Comparing Tag Scheme Variations Using an Abstract Machine Generator[*]

José F. Morales
Facultad de Informática
U. Complutense de Madrid
jfmc@fdi.ucm.es

Manuel Carro
Facultad de Informática
U. Politécnica de Madrid
mcarro@fi.upm.es

Manuel Hermenegildo
Facultad de Informática
U. Politécnica de Madrid
and IMDEA-Software
herme@fi.upm.es

ABSTRACT

In this paper we study, in the context of a WAM-based abstract machine for Prolog, how variations in the encoding of type information in *tagged words* and in their associated basic operations impact performance and memory usage. We use a high-level language to specify encodings and the associated operations. An automatic generator constructs both the abstract machine using this encoding and the associated Prolog-to-bytecode compiler. *Annotations* in this language make it possible to impose constraints on the final representation of tagged words, such as the effectively addressable space (fixing, for example, the word size of the target processor / architecture), the layout of the tag and value bits inside the tagged word, and how the basic operations are implemented. We evaluate a large number of combinations of the different parameters in two scenarios: a) trying to obtain an optimal general-purpose abstract machine and b) automatically generating a specially-tuned abstract machine for a particular program. We conclude that we are able to automatically generate code featuring all the optimizations present in a hand-written, highly-optimized abstract machine and we can also obtain emulators with larger addressable space and better performance.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Compilers, optimization, code generation, translator writing systems and compiler generators; D.1.6 [**Programming Techniques**]: Logic programming; D.3.3 [**Language Constructs and Features**]: Dynamic storage management

General Terms

Languages, performance

Keywords

Warren's abstract machine, Prolog, low-level representation optimization, performance, compilation.

1. INTRODUCTION

Dynamically typed languages, where the type of an expression may not be completely known at compile time, have experienced increased popularity in recent years. These languages offer a number of advantages over statically typed ones: the source code of the programs tend to be more compact (e.g., type declarations are not present and castings are unneeded), faster to develop, and easier to reuse.

In statically typed languages with prescriptive types the type of each expression is to be declared in the source code. This makes it possible to generate tighter data encodings and remove all type checks at compile time, which in general results in faster execution times and reduced memory usage and, hopefully, in the static detection of more programming errors. At the same time, extra redundancy is added in the cases where types can be inferred, and calls for a more rigorous (or strict) developing methodology. Additionally, some program and data structures which are natural for dynamically typed languages need to be adapted significantly to be cast into common, well-known type systems.

It is interesting to note that recent advances in program analysis can potentially reconcile to a large extent this apparent dichotomy by inferring automatically the properties needed to perform the optimizations brought about by static typing and by using such properties to perform non-trivial error and property checking which subsumes traditional type-based static checking [10, 5, 14, 6]. Also, even in the cases where types and other information cannot be statically inferred, performance of dynamic languages can be quite competitive, since modern compilers and abstract machines have evolved significantly and offer a high degree of optimization.

However, the same high degree of specialization and optimization that brings about the performance of modern compilers and abstract machines implies significant complexity in their design which in turn makes it very difficult to explore a large design space of modifications in order to obtain significant further advances, at least "by hand."

In this paper we explore, in the context of a state-of-the-art WAM-based [18, 1] implementation of Prolog, how a number of variations in the way the core data structures of an abstract machine are implemented impact performance and memory usage. Our objective is to draw general conclusions about which are the best optimization options. We use a Prolog-inspired high-level language to specify encod-

[*]The authors have been partially supported by EU projects 215483 *S-Cube*, IST-15905 *MOBIUS*, Spanish projects ITEA2/PROFIT FIT-340005-2007-14 *ES_PASS*, ITEA/PROFIT FIT-350400-2006-44 *GGCC*, MEC TIN2005-09207-C03-01 *MERIT/COMVERS*, Comunidad de Madrid project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo is also partially funded by the Prince of Asturias Chair in Information Science and Technology.

PPDP'08, July 16–18, 2008, Valencia, Spain.
Copyright 2008 ACM 978-1-60558-117-0/08/07 ...$5.00.

ings, the associated operations, and in general to generate complete abstract machines using these variations. This language has a type and property system which, while quite flexible, is decidable at compile time, so that very efficient low-level code (C, in our case) can be generated.

Implementation of Dynamic Typing: Dynamic typing requires type information to be available at run time, and therefore it is customary to store it within the actual data during execution. In this paper we focus on the widespread implementation technique which uses *tagged words*. In this approach a (usually reduced) set of *tags* is used to represent type identifiers which are used as the first level information regarding the type of the contents of the memory word. Other ancillary information, such as the garbage collection (GC) bits as needed by some GC algorithms, is often stored in the tag. These tags are usually attached to the actual data value, and they are typically stored together in the same machine word. There are multiple ways to do this using fewer or more bits for the tag, placing it in the upper or lower part of the word, etc. However, not all schemes to pair up tag and value offer the same performance, memory consumption, and addressable space size. The quality of a tagged word implementation actually depends on a compromise between the available addressable space and performance. The latter is dominated by the cost of operations on the tagged type: setting and getting the tag, reading the value, and manipulating the GC-related information. Without explicit hardware support for tagged words (the norm in today's general-purpose processors), implementers must rely on simple, natively supported operations like shifting and masking, and often the effectively addressable space has to be reduced in order to, e.g., be able to *pack* a (reduced-size) pointer in a word using the space left by the tag and the GC information.

Performance of Encoding Schemes: Some bit layouts appear, at least a priori, to allow implementing more efficiently the most often used operations. In practice, experimental results indicate that with modern processors a realistic cost estimation is very difficult to perform a priori due to the inherent complexity of the processor operation, since too many parameters have to be taken into account: superscalar architectures, out-of-order execution, branch prediction, size of the pipeline, different cache sizes and levels, SIMD instructions, etc. That makes it very difficult to extract performance conclusions from the source code which implements the operations on the tagged data, even if the final assembler code is known and can be analyzed thoroughly: every basic operation depends, in fact, on the previous history and current state of the processor units. The situation is aggravated when the overall abstract machine complexity is taken into account. The implementation of the basic operations is often quite tricky, and high-performance code is usually achieved by creating by hand a number of specialized cases of these operations to be used in different contexts (i.e., when certain preconditions are met). As a result the code describing the abstract machine is usually large and any change is tedious and error-prone, which limits the capacity to explore alternatives systematically.

In order to overcome these problems, and building on the work presented in [13, 12], we have constructed a framework for generating abstract machines (and their corresponding compiler) *automatically* based on high-level descriptions of the placement of tags, GC bits, and values inside a tagged word, the size of the tagged word, the size of the native machine word, as well as some variations on the *shape* of the basic operations which deal with these tagged words. In all combinations, we aim at optimizing the resulting C code as much as possible, up to the point in which the result is often indistinguishable (in some cases identical) from what a highly skilled programmer would have written.

The concrete virtual machine skeleton we base our variations on is an implementation of the well-known WAM for Prolog. We argue that tagging schemes of modern Prolog virtual machines have requirements which are not unlike those of other dynamic languages: basic operations, encodings, and optimization possibilities at this level are similar in all of them. Additionally, many *statically-typed* programming languages present also dynamic features such as class inheritance, dynamic dispatching, disjunction of data structures, etc., which require similar techniques.

In all these cases, variations on the tagging scheme and ancillary operations significantly affect both the generated code and the size of the data structures, and can have a critical impact on execution speed and memory consumption. In fact, as a result of the (non-obvious) complexity of the basic tag operations and their very frequent use, small variations in their implementation can have a significant effect on performance even in small programs.

Note that the purpose of this paper is not to demonstrate that we are using the overall best scheme for implementing a Prolog abstract machine (which includes decisions about term representations [8], whether or not to use stack variables, last call optimizations, . . . , as well as parameters that affect performance including the bytecode encoding, instruction set, unification algorithm, C compiler, etc.), but rather to show how a higher level description (higher than C code) can be parameterized to generate variations of a tagging scheme, and to study the different results in terms of performance and memory usage.

