TECHNICAL REPORT NO. 338

A Template Architecture for the WAM

by

Jonathan W. Mills

October 1991

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

A Template Architecture for the WAM

Jonathan Wayne Mills*
Indiana University
Bloomington, Indiana 47405

Abstract

The similarity and prevalence of Prolog unit clauses is used to develop the concept of *template programming*, where procedures are partitioned into a template and a list of differences for each clause. Code for unit clauses on RISC machines can be reduced to about twice that of the WAM using a four-address architecture to support template programming. WAM bytecode for unit clauses can be reduced approximately 40% by adding instructions for template programming to the WAM instruction set.

1. INTRODUCTION

Prolog programs compiled to native instructions for a RISC are typically three to seven times larger than the same program compiled to WAM byte code (Borriello et al. 1987, Mills 1988). In this paper the similarity and prevalence of unit clauses is used to devise a method to reduce the size of the native coded Prolog programs. The programs that benefit most from this proposal are those that have a large number of similar unit clauses where shallow backtracking is a substantial part of execution time, although non-unit clauses (i.e., rules) also can be compressed to a lesser extent with this technique. In addition, the locality of reference of the Prolog program will be increased, leading to more effective use of the instruction cache.

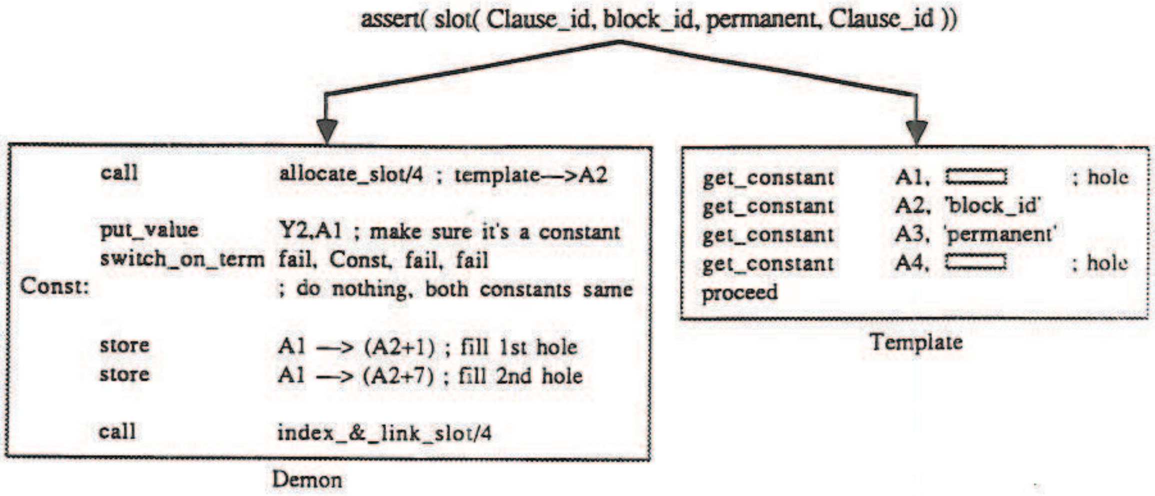
2. TEMPLATE PROGRAMMING

The method proposed is called *template programming*, which consists of dividing procedures into two parts. The first part of the procedure, the *template trace*, contains the invariant code for all clauses in the procedure. This invariant code need not necessarily be contiguous; in fact, it is expected to contain "holes" that may be as small as a single instruction. The second part of the procedure, the *difference trace*, contains the instructions to fill in the holes in the template trace. When the procedure is executed, the template trace executes repeatedly using the instructions from the difference trace to produce the same effects as executing the instruction stream for the original procedure.

