Instruction Merging and Specialization in the SICStus Prolog Virtual Machine

Henrik Nässén
Swedish Institute of Computer Science, Uppsala, Sweden
henrikn@sics.se

Mats Carlsson
Swedish Institute of Computer Science, Uppsala, Sweden
matsc@sics.se

Konstantinos Sagonas
Computing Science Dept.
Uppsala University, Sweden
kostis@csd.uu.se

ABSTRACT

Wanting to improve execution speed and reduce code size of SICStus Prolog programs, we embarked on a project whose aim was to systematically investigate combination and specialization of WAM instructions. Various variants of the SICStus Prolog virtual machine instruction set were designed, implemented, and their performance was evaluated against standard benchmarks and on big Prolog programs. In this paper, we describe our methodology in finding appropriate candicates for instruction merging and specialization, discuss related trade-offs, present detailed statistics and performance measurements that we gathered, and report on our experiences from our involvement in this feat. In short, our experience is positive: the speedup of performing instruction merging and specialization in the context of the SICStus emulator is approximately 10%, while the bytecode size reduction is about 15%.

1. INTRODUCTION

Prolog can be implemented by interpretation, by compilation to and emulation of virtual machine (VM) code, or by compilation to C or directly to native code [16]. Indeed, in some implementations, many of these forms happily coexist. If the virtual machine approach is taken, performance can be enhanced by improving: the abstract instruction set, the compiler, the mapping from abstract instructions to VM instructions, the VM instruction encoding, or the emulator. In this paper, we will be focusing on improving the instruction mapping, using the two well-known transformations:

Instruction merging. A sequence of abstract instructions is replaced by a single VM instruction. This primarily reduces the overheads of emulation and also results in a more compact (byte)code representation. Additional benefits are that the emulator's code locality is enhanced and the C compiler has the chance to perform optimizations that are impossible to make across implementations of the individual instruction routines.

Instruction specialization. An abstract instruction corresponds to a set of VM instructions, all but one of which being special cases of the abstract instruction. A given abstract instruction is replaced by the most specific corresponding VM instruction. In addition to a more compact code representation, this eliminates the cost of decoding some arguments and reduces the register pressure in their implementations.

These transformations are of course not mutually exclusive. So far, both have been applied mostly in an ad hoc fashion in many Prolog systems. In this paper, we will try to systematically study them and quantify the extent of their benefits on an already high performance Prolog implementation. We first review the standard virtual machine emulator techniques.

A virtual machine emulator is implemented either as a non-threaded (bytecode) interpreter or as some form of a threaded code interpreter. We will use the term *routine* to refer to the code implementing a VM instruction. A bytecode interpreter is typically implemented in C as a `switch` statement, where each routine is a `case` in the body of the statement, and ends by fetching the next opcode (simply a byte or a small integer) and branching to the `switch`. Threaded code [3, 6][1] was first made popular by the implementation of the Forth programming language. In *direct threading*, each opcode is represented as the address of its routine. Quintus Prolog uses a space-efficient variant of this: opcodes are represented as 16-bit offsets from a base address; this limits the size of the main emulator function to 64K. In *indirect threading*, each opcode is represented as an offset into a table of such addresses.

The WAM [18] poses some additional challenges to these techniques. WAM executes in *read mode* or in *write mode*. The semantics of many WAM instructions is mode dependent, whereas others change the current mode. To avoid expensive runtime tests, the mode dependent opcodes can be linked to a read mode routine and a write mode routine. In a bytecode interpreter, this can be done either using two `switch` statements (as in [4]), or e.g. as a common one where an opcode $2*i$ refers to a read mode routine and $2*i+1$ refers to the write mode version. An indirect threaded interpreter can use a similar technique with even and odd offsets. These techniques are used in SICStus Prolog. In a direct threaded interpreter, mode dependent instructions must be equipped with two addresses, a space-wise expensive solution. YAP

[1] See also http://www.complang.tuwien.ac.at/forth/threaded-code.html.

can be configured to use this approach; see [15]. Quintus Prolog avoids this problem by letting each read mode routine be preceded by a pointer to its write mode version.

In looking closely for other similarities and differences between the SICStus and Quintus Prolog emulators, we noticed that Quintus aggressively implements instruction specialization, whereas only some instruction mergings were implemented in SICStus. We were thus interested in finding out whether instruction specialization would also pay off in the context of SICStus and whether performing instruction merging in a less ad hoc way gives additional benefits. To satisfy our curiosity, we relied on systematic instruction profiling (see e.g. [13]). To this end, a WAM-like version of the SICStus emulator was implemented and instrumented for collecting statistics about instructions and instruction-pairs. For each program in our benchmark suite, two pieces of dynamic information were collected:

- the frequency of each WAM instruction
- the frequency of each pair of WAM instructions

To the best of our knowledge, such data has not been reported in the literature. Our performance evaluation was then based on time and code size measurements on four versions of the WAM, using different combinations of the two transformations mentioned earlier.

The rest of this paper is structured as follows: The next section briefly reviews the WAM and describes how the instruction sets of SICStus and Quintus Prolog deviate from it. Section 3 discusses methodological issues in identifying appropriate candidates for instruction merging and specialization, and some of the statistics gathered in doing so; the full set can be found in [12]. In Section 4 we present the SICStus-based virtual machine instruction sets that this paper considers: one derived by performing instruction merging in an *ad hoc* way, one based on specialization and one that combines merging and specialization. Section 5 presents a detailed evaluation of the time and space performance characteristics of each instruction set. Finally, after discussing some related work (Section 6), Section 7 concludes.

2. WAM-BASED INSTRUCTION SETS FOR PROLOG IMPLEMENTATION

2.1 The WAM instruction set

The WAM [18] (see also [1] for a tutorial reconstruction) is a register-based abstract machine for executing Prolog programs. In actual implementations, WAM code acts as an intermediate language between compilation and emulation: Prolog code is first compiled to WAM instructions and then emulated. The WAM instruction set can be partitioned into groups according to instruction usage:

Get instructions, for the arguments of the head of a clause.

Put instructions, for the arguments of body goals.

Unify instructions, for the arguments of compound terms.

Control instructions, implementing recursion: `allocate`, `deallocate`, `call`, `execute`, and `proceed`.

These four groups along with the *choice* (`try`, `retry`, and `trust`), *indexing* (`switch_on_*`), and *cut*[2] instructions, comprise the basic WAM instruction set. The *choice* instructions are used for backtracking, the *cut*-handling instructions explicitly prevent all backtracking beyond a certain execution point, and the *indexing* instructions are used to speed up clause selection.