2. SPECIFYING ABSTRACT MACHINES

As mentioned before, classical implementations of abstract machines involve non-trivial algorithms with data representations defined at the bit level, and very involved (hand-)codifications of the operations on them. The C language is a common implementation vehicle because the programmer can control many low level aspects and the quality of assembler code generated by current compilers is quite high, so that it is essentially a high-level way of generating quasi-optimal assembler code. The usual development techniques involve using preprocessor macros and conditional code, inlining C functions, using bit packed structures, etc. However, in more involved cases (e.g., when extensive instruction merging is to be performed, or when variations are being explored), this approach falls short, and some sort of automatic code generation (which can in part be done with advanced preprocessors) needs to be adopted to control complexity and avoid coding errors.

In this paper we use an abstract machine generator framework [13] and we use the ImProlog language [12] to write the abstract machine and to specify the low-level data layout. ImProlog is a restricted Prolog subset extended with *mutable variables* as first-order citizens and specific annotations to declare constraints on how to encode the tagged words in the machine memory. The compilation process ensures that the optimizations which allow efficient native code to be generated (controlled unfolding, specialization, unboxing of types, avoiding unnecessary trailing, backtracking, etc.) are met by the initial code, and rejects the program / module otherwise. Although the compiler and the language is not yet suitable (or intended) for writing general-purpose applications, it has been successfully applied in the implementation of a very efficient Prolog emulator that is highly competitive with hand-written state of the art Prolog emulators such as those of Ciao [4], Yap [16], SICStus [17], hProlog etc., and which is used as the basis for our experiments.

The C code that the ImProlog compiler generates could obviously have been hand-written given enough time and patience. However, as we stated before, our goal is to reduce the burden on the abstract machine designer (which is likely to reduce programming errors) through the use of a higher-level language and improve correctness (the ImProlog compiler can statically detect ill-typed operations with respect to the expected tags, without introducing run-time checks), while at the same time making it easier to maintain and extend the code.

The approach also allows exploring more possibilities of optimization. This includes stating high-level optimization rules and letting the compiler determine (maybe with additional compilation hints) whether applying them to generating acceptable native code is feasible or not at each point. Such optimization rules are used at any point in which they can be applied while generating the C code —i.e., in all parts of the abstract machine code corresponding to the implementation of bytecode instructions and other operations. This relieves the programmer from repeatedly having to remember to apply the same coding tricks at all points where similar operations are implemented, which also reduces the possibilities of introducing errors.

2.1 Types in ImProlog

Since ImProlog is the implementation language, the data representations used in the abstract machines generated are described as ImProlog types. The language used in ImProlog to describe types is (as in Ciao and CiaoPP [10]) the same that is used to write programs. Types are written as predicates, and the set of solutions of the predicate is the set of values in the type. Type definitions in ImProlog can have recursive calls, disjunctions, etc., as long as the compiler is able to analyze them and extract enough information to generate an efficient, unique encoding for the type.

From a syntactic point of view, Ciao's functional notation [7] is convenient to write type definitions, and we will use it for conciseness in this paper. This notation makes the default assumption that function outputs correspond to the last predicate argument (i.e., $\mathsf{X} = {\sim}\mathsf{p}(T_1,\ldots, T_{n-1})$ stands for $\mathsf{p}(T_1,\ldots, T_{n-1}, T_n)$, $\mathsf{X} = T_n$). The syntax for feature terms uses $\mathsf{A}{\cdot}\mathsf{B}$ as a function that returns the contents of feature B in A. For mutables, the goal $\mathsf{A} \Leftarrow \mathsf{B}$ sets B as the content of A, and the function @A obtains the value stored in A. The type ${\sim}\mathsf{mut}(\mathsf{T})$ stands for the mutables whose values are restricted to those of type T.

The ImProlog compilation process distinguishes the type a variable holds at a given time and the type that describes the possible values that can be encoded in that variable (the *encoding* type). Choosing the right encoding for each variable generates a statically typed (ImProlog) program, in which the type of every expression must be either inferred by the ImProlog compiler or explicitly stated so that unique memory encodings for variables and mutables can be statically determined. The compiler will reject programs with non-unique feasible type encodings for expressions / variables.

Although this presents a limitation for general-purpose programming, we think that it can be tolerated in the case of writing an abstract machine, where high performance is paramount. If a unique data representation cannot be statically determined, the compiler would need to generate dynamic tests to distinguish between the different encodings for the same data (e.g., comparing variables containing atoms which are encoded in different ways) which may result in a performance loss. Detecting that this is the case would be difficult if the compiler does not inform the user that this overhead has been introduced.[1]

2.2 Feature Terms and Disjunctions of Types

In [12] ImProlog types were limited to built-in types, which are seen as black boxes, and which reflect machine and low-level types. As a natural step, we have extended the ImProlog language and compiler with the required machinery to express the complex types needed to define *tagged words*: structured types (based on feature terms), and disjunctions of types (based on dependent types). For the sake of efficiency, we have improved the support for low-level encoding at the level of bits. The implemented support for dependent types allows indexing the feature term w.r.t. a single atomic feature. The restriction in that case is that the discriminant field must have the same encoding properties in all cases. When one of the type features is accessed and the discriminant value is known at compile time, no checks are necessary, since the type encoding is precisely defined. On the other hand, if the discriminant value is not unequivocally defined at compile time, the code is wrapped in tests that cover all the possible cases and which associate them with a code version specialized for the case in hand.

2.3 Defining a Hierarchy of Types

Inheritance can be captured by the semantics of logical implication [2], and this idea has been used to define our type hierarchy by means of feature terms and disjunctions of types. We will use this hierarchy to define the *tagged* type (Section 3). Inheritance makes it possible to define a (complex) set of types as a group of predicates making up a hierarchy tree where each type is a node, the root type is the root of the tree and the final types, which are not needed by any other type, its leaves. Using this coding style we can group together definitions belonging to the same type in an "object-oriented" fashion. Although it is out of the scope of this paper, this makes it possible to extend or specialize the type just by adding or removing final types. The general skeleton of each type definition includes the following rules:

Rule 1 : $\mathsf{supertype}(\mathsf{T})$:− $\mathsf{type}(\mathsf{T})$.
Parent-child relationship between some type and its supertype (except for the root type). This is to be read as "type is a kind of supertype".

Rule 2 : $\mathsf{type}(\mathsf{T})$:− $\mathsf{T}{\cdot}\mathsf{kind}$ = 'unique−atom', $\mathsf{type}_\mathsf{i}(\mathsf{T})$.
This rule defines a final type (a leaf in the tree) in the hierarchy. This gives the basic solution for the type,

[1] In fact, these errors can be interpreted as a quality report. An interesting option would be to define a tolerance value that allows or disallows certain ways of generating code.

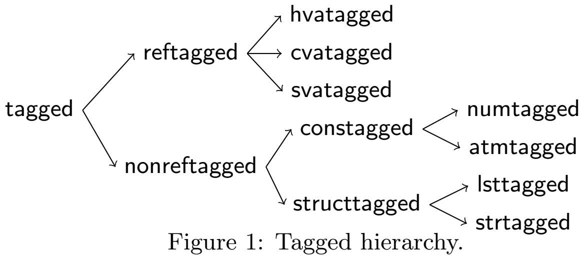


Figure 1: Tagged hierarchy.

which must be mutually exclusive with all other types. Only leaf nodes define the *kind* feature.

Rule 3 : type_i(T) :− supertype_i(T), . . .
The definition of the features that are specific to the type. It may include the parent's invariant and extend it with other features.

Rule 4 : operation(A, . . .) :− type(A), . . .
One rule for each operation on the type, using first argument indexing, when available, for efficiency.

The solutions (i.e., values belonging to the type) given by a type declaration are those generated by the type itself and by all of its descendants. The predicates type_i and type differ as follows: the former defines the contents that are exclusive to type type (and, optionally, the types it inherits from) and the latter defines the type itself (and, recursively, all the types which inherit from it).

3. SPECIFYING THE TAGGED DATA TYPE

We define in ImProlog a type that implements the basic data structure (*tagged* type) for the Ciao abstract machine. It is a discriminated union of heap variables, stack variables, constrained variables, structures, list constructors, atoms, and numbers. This type (or that of mutables constrained to values of this type) appears everywhere in the abstract machine state (heap cells, temporal registers, values saved in a choice point, local values in local frames, etc.).

