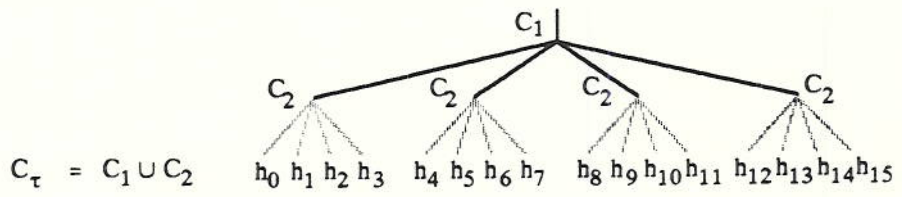

Figure 1. Branch compression with a schema

The operation selected must be distinct, i.e., not a schema, but need not necessarily be unique. The hardware architecture η must support both the operations chosen by the specialization function and the specialization function itself; if η does not provide architectural support for the specialization function then the schema is not executable on η.

Definition 14. *ϑ is an **instruction set** for η composed of instructions and schemata such that $\vartheta = I \cup S$ where $I \cap S = \emptyset$.*

Definition 15. *P is a **program** for the hardware architecture η iff $P \in \vartheta^*$. P is also represented by the sequence $(p_0, p_1, p_2, ..., p_{|P|})$ where each $p_j \in \vartheta$.*

Definition 16. *P is an **unspecialized** program iff $P \in \vartheta^*$ and $\exists p_j \in P$ such that $p_j \in S$.*

Definition 17. *P is a **specialized** program iff $P \in I^*$.*

Definition 18. *P' is a **virtual** instance of P iff*

(a) $P \in \vartheta^*$,

(b) $P' \in \vartheta^*$,

(c) $\neg \exists p'_j \in P'$ *such that* $p'_j \in S$, *and*

(d) $\forall p_j \in P \; \big[\exists e_j \in P' \; \forall C_\tau (\, (p_j \in S \Rightarrow g_j(p_j, C_\tau) \rightarrow \sigma_j) \Rightarrow e_j \text{ defines } \sigma_j \,)$
$\text{or } (\, \exists p'_j \in P' \, (p_j \notin S \Rightarrow p_j = p'_j) \,) \, \big].$

Given these definitions we now define delayed specialization.

2. DELAYED SPECIALIZATION

Definition 19. ***Delayed specialization** is the execution of a specialized virtual instance P' in place of an unspecialized program P, as a consequence of executing P on some hardware architecture η.*

P is composed of instructions and schemata. As P executes, the instructions are invariant with respect to the operations in H they perform, while schemata are specialized to operations in H as they are encountered, and perform various functions in F depending on the processor state. In one sense the program is compiled as it is run, and is described by some specialized program P' which contains only instructions. However, P' does not exist even temporarily because instructions are not substituted for schemata. Instead, schemata are dynamically translated by the associated specialization function g_j, using the processor state C_τ arising from the prior execution of instructions or schemata.

2.1 SIMILAR TECHNIQUES

Delayed specialization that selects one of *n* operations may be implemented using an *n*-way microcode branch, such as the AMD 2910 microsequencer's three-way branch (TWB) or mapped jump (JMAP) (Mick and Brick 1980). However, the hardware architecture η need not be microcoded to use delayed specialization. All that is necessary is a means of generating the specialization functions, which could be done with combinational logic, a programmable logic array or, as is the case in the LIBRA, a mapping read-only memory. Admittedly, since most architectures are now microcoded, it is natural to view delayed specialization as a form of microcoded branch, but to do so is to confuse the implementation with the principle.

Delayed specialization does not require a writable control store, since schemata do not modify the control store as the program executes. Delayed specialization may be coupled with a writable control store to implement dynamic branch compression, but this is an implementation technique.

Delayed specialization differs from partial evaluation (or partial deduction) in that the code sequences of P' are never produced, but are only executed. Although there is a program P' corresponding to the specialization of program P, P' is a virtual program; it never exists.