Conceptually, the operations performed by each WAM instruction are relatively disjoint; no merging of operations takes place. On the other hand, the *get*, *put* and *unify* instructions of the WAM can be seen as an attempt to specialize the unification operation based on type information. Moreover, the *indexing* instructions explicitly perform instruction specialization: the argument register to index on is hardcoded as the first one (A0)[3].

2.2 The core of the SICStus WAM

The core instruction set of the SICStus Prolog virtual machine consists of the basic WAM instructions, with the following deviations:

Support for indexing. In SICStus, clause indexing is performed as part of the procedure call operations (`call` and `execute`), which index on the first argument if the callee is of the appropriate kind, without using the *indexing* instructions of the WAM. Indexable clauses use *get* instructions specialized for matching the first argument, A0, and for the fact that A0 has the right type and principal functor. Thus, that *get* instruction often becomes a no-op on forward execution, in which case the compiler back-end arranges for it to be skipped over.

Almost no choice instructions. Taking the next alternative of a choicepoint, and removing the choicepoint if the last alternative was taken, is done as part of a general backtracking routine. A `try` instruction creates a choicepoint if multiple matching alternatives exist at a procedure call.

Inlined operations. Built-in predicates for e.g. type tests, comparisons, and arithmetic are compiled to special instructions which are denoted as `function_*` (used for arithmetic) and `builtin_*` (for everything else).

Support for conditionals. Inline tests are equipped with an "else" branch which is taken if the test fails. The compiler prefers else-branching over general backtracking. However, clauses are "flattened" by the compiler, turning any disjunctions into calls to anonymous predicates.

Support for garbage collection (GC). At procedure calls, if the heap does not have a prescribed amount of free space, it is garbage collected (as described in [2]) and/or expanded. The compiler emits `heapmargin_call` instructions that perform similar tests inline if needed. Since the WAM instruction set does not support GC, we have taken the approach to ensure that all locations reachable by the garbage collector contain valid

[2]The handling of the Prolog cut is left unspecified in the WAM.

[3]We use 0-based register numbering; 1-based numbering was used in [18]

terms. In particular, an `init` instruction has been added which initializes to unbound all permanent variables before the first `call` instruction of any clause.[4] Consequently, any `put_variable`, `get_variable` and `unify_variable` instructions that mention such variables downstream in the clause must be replaced by the corresponding `*_value` instructions. (SICStus actually uses versions of `*_value` instructions specialized for the fact that their operand is a free variable; the instructions are denoted as `*_first_value`.)

Support for coroutining. SICStus supports goals being suspended on attributed variables [9]. The heap overflow tests described in the previous item also check whether some attributed variables were recently bound, in which case the relevant suspended goals are run.

Support for floats and bignums. Such numbers are represented as "boxes" on the heap. New *get*, *put* and *unify* instructions handle them.

Support for profiling. The SICStus compiler can instrument code for profiling based on counters associated with program points, each containing an instruction which increments its counter.

The SICStus bytecode implements the above instruction set consisting of 136 opcodes in total. On 32-bit architectures, opcodes and short operands are 2 bytes, and long operands are 4 bytes. The size of the instruction set is big compared to the WAM also because instructions that take long operands exist both in an aligned and an unaligned version, to ensure that the long operands reside on word boundaries. We refer to this virtual machine instruction set that we consider as our starting point as M_0.

2.3 The Quintus WAM

The instruction set of the Quintus Prolog 3.4 virtual machine is also based on the basic WAM instruction set, including its *indexing* and *choice* instructions. Similarly to the SICStus WAM, the instruction set features inlined operations, full 32-bit integers and floats. It also supports e.g. the foreign language interface, memory access operations, and the debugger.

Quintus's support for disjunction and thus for conditionals is more ambitious than SICStus's: clauses are not flattened; instead, new *choice* instructions are emitted inline to implement the flow of control.

Quintus Prolog has a cautious garbage collector—it does not assume that all reachable locations have valid contents; therefore, no special treatment of permanent variables is done prior to the first `call` instructions in a clause. Heap overflow checks are performed like in SICStus.

Like in the SICStus WAM, the Quintus instructions come in multiples of 2 bytes, and many instructions exist in an aligned and an unaligned version. The Quintus emulator, probably influenced by the compilation technology of the time that it was written (which coincided with the development of the RISC architecture), and aiming to achieve high performance on the SPARC, assumes that the first four WAM argument registers (A0–A3) reside in dedicated machine registers whereas the rest reside in memory. Hence, all instructions mentioning argument registers have specialized versions w.r.t. A0–A3. Also, the *choice* instructions are specialized w.r.t. the arity of the predicate (for arities 0–4). As a result, the Quintus virtual machine contains many opcodes: 2174 in total. On the other hand, instruction merging is used sparingly in Quintus: only for very frequent combinations.

[4] `init` followed by `call` are combined to `firstcall` in M_0 as well as in M_1.

3. METHODOLOGY FOR PERFORMING INSTRUCTION MERGING AND SPECIALIZATION IN THE WAM

3.1 Finding candidates

To find candidate instructions for merging or specialization, we relied on instruction profiling (see e.g. [13]). To this end, the WAM-like emulator for M_0 was instrumented for collecting statistics about instructions and instruction-pairs. For each program in our benchmark suite, two pieces of information were collected:

1. For each instruction O_i, the number of times that O_i was executed. This instruction count did not take operands into account. Arguably, operands could yield useful information for specialization purposes. For example, in the SICStus bytecode, the idiom `execute(true/0)` is quite common.

2. For each possible instruction-pair (O_i, O_j), the number of times that these two instructions were executed in sequence.[5] To filter out false pairs, we let the WAM emulator simulate the execution of a `fail` instruction each time backtracking occurred. Similarly, pairs where O_i implies a branch are ignored.

When collecting these statistics, the aligned and unaligned version of an instruction were counted as the same instruction. For example, Table 1 shows collected instruction and instruction-pair counts (in millions) and frequencies for the 30 most frequently occurring instructions of M_0. The shown instructions amount to 86% of the total instructions executed and the instruction-pairs constitute 65% of M_0.

Given such information, the most frequent instructions (singles or pairs) might then be good candidates for specialization or merging, respectively. Dually, a rare instruction that is a merger or specialization of extant instructions might be a candidate for deletion. In Table 1, pair entries marked with a † denote dynamically frequent instruction pairs that cross predicate—or sometimes just clause—boundaries. We did not consider these pairs as candidates for merging.