The novelty of this approach —for the generation of abstract machines— is that the full definition is composed of the high level description, used by analysis and optimizations, and the low level annotations and optional user-defined compilation rules that determine how optimal native code is generated for the type definition and operations.

3.1 The Tagged Hierarchy

The tagged type can be viewed as the hierarchy in Figure 1. A simplified subset of the description for the type is shown in Figure 2, for generic nodes, and in Figure 3, for the leaf nodes. In this case the feature that distinguishes all leaves is called *tag* (the type of the *tag* feature returns the set of all required tags). We also describe, as an example, the code for the operation unify_cons, that unifies the tagged in its first argument with a single-cell tagged (a tagged whose value is perfectly defined without consulting any heap or stack location, such as small integers or atoms).

That specification defines the possible tagged words. The defined tagged word may optionally contain two GC bits [3] or place them in an external memory area, depending on whether predicate `ext_gc_bits` is true or false. Partial evaluation is used to statically specialize tagged$_i$ with respect to the usage of external GC bits.

```
The tagged type

tagged_i(T) :−
  (  ext_gc_bits  → true
  ;  T·marked = ∼bool, T·forward = ∼bool ).
unify_cons(R, Cons) :− derefvar(R), unify_cons_d(R, Cons).
. . .

References (a kind of tagged)

tagged(T) :− reftagged(T).
reftagged_i(T) :−
        tagged_i(T),
        T·ref = ∼mut(tagged).
unify_cons_d(R, Cons) :− reftagged(@R),
        (  trail_cond (@R) →
              trail_push (@R)
        ;  true
        ),
        (@R)·ref ⇐ Cons.
. . .

Non-references (a kind of tagged)

tagged(T) :− nonreftagged(T).
nonreftagged_i(T) :− tagged_i(T).
unify_cons_d(R, Cons) :− nonreftagged(@R),
        @R == Cons.
. . .

Constants (a kind of non-reference)

nonreftagged(T) :− constagged(T).
constagged_i(T) :− nonreftagged_i(T).
. . .
```

Figure 2: Some generic nodes in the hierarchy.

```
Heap variables (a kind of reference)

reftagged(T) :− hvatagged(T).
hvatagged(T) :− T·tag = hva, hvatagged_i(T).
hvatagged_i(T) :− reftagged_i(T).
trail_cond (A) :− hvatagged(A), A·ref < (∼s)·heap_uncond.
. . .

Small integers (a kind of constant)

constagged(T) :− numtagged(T).
numtagged(T) :− T·tag = num, numtagged_i(T).
numtagged_i(T) :− constagged_i(T), T·num = ∼int.
```

Figure 3: Some leaf nodes in the tagged hierarchy.

Note that low-level encoding details are not present at this point; they are introduced later as annotations. This separation facilitates automatic handling of more properties that it would be possible with a low-level implementation (where the domain information for tagged words is lost when they are translated to integers). For example, it is possible to specify that all members of the heap have values of type `nonstacktagged`, to detect errors or remove checks for svatagged automatically, where the type is defined as:

```
nonstackreftagged(T) :− hvatagged(T) ; cvatagged(T).
nonstacktagged(T) :− nonstackreftagged(T) ; nonreftagged(T).
```

4. OPTIMIZING TYPE ENCODINGS

The data type that customarily constitutes the building unit of WAM-based abstract machines is a machine word (or *tagged*) where the value of some of its bits (the *tag*) provides the meaning of the rest of them. We obtain the same effect from higher level type descriptions for the *tagged* type by constraining the size available to represent the type and each of its features. In this section we will describe the annotations and low-level optimizations used to implement the *tagged* type. The set of all of these annotations constitute the parameter values that define a *tag scheme variation*.

4.1 Bit-level Encoding

The basic data types supported in ImProlog include the unboxed form of atoms (encoding types that define a set of atoms, where each atom is represented as a number), numbers (limited to native types, such as 32 bit or 64 bit integers, floats, etc.), and mutables (as local C variables or stored as pointers to a memory location where the value is contained). This is possible because of the restrictions on the values which can be stored in a variable to avoid boxing: they must be completely instantiated and contain values defined by their type. Also, dereference chains of arbitrary length are not allowed, only fixed-length chains as specified in the type. With bit-level annotations for features, we limit the total size available to represent the data type, and the bits available for each of the features and their relative bit position.

The final bit layout is determined by annotations that control the bit offset where each feature is placed. In particular, they may be allocated in the *upper* or *lower* free part of the data bits, or even *split* between these two locations. For the *tagged* type, we abbreviate the storage location of the tag bits as `h` for upper bits, `l` for lower bits, or `s` for split bits. For GC marks, we use `H` for upper bits, `L` for lower bits, and *external* when they are allocated in a separate section of the stack (enabled by ext_gc_bits/0).

The bit layout affects how feature values can be extracted: if tags are in the lower part, a mask just gives its value; if they are in the upper part, a left rotation is needed. However, extracting a tag value is not so common when optimized tag tests are used, since most operations are reduced to bit tests. The location of tags and GC bits affects the access to pointers and other values. A field is extracted by masking out other fields and shifting the value accordingly. If the fields to be removed are known at compile time, then subtraction is used instead of masking, so that use can be made of the indirect addressing mode in the assembler code (pointer+offset), and combined at compile time with other arithmetic operations such as, e.g., the additions that need to be performed when we use a displaced pointer from a *tagged* or add two small numbers.

In the case of pointers, the fixed pointer bits must be placed back, with a previous shift to reintroduce the alignment bits, if they have been used (note that the shifting to extract the pointer and reinsert the alignments may be canceled out, such as in `s` and `hL` in 32 bits or `lH` or `lL` in 64 bits). Note that each bit placement has its advantages and disadvantages, since optimizing some operations with one configuration sometimes makes others more costly, which ultimately complicates performance estimation.

4.2 Trade-off: Limited Address Space

Encoding both fixed-length reference chains and mutables in the presence of bit-size restrictions implies some implementation tricks to define pointers where some bits are free to be used for other purposes. For example, alignment bits can be reused or some extra bits can be made available by fixing the location of a memory region with the `mmap` system call. Note that storing n-bit pointers in less than n bits results in address space limitation. Such limitation may not be an issue in 64 bit machines (were the address space is huge, much larger than the physical memory, as it was also the case for 32 bit pointers in the not-so-remote past), but currently it may be a problem in 32 bit machines where more than 1GB of physical memory is common nowadays. In the case of our *tagged* word representation, the available *address space* is determined by the number of bits required for tags and GC, the tagged word size and the use of external or internal GC bits. Here we see even more clearly why a flexible way to implement abstract machines with different tagged schemes matters, since a scheme that represents a good compromise between address space and performance for a 32 bit machine may not be the best for 64 bits (even more, if we consider architectural differences). Also, while it is tempting to decide to simply concentrate on 64-bit machines, since they are now the norm for new general-purpose machines, it should be noted that 32 bit machines are still in widespread use and, furthermore, other computing platforms (such as, e.g., portable gaming devices, cell phones, etc.) have physical limitations which make them have 32-bit address buses.

4.3 More Control Over Tag Representation

Although an automatic assignment of encoding values for each of the atoms in a type may be efficient for most applications, it is not enough to implement optimal operations for the *tagged* type, where choosing the right numeric value for each *tag* may play an important role in reducing the number of assembler instructions required to implement essential basic operations on the *tagged*. The reason is that for some encodings a single assembler instruction can check for more than one tag at the same time, yielding faster code to switch on tag values. To this end, the ImProlog compiler allows explicit definition of atom encodings and a large set of predefined compilation patterns to build indexing trees. ImProlog does not have a *switch* construct, and, as Prolog, it is based on indexing to optimize clause selection, in two steps defined as Figure 4 illustrates.