Delayed specialization differs from interpretation because an interpreter calls a routine for every token it encounters, while delayed specialization compiles an operation for every schema the hardware architecture η encounters. Delayed specialization is equivalent to interpretation if $\vartheta = S$, in which case the entire program is compiled, possibly for a second time, as it is run.

2.2 DELAYED SPECIALIZATION IS NOT SELF-MODIFYING CODE

Delayed specialization is not self-modifying code. A program that modifies itself leaves the modified code in place for some time after the code has executed. A program that uses self-modifying code must embed a code sequence e_j in P for each specialization function associated with a schema $s_j \in S$, and introduce a second code sequence e'_j which is altered by e_j from e'_{j_τ} to $e'_{j_{\tau+1}}$. The lifetime of e'_j is longer than its execution time, making it possible for e_j corresponding to the schema to produce an instruction sequence $e'_{j_{\tau+1}}$ after the unaltered e'_{j_τ} is fetched, either during caching or instruction pre-fetching. This will lead to errors which delayed specialization avoids because it does not separate the schema and its sequence of operations, substituting operations for schema only after the schema has been fetched by the hardware architecture η.

2.3 EXAMPLES OF DELAYED SPECIALIZATION IN COMPUTER ARCHITECTURES

Delayed specialization is commonly used in computer architectures, although no architecture includes all variations of it. The conditional branch instructions found in most computer architectures are by definition schemata. For example, a **branch if zero** instruction can be realized with a schema for which Σ = {**nop**, **goto**} and the associated specialization function g_j chooses **goto** if the zero status bit is set, and **nop** otherwise.

Branch folding in the CRISP microprocessor is an example of delayed specialization (Ditzel and McLellan 1987). All "instructions" are schemata for virtual 192-bit four-address instructions, where Σ = {**<op:inc PC>**, **<op:address>**, **<op:predictedaddress:otheraddress>**}. The specialization function g_j recognizes instances of a non-branch instruction followed by an unconditional branch instruction, and "folds" the branch into the non-branch instruction, using the branch address to construct **<op:address>**. The CRISP follows the definition of delayed specialization exactly, in that conditional branches are not translated to schema, but to a pair of branch addresses. If the conditional test fails, then the processor recovers and restarts execution using an alternate program counter.

Other examples of delayed specialization include the PLM, which translates WAM opcodes into an intermediate fixed-length instruction form with operands in pre-specified positions, and the Intel iAPX-432, which has polymorphic instructions in its instruction set. These examples demonstrate the effectiveness of delayed specialization and schemata. Next, the use of delayed specialization in the LIBRA Prolog architecture is discussed.

3. ARCHITECTURAL SUPPORT IN THE LIBRA

The LIBRA architecture has a number of features that provide architectural support for Prolog, but some, such as the organization of the individual ALUs and the internal pipelining of each functional unit are not easily transferred to other processors. However, the collection of features that provide support for delayed specialization of logic programs are sufficiently independent that they could be added to other machines, although in a restricted form.

3.1 PARTIAL UNIFICATION

Unification requires an execution sequence of five instructions if the **sub** and **switch** instructions are used to build a "tree-structured" unifier in the LIBRA (Figure 2).

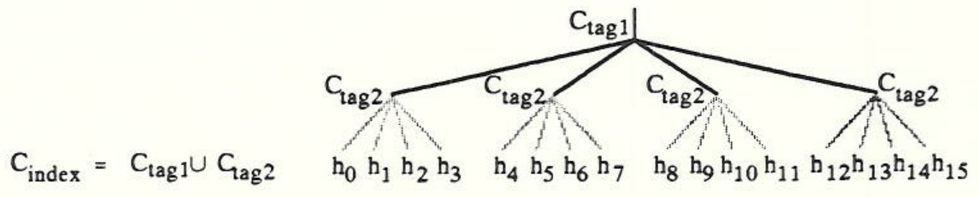
C_{tag1}
C_{tag2}
$C_{index} = C_{tag1} \cup C_{tag2}$
h_0 h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8 h_9 h_{10} h_{11} h_{12} h_{13} h_{14} h_{15}

