LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Novel 64 Bit Data Representation for Garbage Collection and Synchronizing Memory

Computation Structures Group Memo 396
April 9, 1997

Alejandro Caro
`acaro@abp.lcs.mit.edu`

This research was conducted at the MIT Laboratory for Computer Science. Funding for this project was provided in part by the Advanced Research Projects Agency of the Department of Defense under Ft. Huachuca contract C-DABT63-95-C0150.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Novel 64 Bit Data Representation for Garbage Collection and Synchronizing Memory

Alejandro Caro
acaro@abp.lcs.mit.edu

April 9, 1997

Abstract

We have describe a novel 64-bit data representation that facilitates the implementation of garbage collection and synchronizing memory. The representation takes advantage of the structure of IEEE-754 Not-a-Numbers (NaNs) to enable fast pointer/non-pointer and full/empty/deferred tests with minimal loss in the range of scalar representations.

1 Introduction

This research note describes work in the pH (parallel Haskell) compiler pHc. *The presentation is informal and was written to make the ideas available to others as quickly as possible. We assume the reader is familiar with the issues involved in the implementation of a garbage collector and synchronizing memory.*

The *pH* language implementation requires automatic memory management via garbage collection. A garbage collector, and in particular a *copying* garbage collector, must be able distinguish data values of different types as it traverses memory. On specialized machines like Symbolics Lisp machines, the architecture provides *tag bits* for every word so that types can be encoded separately from data. In standard architectures, no such support exists, so types must be encoded directly in the data values. We have developed a novel data representation that encodes tag information with far less negative impact on data range/precision than existing approaches. The representation is also useful for implementing I-structures and M-structures, two types of per-element synchronizing memory supported by *pH*.

The standard approach to encoding type information is to steal a few bits from the data representation. For example, the GNU Emacs Lisp interpreter supports 24-bit integers even though the native machine integer is 32 bits. The remaining 8-bits are used for type information by the interpreter. The precision of floating point numbers would suffer dramatically if an implementation were to steal bits from their representation. Instead what is usually done is to "box" floating point numbers: they are allocated on the heap and are represented by pointers. Boxed floats are terribly inefficient due to the additional memory operations they entail.

It is important to note that Lisp systems implement run-time type checking and require tag bits even if the garbage collector is turned off. The *pH* language is statically type-checked, so tag bits are not strictly necessary, even for garbage collection. There has been previous work in implementing full tagless garbage collection, but unfortunately, this approach requires support in the compiler front-end. Compiler implementors like us who do not want to modify the front-end (yet!) must resort to tags, though the amount of tag information required is dramatically less than in a Lisp system.

2 A Novel Data Representation

We established a number of requirements for the data representation of our compiler:

1. All scalars must be the same size (in bits) to allow implementation of polymorphic functions.
2. Hardware (IEEE-754) floating point range must not change.
3. Floating point numbers must not be boxed.
4. Integer/pointer range can change, but loss of range should be kept to a minimum.
5. The representation should make it easy to distinguish among integers, pointers, and floating point numbers.
6. The representation should make it easy to distinguish full, empty, or deferred I-structure memory locations.
7. The representation should not add significant overhead to basic arithmetic operations.

To the best of our knowledge, there exists no other work that tackles this problem in the same way as we did. Unfortunately, the garbage collection literature is quite fragmented, so we encourage readers to contact the author if they know of similar work in this area. The following sections describe the decisions we took.

Scalar Size Our data representation requires that all scalar quantities be 64-bits wide. In other words, integers and pointers will be 64-bits wide and floating point numbers will be IEEE-754 double precision. The rationale for this choice is simple: polymorphism is hard to implement without a fixed data size, 32-bit floats are too small for many scientific problems, and 64-bit machines are already on the market or can be simulated easily via compiler support (i.e. `long long` types in C).

Integer/Pointer Bit Patterns The fundamental problem that must be addressed in meeting the other requirements is that by encoding tag information into data one cannot represent some otherwise legal values. Consider all patterns of 64 bits. All of these patterns are valid integers (signed or unsigned) and in most cases, all of these patterns are also valid pointers (there are some limitations imposed by different VM systems). In general, looking at a bit pattern, one cannot recognize whether it is an integer or a pointer, hence the need for tags. In addition, there is no room in the "space" of bit patterns to encode additional information. The entire space is mapped to valid integers and pointers.

