Using Squids to Address Forwarding Pointer Aliasing

Abstract

Forwarding pointers allow safe and efficient data compaction. However, they also introduce a new source of aliasing and as a result can have a serious impact on program performance. In this paper we introduce short quasi-unique ID’s (squids), a simple hardware mechanism for avoiding the problems associated with aliasing. In the common case, squids can be used to prove that there is no aliasing and therefore avoid any overhead. The probability of having to perform an expensive dereferencing operation to check for aliasing when comparing pointers to different objects is exponentially small in the number of bits used to implement squids.

1 Introduction

Forwarding pointers are a conceptually simple architectural mechanism that allow references to a memory location to be transparently forwarded to another location. Known variously as “invisible pointers” [Greenblatt74], “forwarding pointers” [Moon85] and “memory forwarding” [Luk99], they are relatively easy to implement in hardware, and are an invaluable tool for safe data compaction ([Luk99], [Brown99]). Why, then, have they been incorporated into so few architectures?

One reason is that forwarding pointers have traditionally been perceived as having limited utility. Their original intent was fairly specific to accelerating the execution of LISP, and even some designers of LISP architectures chose not to implement forwarding pointers [Taylor86]. One common application of forwarding pointers not specific to LISP is incremental garbage collection ([Baker78], [Moon84]), but many other methods of garbage collection exist which do not make use of or benefit from forwarding pointers [Plainfossé95].
Only recently has it become apparent that forwarding pointers are indeed useful constructs that can expedite program execution. In [Luk99] it is shown that using forwarding pointers to perform safe data relocation can result in significant performance gains on arbitrary programs written in C, speeding up some applications by more than a factor of two. In [Brown99] an algorithm is given for performing asynchronous local compacting garbage collection in a massively parallel distributed system. This algorithm uses forwarding pointers to avoid the high run-time costs usually associated with such a system. Thus, there is growing motivation to include hardware support for forwarding pointers in novel architectures.

A second and perhaps more significant reason that forwarding pointers have received little attention from hardware designers is that they create a new set of aliasing problems. On an architecture that supports forwarding pointers, no longer can the hardware and programmer assume that different pointers point to different words in memory (figure 1). In [Luk99] two specific problems are identified. First, direct pointer comparisons are no longer a safe operation; some mechanism must be provided for determining the final addresses of the pointers. Second, seemingly independent memory operations may no longer be reordered in out-of-order machines.

**Figure 1:** Forwarding pointer indirection creates aliasing: two different pointers ($P_1, P_2$) can point to the same piece of data (D).
In this paper we will show how to address these problems using squids – Short Quasi-Unique ID’s. The idea is to assign every object a short random tag, stored in pointers to the object, which is similar in role to a UID, but is not necessarily unique. In the common case squids allow pointer comparison and memory operation reordering to proceed with no overhead. Only in rare cases is it necessary to degrade performance to ensure correctness. Thus, squids allow an architecture to support forwarding pointers with a minimal average run-time overhead. Furthermore, this overhead can be eliminated altogether if the software chooses not to make use of forwarding pointers.

The next section outlines some existing solutions to the problems caused by forwarding pointer aliasing. In section 3 we introduce squids, and in section 4 we outline the overhead required to implement squids. We conclude in section 5.

2 Previous Approaches

In [Luk99] the problem of pointer comparisons is addressed by inserting code to determine the final address for each pointer (unless the compiler can somehow determine that the pointers do not point to relocated objects). In our view this overhead is unacceptable. In the best case, both target memory words will be resident in the cache, neither of them will contain a forwarding pointer, and the pointer comparison will be slowed down by a mere order of magnitude. However, since pointer comparisons often precede a decision to perform operations on an object, a common case will be for one or both dereferences to cause a cache miss, slowing down the comparison by yet another order of magnitude. In the worst case, one of the pointers points to data which is not even resident in main memory, which can occur frequently in programs that
deal with massive datasets. Clearly, then, it is desirable to be able to compare pointers without having to dereference them.

The solution proposed in [Luk99] for reordering memory operations is to use *data dependence speculation*, which allows loads to execute speculatively before it is known that they are independent of any preceding stores. In an architecture that supports data dependence speculation, it is fairly easy to extend the hardware to operate correctly in the presence of forwarding pointers. In [Luk99] it was found that this solution is effective as incorrect speculation occurs only rarely. However, it assumes the presence of some fairly complex hardware. For architectures in which silicon area efficiency is a concern, it may be preferable to develop a lower cost alternative.

Both of these problems are instances of the more general challenge of determining object identity in the presence of multiple and/or changing names. This problem has been studied explicitly [Setrag86]. A natural solution which has appeared time and again is the use of system-wide unique object ID’s (e.g. [Dally85], [Setrag86], [Moss90], [Day93], [Plainfossé95]). UID’s completely solve the aliasing problem, but have two disadvantages:

i. The use of ID’s to reference objects requires an expensive translation each time an object is referenced to obtain the virtual address of the object.

ii. Quite a few bits are required to ensure that there are enough ID’s for all objects and that globally unique ID’s can be easily generated in a distributed computing environment. In a large system, at least sixty-four bits would likely be required in order to avoid any expensive garbage collection of ID’s and to allow each processor to allocate ID’s independently.
Despite these disadvantages, the use of ID’s remains appealing as a way of solving the aliasing problem, and it is tempting to try to find a practical and efficient mechanism based on ID’s. This is the motivation for squids.

3 Squids

We begin by noting that the expensive translations (i) are unnecessary if we adopt a style of capability-based addressing ([Fabry74], [Carter94]) which includes as part of the capability (henceforth referred to simply as a pointer) both the object address and the object ID. In this case we have the best of both worlds; object references make use of the address so that no translation is required, and pointer comparisons and memory operation reordering are based on ID’s which solve the aliasing problem. However, this still leaves us with disadvantage (ii), which also implies that the pointer format must be quite large.