Figure 2. Unification tree

This is because the **sub** instruction, which simply subtracts the tags, allows only for comparison of equality or inequality between tags. This does not allow the relationship between two terms to be identified with a single comparison. Examining a tree-structured unifier shows that its purpose is to reach a leaf node in the tree, at which either binding, branching or dereferencing is performed. In (Mills 1988) the relationship between the tree structured unifier and the table-driven unifier is used to develop a new "instruction", which operates in a single cycle and encodes the leaf node instructions of the unification branch tree, or *partial unification*.[2] Partial unification depends on the observation that, except for recursive unification and failure, all operations in a table-driven unifier are single instructions — and the two exceptions can be made single instructions by replacing them with subroutine calls (Figure 3).

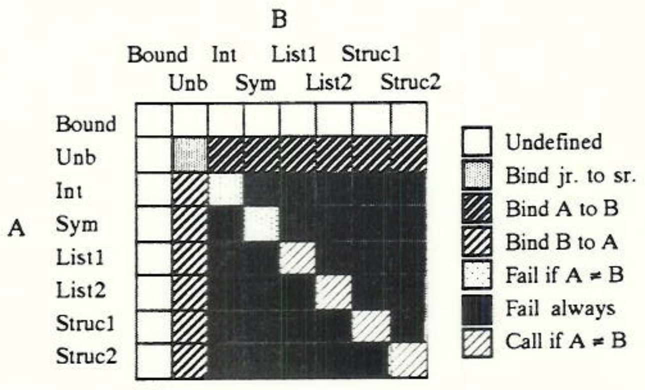

Figure 3. LIBRA partial unification

The cost of executing one of the leaf node instructions is due to the need to determine both operand types, then select the correct table entry based on these types. If the tags of the two operands are used to form the index, then the tag checking and indexing instructions can be eliminated. This is done in the LIBRA, where the look-up table is implemented with a 64 x 8 bit mapping read-only

2 Partial unification was originally viewed as an instruction; it is a schema by the definitions in this paper.

memory activated by **unify**, the partial unify schema. Partial unification detects special cases (such as two structure pointers being equal), which further reduces the frequency of branches, in this case calls to the recursive unifier. This means that **unify** avoids a pipeline break for a **switch** or a **call** instruction whenever it is possible to do so. It is critical to reduce pipeline breaks in a machine that frequently executes sequences of semantically-related short branches, and parallelizing the operations internally to select one of many non-overlapping operations with a single schema is an effective way to do it.

Before providing a detailed description of the operation of the LIBRA's **unify** schema, a block diagram of the architecture is presented. The LIBRA looks very like a commercial RISC architecture, with additional ALUs to perform tag processing and garbage-collection bit management. Functional support for trail-checking and trailing is also present. However, the support for partial unification consumes much less area than any of the other functions (Figure 4).