If an instruction is chosen for specialization, it remains to be decided w.r.t. which argument value it should be specialized. In our experiment with specialization, we mimicked the Quintus WAM which specializes most instructions w.r.t. A0–A3. Many other options are of course possible.

Finally, it should be noted that although statistical information was collected for each program separately, candicates for merging or specialization were identified looking at the collected information in its entirety. This is because we wanted to avoid coming up with instruction set designs

[5] A more ambitiuous approach would have been to collect frequencies of sequences of instructions using e.g. *suffix trees* as in [8].

Instruction	Count	%	Instruction pair			Count	%
`unify_X_variable`	936	18.0	`unify_X_variable`	`unify_X_variable`		379	7.3
`put_X_value`	581	11.2	`put_X_value`	`put_X_value`		293	5.6
`execute`	503	9.7	`put_X_value`	`execute`		242	4.7
`put_Y_value`	351	6.7	`get_X_variable`	`unify_X_variable`		189	3.6
`get_X_variable`	259	5.0	`put_Y_value`	`put_Y_value`		178	3.4
`get_Y_variable`	248	4.8	`get_Y_variable`	`get_Y_variable`		171	3.3
`heapmargin_call`	233	4.5	`unify_X_variable`	`heapmargin_call`		168	3.2
`function_2`	206	4.0	`unify_X_variable`	`put_X_value`		168	3.2
`get_list`	189	3.6	`heapmargin_call`	`function_2`		160	3.1
`unify_X_local_value`	167	3.2	`execute`	`get_X_variable`	†	145	2.8
`unify_X_value`	151	2.9	`get_list`	`unify_X_local_value`		133	2.6
`get_structure`	150	2.9	`execute`	`get_list`	†	125	2.4
`proceed`	90	1.7	`function_2`	`execute`		119	2.3
`allocate`	87	1.7	`unify_X_local_value`	`unify_X_variable`		107	2.0
`firstcall`	87	1.7	`unify_X_variable`	`get_structure`		90	1.7
`deallocate`	78	1.5	`deallocate`	`execute`		78	1.5
`function_2_imm`	70	1.3	`execute`	`unify_X_variable`	†	67	1.3
`get_X_value`	70	1.3	`proceed`	`put_Y_value`	†	65	1.2
`try`	63	1.2	`put_Y_value`	`deallocate`		56	1.1
`put_structure`	57	1.1	`get_structure`	`unify_X_variable`		55	1.1
`put_Y_unsafe_value`	56	1.1	`put_structure`	`unify_X_value`		49	0.9
`cutB`	51	1.0	`unify_X_value`	`unify_X_value`		49	0.9
`fail`(opcode+inter.proc.calls)	50	1.0	`allocate`	`get_Y_variable`		44	0.8
`unify_Y_variable`	46	0.9	`put_Y_unsafe_value`	`put_Y_value`		43	0.8
`put_constant`	45	0.9	`unify_X_variable`	`get_list`		42	0.8
`put_Y_variable`	41	0.8	`execute`	`try`	†	37	0.7
`unify_void`	41	0.8	`put_Y_value`	`put_Y_unsafe_value`		36	0.7
`builtin_2`	35	0.7	`get_X_variable`	`put_X_value`		32	0.6
`builtin_2_imm`	34	0.6	`get_X_variable`	`get_X_variable`		31	0.6
`builtin_1`	30	0.6	`get_structure`	`get_X_variable`		31	0.6

Table 1: Instruction and instruction-pair counts and frequencies for the 30 most frequent occurrences in M_0.

which are solely based on the idioms of the particular benchmark programs chosen for training set. For example, we saw no reason to add the `append` instruction to the instruction set given that naive reverse was used as a benchmark program.

3.2 Pragmatics and discussion

Since this work was performed in SICStus, opportunities for merging and specialization were constrained by some prior design decisions and peculiarities of SICStus' instruction set. For example, a type of WAM instruction specialization often performed in other Prolog systems (e.g. Quintus, YAP or dProlog) is to specialize the *choice* instructions for small and thus commonly occurring arities. Because in SICStus choice point manipulation is part of the call mechanism (cf. Section 2.2), this specialization is not applicable in our setting.

Also, merging of instructions sometimes interacts with optimizations performed by the compiler. For example, in predicates manipulating lists (e.g. in `append/3`), it is commonplace for a `get_list` to be immediately followed by a pair of `unify_variable` instructions; see Figure 1(a). Some compilers (e.g. XSB's) greedily merge these three instructions into one; see Figure 1(b). In the SICStus compiler this naive merging is usually not a good idea because it prevents indexing from (dynamically) skipping over the `get_list` instruction when the first argument is bound to a list. Merging the two `unify_variable` instructions is a better option in this case; see Figure 1(c). A design decision was taken not to consider an instruction merging if this prevented some other optimization of the SICStus compiler.

Finally, note that if a compiler is highly optimizing or if it already has a big instruction set, there are less opportunities for specialization and effective merging of instructions. As a SICStus-specific example, consider the `*_variable` instructions. As mentioned in Section 2.2, because of SICStus's support for garbage collection, some of these instructions get changed by the compiler to `*_first_value` instructions. As a consequence, the frequency of `*_variable` pairs decreases, and this either results in not considering these pairs for merging or to obtaining a significantly larger instruction set after merging. The experience reported in [4] is similar.

3.3 Evaluation

To evaluate a particular version of the WAM, we ran our benchmark suite on it and measured the execution time. The experiments were conducted both on SPARC (Solaris) and Pentium (Linux) processors, to reduce any architectural bias. We also considered counting machine instructions using a hardware simulator, but a suitable tool for both architectures was not available. The time measurements were collected on versions that were compiled with optimization (`gcc -O2`) and with instruction profiling instrumentation turned off.

```
    switch_on_term V1 Lc Ll fail
V1  try_me_else V2
Lc  get_nil A0
    ...
V2  trust_me_else fail
Ll  get_list A0
    unify_variable X3
    unify_variable A0
    get_list A2
    ...
```

(a) append/3 in WAM.

```
    switch_on_term V1 Lc Ll fail
V1  try_me_else V2
Lc  get_nil A0
    ...
V2  trust_me_else fail
Ll  get_list_Xvar_Xvar A0 X3 A0
    get_list A2
    ...
```

(b) append/3 with naive merging.

```
    switch_on_term V1 Lc Ll fail
V1  try_me_else V2
Lc  get_nil A0
    ...
V2  trust_me_else fail
    get_list A0
Ll  unify2_Xvar_Xvar X3 A0
    get_list A2
    ...
```

(c) append/3 with better merging.

Figure 1: Interaction of instruction merging with indexing optimizations.