The concept of template programming was suggested by earlier work with *assertive demons* to reduce the run-time overhead for **assert** and **retract** (Mills and Buettner 1988). If a demon can be created for a clause that is invariant in most of its components (such as the **slot/4** clause in Figure 1), then it is a natural next step to partition such a clause so that the template trace is present only once in the code space. Instances of the clause are then defined by the values placed in the holes of the template at run time. Fetching a template for a clause during shallow backtracking which is then executed repeatedly with the "holes" filled in with instructions from the difference trace may result in an appreciable reduction in code size.

This work was supported in part by the National Science Foundation under Grant No. MIP 90-10878.

* Lindley Hall, Department of Computer Science, (812) 855-7081, jwmills@iuvax.cs.indiana.edu



assert(slot(Clause_id, block_id, permanent, Clause_id))

```
        call            allocate_slot/4 ; template-->A2

        put_value       Y2,A1 ; make sure it's a constant
        switch_on_term  fail, Const, fail, fail
Const:                  ; do nothing, both constants same

        store           A1 --> (A2+1) ; fill 1st hole
        store           A1 --> (A2+7) ; fill 2nd hole

        call            index_&_link_slot/4
```

Demon

```
get_constant    A1, ▭          ; hole
get_constant    A2, 'block_id'
get_constant    A3, 'permanent'
get_constant    A4, ▭          ; hole
proceed
```

Template

Figure 1. Assertive demon and template

The holes that are shown in the **slot/4** clause are the values that are encoded in a WAM bytecode. However, in the native code for a RISC architecture such as the LIBRA, the values are encoded in single instructions (Figure 2).

get constant C, Ai

```
drf                  Ai  T1
add                  r0  C16      con: T2
sub               sc T1  T2  r0
unify             sc T1  T2  $+2 (no mode splitting)
if traill push+      TR  T1
```

Figure 2. Value for constant *C* encoded in `add` instruction as constant `C16` and tag `con`

A straightforward way to implement template programming is to create the template and difference traces as coroutines using **`call`** and **`return`**. To do this an initial block must use **`call`** to establish the addresses for subsequent **`returns`** in the main part of the template and difference traces (Figure 3).

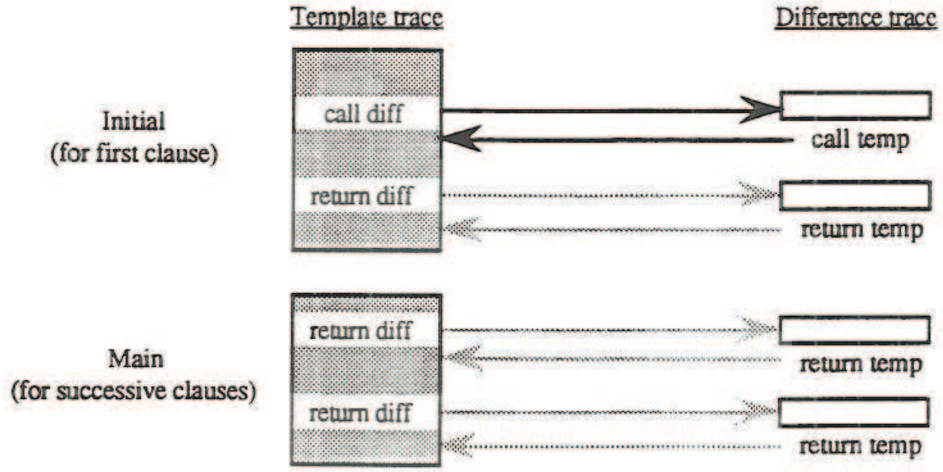

Template trace
Difference trace
Initial
(for first clause)
call diff
call temp
return diff
return temp
Main
(for successive clauses)
return diff
return temp
return diff
return temp

Figure 3. Template programming with **call** and **return**

This structure can be extended by equating the **try** clause to the initial template and the **retry** clauses to the main template, and by adding a final template for the **trust** clause of the procedure. This has the advantage of allowing the retry code to be placed into the main template instead of the difference trace.

