Tagged Representations in WIL

by

Daria Craciunoiu

A senior thesis submitted in partial fulfillment of

the requirements for the degree of

Bachelor of Science
With Departmental Honors
Computer Science & Engineering
University of Washington
August 2007

Presentation of work given on **January 18, 2006**

Thesis and presentation approved by _____

Date _____

Daria Craciunoiu

Tagged Representations in WIL

1. Introduction

High level programming languages such as ML and Smalltalk have a multitude of features designed to make a programmer's life easier. One of the remarkable foundational features is uniform data structure representation as heap-allocated structures references indirectly via pointers - also known as *boxed representations*. Among the advantages of boxed representations we can count:

- uniformity

  Programmers can treat all first class objects uniformly as pointers. They can implement generic data structures such as containers (e.g. stacks) that will behave identically regardless of their content.

- subtyping

  Programmers can implement generic mechanisms for handling data structures at the top level of a hierarchy and then rely on dynamic dispatch if necessary to specialize messages at any sublevel.

- sharing of mutable data

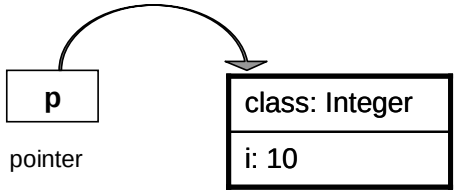
  Program variables contain implicit references to objects. Assignment or parameter passing preserve the identity of the target object because they copy the reference instead of the referenced object.

There is however a price to pay in efficiency for these software engineering advantages of boxed representations:

- cost of indirect access to objects via pointers
- cost of allocation and deallocation
- cost of storage space (overhead of a pointer for every object)

There is a more efficient data layout alternative to boxed representations: inline representations also known as *unboxed representations*. In this case we can reverse the discussion of advantages and disadvantages evaluated for boxed representations without reaching the optimal result of preserving flexibility and increasing efficiency.

The pursuit of this efficient flexibility goal has created languages with a hybrid data model such as Java. Java allows the representation of pointers and objects such as Integer objects alongside the representations of inline primitive values such as int.



Values of type Integer are boxed.
Values of type int are unboxed.

Figure 1. Java memory model.

The hybrid data model has its benefits, but it also foregoes a generic way of manipulating all data values and it makes generic code only conditionally reusable. We cannot implement a Stack class with uniformly represented numeric content regardless of the content type because an int value is not referred to by a pointer like the Integer class or any other subclass of Object. We cannot take advantage of dynamic dispatch for int values either. In order to address some of these issues Java implemented autoboxing to automatically convert between primitive values and their appropriate wrapper classes. This solution further complicates the data model and reduces efficiency.

A different approach is the implementation of tagged representations which will be discussed in this paper. Tagged representations aim to optimize the manipulation of commonly occurring kinds of values such as integers by representing these values inline in place of the pointer. In order to distinguish these kinds of values from normal pointers the low order bits of a variable's content are used to encode the interpretation of the high order bits. Tag values can be chosen to minimize tag manipulation overhead. In this research project the support for tagged (pointer and integer) representations is implemented in the Diesel front end of the Whirlwind compiler. The result was a significant increase in the efficiency of arithmetic operations performed on integers. The following sections will present the

environment and implementation of the project, as well as discuss benchmarking results and mention previous work related to tagged representations.

2. Tagged Representations

2.1. Environment: WIL/Whirlwind

The environment for studying tagged representations in this project is the Whirlwind compiler developed by Dr. Craig Chambers and the WASP research group at the University of Washington. Whirlwind is a multilingual optimizing compiler written in the Diesel experimental programming language. The Java and Diesel front ends shown in Figure 2. are currently functional, while the C++ front end is used for illustration purposes at this point.

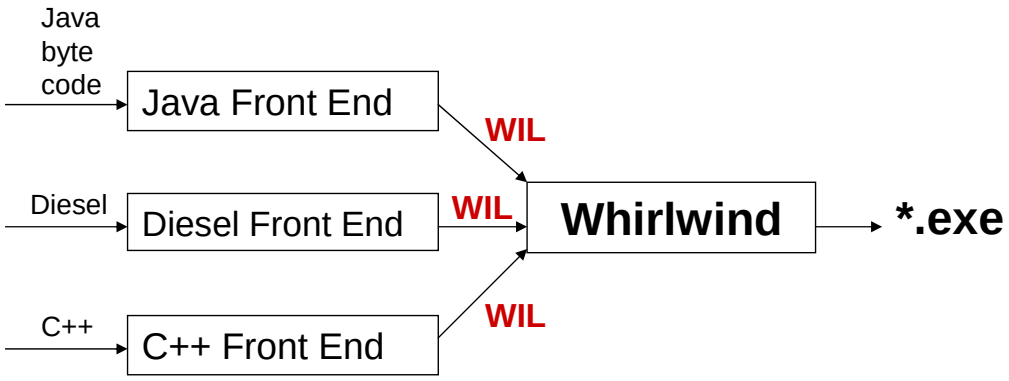


Figure 2. Whirlwind.