4. THREE SICSTUS PROLOG VIRTUAL MACHINE VARIANTS

4.1 M_1: SICStus Prolog 3.8 virtual machine

The SICStus Prolog 3.8 bytecode instruction set was derived from the core SICStus virtual machine instruction set described in Section 2.2 mainly by performing instruction merging. The merging was done in an *ad hoc* way, gleaning frequent combinations from manual inspection of compiled code. That is, merging was performed to sequences of instructions that are *statically* commonplace. Instruction specialization was not considered, except in cases where it brings more benefit than just avoiding to decode an operand; see Section 2.2. The following instruction sequences were merged:

- All occurrences of `allocate` or `deallocate` followed by another instruction. Consequently `allocate` and `deallocate` are not needed anymore and were removed from the instruction set.
- `init` followed by `call`. The resulting instruction is called `firstcall`.
- Multiple `put_value` or `put_unsafe_value` followed by `call` or a `lastcall` instruction (the result of merging `deallocate` + `execute`).
- Pairs of `get_variable` or `get_constant`.
- Pairs of `put_variable`, `put_value`, or `put_unsafe_value`.
- Any two consecutive `unify_variable`, `unify_value`, `unify_void`, or `unify_unsafe_value` instructions.
- A `get_constant`, `get_nil`, `get_value`, `unify_constant`, `unify_nil`, or `cutB`, followed by `proceed`.
- A `get_structure` or a `get_list` instruction that cannot be skipped by indexing (see Section 3.2) followed by two `unify_variable` instructions.
- `put_constant` followed by an inline test or arithmetic.

The resulting instruction set consists of 189 opcodes in total.

4.2 M_2: SICStus core + specialization

Starting from M_0, the M_2 virtual machine instruction set was built on ideas from the Quintus instruction set and contains 355 opcodes. All transformations are in the form of instruction specializations. Notably, Quintus Prolog specializes instructions for A0–A3, because on some architectures these are directly mapped to hardware registers. We follow this pattern in M_2.

The names chosen for specialized instructions (some are shown on Table 2; the complete set can be found at [12]) can be a bit cryptic, but "syllables" like `A0`...`A3` and `X0`...`X3` denote specialized operands whereas `AN` and `XN` denote generic operands which must be decoded. For example, the VM instruction `get_A1_variable_XN` corresponds to the WAM instruction `get_variable`(X_n, A_i) specialized for $i = 1$.

The following instructions were specialized for $0 \leq i, j \leq 3$:

- `put_void`(A_i), `put_constant`(C, A_i), `put_structure`(S, A_i), `put_nil`(A_i), `put_list`(A_i), `get_value`(Y_n, A_i), `get_constant`(C, A_i), `get_structure`(S, A_i), `get_nil`(A_i), `get_list`(A_i), `unify_void`($i + 1$), `unify_variable`(X_i), `unify_variable`(Y_i), `unify_value`(X_i), `unify_value`(Y_i), `unify_local_value`(X_i), and `unify_local_value`(Y_i)
- `get_variable`(X_j, A_i); covers `put_value` too
- `get_variable`(Y_j, A_i), `get_value`(X_j, A_i), `put_variable`(Y_j, A_i), `put_value`(Y_j, A_i), and `put_unsafe_value`(Y_j, A_i)

4.3 M_3: SICStus 3.8 + specialization

Finally, the M_3 virtual machine instruction set was derived starting from M_1 and specializing its most common opcodes, as determined by instruction profiling (the M_1 column of Table 2). It contains 244 opcodes.

The following instructions and instruction pairs were specialized:

- `get_structure`(S, A_i), `get_list`(A_i), and `unify_variable`(X_i), for $0 \leq i \leq 3$
- `get_variable`(X_j, A_i), for $0 \leq i, j \leq 3$; covers `put_value` too
- `unify_variable`(X_i) + `unify_variable`(X_j), for $0 \leq i, j \leq 3$.

M_0		M_1		M_2		M_3	
Instruction	%	Instruction	%	Instruction	%	Instruction	%
unify_X_variable	18.0	execute	11.7	unify_X_variable	10.0	execute	11.7
put_X_value	11.2	unify2_Xvar_Xvar	8.2	execute	9.7	get_Xvar_Xvar	6.9
execute	9.7	heapmargin_call	6.4	heapmargin_call	4.5	heapmargin_call	6.4
put_Y_value	6.7	put_Xval_Xval	6.2	unify_variable_X3	4.1	function_2	5.7
get_X_variable	5.0	get_X_variable	5.8	function_2	4.0	get_list	3.5
get_Y_variable	4.8	function_2	5.7	get_list	2.5	get_AN_variable_X3	3.4
heapmargin_call	4.5	get_list	5.0	get_AN_variable_X3	2.5	unify_vars_X3_XN	3.1
function_2	4.0	put_X_value	3.6	unify_X_value	2.4	unify2_Xlval_Xvar	2.9
get_list	3.6	unify2_Xlval_Xvar	2.9	unify_variable_X2	2.0	firstcall	2.4
unify_X_local_value	3.2	get_structure	2.7	proceed	1.7	lastcall	2.1
unify_X_value	2.9	firstcall	2.4	allocate	1.7	unify2_Xvar_Xvar	2.0
get_structure	2.9	lastcall	2.1	firstcall	1.7	get_A0_variable_XN	2.0
proceed	1.7	function_2_imm	1.9	get_A1_variable_XN	1.6	function_2_imm	1.9
allocate	1.7	get_Yvar_Yvar	1.8	get_A0_variable_XN	1.6	get_Yvar_Yvar	1.8
firstcall	1.7	unify_X_variable	1.8	deallocate	1.5	try	1.7
deallocate	1.5	try	1.7	get_A0_variable_X1	1.5	get_A3_variable_X2	1.6
function_2_imm	1.3	put_structure	1.6	get_A2_variable_XN	1.4	unify_vars_XN_X2	1.6
get_X_value	1.3	put_Y_value	1.4	get_A0_variable_X2	1.4	put_structure	1.6
try	1.2	get_structure_Xvar_Xvar	1.4	function_2_imm	1.3	put_Y_value	1.4
put_structure	1.1	fail	1.3	unify_local_value_X1	1.3	get_structure_Xvar_Xvar	1.4
put_Y_unsafe_value	1.1	unify2_Xval_Xval	1.3	unify_local_value_X3	1.3	fail	1.3
cutB	1.0	put_constant	1.2	unify_variable_X0	1.2	get_A1_structure	1.3
fail	1.0	get_Yfvar_Yvar	1.1	put_y_value	1.2	unify2_Xval_Xval	1.3
unify_Y_variable	0.9	get_X_value	1.1	get_A3_variable_X2	1.2	put_constant	1.2
put_constant	0.9	proceed	1.0	get_A1_variable_X3	1.1	get_Yfvar_Yvar	1.2
put_Y_variable	0.8	builtin_2	1.0	get_A3_variable_XN	1.1	get_X_value	1.1
unify_void	0.8	put_Y_variable	0.9	fail	1.1	proceed	1.0
builtin_2	0.7	builtin_2_imm	0.9	get_Y_variable	1.0	builtin_2	1.0
builtin_2_imm	0.6	get_Y_variable	0.9	try	1.0	put_Y_variable	0.9
builtin_1	0.6	cutB	0.9	get_structure	1.0	builtin_2_imm	0.9
				cutB	1.0		
Total %	**86**	**Total %**	**86**	**Total %**	**71**	**Total %**	**77**