Given the low density of RISC code compared to WAM bytecode, the overhead of template programming implemented using **call** and **return** instructions may be acceptable to reduce code size, and, for existing commercial RISCs is the only implementation technique possible. However, for Prolog RISCs such as the LIBRA, which have higher-density code, the resulting performance decrease is noticeable. In the next section a hardware solution is offered that implements the template and difference traces as instruction-level coroutines.

3. INSTRUCTION-LEVEL COROUTINING

Template programming was shown to be a form of coroutining within a clause, where a single thread of execution is decomposed into a pair of instruction traces. The template trace common to all clauses in a procedure is one coroutine. The rest of the instructions in the procedure, i.e., the difference trace, is the other coroutine. Recombining the pair of template and difference traces at execution time by executing them as coroutines generates the instruction stream for the unpartitioned procedure.

The template architecture for the WAM is an extension of the LIBRA. The first part of the extension adds a second program counter to the LIBRA processor. The only difference between the program counters is that one is selected for initial use when the LIBRA is reset, otherwise neither program counter is preferred. The second part of the extension adds a one-bit fourth address field, *next address PC select*, to each instruction. The fourth address field need not require a longer instruction word. The single-bit field could replace one of the skip condition field bits in the LIBRA instruction format (Figure 4).

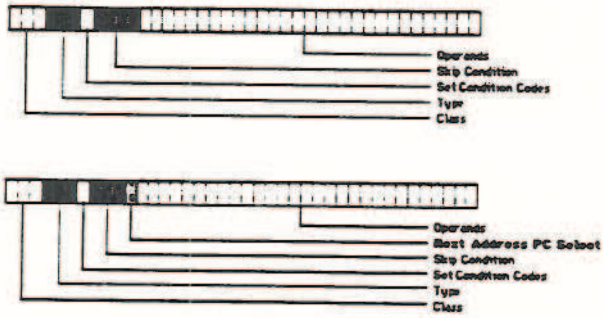


Figure 4. Original (top) and four-address (bottom) LIBRA instruction formats

The **next address PC select** field allows zero-cost zero-delay branches to be executed between any pair of instruction streams. This is possible because the fourth address field is available without decoding as soon as the instruction has been fetched. Thus, **next address PC select** can be used to steer the choice of program counter for the next instruction fetch, even in a pipelined machine (note that there is still a delay if instruction streams are switched at the same time as a branch is executed). The overhead of one **call** or **return** instruction for each difference trace instruction or code segment that fills a template hole can be avoided by switching between program counters (Figure 5).

There is no overhead for filling a hole in a template, even when only a single instruction is required. Nor is there a performance degradation, because the dual program counters address each trace independently, with each instruction selecting the program counter used for the following instruction. There is a set-up overhead required: each program

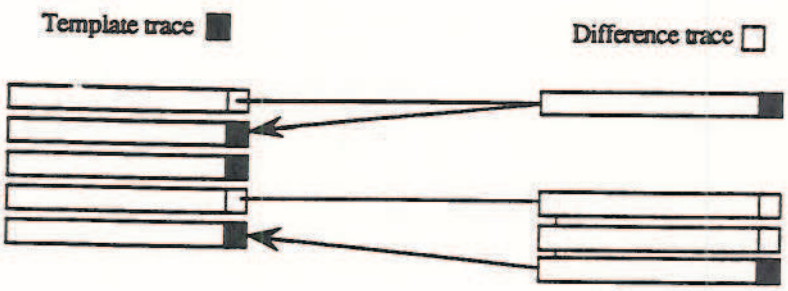


Figure 5. Using the **next address PC select** field to implement instruction-level coroutining

counter must be loaded with an address by executing an execution control instruction such as a **jmp** or **call** with the **next address PC select** field set to select the desired program counter. This also forces the subsequent branch delay slot to belong to the instruction stream that executed the branch. There is also the overhead of saving the second program counter in the choicepoint if a clause succeeds on a **try** or **retry** instruction, and restoring the second program counter if failure later occurs. However, the total overhead is only a few instructions per procedure, and is absorbed in the overall reduction of code size.