Whirlwind features an object-oriented explicitly typed intermediate language named WIL into which other languages are first translated. In addition to the source languages supported we can also write WIL code directly when working with Whirlwind.

WIL has similar constructs to other programming languages such as C/C++, including the following:

- primitives (integers, floats etc.)

- pointer types and operations
- records
- objects (which are records with an implicit class field)
- class declarations

  class Point isa Object;
- variable declarations

  decl p:Point* :== new_point(3,4) ;
- function declarations

  fun f (x:int): void {…}

From the two source languages currently supported by Whirlwind, Diesel was the ideal candidate for this project because it has a uniform pointer model. The following example shows a sample Diesel to WIL translation simplified on the WIL side[1]:

Diesel	WIL
abstract class Object;	abstract class Object_class; ...
2 class Cell isa Object;	concrete class Cell_class inherits Object_class; … rep Cell_rep := * object Cell_class;
field contents(:Cell):int;	rep Cell_rep_fields :== {var Cell_contents:oop_rep, ...}; … rep Cell_rep_target :== object Cell_class Cell_rep_fields;

fun new_cell(x:int):Cell {
new Cell { contents := x }
}
	fun new_cell_(generic _rep):generic_rep {
	decl t1 :== x;
	decl t2 :== new Cell_rep_target;
	*t2.Cell_contents :== t1;
	return t2;
	}

let c := new_cell(10);	decl c :== new_cell_(new_prim_int(10));

[1]Some complicated WIL syntax and constructs have been omitted. Among other things this code does not reflect that WIL implements generic functions and methods along with functions for a dynamic dispatch mechanism.

Diesel	WIL

	--an int represented in boxed format
	fun new_prim_int(v:1 word):generic_rep {
	  decl r :== new G_prim_int_rep_target :==
	  { G_prim_int_G_value :== v };
	  return r;
	}

The code example presented shows how we can declare and instantiate a simple Cell class with an integer field. A sketch of the memory layout is presented in Figure 3:

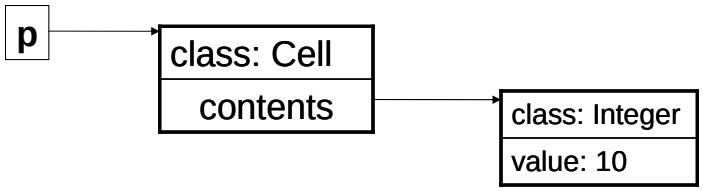


Figure 3. Diesel pointers.

The primitive value for the contents of this Cell object is represented indirectly in boxed format. A more efficient approach would be to find a way for inlining the value. The next section describes the tagging idea.

2.2. Tagging

Tagging in the context of this research project is focused on optimizing the data layout and manipulation of integer values. For all WIL representations the two low-order bits in the content of a variable indicate the interpretation of higher order bits. Figure 4 revisits the original Diesel memory model, making the tag bits explicit:

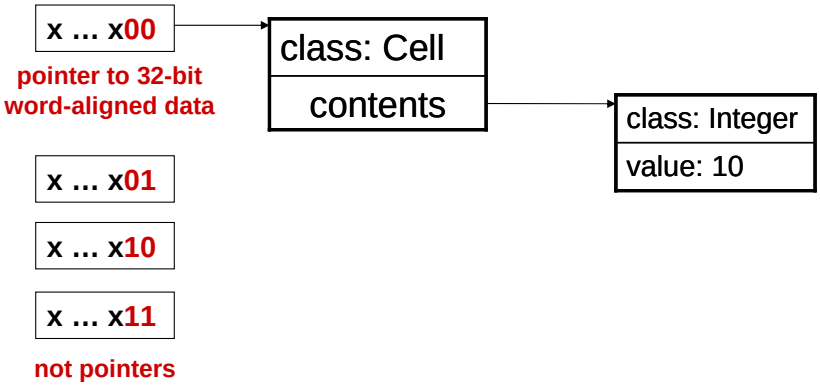


Figure 4. Diesel pointers with tag bits explicit.