Floating Point Bit Patterns The key observation is that all 64-bit patterns are *not* valid floating point numbers. The IEEE-754 standard defines that positive and negative double precision numbers with an exponent field of all ones and a non-zero mantissa are Not-a-Numbers (NaNs). NaNs are invalid floating point numbers and result from illegal computations such as arithmetic on infinities and computations on NaNs. How many NaNs are there? Lots! It turns out there are roughly 2^{53} double precision NaNs.

NaNs, Integers, and Pointers A particularly interesting part of the NaN "space" are those NaN bit patterns with a sign bit of 0. These bit patterns are identical to those of the largest positive integers in signed twos-complement representation. More precisely, the bit patterns for "positive" NaNs correspond exactly to largest 0.024% (1/4096) of the integer range. We propose to remove these large positive integers from the range of integers than can be represented in our implementation of the *pH* language. In other words, the `Int` type in *pH* will represent the following range of integers: $[-2^{63}, 2^{52}-1]$.

What will we do with this 52-bit space of "illegal" large integers? We will use it to represent pointers. If possible, we will align the heap managed by the *pH* run-time system to correspond to the 52-bit space starting at virtual address `0x8FF0000000000000`. This alignment in VM is not absolutely necessary, but it does make the implementation of our data representation particularly simple. In this representation, the size of the heap space decreases from 64 bits to 52 bits. We feel this is not unduly restrictive since the current world production of DRAM is about 2^{52} bytes.

We divide the 52-bit heap into two subspaces: the full area and the deferred area. We also designate a location within one of these areas (it does not matter which area) to serve as the "empty value". This pointer represents a value that has not been computed and is crucial for implementing the I-structure protocol. The full area contains data structures that have been allocated but whose contents may not be computed. These data structures can contain the empty value or pointers to the deferred area. The deferred area contains the data structures to implement "deferred lists". These lists are used to keep track of threads waiting for the computation of a particular value.

3 Pros/Cons

So far we have presented the "mechanics" of our data representation. Now, consider an arbitrary 64-bit pattern in this representation with respect to the following criteria:

Integrity of Floating Point Numbers Floating point numbers are not altered at all by the new data representation. Full precision is maintained and floats can be represented directly rather than via boxing.

Pointer/Non-Pointer? If the bit pattern, when interpreted as a signed integer, is less than `0x8FF0...`, then it is an integer or a floating point number. Otherwise, it is a pointer. For the purposes of garbage collection, we do not need to distinguish between integers and floating point numbers. Thus, the pointer/non-pointer test involves two instructions.

Empty, Deferred, Full? Once it is determined that the bit pattern is a pointer, only one or two additional comparisons are required to determine the synchronization state of the memory location.

Overhead in Arithmetic Operations Arithmetic operations can be implemented directly without additional any untagging/retagging overhead. Overflow handling does change and is discussed below.

In summary, our representation can unambiguously represent pointers, integers, floating point numbers, and synchronization information. The tests used to determine the identity of a particular pattern involve a handful of instructions that execute quickly. Basic arithmetic operations are unaffected.

There seem to be some disadvantages to our data representation. As we have mentioned earlier, we reduce the size of the integer space. But unlike other representations, the reductions are minimal.

The range of GNU Lisp integers in about 0.4% of the original range (24 bits vs. 32 bits). The range of integers in our representation is 99.976% of the original. This is a big improvement!

The space of pointers, however, is reduced dramatically in our representation, from 64 to 52 bits. But we believe strongly that this is a minor incovenience. Large machines, even parallel machines, are very far away from supporting 52-bit addressing of real memory. Furthermore, if we regard VM space as a large namespace, 52 bits still provide lots of room for clever naming schemes.

One might argue that an additional disadvantage of our representation is the need to handle overflow in integer arithmetic. If unhandled overflows occur in this data representation, the results might be interpreted as pointers rather than as negative numbers, as is the case in C code. It is hard to argue that one sort of incorrect answer is preferable to any other sort, so we believe most users will not be concerned about running code with overflow checking disabled. If users do enable overflow checking in the compiler, the cost of such code is perhaps one instruction more expensive that in a compiler that uses an untagged data representation. We consider this additional cost insignificant.

4 Conclusions

We have described a novel 64-bit data representation that facilitates the implementation of garbage collection and synchronizing memory. The representation enables fast pointer/non-pointer and full/empty/deferred tests with minimal loss in the range of scalar representations. This representation was designed for the pHc compiler currently under development at the Computation Structures Group of the MIT Laboratory for Computer Science.

We would like to acknowledge Jan-Willem Maessen and Boon S. Ang for the significant contributions they made to early versions of this data representation.