4. COMPARISON OF CODE SIZES

A simple but representative unit clause, **p/4**, will be used to compare the reduction in code size due to different implementations of template programming:

`p( a, y, z, t )`

In the clause shown, **a** is a constant common to all clauses, and **x**, **y**, and **z** are literals which represent WAM symbolic constants. Thus, the arguments have similar types, but only the first value is identical in all clauses, which is reasonable if the database is indexed on the first argument of the clause. This results in a template with three holes. Given that there are *n* clauses in the **p/4** procedure, then the general formula for code size in bits is:

$$\sum \text{bits}_{\text{try}} + n(\text{bits}_{\text{arguments}}) + \text{bits}_{\text{exit}}$$

where bits_{try} is determined from the code for **try**-family instructions, $\text{bits}_{\text{arguments}}$ from code for the arguments, typically **get** and **unify** instructions, and $\text{bits}_{\text{exit}}$ from **proceed** and **execute**. Because these divisions are not natural in a procedure to which template programming has been applied, code size is calculated by summing the number of bits in the template and difference traces:

$$\sum \text{bits}_{\text{template trace}} + \text{bits}_{\text{difference trace}}$$

The implementations compared are the WAM, the LIBRA without template programming, the LIBRA with **call** and **return** coroutining, the LIBRA with instruction-level coroutining, and a template-programming version of the WAM. To ensure that the procedure contains all **try**-family instructions, $n = 10$ will be chosen. All indexing is assumed to be done outside the block of code whose size is being determined.

```
try_family
get_constant a, A1
get_constant x_n, A2
get_constant y_n, A3
get_constant z_n, A4
proceed_or_execute
```

Using the instruction encodings for the WAM given in (Warren 1983), the code size for this representation of the clauses is 240 + 960 + 80 bits, or 1280 bits.

For the LIBRA without template programming, the WAM bytecode expands into the following instruction sequence:

```
ldhi                  LaddrHi               ;retry_me_else
add                   r0   LaddrLo   T1
st                    B    -1   T1
drf                   A1   T1               ;get_constant a, A1
add                   r0   a   con: T2
sub              sc   T1   T2   r0
unify            sc   T1   T2   S+2
if traill push+       TR   T1
drf                   A2   T1               ;get_constant xn, A2
add                   r0   xn   con: T2
sub              sc   T1   T2   r0
unify            sc   T1   T2   S+2
if traill push+       TR   T1
drf                   A3   T1               ;get_constant yn, A3
add                   r0   yn   con: T2
sub              sc   T1   T2   r0
unify            sc   T1   T2   S+2
if traill push+       TR   T1
drf                   A4   T1               ;get_constant zn, A4
add                   r0   zn   con: T2
sub              sc   T1   T2   r0
unify            sc   T1   T2   S+2
if traill push+       TR   T1
ret                   CPC                   ;proceed
```

These expansion for the WAM instructions are found in (Mills 1989). The **try_me_else** instruction (not shown) includes the register saves needed to create a choicepoint, and is thus substantially longer than the **retry_me_else** shown in the example. All instructions are 40 bits long; thus the number of bits needed for this representation of the clauses is 960 + 8000 + 400, or 9360 bits.