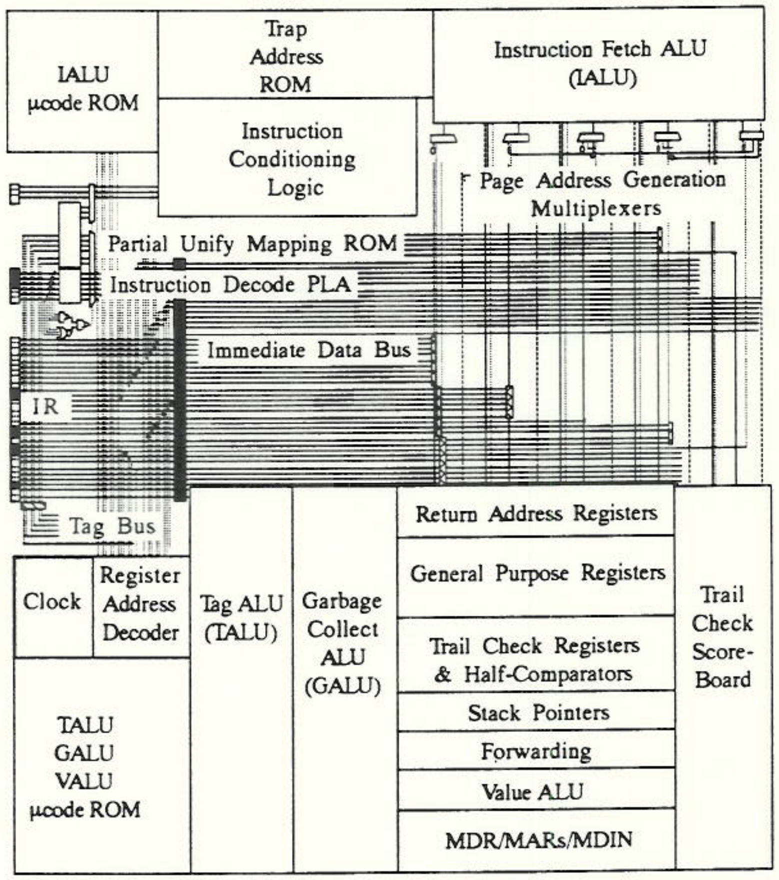

Figure 4. Functional schematic of the LIBRA

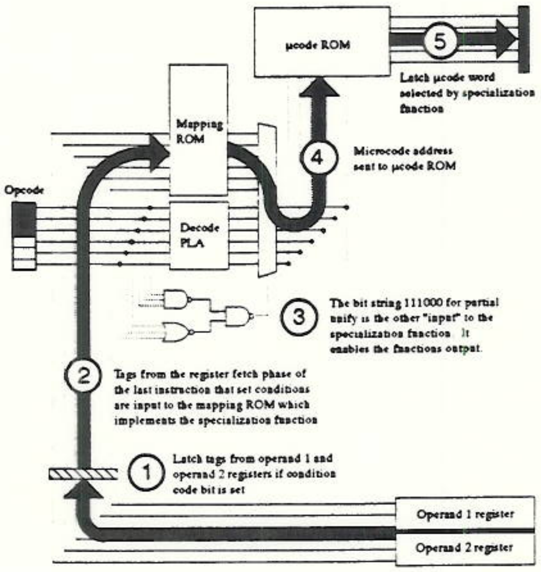
µcode ROM
5
Latch µcode word selected by specialization function
Mapping ROM
4
Microcode address sent to µcode ROM
Opcode
Decode PLA
3
The bit string 111000 for partial unify is the other "input" to the specialization function. It enables the functions output.
2
Tags from the register fetch phase of the last instruction that set conditions are input to the mapping ROM which implements the specialization function
1
Latch tags from operand 1 and operand 2 registers if condition code bit is set
Operand 1 register
Operand 2 register

Figure 5. Operation of partial unification in the LIBRA

The LIBRA's **unify** schema operates as follows (Figure 5):

1. Both operands' tags are latched if an instruction or schema operation sets condition flags, however this is practically restricted to dyadic operations such as **sub** or **add**,

2. The latched tags and the zero status flag are presented to the mapping read-only memory during the decoding of all subsequent instructions, until some instruction changes them by setting condition codes,

3. If the **unify** schema is detected, then the microcode address select multiplexer is steered to output the microcode address from the **unify** mapping read-only memory instead of the opcode mapping read-only memory,

4. The **unify** microcode address is presented to the microcode read-only memory, and

5. The microcode word selected is latched to control the subsequent execution.

At this point it is clear that partial unification is not an instruction, but a schema, with Σ = {**store**, **goto**, **nop**} and a specialization function that uses the operand tags and the machine state to select an operation from Σ. Using this insight partial unification can be further improved. In the LIBRA the partial unifier had to be protected from bound variables because no entry in the mapping read-only memory was defined for them. However, given the parallel functional units in the LIBRA, it