Table 2: The 30 most frequent instructions for the four different virtual machine instruction sets.

4.4 Instruction frequencies in different machines

Table 2 shows the 30 most frequent instructions for each of the four different virtual machine instruction sets as well as the percentage of their frequencies. It is worth noticing that execute has such a dominating role in M_1, M_2 and M_3. The higher percentage of execute in M_1 and M_3 compared to M_0 is due to the extensive instruction merging that takes place in M_1 and M_3 which decreases the total instruction count. In addition, there are few opportunities for execute to merge with other instructions, compared to other common instructions.

Also, notice the interactions of merging and specialization. For example, in M_1, unify2_Xvar_Xvar is the second most frequent instruction; its percentage drops dramatically in M_3 because of the introduction of instructions unify_vars_X3_XN, unify_vars_XN_X2, and others not shown in the table.

5. PERFORMANCE MEASUREMENTS

5.1 The setting and the programs

As mentioned, we conducted our experiments on two platforms. The first was a SUN UltraSPARC multiprocessor machine (8 processors at 248 MHz), running Solaris 2.7. The second was an Intel x86 machine (i686 dual processor Celeron 600 MHz), running Red Hat Linux release 6.1. In the sequel we refer to these platforms simply as SPARC and x86.

Currently, there is no agreed upon set of reasonably big-sized programs that can be used by the Prolog implementation community as a benchmark set. A related difficulty in coming up with such a benchmark set is that even when some big "real-world" Prolog application from a user is available, it is not always clear what part of the application can be used as a "portable" benchmark or what its performance properties are. Thus, many implementors still resort almost exclusively to the slightly outdated but "standard" Aquarius benchmark suite [17]; for recent such cases see [15, 5, 7, 4]. Another advantage of this testsuite is that the benchmark programs and most of their properties are by now familiar to implementors and since each of the programs usually spends most of its time on a relatively small set of WAM instructions or Prolog builtins, tracing the results to the source code is relatively easy. On the other hand, it is questionable whether these small programs accurately represent the behavior of real Prolog applications since they tend to e.g.

require no garbage collection or shadow caching effects and do not provide a clear picture of how compiler or emulator optimizations scale.

Despite our reservations, we decided not to completely depart from this benchmarking tradition because we want our results to make sense to and be reproducible by other implementors. However, to get an indication of the effect of instruction merging and specialization in real programs, in addition to most of the programs of the Aquarius benchmark suite, we also used the following set of big Prolog programs:

FSA: A Finite State Automata toolbox which is a collection of utilities to manipulate regular expressions, finite-state automata and finite-state transducers. From the standard FSA tests, we used test1 and test3. More information on these utilities and their sources can be found at: http://odur.let.rug.nl/~vannoord/fsa/fsa.html.

Aquarius compiler: Compilation of the BAM (Berkeley Abstract Machine [17]) compiler and a somewhat I/O related test-run on it. Because of reads and writes to files, time measurements partly depend on the speed of performing I/O, which is not what we seek to investigate here. Despite this, we chose to keep the benchmark unmodified so as to provide a range of programs which is broader and closer to "real-world".

SICStus compiler: This benchmark consists in compiling the SICStus 3.8 Prolog compiler by itself.

XSB compiler: The benchmark consists of a compilation of (a relatively stripped down version of) the XSB compiler by itself. This benchmark was also used in [5].

Each of the big programs was run for only one iteration. In contrast, the programs of the Aquarius testsuite were run for a number of iterations so as to get reasonably accurate times and so that each program contributes approximately the same to the total execution time; see e.g. Table 3. The complete set of benchmark programs we used in this paper are available at http://www.csd.uu.se/~kostis/PrologBenchmarks.

5.2 Time performance

To obtain the performance data reported, a pre-compiled bytecode file of each of the benchmark programs was loaded on a fresh version of the SICStus Prolog interpreter and was then run for a number of times, typically 6. Especially on the SPARC, some of the benchmarks showed high variance in execution times between runs. We always report the smallest time value obtained.

Tables 3–6 present execution times (both totals and of each benchmark separately) for each virtual machine instruction set considered. For each machine the absolute time values are shown in milliseconds and for machines M_1–M_3 the relative value compared to M_0 is also presented.

As mentioned, instruction merging results in fewer instruction dispatches during a program's execution which in turn is expected to result in decreased execution time proportional to the reduction in number of executed instructions. Similarly, instruction specialization is expected to result in speedup because of avoidance of argument decoding. In practice, the performance is sometimes also affected by caching effects or C compiler features, and the obtained picture might not be so clear. To some extent, we experienced this phenomenon in our benchmarking. So as to get a less blurred picture we decided to measure and report the performance of various different configurations of the four virtual machine instruction sets.

Tables 3 and 5 present execution times on the SPARC and on the x86 respectively using the indirect threaded (i.e., jump-table based) implementation of the SICStus emulator. The results do not always show the same picture across platforms, but the following conclusions can safely be drawn:

1. Instruction merging, either as in M_1 or as in M_3, pays off in the context of SICStus virtual machine instruction set. On a threaded emulator, it gives on the average a speedup of 8–10%.

2. Instruction specialization alone did not yield any speedup except in a few cases, contrary to expectations.

3. Execution time wise, instruction merging is more beneficial than specialization.

4. Combining instruction merging with specialization is a good idea, but the additional speedup over performing only merging, is usually not big.

We also disabled threading and the corresponding results are shown in Tables 4 and 6. We postpone their analysis until Section 5.5.