Index extraction: Group together discontiguous clauses and transform the program to make explicit the possible indexing keys (unfolding cheap type checks, removing impossible cases, adding the default case). E.g., suppose that nonstacktagged(X) holds on entry, then for the code on the left we obtain the index on the `tag` feature of X on the right:

(hvatagged(X) $\rightarrow C_H$; cvatagged(X) $\rightarrow C_C$; svatagged(X) $\rightarrow C_S$; nonreftagged(X) $\rightarrow C_N$)	(X·tag = hva $\rightarrow C_H$; X·tag = cva $\rightarrow C_C$; (X·tag = num ; X·tag = atm ; X·tag = lst ; X·tag = str) $\rightarrow C_N$)

Low level indexing: Compile low level indexing, by taking into account where the tag is located (e.g., lower or upper bits) and how it is encoded (e.g., hva: 0, cva:1, ..., str:7). Depending on whether optimized tests are used or not, we distinguish several compilation modes, which we will call the `swmode` parameter: **sw0**, **sw1**, and **sw2**.

In **sw0** mode, the code generator emits a simple if-then-else (when two cases are used), or a C switch statement

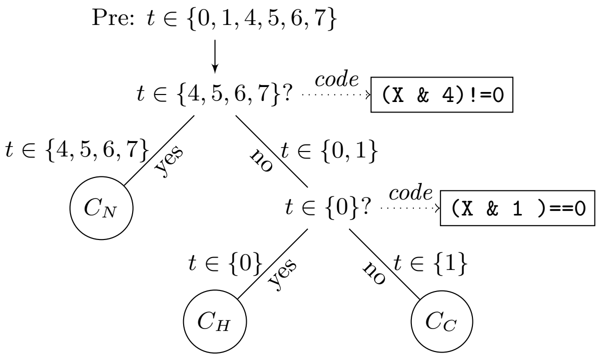


Figure 4: Obtaining indexing code.

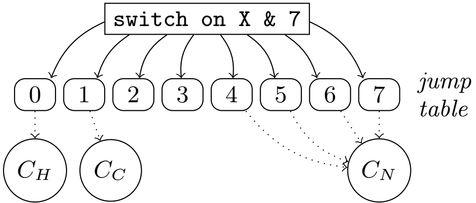


Figure 5: Code generation using **sw0**.

(usually compiled internally as a indirect jump to an array that maps from tag values to code labels) — see Figure 5 for an example where the tag is stored in the lower 3 bits of `X`.

Note that although C compilers can generate very clever code for switches, they are unable to infer the value of bit-encoded members or make use of user-provided assertions (so they cannot optimize some tests). Although the indirect jump version may look very optimized, it may hinder the benefits of the branch prediction unit in modern processors [11]. In **sw1** the compiler tries to emit better code by grouping together tests for several values in a single expression (that can be cheaply compiled in few assembler instructions by the C compiler) that we will call *bitfield test*. We call a bitfield test with precondition S and postcondition P, for the k-bits in the p-bit offset, a low level test *Test* that given a n-bit word `X`, where t=`(X >>` p`)&((1<<`k`)-1)`, checks that if $t \in S$ and *Test* is true, then $t \in P$. Examples of tests for the 3 lower bits are:

- Pre:$t \in \{0..7\}$ Post:$t \in \{4,5,6,7\}$ Code: `(X & 4)!=0`
- Pre:$t \in \{0,1\}$ Post:$t \in \{0\}$ Code: `(X & 1)==0`
- Pre:$t \in \{4,5,6,7\}$ Post:$t \in \{4,6\}$ Code: `(X & 1)==0`

With the optimized tests, the search space is divided in two (the values in postcondition, and the values in the precondition minus the postcondition). When **sw2** is selected, more complex indexing trees are treated, that would otherwise be translated to C switches. Switch rules represent heuristics to reach the desired branches more efficiently. The rules are specified as nested disjunctions (that define a binary tree where the leaf nodes are sets of values) and are supposed to cover all cases so that the implicit precondition is the union of all the sets. For example: $\{4,5,6,7\} \vee (\{0\} \vee \{1\})$ states that it should do a test to distinguish between $\{4,5,6,7\}$ (which is a leaf node) $\{0,1\}$, then do the same for $\{0\}$ to distinguish between $\{0\}$ and $\{1\}$. When the test code is inserted, we obtain the low level definition of the indexing code (Figure 4).

4.4 Extending GC for External and Internal GC bits

GC bits are only active during garbage collection. When GC starts, a special section is reserved to store GC bits for every WAM stack. A GC pointer resolution operation which obtains the location of the GC bits for a pointer from some memory region, while reasonably fast, accumulates an excessive overhead when it is performed for every GC bit access.

To minimize the number of GC pointer resolutions without having to maintain two slightly different GC algorithm codifications (for internal and external GC bits), we define an abstract "pointer to tagged" data type. When internal GC bits are selected, the pointer is a normal pointer. When external GC bits are used, the pointer to tagged is actually a pair of pointers: one points to the actual tagged word and the other one to the byte that contains the GC bits. Pointer displacement, read, and write operations are done for both pointers, obtaining a similar performance for the case of external and internal GC bits.

The GC algorithm (and the code itself) is therefore generic with respect to the format of the tagged words.

4.5 Interactions with Bytecode

There is an implicit dependence between the abstract machine definition and the compiler that generates bytecode to be executed in that machine. By extending the framework to describe the *tagged* type, the assumptions about data sizes become a parameter for the compiler (in addition to other parameters related to the accepted instruction set, instruction merging, how instructions are encoded, etc. [12, 13]). When the abstract machine code is generated, all this information is fed into the compiler; this approach ensures that the compiler and the abstract machine are synchronized. This is needed, for example, when the compiler requires the size of a *tagged* to insert heap overflow tests prior to the construction of known terms, or to decide whether multiple-precision arithmetic is needed when building an integer, which depends on whether the number of bits used to represent the value of a small number (num feature in numtagged) is enough to store the value.

The *tagged* size also affects the size of opcodes and operands in bytecode instructions. In some architectures n-bit words must be n-bit aligned (for efficiency or architectural constraints). While it is possible to introduce paddings to keep operands always aligned [15], in order to allow a more compact bytecode representation, the instruction set in the standard Ciao system uses instructions whose size is not a multiple of the largest machine word (32 bits in a 32-bit machine). In that case, several opcodes stand for differently padded versions of the same instruction.

In our system the instruction set is generated automatically from a higher-level description, creating all the necessary padded versions automatically. E.g., the following code defines an instruction that unifies an X register (a mutable variable whose low-level address is specified by an operand in the bytecode representation containing an indirect offset to an address in the global state), and a `constagged`:

```
:− entry(u_cons(xreg,constagged)).
u_cons(A, Cons) :− T ⇐ @A, unify_cons(T, Cons).
```

where the code that links the operands of that bytecode with the arguments is synthesized by the abstract machine gen-

boyer	Simplified Boyer-Moore theorem prover.
crypt	Arithmetic puzzle involving multiplication.
deriv	Symbolic derivation of polynomials.
exp	Work out 13^{7111}.
factorial	Compute the factorial of a number.
fft	Fast Fourier transform
fib	Simply recursive computation of the n^{th} Fibonacci number.
guardians	Prison guards playing game.
jugs	Jugs problem.
knights	Chess knight tour, visiting only once every board cell.
nreverse	Naive reversal of a list using append.
poly	Raises symbolically the expression `1+x+y+z` to the n^{th} power.
primes	Sieve of Eratosthenes.
qsort	Implementation of QuickSort.
queens11	N-Queens with $N = 11$.
query	Natural language query to a database with information about countries.
tak	Computation of the Takeuchi function.
trie	Word indexer using tries.
wave_arr	Signal processing using updatable arrays.
wave_dyn	Signal processing using dynamic facts.
witt	A conceptual clustering algorithm.
wumpus	Wumpus world game.

Table 1: Benchmark descriptions.

erator. In the instruction above, if the instruction opcodes are stored as a 16 bit word and the X registers as a 16-bit word, the second operand (that is 32-bit sized) is aligned to 32 bits or not, depending on whether the instruction begins at a 32-bit aligned address or not. Then, two versions of the same instruction (padded and non-padded) are emitted; in the latter a 16-bit pad before the Cons operand is inserted to ensure that it is aligned to 32-bits.