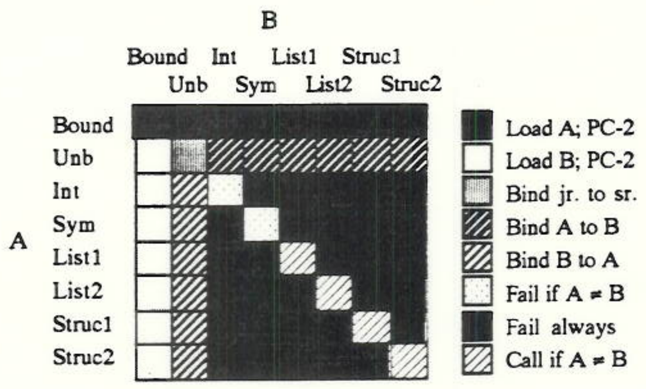

Figure 6. Modified LIBRA partial unification includes dereferencing for bound variables

is possible to add the **load** operation to Σ and obtain a partial unifier that dereferences its arguments if necessary (Figure 6).

3.2 FREQUENCY OF UNIFICATION

Partial unification does not provide a fully recursive unifier in hardware as has been suggested by (Shobatake and Aiso 1986, Woo 1985) because it is not cost-effective. Although the frequency of unification appears to call for unification hardware (Table 1), the majority of these unifications (>95%) are non-recursive (Tick 1987). Thus a minimalist approach such as partial unification is sufficient.

Table 1. Dynamic WAM Instruction Frequency for Four Prolog Implementations

	Frequency (%)			
WAM Instruction group	PLM[3]	PWAM[4]	LOGIX[5]	WAM[6]
Unification[7]	48.54	39.67	41.94	44.02
Procedure calls[8]	37.16	41.26	40.42	28.26
Backtracking[9]	14.12	15.86	17.57	26.50
(Arithmetic)				0.39

3 Dobry, Despain and Patt (1985).

4 Hermenegildo (1986). Parallel WAM.

5 Ginosar and Harsat (1987). Flat Concurrent Prolog machine.

6 Tick (1987).

7 Includes dereferencing, term and structure matching, binding and recursive unification.

8 Goal matching instructions are included in this category because they correspond to parameter passing instructions in an imperative language

9 Includes clause indexing and failure.

4. ADDING A PARTIAL UNIFICATION SCHEMA TO THE SPARC

The SPARC RISC architecture includes tagged addition and subtraction instructions (Sun Microsystems Inc. 1987) which are not useful in a Prolog implementation because they invoke an overflow trap on unaligned address references instead of providing type comparisons. Prolog implementations typically rely on fast conditional branches; the overhead of a trap, and even a subroutine call, is too costly in many cases. This analysis was suggested by Lindholm (1989), who states that the tagged operations are not used in the SPARC implementation of Quintus Prolog. However, delayed specialization can be used to make use of the SPARC's tag bits more effective.

4.1 PARTIAL UNIFICATION

Partial unification on the SPARC is even more restricted than partial unification on the LIBRA because fewer tag bits and no specialized functional units are available. However, performance gains could be realized by using the two tag bits to define the following WAM operand types:

00	bound variable
01	unbound variable
10	atom
11	structure pointer

Next define a **unify** schema for partial unification over these types (Figure 7).

