Arranging for Safety Checks with Hardware Traps

Göran Weinholt

March 2012

Abstract

Many programming languages are untyped and perform all type checks at run-time. This requires that type information for objects is kept around for as long as the objects are live. One way of achieving this involves tagging all pointers with the type of the object to which they refer. The compiler inserts safety checks before all object references in order to provide safe semantics and error reporting.

The run-time safety checks can however be expensive and compilers sometimes provide an option to use unsafe semantics. The resulting programs are faster and more compact, but there are some obvious drawbacks. This paper describes a technique whereby a compiler can generate code that arranges for the processor to perform safety checks. The technique is usable on commodity hardware.

I. Introduction

When a programming language is untyped it becomes necessary to keep object type information around at runtime. Some garbage collectors can also require the use of runtime type information [1]. One method for storing this data is to add a header to all objects [1, 2]. Another common technique is to reserve some of the bits in machine words for use as tags [1, 3, 4]. The tag can either be in the high bits or in the low bits of the word. When the low bits are used the system relies on the fact that objects are allocated on word-aligned addresses. This makes two or three bits available to store a type [5].

Often one of the tags indicates that there is a small integer encoded immediately in the machine word [1, 6]. When the result of an arithmetic operation is small enough there is no need to allocate memory to store the result. These small integers are sometimes referred to as fixed-point numbers or fixnums [4, 3, 7]. On a machine with 64-bit words and three tag bits the fixnum range is quite reasonable.

In Lisp implementations one of the tags is often used for other miscellaneous types which can be stored immediately in machine words, e.g. characters and the empty list [5, 6]. The remaining tags can be used to indicate that the machine word contains a pointer to a specific type of object, e.g. a pair, a vector, a procedure or some other common type. One of the tags will need to indicate that the real type of the object is stored somewhere else [6].

There are some disadvantages to tagging machine words in this way. For some operations the tag must first be removed, a process known as unboxing [1]. The compiler must also insert explicit checks before using an object in order to guarantee memory-safe semantics. There have been several approaches to solving this problem. One approach is to use specialized hardware. The Symbolics 3600 had special hardware that performed type-checking in parallel with the normal operations of the processor [4]. Another example is the SPARC which has the TADDcc and TSUBcc instruction that work with fixnums [8].

Another approach comes from the software side. There are a number of techniques available that let the compiler generate code which does not need to do any unboxing or safety checking. The language can be extended with optional static typing and the compiler can use type inference [9]. The compiler can also provide different safety levels [10]. The default safety level generally guarantees memory-safe semantics. When the programmer is comfortable with the type safety of the program he can instruct the compiler to generate faster code by removing the safety checks.

This paper presents a novel technique that uses a combination of software and existing commodity hardware. The software arranges for the hardware to automatically perform the safety checks with virtually no overhead for programs without dynamic type errors.

II. Alignment Checks

The widely available Intel IA-32 and AMD64 architectures have an optional feature called alignment

Listing 1: Code with an explicit safety check

```
mov ecx, edx
and cl, 7
cmp cl, 2
jne error
mov rax, [rdx - 2]
```

checking. When a program performs an unaligned memory reference the processor will normally handle this transparently. By setting the Alignment Check (AC) flag in the processor's flags register all unaligned memory references will instead cause an exception to be raised [11, 12]. This feature can be used in combination with tags in the lower bits to make unsafe code behave according to memory-safe semantics.

The technique presented here relies on the use of tags in the low bits, as described in the introduction. When memory is allocated for a new object the system requests a free address and then adds the tag to the least significant bits of the address. Later when the object is used the code removes the tag using the architecture's built-in support for additions and subtractions in memory references.

Most compilers using this tagging scheme will generate explicit checks to make sure that the tag is correct. Listing 1 shows what the code might look like. Let's suppose that pairs have the tag 010_2 and fixnums have the tag 000_2. If the code expects a pair it first masks off the lower three bits, compares them to 010_2 and finally branches to an error handler if the tag does not match. If the procedure is passed a fixnum instead of a pair the tag will not match and it will be detected as an error. This method requires an extra register to temporarily hold the tag and there are now more instructions than should be necessary. These safety checks must in general be generated for every memory reference if memory-safe semantics are required (except when the compiler has inferred the type somehow).

When alignment checking is enabled this explicit tag comparison is not needed. Suppose that the code in listing 1 did not check the tag. If RDX is a pair object then subtracting 010_2 will result in an effective address where the lower three bits are zero, i.e. the memory reference is aligned. The compiler and the runtime work together to guarantee that iff the lower bits of a word are 010_2 then the word must point to a pair. If RDX is any kind of object other than a pair then the code will compute an effective address where the lower bits are not zero. The memory reference then becomes misaligned and the processor raises an exception.