5. EVALUATING TAG SCHEME VARIATIONS

We tested 48 combinations of the available address space possibilities, bit layouts, and optimizations. For each combination we generate an abstract machine on which 22 benchmarks (see table 1) are executed. Some of these benchmarks are well known and relatively small, while others (for example, `witt` and `wumpus`) can be considered having a medium size (from 400 to 500 lines). In any case, these benchmarks exercise operations whose performance will be greatly affected by the different compilation options under study, so they can be taken as reasonable witnesses of this impact in efficiency. We measured the memory usage and the total execution time, with and without GC. Each benchmark was executed three times, taking the shortest execution. The experiments were run on four machines, with different architectures or processors, taking from 5 to 12 hours, depending on the machine speed. In the machines where it was possible, the O.S. was running in native 64-bit mode (executing also the 32-bit tests).

We will use an abbreviated notation for the different combinations of compilation options and architectures, using the following codes:

Benchmark	**Yap**	**hProlog**	**SWI**	**Def.**	**1.13**
boyer	1392	1532	11169	1604	2560
crypt	3208	2108	36159	3460	6308
deriv	3924	3824	12610	3860	6676
exp	1308	1740	2599	1624	1400
factorial	4928	2368	16979	2736	3404
fft	1020	1652	14351	1548	2236
fib	2424	1180	8159	1332	1416
knights	2116	1968	11980	2352	3432
nreverse	1820	908	18950	2216	3900
poly	1328	1104	6850	1160	1896
primes	4060	2004	28050	2520	3936
qsort	1604	1528	8810	1704	2600
queens11	1408	1308	24669	1676	3200
query	632	676	6180	968	1448
tak	3068	1816	27500	2964	5124

Table 2: Speed comparison (`coreduo-32`).

highbittags	`h`	splitbittags	`s`	lowbittags	`l`
lowbitgc	`L`	highbitgc	`H`		
sw0	`0`	sw1	`1`	sw2	`2`

`sparc64`	64-b. Sparc/Sol. 5.10, 3MB cache
`p4-64`	64-b. Intel P4/Lin. 2.6, 2MB cache
`xeon-32`	32-b. Intel Xeon/Lin. 2.6, 512Kb cache
`coreduo-32`	32-b. Intel Core Duo/Lin. 2.6, 2Mb cache

The C compiler (gcc) we used to compile the emulators had a different version in each architecture. This may affect the conclusions in which speedups from different architectures are compared, which we think is sensible only for `xeon-32` and `coreduo-32` (Section 5.3). In these two cases, however, the "best options" happen to be the same. This makes us confident that the results will also be applicable in a more long term, when current compilers have been outdated.

For each architecture the speedups and memory usage figures are normalized w.r.t. a default case which corresponds to `hL2` with 26 bits of address space (the smallest we tested). This normalization makes comparisons among different address spaces meaningful using speedups w.r.t. the fixed basic case. We want to note that this *default case* is in itself quite efficient: it was generated using as basis the Ciao virtual machine (itself an offspring of the SICStus 0.6/0.7 virtual machine) with some improvements (computed goto/threaded bytecode, smarter instruction opcode assignment, ...) written in ImProlog which made the virtual machine faster than the stock Ciao one. Table 2 helps to evaluate the speed of the *default* emulator (in the `coreduo-32` machine) w.r.t. to Ciao 1.13 and other well-know Prolog systems: Yap 5.1.2, hProlog 2.7, SWI-Prolog 5.6.55.

5.1 Address Limits and Memory Usage

The different addressing possibilities are described in Table 3. The leftmost column gives a name to each of the combinations and the one to its right describes the (physical) size of the *tagged* word (32 or 64 bits), the size of the stored pointer (64 bits when running in a 64 bit architecture and O.S.), and whether GC bits are internal or not to the tagged word (`extgc`). The option `qtag`, used in the default tag scheme in Ciao, reserves one bit to represent special functors for *blobs*. Note that not all combinations are possible in all studied architectures. The following columns provide the number of bits available for pointers, the number of bytes to be aligned to, and the resulting limits to addressable space.

The growth ratio in memory consumption that a given choice implies (w.r.t. the default case) appears in Table 4. Within each case, the figures are the same for every archi-

Name	Options	Ptr. bits	Align. (bytes)	Limits (Mb)
addr26	tagged32*pointer32*qtag	26	4	256
addr27	tagged32*pointer32	27	4	512
addr29	tagged32*pointer32*extgc	29	4	2048
addr30	tagged64*pointer32	30	8	4096
addr32	tagged64*pointer32*extgc	32	8	4096
addr59	tagged64*pointer64	59	8	full
addr61	tagged64*pointer64*extgc	61	8	full

Table 3: Options related to the address space.

tecture in which the case can be implemented. However, every memory zone has a different growth ratio, and the relative growth for every area is not homogeneous for all the combinations. The reason is that the memory space used by objects does not change uniformly for all objects.[2] Therefore, data about actual memory usage has to be taken experimentally, as the ratio between these object types is different for each program.

The differences in program memory come from the need to use extra *padding* bytes in some instructions, which have a different impact in the different tag schemes.

As objects in different tagging schemes take different amounts of memory (for example, `addr30` can address 4GB of memory, but each tagged word takes twice as much space as in `addr29`), Table 3 is not really useful to decide which is the best scheme memory-wise: it is much more useful to reason about the number of *objects* which can actually fit in the memory addressable so that, leaving aside speed considerations, using `addr29` may be more advantageous, as it uses half as much memory.[3] This is shown in Table 5, where the memory address limits have been *adjusted* taking into account the data in Table 4, to finally work out a ratio of the number of objects which can actually be created. Note that in practice all the physically available memory is addressable in 64-bit architectures.

Both `addr30` and `addr32` use two 32-bit (4 bytes) words for a tagged word in a 32-bit architecture. Pointers must therefore be aligned to 4-byte boundaries, and their two less significant bits have to be zero. Hence, both `addr32` and `addr30` can address 4GB of memory. The latter is included in order to study how using external GC bits impacts performance. Although the underlying emulator differs, those schemes are similar to the one used in ECLiPSe [9].

5.2 General Speed-Up Analysis

We have summarized in a series of plots (Figure 6) the arithmetic[4] average of speedups obtained assigning the different combinations of placements of GC bits, tags, and tag-related primitives in the `Y` axis, the option which selects the address space in the shapes of the dots, and the speedup in the `X` axis.

Some general conclusions can be drawn from these plots. First, the Sparc 64 (Figure 6(a)) architecture seems to have a very regular behavior (probably due to its large number of registers), except for some specific combinations of options which, non surprisingly, perform badly in all architectures. The Intel processors, on the other hand, show wide variations, both for the different options in a given processor and also among the tested processor families. The latter can be attributed to the large differences in the internal architecture among Intel processors, which share little more than a common assembler language.

Although careful inspection of the plots (including the extensive per-program data available at `http://clip.dia.fi.upm.es/~jfran/tagschemes`) can be of help to draw some definite conclusion, resorting to visual inspection is too error-prone, so we performed an analytical study of the available data.

5.3 General-Purpose Abstract Machine

Determining the combination of options which gives the best performance for a general-purpose abstract machine (where *general-purpose* is in our case reduced to the universe of programs in Table 1) is not easy: there is no absolute winner combination of tagging scheme / access primitive for any architecture / addressing range, and combinations which perform well (or very well) for some programs often perform badly (or very badly) for other programs.

Therefore, we decided to resort to a method based on *ballots* to decide the best option encoding / tag primitive implementation for each addressing space. In this setting, programs are voters, and the different combinations are candidates which voters *rank* using the runtime they offer for each voter. The idea is to select the candidate which is, in some sense, most preferred. Among the many ways to decide between candidates given this information, *Condorcet* methods are among the best known and preferred in many situations. We have selected the *Schulze* method [5] which it is used in other related areas and which determines a winner combination. This winner is removed from the list of candidates and the algorithm is applied again, to generate a global preference list, its last member being the the worst combination – the *loser*.[6]

The reason not to simply select the combination with the best average is that extreme cases can affect this average more than what is desirable. Removing the best and worst combination (to exclude extreme elements) was an option we did not want to take, because we wanted to take into account all the available information.