5.3 Space performance

By allowing a more compact representation, instruction merging and specialization also decrease the size of the generated bytecode. In Table 7 the impact on the bytecode size for the benchmark programs is shown. Contrary to timing measurements, space measurements are precise and independent of the platform or the emulator implementation. Here, conclusions can be drawn with complete confidence:

1. Bytecode size wise, performing aggressive specialization pays off; the more the specializations that are used, the smaller the size of the bytecode file that the compiler generates. M_2 is the clear winner here and the space savings are in the order of 8–16%.

2. Instruction merging alone is also beneficial and on the average it results in a bytecode size reduction of approximately 9%.

If we consider the effects of performing instruction merging and specialization both in execution time and in bytecode size, it is clear that M_3 is the overall winner.

5.4 Emulator sizes

As mentioned above, aggressive specialization reduces the sizes of the generated bytecode files, but on the other hand, by increasing the number of opcodes, the underlying virtual machine emulator becomes bigger in size. To complete the picture, we also report on the effect that instruction merging and specialization have on the size of the emulator. Rather than simply using the size of the SICStus executable as a measure, we considered as more appropriate to measure directly the size of the main emulator function. We thus used the `nm` UNIX command to obtain the exact size of the `wam()` function in the SICStus emulator object file `wam.o` for each of the four virtual machine instruction sets. The optimized versions (without debugging information) of the `wam.o` files

Benchmark	Iterations	M_0	M_1	$\%M_0$	M_2	$\%M_0$	M_3	$\%M_0$
boyer	10	4910	4810	.979	4880	.994	4840	.985
browse	5	3510	3110	.886	3490	.994	3060	.871
chat_parser	40	4620	4270	.924	4480	.970	4150	.898
crypt	1200	3560	3500	.983	3700	1.04	3610	1.01
deriv	50000	4600	4130	.897	4670	1.02	4260	.926
fast_mu	5000	4430	4110	.927	4670	1.05	4410	.995
flatten	8000	4480	4350	.971	4910	1.10	4480	1.00
meta_qsort	1000	4370	4190	.958	4280	.979	3960	.906
mu	6000	4020	3940	.980	4150	1.03	4100	1.01
nand	250	4640	4410	.950	4690	1.01	4540	.978
nrev	15000	4150	3540	.853	4010	.966	3410	.821
poly_10	100	3830	3280	.856	4120	1.08	3310	.864
prover	5000	4010	3780	.942	4050	1.01	3870	.965
qsort	8000	4370	3920	.897	3990	.913	3620	.828
queens8	100	4450	4130	.928	4650	1.04	4010	.901
query	1500	4600	4540	.987	5160	1.12	4530	.984
reducer	200	5700	5220	.915	5390	.946	4860	.852
sdda	13000	4090	4050	.990	4460	1.09	4040	.987
sendmore	60	3790	3930	1.03	4210	1.11	4010	1.05
serialise	14000	5160	4720	.933	4610	.893	4360	.845
simple_analyser	250	4190	3910	.933	4510	1.08	4070	.971
tak	40	4580	4060	.886	4930	1.08	4440	.969
unify	2500	4180	3780	.904	4010	.959	3840	.918
zebra	150	4070	4080	1.00	4320	1.06	4230	1.03
Total in benchmarks		**104310**	**97760**	**.937**	**106340**	**1.02**	**98010**	**.939**
FSA I	1	25570	22930	.897	26050	1.02	23380	.914
FSA III	1	366970	330070	.899	382490	1.04	339800	.926
Aquarius compiler	1	131500	128380	.976	141290	1.07	129330	.983
SICStus compiler	1	5700	5470	.959	6140	1.08	5630	.987
XSB compiler	1	10560	10200	.965	11010	1.04	10200	.965
Total in real programs		**540300**	**497050**	**.920**	**566980**	**1.05**	**508340**	**.940**

Table 3: Execution times in milliseconds, on the SPARC, for the different virtual machine instruction sets.

Benchmark	M_0	M_1	$\%M_0$	M_2	$\%M_0$	M_3	$\%M_0$
Total in benchmarks	**123620**	**117280**	**.949**	**147290**	**1.19**	**115190**	**.931**
FSA I	28660	26170	.913	29310	1.02	26940	.940
FSA III	441880	388330	.879	446590	1.01	386480	.875
Aquarius compiler	152350	143720	.943	161930	1.06	147770	.970
SICStus compiler	6770	6010	.882	7500	1.11	6100	.901
XSB compiler	13770	11370	.826	13000	.944	11350	.881
Total in real programs	**643430**	**575600**	**.895**	**658330**	**1.02**	**578640**	**.899**

Table 4: Execution times, on the SPARC, for the different VM instruction sets. Here threading is disabled.

were used and threading was enabled. It should be mentioned that some other parts of the SICStus emulator (e.g. the loader or the dissassembler) slightly differ in size when the number of opcodes is changed, but these differences are not of interest for this work as they have no effect on the speed of execution.

The main emulator function has different sizes in different platforms; see Table 8 which shows its size in bytes. As expected, there is a clear correlation between the size of the main emulator loop and the number of opcodes in the virtual machine. Also note that the size of the main emulator function is kept small enough (less than 64 Kbytes) so as to fit in the cache of the architectures considered.

Given the number of opcodes in each virtual machine and the size of the corresponding routine, the average number of bytes required to implement an instruction can be calculated. This number is shown in Table 9. Again, results are as expected: the M_2 machine which contains a big number of opcodes (in the form of instruction specializations) requires the lowest number of machine instructions for the

Machine	M_0	M_1	M_2	M_3
Number of opcodes	136	189	355	244
`wam()` size on SPARC	17148	23196	32452	28248
`wam()` size on x86	18316	23548	33632	31240