We can choose the three non-pointer formats to encode some kinds of values inline, e.g. integers. In fact, we don't need to use the 00 tag for pointers. In this work, we choose to represent pointers with a 11 tag, and integers with a 00 tag, as described below:

- **pointers: 11**

  The low order bits of a pointer to 32-bit word-aligned data are 00 so using these bits for a different purpose does not "steal" any bits from the address. When accessing a field of an object pointed to by a tagged pointer, since field offsets are known at compile time, we can subtract the tag from the offset at compile time. This operation helps avoid any run-time tag manipulation overhead.

  Example: **p.x**

  - untagged: *(p + x.offset)
  - tagged: *(p + (x.offset-tag))

- **integers: 00**

  Integers that are 30-bits or shorter receive a 00 tag by a simple left shift. Because tagging "steals" two bits from integer variables, the tagging code detects overflow above the 30-bits length and encodes the numbers in boxed format as tagged pointers to objects. The ability to combine data layout representations according to need shows this project optimizes the common case without losing any of the flexibility already existent in the Diesel memory model. The choice of a 00 tag value for integers allows operations such as addition, subtraction and comparisons to be performed without untagging. For operations like division and multiplication untagging is implemented as a right shift and it is fairly efficient.

2.3. Method/Implementation

Adding tagging support to Whirlwind was done in a minimally invasive way. The Diesel front end was modified to emit slightly different WIL code, but no changes to Whirlwind were necessary. The three components of the implementation are discussed below:

- **model for tagged data representation**

A new declaration was added to represent and propagate tagged values throughout the program. This is intuitively analogous to declaring a new type of value:

```
rep boxed_rep :== * class Object;
rep tagged_rep :== word;
```

- **WIL support for manipulating tagged values**

Originally Whirlwind used a file named *prims.wil* to store WIL code implicitly included by any Diesel program translated to WIL. This file now has two versions: the original version and the tagging specific version. The tagged version of the file uses and propagates parameters of tagged_rep type. Similar modifications had to be extended to a few files containing WIL primitives in the Diesel standard library. Functions for tagging, untagging and manipulating tags had to be added to the already existing WIL code library. The following example is simplified WIL code used for tagging and untagging a pointer:

```
--number of low-order bits used to tag a value
fun num_tag_bits(): word {
      return 2;
}

--define a magic number derived from the number of tag bits
fun get_mask(): word {
      return (1 << num_tag_bits()) - 1;
}

--compute pointer tag (low-order bits = 11)
fun ptr_tag():word {
      return get_mask();
}

---test if value is tagged pointer (low-order bits = 11)
fun is_tagged_ptr(v: tagged_rep):bool{
      return (v & get_mask()) = ptr_tag();
}

--tag pointer (low-order bits = 11)
fun ptr_to_tagged(v: boxed_rep):tagged_rep{
      return (cast v as tagged_rep) | ptr_tag();
}
```

```
--retrieve value of tagged pointer by removing tag
fun tagged_to_ptr(v:tagged_rep):boxed_rep{
      if is_tagged_ptr(v) goto value;
      fatal "tagged value is not a pointer";
      label value;
      return cast (v & ~get_mask()) as boxed_rep;
}
```

The following sample code handles similar tag manipulations for integers:

```
--compute int tag (low-order bits = 00)
fun int_tag():word {
      return 0;
}

--test if value is tagged integer (low-order bits = 00)
fun is_tagged_int(v: tagged_rep):bool{
      return  (v & get_mask()) = int_tag();
}

--create a tagged int or tagged pointer to int depending on size
fun new_prim_int(v:word):tagged_rep{
      decl result:tagged_rep :== (v << num_tag_bits()) | int_tag();
      decl orig:word :== tagged_to_int(result);

      if v = orig goto ok;
      return ptr_int_helper(v);
      label ok;
      return result;
}

--returns a tagged pointer to int object like the original model
fun ptr_int_helper(v:1 word):tagged_rep {
      decl r :== new G_prim_int_rep_target :== { G_prim_int_G_value :== v };
      return ptr_to_tagged(r);
}

--retrieve value of tagged int
fun tagged_to_int (v:tagged_rep):word{
      if is_tagged_int(v) goto value;
      fatal "tagged value is not an int";
      label value;
      return v +>> num_tag_bits();
}
```

- **Diesel front end modifications for translating input to optimized WIL**

The *tag_values* option added to the compiler can be enabled and it will indicate to the Diesel front-end the generated WIL code needs to handle program variables as tagged representations. The tag manipulation functions will get inlined by Whirlwind, so there will be no function call overhead in the final code.