Alignment checking is a natural part of a processor's operation. Misaligned memory references can affect multiple cache lines or multiple pages and this is handled transparently by the processor if alignment checking is disabled [13]. Using alignment checking does therefore not appear to incur any overhead as long as programs do not generate an excessive number of exceptions.

III. Applications

How widely applicable the technique is depends on whether the code is running in 32-bit mode or 64-bit mode. For 32-bit mode the use is rather limited because there are only four tags available and at least two of those tags would be needed for miscellaneous immediate and boxed objects. In the 64-bit mode there are eight tags available, so there are that many more opportunities.

The alignment checking technique has been implemented in an experimental compiler for the Scheme language [7], which generates machine code for the AMD64 instruction set. It stores type tags in the lower bits of machine words. The tags it uses identify objects that are fixnums, pairs, procedures, strings, vectors, bytevectors, records and miscellaneous immediate types. The rest of this section discusses how the technique is used in the experimental compiler.

Given that pair objects (conses) are tagged it is possible to implement `car` and `cdr` in a single instruction. This is the traditional unsafe implementation, but alignment checking ensures memory-safe semantics. Higher-order programs can also benefit from the technique. When the compiler generates code for calling a procedure with a statically unknown entry point (e.g. a closure) it normally needs to ensure that the object in the operator position really is a procedure. When procedure objects use one of the tags the check is done for free.

Let's look at a concrete example of when alignment checking is useful. At the cost of a slight complication in the error reporting routine it is possible to implement a safe `vector-ref` in only three instructions. The `vector-ref` procedure is Scheme's array referencing primitive: it takes a vector object and an index and returns the element at the given index. In listing 2 the RDX register contains the vector reference, RCX contains the index, and the return value is to be stored in the RAX register. The program could actually have any type of object in RDX and RCX.

This code looks unsafe, but it's not. The first machine word in every vector is a fixnum which represents the length of the vector. Following this

Listing 2: `vector-ref` using alignment checks

```
cmp rcx, [rdx - 6]
jnb error
mov rax, [rdx - 6 + rcx + 8]
```

Listing 3: Branchless and safe `vector-ref`

```
cmp rcx, [rdx - 6]
cmovnb rcx, rdx
mov rax, [rdx - 6 + rcx + 8]
```

are the elements of the vector. In this version of `vector-ref` the comparison and move from memory instructions have dual purposes. If alignment checking was disabled then this would be a decidedly unsafe implementation. It would only be making sure that the index is in bounds. There would be nothing to guarantee that the index actually is a fixnum or that RDX is a vector object.

When alignment checking is enabled the code in listing 2 becomes safe. The comparison instruction has the side-effect of checking that RDX is a vector object. The comparison together with the branch implements an unsigned comparison on the index, which is sufficient to verify that it is in bounds [14]. It does not yet check that the index is a fixnum, so if the branch to the error handler is taken it is possible that the index has some other type.

The last instruction in listing 2 loads the vector element from memory and it also has the side-effect of checking that the index is a fixnum. The displacement part of this memory reference removes the tag by subtracting 6 and adds 8 in order to skip over the length word. The compare instruction already verified that RDX is a vector, so it is known that the lower three bits of RDX are 110_2. This means that the only way that the last memory reference can be misaligned is if RCX is not a fixnum. The code does all the safety checks and references the vector only if the types are correct.

The traditional implementation of `vector-ref` needs three branches: one to check that the vector really is a vector, one to check that the index is a fixnum, and one to check that the index is in bounds. The bounds check can be done with only one compare and branch, but the author has observed that many implementations needlessly do two signed comparisons on the index. Using alignment checking therefore makes it possible to remove two or three branches per `vector-ref`.

Is a completely branchless version possible? Listing 3 shows the ultimate general `vector-ref`. The first and last instructions are the same as before. The branch is replaced by a conditional move. If the index is out of bounds then the conditional move overwrites RCX with the pointer to the vector and the memory reference in the last instruction will therefore be misaligned. One problem with this approach is that it does not account for how to accurately report the error. The original index is unavailable to the exception handler because it has been overwritten.

Listing 4: Code to enable alignment checking

```
pushf
or dword ptr [rsp],0x40000
popf
```

IV. Implementation details

Before entering the compiled code the runtime environment enables alignment checking. This is done with the code in listing 4.

After popf is finished the alignment check flag is set in the flags register. Any misaligned memory references performed by the program will now result in an alignment check exception. The Linux kernel translates this into a BUS signal, so a signal handler must be installed.

It is possible for the program to raise other exceptions as well. The AMD64 architecture defines a *canonical address form*. An address is canonical if the most significant bits are all zero or all one. How many bits are affected depends on how many bits of virtual address space a particular processor supports [11]. A program can easily construct a fixnum which corresponds to a non-canonical address and then pass it to `car`. The result is a general-protection exception which Linux translates into SEGV signal. It is also possible to trigger a stack exception if the code generator uses RSP or RBP in a trapping memory reference.