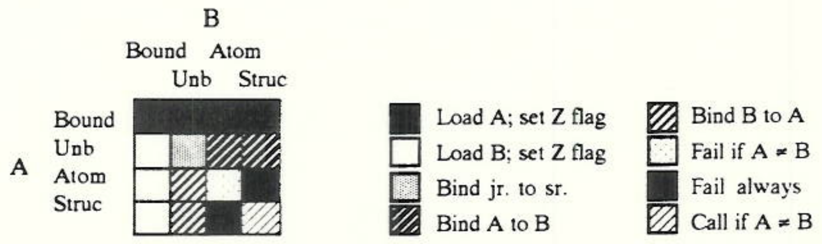

Figure 7. Partial unification for the SPARC

This definition of partial unification overloads the *atom* type with both *integer* and *symbol* types, and the *structure* type with both *list* and *structure* types. If bit 31, the sign bit, is used to differentiate between integers and symbols, and lists and structures, then partial unification succeeds in the case that the two objects unify, and fails when they do not. Recursive unification is handled by the proposed SPARC **unify** schema just as it is handled by the LIBRA, but dereferencing is handled differently. If a bound variable is encountered **unify** sets the zero flag.

This condition can be intercepted by a conditional branch following **unify**, and used to repeat the **unify** until both arguments are dereferenced. If the compiler can determine that zero or one dereferences will be sufficient, then the conditional branch can be omitted, and the instruction following **unify** will always be executed. Although emulation of WAM code using the SPARC's instruction set and the proposed **unify** schema is not presented in this paper, examples follow those of the LIBRA, which are given in Mills (1988, 1989).

The block diagram of a SPARC modified for partial unification is mostly unchanged (Figure 8). The simplicity of the modification in the diagram may not represent the actual difficulty that would be encountered in extending the SPARC's design, particularly any effects adding the tag buses would have on worst-case timing. However, if the bus is pre-charged a ratioed driver could reduce added delays.

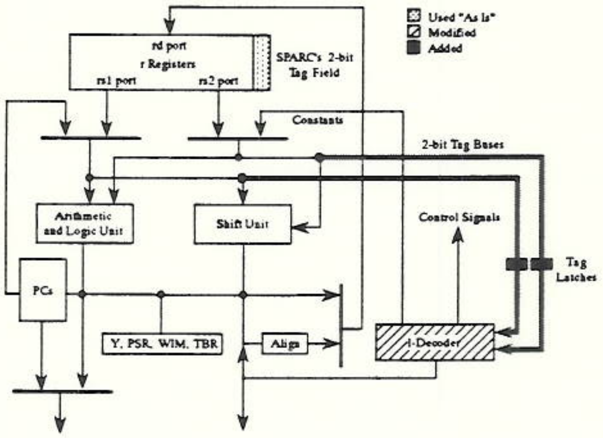

Figure 8. SPARC Integer Unit modified for partial unification

5. CONCLUSIONS AND FURTHER WORK

Delayed specialization was defined and shown to be generally useful in compressing sequences of semantically-related branches. Partial unification as defined in the LIBRA was shown to be a form of delayed specialization, and to be one of the most area efficient support functions for logic programming of the LIBRA. The generality and simplicity of partial unification was demonstrated by adding a new schema, **unify**, is added to the SPARC instruction set. A thorough analysis of the effect of the **unify** instruction on the SPARC's performance remains to be carried out, but based on an early comparison to the LIBRA, a modified SPARC should be capable of attaining 75% of the LIBRA's performance when emulating the WAM.

6. ACKNOWLEDGEMENTS

Delayed specialization was first recognized during discussions with Olivier Danvy about the relationship of partial unification to self-applicable partial evaluation and program specialization; the term "delayed specialization" is due to Danvy, but the definitions in this paper are due to Mills.

Peter Van Roy's comments on the ideas presented here led to a stronger distinction between the principle of delayed specialization and its various implementations.

REFERENCES

Abe, T., T. Bandoh, S. Yamaguchi, K. Kurosawa, and K. Kiriyama. 1987. High performance integrated Prolog processor IPP. *Proceedings of 14th Annual International Symposium on Computer Architecture*. Pittsburgh, Pennsylvania. Washington, D.C.: IEEE Computer Society Press. pp. 100-107.