The results of this process are summarized in Tables 6 (without GC time) and 7 (including GC time), where for each architecture and address space, the best and worst tag scheme combinations are shown, together with the average speedup w.r.t. a default combination. For each of these, the best and worst speedup in the benchmarks is shown, as well as the ratio (W/L) between these speedups. This ratio gives a raw idea of what variation can be expected for each address space and architecture combination.

Let us note first that comparing speedups in different architectures is not really needed (or even meaningful). If a single tagging scheme / address space option were the winner hands down, then a single answer (a single code scheme) could be given for a machine, but a per-architecture best option can typically be decided at compile time with adequate macro definitions. On the other hand, comparing the best schemes in each of the architectures may give some insight on which code schemes are favored by the C compiler, opti-

[2] For example, large numbers or floating point numbers have a special, structure-based representation (a *blob*) whose size does not grow linearly with the size of the tagged word.

[3] Also, note that stock Linux kernels make only 3Gb available for user processes in 32-bit architectures.

[4] Plots with geometric average were practically identical.

[5] `http://m-schulze.webhop.net/schulze1.pdf`

[6] The Schulze method may end up in a draw, in which case we select the combination with the best average speedup to be the winner.

Addr.	Total	Program	Heap	Local	Trail	Choice	GC'ed memory
addr26	1.00	1.00	1.00	1.00	1.00	1.00	1.00
addr27	1.00	1.00	1.00	1.00	1.00	1.00	1.00
addr29	1.09	1.00	1.25	1.25	1.25	1.25	1.10
addr30	1.53	1.22	2.03	1.59	2.15	1.33	1.29
addr32	1.63	1.22	2.28	1.79	2.42	1.50	1.38
addr59	1.92	1.81	2.03	2.00	2.10	2.00	1.29
addr61	2.02	1.81	2.28	2.25	2.36	2.25	1.38

Table 4: Memory growth ratio.

Addr.	Total	Program	Heap	Local	Trail	Choice
addr26	256 (1.00)	256 (1.00)	256 (1.00)	256 (1.00)	256 (1.00)	256 (1.00)
addr27	512 (2.00)	512 (2.00)	512 (2.00)	512 (2.00)	512 (2.00)	512 (2.00)
addr29	1876 (7.33)	2048 (8.00)	1638 (6.40)	1638 (6.40)	1638 (6.40)	1638 (6.40)
addr30	2669 (10.43)	3355 (13.11)	2018 (7.89)	2579 (10.08)	1906 (7.45)	3076 (12.02)
addr32	2513 (9.82)	3355 (13.11)	1794 (7.01)	2293 (8.96)	1694 (6.62)	2734 (10.68)
addr59	all memory	all memory	all memory	all memory	all memory	all memory
addr61	all memory	all memory	all memory	all memory	all memory	all memory

Table 5: Effective addressable limits (and ratio w.r.t. default case) for address space options.

mizer, and processor, which may guide future decisions.

Selecting between tag and GC bit placement and code for tag management operations is somewhat independent from the addressing scheme selected. On one hand, schemes which need native 64 bits (e.g, `addr59` and `addr61`) are only applicable to 64-bit machines, and therefore cannot be compared with those designed to be used in 32-bit machines. Additionally, if some tag scheme A gives better performance, but less effective address space than some other tag scheme B, then a decision should be made depending on the expected needs of speed vs. runtime memory needs.

In order to decide which schemes are the best, we first observe that the options favored by the Schulze method are the same regardless of whether GC time is taken into account or not in all cases but one: for the `p4-64` architecture and the `addr29` tag scheme in tables 6 and 7, the winners are, respectively, `h2` and `h1`. However, these two code generation options gave almost identical speedup results, so the difference can be safely ignored, and we can, therefore, focus on just one of the speedup tables.

Options for Native 64-bit Implementations: In the two 64-bit architectures the native 64-bit tagging scheme is reasonably fast w.r.t. the default 32-bit one, and both (`addr59` and `addr61`) give a similar speedup. Therefore, and looking at the actual memory consumption figures in Table 4, we select `addr59`, since it would make better use of the really available memory in the machine. In that case, GC bits in the higher part (`H`) is the recommended option, while tag bits are better placed in the lower (`l`) part for `sparc64` and in the higher (`h`) part for `p4-64`. In general, predefined switch rules (`2`) gave the best results, except in the scheme `addr95` in `p4-64`.

Options for 32-bit Architectures: Architectures with only 32-bit addresses (`xeon-32` and `coreduo-32`, in our case), can take advantage of the full address range by using two 32-bit words (i.e., schemes `addr30` and `addr32`). This, however, comes at the cost of a noticeable slowdown. In return, a larger set of objects can be kept in memory. Between these two schemes, and if memory usage is a concern, we would select `addr30`. If speed is a primary concern, then the default setup (`addr26`) gives the best results. However, for a not very large price in speed (especially in `coreduo-32` machines), the number of objects in memory can be increased almost eightfold. For memory-demanding applications (which may even not run in the memory available with `addr26`) this would obviously be an advantage and the results suggest using `addr29` in these cases. For the selected case, the best performance is obtained by using external GC bits and split tag bits (`s`).

Options for 32-bit Tagged Words in 64-bit Machines: Although the natural option for a 64-bit machine would be a 64-bit native implementation, there are cases where this may not be completely advantageous. The space that a 64-bit tagged word takes up is 8 bytes, and if a 64-bit machine has less than 4Gb of memory, as is often the case currently on, e.g., normal desktops and laptops, more 32-bit-based objects can fit in it without leaving any memory unreachable. In this case, again, the best options in terms of addressing space, `addr30` and `addr32` pay a high price in speed. A better compromise uses `addr29`, as it provides good speed and a reasonably large memory address space. For the selected case, best options are similar to those for 64-bit in the same machines: tag bits are better placed in the lower (`l`) part for `sparc64` and in the higher (`h`) part for `p4-64`.

Impact of External GC Bits on Performance: Turning on external GC bit support (moving the GC bits out of the word) increases addressable space, at the expense of a somewhat more complex access to the GC bits and, perhaps more importantly, less cache locality, which however affects execution only when GC is performed. This is the reason why speed-ups are more modest when GC is turned on in the combinations where GC bits are external to the tagged word (`addr29`, `addr32`, `addr61`). Its impact is however not dramatically large, and can be accepted in exchange for the increased address space if it is really needed.

5.4 Per-Program Abstract Machines

Our framework makes it possible to generate an executable which contains bytecode, native code, and an abstract machine specialized for a given Prolog program, using any of the generation options we have discussed so far. It is, thus, natural, to generate program-specific abstract machines and measure their performance. Table 8 shows the results of this experiment. Each case was obtained by finding the best combination of tag/GC placement options for each benchmark, then averaging over the benchmarks.