Table 8: Sizes of the main emulator function in bytes

Benchmark	Iterations	M_0	M_1	$\%M_0$	M_2	$\%M_0$	M_3	$\%M_0$
boyer	10	2090	1780	.852	2010	.961	1880	.900
browse	5	1340	1190	.888	1260	.940	930	.694
chat_parser	40	2090	2100	1.00	2310	1.11	2140	1.02
crypt	1200	1430	1450	1.01	1480	1.03	1430	1.00
deriv	50000	1810	1610	.890	1860	1.03	1630	.901
fast_mu	5000	2000	1830	.915	2080	1.04	1940	.970
flatten	8000	2040	1990	.975	2350	1.15	2100	1.02
meta_qsort	1000	1740	1670	.960	1790	1.03	1680	.966
mu	6000	1620	1330	.821	1370	.846	1320	.815
nand	250	2040	1960	.961	2220	1.09	1920	.941
nrev	15000	1490	1100	.738	1160	.779	990	.664
poly_10	100	1590	1340	.843	1460	.918	1340	.843
prover	5000	1870	1790	.957	1970	1.05	1710	.914
qsort	8000	1580	1260	.797	1380	.873	1300	.823
queens8	100	1680	1570	.935	1600	.952	1720	1.02
query	1500	1790	1670	.933	1800	1.01	1730	.966
reducer	200	2320	2200	.948	2380	1.03	2250	.970
sdda	13000	1940	1920	.990	2270	1.17	2020	1.04
sendmore	60	1790	1480	.827	1440	.804	1440	.804
serialise	14000	1950	1670	.856	1960	1.01	1810	.928
simple_analyser	250	1930	1880	.974	2280	1.18	2000	1.03
tak	40	1900	1870	.984	1870	.984	1870	.984
unify	2500	1760	1630	.926	1840	1.05	1670	.949
zebra	150	1830	1870	1.02	1910	1.04	1840	1.00
Total in benchmarks		**43620**	**40160**	**.920**	**44050**	**1.01**	**40660**	**.932**
FSA I	1	11660	11110	.953	12080	1.04	11340	.973
FSA III	1	160620	144780	.901	158930	.989	141380	.880
Aquarius compiler	1	59750	59700	1.00	68620	1.15	59070	.989
SICStus compiler	1	2950	2870	.973	3320	1.13	2950	1.00
XSB compiler	1	4920	4560	.927	4950	1.01	4560	.927
Total in real programs		**239900**	**223020**	**.929**	**247900**	**1.03**	**219300**	**.914**

Table 5: Execution times in millisecs, on the x86 machine, for the different virtual machine instruction sets.

Benchmark	M_0	M_1	$\%M_0$	M_2	$\%M_0$	M_3	$\%M_0$
Total in benchmarks	**53760**	**51260**	**.953**	**60650**	**1.13**	**51230**	**.952**
FSA I	12950	13060	1.01	15140	1.17	13140	1.02
FSA III	191920	182570	.951	221620	1.15	180720	.942
Aquarius compiler	66310	70060	1.05	80650	1.22	69810	1.05
SICStus compiler	3230	3230	1.00	3910	1.21	3230	1.00
XSB compiler	5730	5700	.995	6540	1.14	5580	.974
Total in real programs	**280140**	**274620**	**.980**	**327860**	**1.17**	**272480**	**.972**

Table 6: Execution times in milliseconds, on the x86, for the different virtual machines. Threading is disabled.

Machine	M_0	M_1	M_2	M_3
Size per VM instruction on SPARC	126	123	91	116
Size per VM instruction on x86	135	125	95	128

Table 9: Average VM instruction size in bytes.

implementation of each opcode; M_3 which also contains specializations comes second. The low number for M_2 suggests that quite a few of the specialized opcodes consist of only a very small number of machine instructions. This in turn means that for these opcodes, the cost of the instruction dispatch becomes a bottleneck.

5.5 Threading vs. no threading

In a non-threaded emulator, the additional overhead over a threaded one is primarily due to the following reasons:

Instruction dispatch cost. A non-threaded emulator implements the dispatch with a general switch statement whereas a threaded one implements it with an indirect jump, e.g. using program labels as values which is allowed by the GNU C compiler. The former case needs more machine instructions, including an unavoidable range check.

Branch prediction misses. The non-threaded emulator is based on a central switch statement, and every routine ends with a branch to this switch code; since

Benchmark	**M_0**	**M_1**	**%M_0**	**M_2**	**%M_0**	**M_3**	**%M_0**
boyer	12792	11976	.936	11272	.881	11320	.885
browse	3384	3096	.915	2904	.858	3048	.901
chat_parser	38992	35392	.908	34320	.880	35056	.899
crypt	2056	1944	.946	1952	.949	1912	.930
deriv	1624	1520	.936	1408	.867	1472	.906
fast_mu	1744	1624	.931	1600	.917	1608	.922
flatten	4832	4352	.901	4008	.829	4208	.871
meta_qsort	2312	2160	.934	2096	.875	2096	.907
mu	1040	1016	.977	904	.907	960	.923
nand	21792	19680	.903	19288	.885	19368	.889
nrev	512	504	.984	472	.922	496	.969
poly_10	2968	2736	.922	2504	.844	2632	.887
prover	3464	3112	.898	3096	.894	3064	.885
qsort	792	768	.970	744	.939	752	.949
queens8	752	696	.926	656	.872	680	.904
query	2464	2448	.994	2400	.974	2448	.994
reducer	10296	9512	.924	8824	.857	9176	.891
sdda	7512	6904	.919	6344	.845	6712	.894
sendmore	1928	1768	.917	1680	.871	1768	.917
serialise	1240	1128	.910	1032	.832	1096	.884
simple_analyser	13912	12720	.914	12152	.873	12504	.899
tak	408	392	.961	344	.843	384	.941
unify	8456	7720	.913	7136	.844	7568	.895
zebra	1432	1288	.899	1048	.732	1232	.860
Total in benchmarks	**146704**	**134456**	**.917**	**128184**	**.874**	**131560**	**.897**
FSA	212896	200848	.943	195408	.918	198488	.932
Aquarius compiler	38768	34744	.896	33376	.861	34272	.884
SICStus compiler	194488	175432	.902	163280	.840	170608	.877
XSB compiler	138912	124408	.896	117936	.849	121824	.877
Total in real programs	**585064**	**535432**	**.915**	**510000**	**.872**	**525192**	**.897**

Table 7: Bytecode size of programs in the benchmark suite for the different virtual machine instruction sets.

this is a jump to a known target, it can be executed quite efficiently. However, the branch prediction hit rate of the switch code is likely to be near zero. On the other hand, in a threaded emulator, where each routine branches directly to some other routine, the branch prediction hit rate can be quite high. The miss penalty is highly architecture dependent.

Loss of code locality. The central switch code of the non-threaded emulator is a "hot spot" which increases the pressure on the instruction cache.

Tables 4 and 6 show execution times for the non-threaded emulator on the SPARC and the x86, respectively. Due to space limitations, we only present times for the big programs and for the Aquarius benchmark suite as a whole; the complete set of results can be found in [12]. We expect that instruction merging pays off even more in a non-threaded emulator than in a threaded one, due to the higher instruction dispatch cost. On the SPARC, this expectation is confirmed: instruction merging gives an additional speedup of 2–4% while times for M_2 which is based on specialization show a slowdown (cf. Tables 3 and 4). Contrary to expectations, on the x86, the situation for M_1 and M_3 is reversed; see Tables 5 and 6. To find out the reason, we looked at the generated assembly code for the emulators.