Ditzel, D. R., and H. R. McLellan. 1987. Branch folding in the CRISP microprocessor: Reducing branch delay to zero. *Proceedings of 14th International Symposium on Computer Architecture*. Boston, Massachusetts. Washington, D.C.: IEEE Computer Society Press. pp. 2-9.

Dobry, T. 1987. *A high performance architecture for Prolog*. Report No. UCB/CSD 87/352. Computer Science Division (EECS), University of California, Berkeley, California.

Dobry, T., Y. Patt, and A. Despain. 1985. Performance studies of a Prolog machine architecture. *Proceedings of 12th International Symposium on Computer Architecture*. Boston, Massachusetts. Washington, D.C.: IEEE Computer Society Press. pp. 180-190.

Ginosar, R., and A. Harsat. 1987. *Profiling LOGIX: A step towards a flat concurrent Prolog processor*. Department of Electrical Engineering, Technion - Israel Institute of Technology, Haifa, Israel.

Hermenegildo, M. 1986. "An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel." Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin., Austin, Texas.

Lindholm, T. 1989. Personal communication.

Matoba, T., T. Aikawa, K. Maeda, M. Okamura, K. Minagawa, T. Takamiya, and M. Saito. 1989. Twin register architecture for an AI processor. *Proceedings of First International Tools for Artificial Intelligence Workshop*. Fairfax, Virginia. Los Alamitos, California: IEEE Computer Society Press. pp. 168-173.

Mick, J., and J. Brick. 1980. *Bit-Slice Microprocessor Design*. New York: McGraw-Hill.

Mills, J. W. 1988. "LIBRA: A high performance balanced RISC architecture for Prolog." PhD Dissertation, Arizona State University, Tempe, Arizona.

Mills, J. W. 1989. A pipelined architecture for logic programming with a complex but single-cycle instruction set. *Proceedings of IEEE 1st International Tools for Artificial Intelligence Workshop*. Fairfax, Virginia: IEEE Computer Society Press. pp. 526-533.

Myer, T. H., and I. E. Sutherland. 1967. On the design of display processors. *CACM* **11** (6): pp. 410.

Nakazaki, R., A. Konagaya, S. Habata, H. Shimazu, M. Umemura, M. Yamamoto, M. Yokota, and T. Chikayama. 1985. Design of a high-speed Prolog machine (HPM). *Proceedings of 12th Annual International Symposium on Computer Architecture*. Boston, Massachusetts. Washington, D.C.: IEEE Computer Society Press. pp. 191-197.

Seo, K., and T. Yokota. n.d. *Pegasus: A RISC processor for high-performance execution of Prolog programs*. (unpublished)

Shobatake, Y., and H. Aiso. 1986. A unification processor based on a uniformly structured cellular hardware. *Proceedings of 13th Annual International Symposium on Computer Architecture*. Tokyo, Japan. Washington, D.C.: IEEE Computer Society Press. pp. 140-148.

Sun Microsystems Inc. 1987. *The SPARC™ Architecture Manual Version 7*. Part No: 800-1399-08, Revision A. Sun Microsystems, Inc., Mountain View, California. October 22, 1988.

Taki, K., K. Nakajima, H. Nakashima, and M. Ikeda. 1987. Performance and architectural evaluation of the PSI machine. *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*. Palo Alto, California. In ACM SIGPLAN Notices **22**: pp. 128-135.

Tick, E. 1987. *Studies in Prolog architectures*. Technical Report No. CSL-TR-87-329. Computer Systems Laboratory, Stanford University, Stanford, California.

Van Roy, P. 1989. Personal communication.

Warren, D. H. D. 1983. *An abstract Prolog instruction set*. Technical Note 309. SRI International, Stanford, California.

Woo, N.-S. 1985. A hardware unification unit: Design and analysis. *Proceedings of 12th Annual International Symposium on Computer Architecture*. Boston, Massachusetts. Washington, D.C.: IEEE Computer Society Press. pp. 198-205.