We can solve the remaining size problems by dropping the restriction that the ID’s be unique. Instead of long unique ID’s we use short (say between eight and sixteen bits) quasi-unique ID’s (squids). At first this seems to defeat the entire purpose of having ID’s, but we make the following observation: while squids can’t be used to determine that two pointers reference the same object, they can in most cases be used to determine that two pointers reference different objects. If we randomly generate an $n$ bit squid every time an object is allocated, then the probability that pointers to distinct objects cannot be distinguished by their squids is $2^{-n}$.

3.1 Pointer Comparisons

We can efficiently compare two pointers by comparing both their addresses and their squids. Assume for the moment that all pointers point to object heads; we will relax this restriction in section 3.4. If the addresses are the same then the pointers point to the same object. If the squids are different then they point to different objects. In the rare case that the addresses are different
but the squids are the same, we trap to a software routine which performs the expensive
dereferences necessary to determine whether or not the final addresses are equal.

To see that this latter case is indeed rare, we observe that it occurs in two circumstances:
either the pointers reference different objects which have the same squid, or the pointers
reference the same object through different levels of indirection. The former of these occurs
with probability $2^{-n}$. The latter is application dependent, but we note that (1) applications tend to
compare pointers to different objects more frequently than they compare pointers to the same
object, and (2) the results of the simulations in [Luk99] indicate that it is reasonable to expect the
majority of pointers to migrated data to be updated, so that two pointers to the same object will
usually have the same level of indirection.

3.2 Reordering Memory Operations
In a similar manner, the hardware can decide whether or not it is possible to reorder memory
operations based on squids. Assuming still that pointers point to object heads, memory addresses
being read/written must be specified as \textit{pointer} + \textit{index}. If the squids are different, it is safe to
reorder. If the squids and pointer addresses are the same but the indices are different, it is again
safe to reorder. If the squids are the same but the pointer addresses are different, the hardware
assumes that the operations cannot be reordered. No data speculation is required, and the
probability of failing to reorder references to different objects is $2^{-n}$.

3.3 Improving Performance
The use of squids reduces the average overhead necessary to check for aliasing to a small but
still non-zero amount. Ideally, software that chooses not to make use of forwarding pointers
should be allowed to run with no overhead whatsoever. We can achieve this by adding a single
‘migrated’ bit (M) to the pointer format which indicates whether or not the pointer points to the
original address at which the object was allocated. When a new object is created, pointers to that object have \( M=0 \). When the object is migrated, pointers to the new location (and all subsequent locations) have \( M=1 \). If the hardware is comparing two pointers with \( M=0 \) (either as the result of a user comparison instruction, or to determine whether or not memory operations can be reordered), it can ignore the squids and perform the comparison based on addresses alone. Hence, there is no runtime cost associated with support for forwarding pointers if an application does not use them.

### 3.4 Pointers to Object Interiors

The preceding discussion assumed that pointers always point to object heads. However, in many cases it is desirable to support pointers to object interiors. For example, it is common practice in C programs to create an array of data structures and then work with pointers to individual structures within the array.

This can cause problems for squids. Consider the worst case in which a program allocates a single massive object containing as sub-objects all the data structures it will ever need. A single squid is created for this super object which will appear in every pointer used by the program. In this case it becomes a common occurrence to have two pointers to different objects with the same squid. As a result, virtually every pointer comparison will result in a trap, and no memory references will be reordered. The program will still execute correctly, but performance will suffer horribly.

We can solve this problem by incorporating a mechanism for extracting the base address of an object from pointers to the object’s interior. This allows the hardware to determine when two pointers point to different words of the same object; when the pointers are decomposed as \( \text{base} + \text{offset} \), the hardware will see that the base addresses are the same, but the offsets are
different. We note that it is possible to implement such a mechanism with an efficient pointer representation and very little hardware, e.g. [Carter94].

4 Hardware and Software Overhead

The only software overhead required to support squids is the code that generates them when objects are allocated, which adds a few instructions to memory allocation. A trap handler is needed to check for aliasing when comparing different addresses with the same squid, but this code (or an equivalent hardware mechanism) is a general requirement for supporting forwarding pointers without UIDs and is not specific to the implementation of squids. Moreover, placing a single copy of this code in a trap handler creates much less software overhead than inlining the code at every pointer comparison as in [Luk99].

Since squids require only a small number of bits, they can be incorporated directly into a 64 bit pointer without actually adding any register/datapath bits. For example, the high bit can be used for M, the next eleven bits for the squid, and the remaining 52 bits for the actual address. Hence, the only hardware required to implement squids consists of some simple logic to inspect squid bits for pointer comparisons and memory operation reordering, and support for the comparison trap. If the architecture and programming model support pointers to object interiors, then additional logic is required for computing base addresses.

5 Conclusion

Forwarding pointers are a key enabling mechanism for safe data compaction and efficient garbage collection. Hardware support for fast garbage collection is especially important given the growing prevalence of the Java programming environment, which specifies a garbage collected memory model [Gosling96]. It is therefore becoming increasingly desirable to
incorporate forwarding pointer support into new architectures, but in order to do so it is necessary to address the aliasing problems that arise.

We have presented squids, a novel approach to the problem of forwarding pointer aliasing. The hardware and software required to implement squids are minimal. The use of squids dramatically reduces the cost of checking for aliasing; the probability of having to perform expensive dereferencing operations when comparing pointers to different objects decreases exponentially with the size of the squids. It is therefore possible to provide practical and efficient hardware support for forwarding pointers.

References