For the LIBRA with template programming implemented by **call** and **return** coroutining, two code sequences are generated. The main template trace is given by:

```
ldhi                  LaddrHi               ;retry_me_else
add                   r0   LaddrLo   T1
st                    B    -1   T1
drf                   A1   T1               ;get_constant a, A1
add                   r0   a   con: T2
sub              sc   T1   T2   r0
unify            sc   T1   T2   S+2
if traill push+       TR   T1
drf                   A2   T1               ;get_constant xn, A2
ret                   CP1
sub              sc   T1   T2   r0
unify            sc   T1   T2   S+2
if traill push+       TR   T1
drf                   A3   T1               ;get_constant yn, A3
ret                   CP1
sub              sc   T1   T2   r0
unify            sc   T1   T2   S+2
if traill push+       TR   T1
drf                   A4   T1               ;get_constant zn, A4
ret                   CP1
sub              sc   T1   T2   r0
unify            sc   T1   T2   S+2
if traill push+       TR   T1
ret                   CPC                   ;proceed
nop
goto                  <start of the main template trace>
```

and an example block from the difference trace is given by:

```
add                     r0   xn    con: T2
ret                     CP2
add                     r0   yn    con: T2
ret                     CP2
add                     r0   zn    con: T2
ret                     CP2
```

The initial block in the template trace must set up the choicepoint and the coroutining into the difference trace; the first block of the difference trace must establish coroutining into the template trace; and the last block in the difference trace must remove the choicepoint, and will not branch back to the template trace, but will perform the equivalent of the WAM **execute** instruction if the clause succeeds. The number of bits needed for this representation of the clauses is 1840 bits for the template trace + 2640 bits for the difference trace, or 4480 bits.

For the LIBRA with instruction-level coroutining the **call** and **return** instructions are replaced by the one-bit **next address PC select field**, indicated by a "D" if the next instruction executed comes from the difference trace (while executing an instruction from the template trace), or a "T" for the opposite case:

```
ldhi                   LaddrHi                    ;retry_me_else
add                    r0   LaddrLo   T1
st                     B    -1    T1
drf                    A1   T1                    ;get_constant a, A1
add                    r0   a   con: T2
sub              sc    T1   T2    r0
unify            sc    T1   T2    S+2
if traill push+        TR   T1
drf                    A2   T1           D        ;get_constant xn, A2
sub              sc    T1   T2    r0
unify            sc    T1   T2    S+2
if traill push+        TR   T1
drf                    A3   T1           D        ;get_constant yn, A3
sub              sc    T1   T2    r0
unify            sc    T1   T2    S+2
if traill push+        TR   T1
drf                    A4   T1           D        ;get_constant zn, A4
sub              sc    T1   T2    r0
unify            sc    T1   T2    S+2
if traill push+        TR   T1
ret                    CP2                        ;proceed
nop
goto                   <start of the main template trace>
```

and an example block from the difference trace is given by:

```
add                    r0   xn    con: T2   T
add                    r0   yn    con: T2   T
add                    r0   zn    con: T2   T
```

The number of bits needed for this representation of the clauses is 1520 bits for the template trace + 1440 bits for the difference trace, or 2960 bits.

Finally, if a template-programming version of the WAM is emulated, the already dense encoding scheme of WAM instructions is further compressed. The emulator must maintain a pointer to the difference trace, which now consists solely of the data to fill holes in the template. One argument is added to the backtracking and control instructions to select their mode of operation. This is necessary because these instructions are present only once in a template. This leads to three new WAM instructions:

set_t_pointer DTaddress, NCaddress	loads the address of the difference trace and the "next clause" address — which is always the template

try_how	uses an argument from the difference trace to perform either a **try**, **retry**, or **trust** instruction but do *not* update the next clause address
proceed_how	uses one argument from the difference trace to perform either a **proceed** or **execute** instruction

and extensions to two families of WAM instructions:

tget_family	get with an argument from the difference trace
tunify_family	unify with an argument from the difference trace

Using the new WAM instructions the p/4 procedure is split into the following template trace:

```
set_t_pointer <difference trace>, <start of main template>

try_how
get_constant a, A1
tget_constant A2
tget_constant A3
tget_constant A4
proceed_how
```

and an example block from the difference trace:

```
retry
x_n
y_n
z_n
proceed
```

Extending the instruction encodings for the WAM, the code size for this representation of the clauses is 88 bits for the template + 640 bits for the difference list, or 728 bits.