For the non-threaded SICStus emulator, we observed that `gcc` version 2.95.2 tries to generate *shared* code for tails of routines as far as possible. This phenomenon was *never* observed in the threaded emulator. The reason for this is unknown; perhaps the `gcc` idiom used for the dispatch disables this form of code sharing. Code sharing implies more compact code, but also more branches and loss of code locality.

Figure 2 displays the x86 assembly code of a threaded and a non-threaded version of an opcode routine. The extra branch in the non-threaded version is due to code sharing.

Presumably, branch prediction misses and/or loss of code locality have a high but difficult to quantify penalty on the x86. Perhaps these overheads outweigh the savings in instruction dispatch cost on M_1 and M_3.

6. RELATED WORK

In bytecode interpreted implementations of programming languages, instruction profiling is a well-known technique for determining instructions as candicates for merging or specialization; see e.g. the BrouHaHa Smalltalk system [11] and the Objective Caml implementation [10]. Quite recently, in [13], this profiling-based technique is extended to creating merged instructions at *runtime* which allows the set of common sequences to be nearly optimal for the particular application being run. This is similar to finding *superoperators* [14] in the context of an ANSI C bytecode interpreter, although the latter technique relies on a compile-time anal-

```
wam+5344:   mov      0x1e20(%esp,1),%ecx
wam+5351:   movzwl   0x0(%ebp),%eax
wam+5355:   add      $0x4,%ebp
wam+5358:   mov      0xcc(%ecx),%edx
wam+5364:   mov      %edx,(%eax,%ecx,1)
wam+5367:   movzwl   0xfffffffe(%ebp),%eax
wam+5371:   jmp      *0x10b4(%esp,%eax,1)
```

(a) A threaded x86 routine.

```
 wam+8288:   mov      0x10b0(%esp,1),%ecx
 wam+8295:   movzwl   0x0(%ebp),%edx
 wam+8299:   add      $0x4,%ebp
 wam+8302:   mov      0xcc(%ecx),%eax
 wam+8308:   jmp      wam+25580
             /* shared tail of routine */
wam+25580:   mov      %eax,(%edx,%ecx,1)
wam+25583:   movzwl   0xfffffffe(%ebp),%eax
wam+25587:   jmp      wam+3897
             /* central switch code */
 wam+3897:   cmp      $0x355,%eax
 wam+3902:   ja       wam+35096
 wam+3908:   mov      %ebx,%edx
 wam+3910:   sub      0xfffa9aac(%ebx,%eax,4),%edx
 wam+3917:   jmp      *%edx
```

(b) A non-threaded x86 routine.

Figure 2: Threaded and non-threaded assembly code versions of a routine produced by gcc on the x86.

ysis which, looking at parse tree nodes, chooses instructions as likely candicates for forming superoperators which are then implemented as new bytecodes.

In the Prolog implementation community, both instruction merging and specialization have been used in an ad hoc way in various Prolog systems. As mentioned, Quintus Prolog aggressively implements instruction specialization. Also, as reported in [4], B-Prolog relies on instruction merging for some of its speed. However, to the best of our knowledge, descriptions of the instruction sets of these systems (or of SICStus) have never been published. On the other hand, the relatively few ad hoc instruction mergings which are implemented in YAP are presented in [15, Section 4], and instruction mergings and specializations implemented in dProlog 1.0 can be found in [5, Section 6.2].

7. CONCLUDING REMARKS

Partly wanting to improve execution speed and reduce code size of SICStus Prolog programs, and partly to satisfy our curiosity on how much instruction mapping transformations pay off in the context of an industrial-quality Prolog implementation, we embarked on a project which systematically investigated the issues that are involved. This paper documents our experiences and reports on the obtained results.

In short, given that the Prolog system on which we conducted our experiment was already high-performance and relatively well-tuned, our overall recommendation is positive: with relatively little implementation effort, systematic profile-based instruction merging and specialization pays off both in time and in space in the WAM. Speed-wise, performing instruction merging is more worthwhile than specialization; especially since it creates bigger instruction routines which can then be better optimized by the C compiler. Also, in modern architectures and given the current C compilers, threading considerably improves an emulator's performance. Finally, instruction specialization, although quite effective space-wise, does not necessarily result in speedup when performed very aggressively or in conjunction with extensive instruction merging.

8. ACKNOWLEDGEMENTS

We thank Per Mildner for his comments on an earlier version of this paper. This research was partly supported by grant # 221-99-555 from the Swedish Research Council for Engineering Sciences (TFR).

9. REFERENCES

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.

[2] K. Appelby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Commun. ACM*, 31(6):719–741, Feb. 1988.

[3] J. R. Bell. Threaded code. *Commun. ACM*, 16(8):370–373, June 1973.

[4] B. Demoen and P.-L. Nguyen. On the impact of argument passing on the performance of the WAM and B-Prolog. Technical Report CW 300, K.U. Leuven, 2000.

[5] B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd et al., editors, *Proceedings of Computational Logic — CL-2000*, number 1861 in LNAI, pages 1240–1254. Springer, July 2000.

[6] M. A. Ertl. Stack caching for interpreters. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 315–327. ACM Press, June 1995.

[7] M. A. Ertl and D. Gregg. Hardware support for efficient interpreters: Fast indirect branches. Unpublished manuscript, 2000.

[8] C. W. Fraser, E. W. Myers, and A. L. Wendt. Analysing and compressing assembly code. In *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, pages 117–121. ACM, June 1984.

[9] C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on PLILP*, number 631 in LNCS, pages 260–268. Springer-Verlag, Aug. 1992.

[10] X. Leroy et al. *The Objective Caml system release 3.01*. INRIA, Mar. 2001. See also http://caml.inria.fr/ocaml/.

[11] E. Miranda. BrouHaHa—a portable Smalltalk interpreter. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, pages 354–365. ACM Press, Oct. 1987.

[12] H. Nässén. Optimizing the SICStus Prolog virtual machine instruction set. Master's thesis, Computing Science Department, Uppsala University, Mar. 2001.

[13] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 291–300. ACM Press, June 1998.

[14] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332. ACM Press, Jan. 1995.

[15] V. Santos Costa. Optimising bytecode emulation for Prolog. In G. Nadathur, editor, *Principles and Practice of Declarative Programming: Proceedings of PPDP'99*, number 1702 in LNCS, pages 261–267. Springer-Verlag, Sept./Oct. 1999.

[16] P. Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming*, 19/20:385–441, May/July 1994.

[17] P. Van Roy and A. M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1):54–68, Jan. 1992.

[18] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.