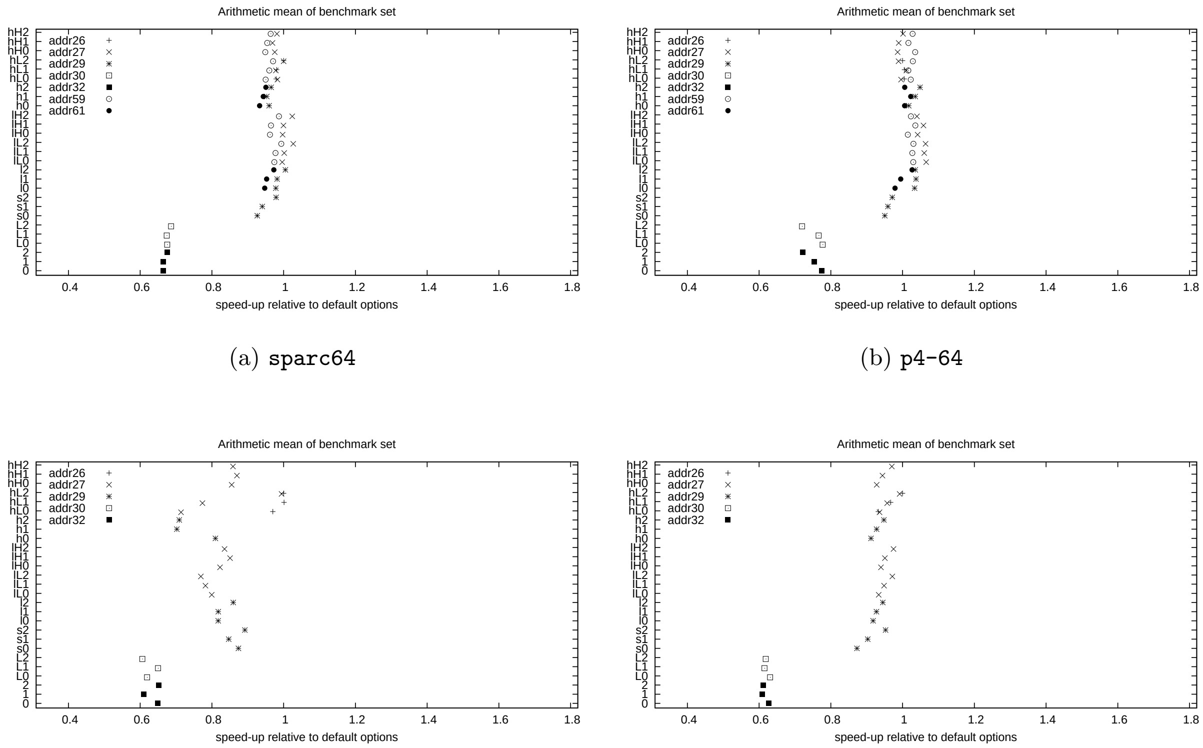
A comparison of the speedups in Table 6 and 7 with Table 8 shows that program-specific abstract machines have, on average, better performance than an "agreed" single abstract machine. The difference is not very big and, for general us-

(tags,gcbits,swmode) and speedups for host sparc64											
Addr.	Winner				Loser				W/L		
	best	avg.	max	min.	**worst**	avg.	max	min.	avg.	max	min
addr26	hL2	1.00	1.00	1.00	hL1	0.98	1.00	0.94	1.02	1.06	1.00
addr27	lL2	1.04	1.33	1.00	hH1	0.97	1.00	0.90	1.07	1.47	1.00
addr29	l2	1.04	1.28	0.96	s0	0.95	1.00	0.89	1.09	1.31	1.00
addr30	2	0.69	0.96	0.59	1	0.68	0.96	0.58	1.02	1.05	0.99
addr32	2	0.69	0.96	0.60	1	0.68	0.96	0.58	1.02	1.05	0.99
addr59	lH2	1.00	1.35	0.86	hH0	0.96	1.35	0.83	1.04	1.11	1.00
addr61	l2	1.00	1.35	0.85	h0	0.96	1.35	0.83	1.05	1.11	1.00
(tags,gcbits,swmode) and speedups for host p4-64											
addr26	hL1	1.01	1.09	0.94	hL2	1.00	1.00	1.00	1.01	1.09	0.94
addr27	lL2	1.08	1.24	0.86	hL2	0.99	1.07	0.88	1.09	1.24	0.84
addr29	h2	1.08	1.25	0.86	s1	0.98	1.05	0.89	1.09	1.28	0.87
addr30	0	0.80	0.99	0.69	2	0.73	1.00	0.62	1.09	1.19	0.99
addr32	0	0.81	1.00	0.67	2	0.74	0.99	0.62	1.09	1.26	1.01
addr59	hH0	1.07	1.31	0.77	hL1	1.04	1.29	0.77	1.03	1.09	0.96
addr61	l2	1.08	1.31	0.79	l0	1.04	1.24	0.70	1.05	1.32	0.91
(tags,gcbits,swmode) and speedups for host xeon-32											
addr26	hL2	1.00	1.00	1.00	hL0	0.97	1.31	0.64	1.07	1.55	0.77
addr27	hL2	0.99	1.36	0.62	hL0	0.72	0.95	0.58	1.40	1.72	1.05
addr29	s2	0.92	1.21	0.65	h1	0.72	0.93	0.60	1.28	1.67	0.97
addr30	0	0.62	0.95	0.39	2	0.61	0.76	0.37	1.02	1.27	0.80
addr32	0	0.66	0.89	0.42	1	0.62	0.80	0.44	1.07	1.48	0.71
(tags,gcbits,swmode) and speedups for host coreduo-32											
addr26	hL2	1.00	1.00	1.00	hL0	0.94	1.01	0.70	1.07	1.44	0.99
addr27	hL2	0.99	1.02	0.90	lL0	0.95	1.17	0.70	1.05	1.42	0.85
addr29	s2	0.98	1.07	0.82	s0	0.90	1.00	0.69	1.10	1.18	1.00
addr30	0	0.64	0.98	0.51	1	0.62	0.97	0.51	1.03	1.12	0.97
addr32	0	0.64	0.99	0.55	1	0.62	0.98	0.52	1.03	1.11	0.99

Table 6: Schulze winner for each address space and machine (GC time not taken into account).

(tags,gcbits,swmode) and speedups for host sparc64											
Addr.	Winner				Loser				W/L		
	best	avg.	max	min.	**worst**	avg.	max	min.	avg.	max	min
addr26	hL2	1.00	1.00	1.00	hL0	0.98	1.02	0.94	1.02	1.07	0.98
addr27	lL2	1.03	1.22	0.98	hH1	0.97	1.00	0.89	1.06	1.37	1.00
addr29	l2	1.00	1.11	0.85	s0	0.93	0.99	0.79	1.09	1.23	1.01
addr30	2	0.69	0.97	0.59	1	0.67	0.97	0.58	1.02	1.05	0.99
addr32	2	0.68	0.96	0.57	1	0.66	0.96	0.56	1.02	1.05	0.99
addr59	lH2	0.99	1.36	0.85	hH0	0.95	1.35	0.82	1.04	1.10	1.00
addr61	l2	0.97	1.34	0.80	h0	0.93	1.33	0.80	1.04	1.11	1.00
(tags,gcbits,swmode) and speedups for host p4-64											
addr26	hL1	1.01	1.09	0.94	hL2	1.00	1.00	1.00	1.01	1.09	0.94
addr27	lL2	1.06	1.24	0.86	hH1	0.99	1.11	0.90	1.08	1.25	0.90
addr29	h1	1.04	1.26	0.74	s0	0.95	1.04	0.73	1.09	1.29	0.92
addr30	0	0.78	1.02	0.65	2	0.72	0.99	0.61	1.08	1.19	0.99
addr32	0	0.77	0.99	0.58	2	0.72	0.98	0.53	1.08	1.18	1.01
addr59	hH0	1.04	1.30	0.77	lH0	1.01	1.23	0.80	1.02	1.18	0.90
addr61	l2	1.03	1.30	0.70	l0	0.98	1.24	0.70	1.05	1.32	0.91
(tags,gcbits,swmode) and speedups for host xeon-32											
addr26	hL2	1.00	1.00	1.00	hL0	0.97	1.31	0.64	1.06	1.55	0.77
addr27	hL2	0.99	1.36	0.62	hL0	0.71	0.95	0.58	1.40	1.72	1.05
addr29	s2	0.89	1.18	0.61	h1	0.70	0.93	0.60	1.28	1.67	0.98
addr30	0	0.62	0.95	0.40	2	0.61	0.76	0.38	1.02	1.27	0.80
addr32	0	0.65	0.89	0.42	1	0.61	0.80	0.43	1.07	1.48	0.71
(tags,gcbits,swmode) and speedups for host coreduo-32											
addr26	hL2	1.00	1.00	1.00	hL0	0.93	0.99	0.70	1.08	1.42	1.01
addr27	hL2	0.99	1.02	0.90	lL0	0.93	1.07	0.70	1.07	1.40	0.94
addr29	s2	0.95	1.01	0.75	s0	0.87	0.98	0.68	1.09	1.18	1.01
addr30	0	0.63	0.97	0.51	1	0.61	0.96	0.51	1.03	1.12	0.97
addr32	0	0.63	0.95	0.52	1	0.61	0.93	0.52	1.03	1.12	0.99

Table 7: Schulze winner for each address space and machine (GC time included).