Template programming can also be applied to rules, although with less effective compression as is shown in the following set of rules from a theorem prover written in Prolog:

```
ir( min(X,Z,Z), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-
         sc( max(Y,Z,Y1), H1),
          sc( min(X,Y,X1), H2),
          sc( min(X,Y1,Z1), H3).
ir( min(X,Y,X1), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-
         sc( max(Y,Z,Y1), H1),
          sc( min(X,Z,Z), H2),
          sc( min(X,Y1,Z1), H3).
ir( min(X,Y1,Z1), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-
         sc( max(Y,Z,Y1), H1),
          sc( min(X,Y,X1), H2),
          sc( min(X,Z,Z), H3).
```

In this example the native LIBRA code for the original clauses requires 120 instructions, or 4800 bits. Using instruction-level coroutining and template programming reduces this to 34 instructions for the template and 3 x 6, or 18, difference instructions for a total of 52 instructions or 2080 bits:

```
ir( min(X,•,•), max(Z,X1,Z1), 17, [ H1,H2,H3 ] ) :-
         sc( max(Y,Z,Y1), H1),
          sc( min(X,•,•), H2),
          sc( min(X,•,•), H3).

Z,  Z,  Y,   X1,   Y1,  Z1.
Y,  X1, Z,   Z,    Y1,  Z1.
Y1, Z1, Y,   X1,   Z,   Z.
```

5. SUMMARY AND CONCLUSIONS

The reductions in code size are summarized relative to the original WAM code. In the example unit clause, even a software implementation of template programming reduced the size of the native code by a factor of two. Thus, template programming may be a useful technique to optimize the size of code generated by native-code compiler. Combining template programming with other optimization techniques, such as global analysis to remove trailing and dereferencing (Holmer et al. 1990).

	Bits	Model/WAM Ratio
WAM	1280	1.0
LIBRA	9360	7.3
LIBRA, call/return coroutining	4480	3.5
LIBRA, instruction-level coroutining	2960	2.3
WAM, template instructions	728	0.57

In all cases locality of reference is improved, which should increase the cache hit ratio. This is because the template will remain in the cache throughout shallow backtracking, while only the difference trace will be fetched. In addition, more difference trace code will execute out of the cache since this code is small. In the four-address LIBRA this advantage translates directly into a performance gain because there is no overhead once coroutining is established. Further work is needed to determine whether the overhead of **call** and **return** coroutining precludes performance advantages gained by a higher cache hit ratio.

REFERENCES

Borriello, G., A. Cherenson, P. Danzig, and M. Nelson. 1987. RISCs vs. CISCs for Prolog: A case study. *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. 136-145.

Holmer, B. K., B. Sano, M. Carlton, P. Van Roy, R. Haygood, W. R. Bush, A. M. Despain, J. M. Pendleton, and T. Dobry. 1990. Fast Prolog with an Extended General Purpose Architecture. *Proceedings of Seventeenth Annual International Symposium on Computer Architecture*. IEEE Computer Society Press.

Mills, J., and K. Buettner. 1988. Assertive Demons. *Proceedings of Fifth Joint International Conference on Logic Programming*. Seattle.

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.

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

Proceedings of the

2nd NACLP Workshop on Logic Programming Architectures and Implementations

Held as a part of NACLP90, The 1990 North American Conference on Logic Programming

November 1, 1990
Hyatt Regency Hotel
Austin, Texas

Organized by:

Jonathan W. Mills
Computer Science Department
101 Lindley Hall
Indiana University
Bloomington, Indiana 47405-4101
USA

jwmills@iuvax.cs.indiana.edu

Micha Meier
ECRC, Arabellastr. 17
8000 Munich 81
West Germany

EUROPE micha@ecrc.de
USA micha%ecrc.de@pyramid.com

Edited by Jonathan W. Mills

Proceedings not including articles Copyright ©1990 by the Association for Logic Programming
Articles Copyright ©1990 by the respective authors