Linux allows user programs to distinguish between the different exception types by examining the third argument to the signal handler. This argument is a *ucontext_t* structure which contains the state of the program at the time of the exception. The *REG_TRAPNO* field has the vector number of the exception, which can be used to distinguish between general-protection, page fault, alignment check and other exceptions.

When the hardware finds an error the runtime needs to raise an appropriate condition. The signal handler will need to examine the state of the user program at the time of the signal to deter-

mine which operation it was trying to perform. One possibility is to generate a table which contains an entry for every instruction that can cause an exception. Another approach is implemented in the experimental Scheme compiler. The runtime uses a symbolic disassembler to find the instruction that caused the exception. From this it is trivial to find out if the instruction tried to reference a pair, a vector, etc.

It should also be noted that alignment checking requires operating system support. The AM bit in the CR0 register must be set and the processor must be executing in user mode [11].

V. Pitfalls

Using alignment checks may not be straightforward and there are some known drawbacks. When adopting this technique in an existing compiler it is necessary to vet the code generator and the runtime for any misaligned memory references.

Programs for the AMD64 architecture will in practice never run with alignment checking enabled. This means that the majority of existing code has not been checked for misaligned memory references, so alignment checks will need to be disabled before entering external code, e.g. before foreign function calls or even C library calls.

Alignment checking is an underutilized feature it might be missing in some processor implementations. This has been observed in valgrind [15] and QEMU [16]. There is no problem for QEMU when it runs with hardware support for virtualization. It would be necessary to do check at startup to see if the machine supports alignment checking. The code could perform an intentionally misaligned memory reference which is guarded by an exception handler. If the handler is not run the program would print an error message and exit. A production compiler will likely need to support machines that do not have alignment checking.

A soft type system has been shown to eliminate 90% of type checks [9]. It is conceivable that a compiler which uses soft typing would generate code with almost no opportunities for using the alignment check technique.

VI. Conclusion

Alignment checking can be used to perform some dynamic type safety checks in a manner that used to require specialized hardware. The actual effect achieved will depend very much on what other kind of analysis the compiler performs. The question of how it affects performance is therefore left open.

Implementing the technique in an existing system is non-trivial and would make the resulting code unsafe when run in some emulators. It also appears to be impractical on 32-bit systems.

VII. Acknowledgments

Christian Häggström came up with the branchless version of `vector-ref` after having seen the version with only one branch.

References

[1] Appel, A. W., "Runtime Tags Aren't Necessary," Tech. Rep. CS-TR-142-88, Princeton University, March 1988.

[2] Venstermans, K., Eeckhout, L., and Bosschere, K. D., "Java object header elimination for reduced memory consumption in 64-bit virtual machines," *ACM Trans. Archit. Code Optim.*, Vol. 4, September 2007.

[3] Steele Jr., G. L., "Data Representations in the PDP-10 MacLISP," Tech. Rep. AI Memo 420, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, September 1977.

[4] Moon, D. A., "Architecture of the Symbolics 3600," *Proceedings of the 12th annual international symposium on Computer architecture*, ISCA '85, IEEE Computer Society Press, Los Alamitos, CA, USA, 1985, pp. 76–83.

[5] Rees, J. A. and Adams IV, N. I., "T: A Dialect of Lisp, or: LAMBDA: The Ultimate Software Tool," *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, August 1982, pp. 114–122.

[6] Dybvig, R. K., Eby, D., and Bruggeman, C., "Don't Stop the BIBOP: Flexible and Efficient Storage Management for Dynamically Typed Languages," Tech. rep.

[7] Sperber, M., Dybvig, R. K., Flatt, M., van Straaten, A., Findler, R., and Matthews, J., "Revised^6 Report on the Algorithmic Language Scheme," *Journal of Functional Programming*, 2009, pp. 1–301.

[8] SPARC International, *The SPARC Architecture Manual Version 9*, Prentice Hall, 1994.

[9] Wright, A. K. and Cartwright, R., "A practical soft type system for scheme," *ACM Trans. Program. Lang. Syst.*, Vol. 19, January 1997, pp. 87–152.

[10] Verna, D., "How to make Lisp go faster than C," *IAENG International Journal of Computer Science*, Vol. 32, 2006, pp. 499–504.

[11] *AMD64 Architecture Programmer's Manual*, Advanced Micro Devices, 2011.

[12] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2011.

[13] *Software Optimization Guide for AMD Family 15h Processors*, Advanced Micro Devices, 2012.

[14] Warren, Jr., H. S., *Hacker's Delight*, Addison-Wesley, 2003.

[15] Nethercote, N. and Seward, J., "Valgrind: a framework for heavyweight dynamic binary instrumentation," *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, ACM, New York, NY, USA, 2007, pp. 89–100.

[16] Bellard, F., "QEMU, a Fast and Portable Dynamic Translator," *USENIX 2005 Annual Technical Conference, FREENIX Track*, April 2005, p. 41–46.