(a) `sparc64`

(b) `p4-64`

(c) `xeon-32`

(d) `coreduo-32`

Figure 6: Arithmetic average of speedups for several architectures.

age, a single abstract machine with the right selection of options (see Section 5.3) is probably enough. However, for some benchmarks the best abstract machine is more than twice as fast as the worst one (e.g., `addr27` in `xeon-32`, column "W/L Sabs"), and in some other benchmarks they difference reaches the 38% (again, `xeon-32` but in the `addr26` row in the "Sabs" column).

6. FINAL REMARKS

It is interesting to observe that the 32 and 64 bit cases have a close performance, but of course there is a huge difference in address space (and a noticeable increase in memory consumption). Double word representations, which are easier to implement, have been observed to be in general slower. Thus, for languages or applications with a reduced set (around 8) of types of small objects requiring runtime type information it is clearly worth using in-word tags.

It is also interesting to note that with our approach we have been able to automatically generate code featuring all the optimizations present in a hand-written, highly-optimized abstract machine (the base case that we compare to) and we have also been able to obtain emulators with larger addressable space and/or better performance.

The results indicate that it is difficult to recommend a particular bit layout or tag switch implementation that will be ideal for all situations, since the best option set changes depending on the architecture and the memory requirements of the application. In this sense the results presented provide a recommendation table which indicates the best option for each situation (the results are of course limited to the architectures studied). In this sense, further interesting results of this work are a) that it is possible to construct a framework that allows a flexible and parametric implementation of tagged data which can be adapted to support different schemes with modest effort, and b) that the variability in results observed indicates that constructing such as framework is indeed useful not just for experimentation, but also for production since it allows generating the right machine for each architecture or even for each application.

The system in which these techniques have been implemented and wich which the experiments have been performed is part of the development branch of Ciao Prolog, which can be obtained by contacting the paper authors.

7. REFERENCES

[1] H. Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* MIT Press, 1991.

[2] H. Ait-Kaci and R. Nasr. Integrating Data Type Inheritance into Logic Programming. In P. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, pages 121–136. Springer, Berlin, Heidelberg, 1988.

[3] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage Collection for Prolog Based on WAM. *Communications of the ACM*, 31(6):719–741, 1988.

[4] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. P. (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

[5] M. Carro, J. Morales, H. Muller, G. Puebla, and M. Hermenegildo. High-Level Languages for Small Devices: A Case Study. In K. Flautner and T. Kim, editors, *Compilers, Architecture, and Synthesis for*

speedups for host `sparc64`												
Addr.	Without GC						With GC					
	Sabs			W/L Sabs			Sabs			W/L Sabs		
	avg.	max	min	avg.	max	min	avg.	max	min	avg.	max	min
addr26	1.00	1.03	1.00	1.03	1.07	1.00	1.00	1.02	1.00	1.03	1.07	1.00
addr27	1.04	1.33	1.00	1.08	1.47	1.00	1.03	1.22	1.00	1.07	1.37	1.01
addr29	1.04	1.29	0.99	1.10	1.40	1.00	1.01	1.11	0.85	1.09	1.29	1.01
addr30	0.69	0.96	0.59	1.02	1.05	1.00	0.69	0.97	0.59	1.02	1.05	1.01
addr32	0.69	0.96	0.60	1.02	1.06	1.00	0.68	0.96	0.57	1.02	1.06	1.00
addr59	1.02	1.35	0.86	1.07	1.43	1.00	1.00	1.36	0.85	1.06	1.35	1.00
addr61	1.01	1.35	0.86	1.05	1.11	1.00	0.97	1.34	0.82	1.05	1.11	1.00
speedups for host `p4-64`												
addr26	1.03	1.13	1.00	1.05	1.13	1.01	1.02	1.13	1.00	1.05	1.13	1.01
addr27	1.12	1.29	1.01	1.18	1.41	1.03	1.11	1.29	1.01	1.17	1.41	1.03
addr29	1.12	1.29	1.01	1.17	1.32	1.02	1.08	1.29	0.78	1.16	1.32	1.02
addr30	0.80	1.01	0.69	1.10	1.19	1.01	0.78	1.02	0.65	1.09	1.19	1.01
addr32	0.81	1.00	0.67	1.10	1.26	1.01	0.78	0.99	0.58	1.09	1.18	1.01
addr59	1.14	1.33	0.83	1.15	1.40	1.04	1.09	1.32	0.81	1.14	1.38	1.04
addr61	1.11	1.31	0.81	1.12	1.33	1.02	1.06	1.30	0.71	1.11	1.33	1.02
speedups for host `xeon-32`												
addr26	1.08	1.38	1.00	1.24	1.56	1.00	1.08	1.38	1.00	1.23	1.56	1.00
addr27	1.03	1.36	0.77	1.54	2.13	1.27	1.02	1.36	0.77	1.53	2.13	1.25
addr29	1.03	1.23	0.75	1.48	1.74	1.12	1.00	1.23	0.73	1.47	1.74	1.11
addr30	0.69	0.99	0.39	1.20	1.58	1.03	0.69	0.99	0.40	1.19	1.58	1.03
addr32	0.73	0.89	0.45	1.27	1.51	1.04	0.71	0.89	0.45	1.27	1.51	1.04
speedups for host `coreduo-32`												
addr26	1.00	1.04	1.00	1.08	1.44	1.01	1.00	1.03	1.00	1.08	1.42	1.01
addr27	1.02	1.24	0.98	1.13	1.43	1.05	1.01	1.16	0.94	1.13	1.40	1.05
addr29	1.01	1.19	0.97	1.14	1.43	1.02	0.98	1.04	0.75	1.13	1.40	1.05
addr30	0.64	0.98	0.55	1.04	1.12	1.01	0.63	0.97	0.55	1.04	1.12	1.01
addr32	0.64	0.99	0.55	1.04	1.13	1.00	0.63	0.95	0.53	1.04	1.13	1.00

Table 8: Speedup results using the best options for each benchmark.

Embedded Systems, pages 271–281. ACM Press / Sheridan, October 2006.

[6] R. Cartwright and M. Fagan. Soft Typing. In *Programming Language Design and Implementation (PLDI 1991)*, pages 278–292. SIGPLAN, ACM, 1991.

[7] A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *FLOPS'06*, Fuji Susono (Japan), April 2006.

[8] B. Demoen and P.-L. Nguyen. So Many WAM Variations, So Little Time. In *Computational Logic 2000*, pages 1240–1254. Springer Verlag, July 2000.

[9] ECRC. *Eclipse User's Guide*. European Computer Research Center, 1993.

[10] M. Hermenegildo, G. Puebla, F. Bueno, and P. L. García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[11] S. Marlow, A. R. Yakushev, and S. P. Jones. Faster Laziness Using Dynamic Pointer Tagging. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 277–288, New York, NY, USA, 2007. ACM.

[12] J. Morales, M. Carro, and M. Hermenegildo. Towards Description and Optimization of Abstract Machines in an Extension of Prolog. In G. Puebla, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'06)*, number 4407 in LNCS, pages 77–93, July 2007.

[13] J. Morales, M. Carro, G. Puebla, and M. Hermenegildo. A Generator of Efficient Abstract Machine Implementations and its Application to Emulator Minimization. In M. Gabbrielli and G. Gupta, editors, *International Conference on Logic Programming*, number 3668 in LNCS, pages 21–36. Springer Verlag, October 2005.

[14] A. Rigo. Representation-based Just-In-Time Specialization and the Psyco Prototype for Python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 15–26, New York, NY, USA, 2004. ACM Press.

[15] V. Santos-Costa. Optimising Bytecode Emulation for Prolog. In *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *LNCS*, pages 261–277. Springer-Verlag, 1999.

[16] V. Santos-Costa, L. Damas, R. Reis, and R. Azevedo. *The Yap Prolog User's Manual*, 2000. Available from `http://www.ncc.up.pt/~vsc/Yap`.

[17] Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog 3.8 User's Manual*, 3.8 edition, Oct. 1999. Available from `http://www.sics.se/sicstus/`.

[18] D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.