Some of the main modifications for generating tagged WIL code are in the areas of: objects creation (tag every object upon creation), class analysis (started by tag examination to differentiate integers from pointers), field access (untag before accessing fields through pointers).

3. Results

There is a significant speed-up in integer arithmetic operations when the tagging optimization is used. The micro-benchmark discussed in this paper focuses on measuring integer addition speed-up. If we consider a program that performs addition a number of times *n* we can compare the performance of the program for different values of *n* in tagged and untagged versions. A *generic_rep* type of value has been declared to represent either tagged or boxed values, depending on an optimization flag set at compile time to indicate whether or not the tagging optimization should be used.

This is the WIL function used to perform integer addition:

```
fun add(l:generic_rep, r:generic_rep):generic_rep {
   return new_prim_int(
      generic_to_int(l) + generic_to_int(r)
      );
}
```

In the case of tagged optimizations the addition operation can be optimized from its generic format by forcing the elimination of untagging/tagging shifts:

```
fun optimized_add(t1:tagged_rep, t2:tagged_rep):tagged_rep {
   return (t1+t2);
}
```

In order to measure the cost of generic addition and ignore the overhead of function calls etc., we can normalize results by performing tests calling the identity function instead of additions for both tagged and untagged versions at each considered value of *n*:

```
fun identity(i:generic_rep):generic_rep {
   return i;
}
```

The following tables present the results (reported in milliseconds) computed as the median of 11 time test results for each of the function versions described:

Table 1. Untagged Operations (ms)

Iterations (x 10 000 000)	0	1	5	10	15	20
Untagged Identity	464	596	812	1108	1344	1652
Untagged Addition	464	1028	3408	6252	9080	11960
Untagged Addition Cost	0	423	2596	5144	7736	10308

Table 2. Tagged Operations (ms)

Iterations (x 10 000 000)	0	1	5	10	15	20
Tagged Identity	72	144	372	660	956	1234
Tagged Addition	72	176	584	1116	1648	2116
Tagged Addition Cost	0	32	212	456	692	882
Optimized Tagged Addition	72	144	400	740	1064	1400
Optimized Addition Cost	0	0	28	80	108	166

The results for addition cost are visually summarized in Figure 5.

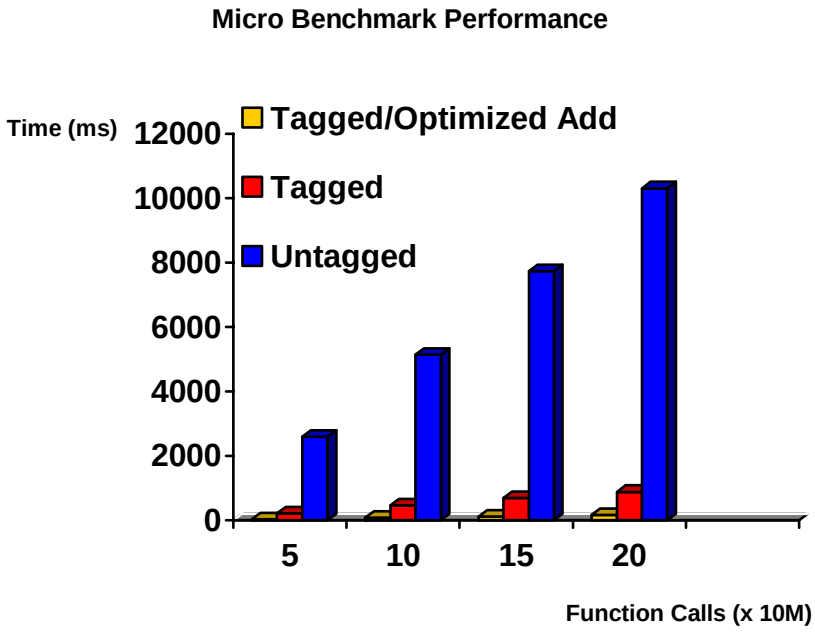


Figure 5. Benchmarking Results

The cost of generic tagged addition is only about 8.7% of the cost of untagged addition. This number is computed by comparing the slopes of linear fits computed for each of the data series:

- the linear fit for untagged additions is `time = 5.60 + 515.12*n`

From the equation above we identify the slope $m_{untagged}$ = 515.12.

- the linear fit for tagged additions is `time = -0.40 + 44.88*n`

From the equation above we identify the slope m_{tagged} = 44.88.

The ration of the slopes m_{tagged} /$m_{untagged}$ = 0.0871.

From the graph in Figure 5 we can observe an even more dramatic speed-up when tagged addition is optimized. The linear fit for optimized tagged additions is `time = -6.00 + 8.24*n` thus $m_{otagged}$ = 1.6. The cost of optimized tagged addition is about 1.6% of the cost of untagged addition.

4. Future Work

More things to explore in this project would be:

- extend tagging to other kinds of values such as characters and floats

There are more tags available and it is possible to expand the set by extending the tagging levels (2 level tagging etc.).

- move the implementation of tagging mechanisms from the Diesel front end to the Whirlwind compiler back end to make this optimization available to all source languages and integrate better with other Whirlwind optimizations such as class analysis

5. Related Work

Various researchers have explored both the method for implementing and the policy for applying representation optimizations before.

EuLisp implements an extensible model for describing data representations (Quinnec & Cointe 88). It is the stated intention of the authors to replace Objects by entities with a wider spectrum of physical representation. They define entities as sequences of bits from which a unique type can be retrieved. In EuLisp instantiation is possible either as entity (which actually allocates a manageable entity in memory) or as slot (which is an encoding of the size needed to hold a value of the instantiated type).

The SchemeXerox compiler provides a user programmable interface for describing data layouts (Adams et al. 93). To bypass the slow data structure operations cost associated with the general and modular type system implementation the compiler uses a series of optimization techniques among which are programmer supplied inlining declarations.

Policy decisions explored before include local or intra-procedural dataflow analysis to identity the possibility of representation optimizations such as the primitive inlining mechanism implemented in the SELF programming language (Chambers et al. 89). A different approach is allowing user annotations to drive unboxing techniques (Peyton Jones & Launchbury 91). Along with turning “unboxed” into a keyword at the programmer’s disposal the authors extend a language and its type system to handle unboxed values. Optimizations involving unboxed values become correctness preserving program transformations.

6. Retrospect

Undergraduate research is the most valuable part of my Computer Engineering education. I started working on research with the hope of finding my own definition of success in a time when I didn’t seem to fit any definition of success in my department. Knowing that I would always be more interested in the mechanisms of international economy and the welfare effects of globalization than I will be interested in architecting the next operating system, I chose to study Computer Engineering with the main purpose of acquiring a new perspective and new skills to analyze the business world with. My first Computer Engineering classes showed my learning pace was too slow at times and my thoughts were not always well technically articulated. By that time I liked everything I knew about Computer Engineering enough to continue studying it regardless of the challenge, and my results greatly improved by the end of my undergraduate career.

I chose my research project because I was particularly intrigued and intimidated by my Programming Languages class and thus compilers by association. I wanted to learn more about something I didn’t quite understand and I wanted to face the fear of

failure. My definition of success in research was shaped by every new concept that became clear and every small milestone where some of my code produced a usable result. Towards the end of my project I realized that in my eyes the greatest accomplishment was not any individual result along the way, but simply the fact that I never gave up and walked away. The greatest help was my research adviser's patience because he allowed me to come back and try again even when I didn't perform as expected. I am very grateful for everything I learnt about compilers, programming languages and software engineering concepts, but even more grateful for discovering my character managed to grow some roots of perseverance and confidence in a vision of success as multifaceted as the world of students who ever attempted to find their way around a research project.

7. References

[Adams et al. 93] Norman Adams, Pavel Curtis, and Mike Spreitzer. First-Class Data-Type Representations in SchemeXerox. In *Proceedings of the ACM SIGPLAN O93 Conference on Programming Language Design and Implementation*, pages 139-146, June 1993.

[Chambers et al. 89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 49-70, New Orleans, LA, October 1989.

[Chambers 99] Craig Chambers. Representation Specification and Optimization in Object-Oriented Languages, November 1999.

[Peyton Jones & Launchbury 91] SimonPeyton Jones and John Launchbury. Unboxed Values as First-Class Citizens. In *Proceedings of the Fifth Conference on Functional Programming Languages and Computer Architecture*, pages 636-666, September 1991.

[Queinnec & Cointe 88] Christian Queinnec & Pierre Cointe. An Open-Ended Data Representation Model for EuLisp. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 298-308, Snowbird, UT, July 1988.