C Is *Not* a Low-level Language

DAVID CHISNALL

YOUR COMPUTER IS NOT A FAST PDP-11

In the wake of the recent Meltdown and Spectre vulnerabilities, it's worth spending some time looking at root causes. Both of these vulnerabilities involved processors speculatively executing instructions past some kind of access check and allowing the attacker to observe the results via a side channel. The features that led to these vulnerabilities, along with several others, were added to let C programmers continue to believe they were programming in a low-level language, when this hasn't been the case for decades.

Processor vendors are not alone in this. Those of us working on C/C++ compilers have also participated.

WHAT IS A LOW-LEVEL LANGUAGE?

Computer science pioneer Alan Perlis defined low-level languages this way:

> *"A programming language is low level when its programs require attention to the irrelevant."*[5]

While, yes, this definition applies to C, it does not capture what people desire in a low-level language. Various attributes cause people to regard a language as low-level. Think of programming languages as belonging on a continuum, with assembly at one end and the interface to the Starship *Enterprise*'s computer at the other. Low-level languages are "close to the metal," whereas high-level languages are closer to how humans think.

For a language to be "close to the metal," it must provide an abstract machine that maps easily to the abstractions exposed by the target platform. It's easy to argue that C was a low-level language for the PDP-11. They both described a model in which programs executed sequentially, in which memory was a flat space, and even the pre- and post-increment operators cleanly lined up with the PDP-11 addressing modes.

FAST PDP-11 EMULATORS

The root cause of the Spectre and Meltdown vulnerabilities was that processor architects were trying to build not just fast processors, but fast processors that expose the same abstract machine as a PDP-11. This is essential because it allows C programmers to continue in the belief that their language is close to the underlying hardware.

C code provides a mostly serial abstract machine (until C11, an entirely serial machine if nonstandard vendor extensions were excluded). Creating a new thread is a library operation known to be expensive, so processors wishing to keep their execution units busy running C code rely on ILP (instruction-level parallelism). They inspect adjacent operations and issue independent ones in parallel.

The quest for high ILP was the direct cause of Spectre and Meltdown.

This adds a significant amount of complexity (and power consumption) to allow programmers to write mostly sequential code. In contrast, GPUs achieve very high performance without any of this logic, at the expense of requiring explicitly parallel programs.

The quest for high ILP was the direct cause of Spectre and Meltdown. A modern Intel processor has up to 180 instructions in flight at a time (in stark contrast to a sequential C abstract machine, which expects each operation to complete before the next one begins). A typical heuristic for C code is that there is a branch, on average, every seven instructions. If you wish to keep such a pipeline full from a single thread, then you must guess the targets of the next 25 branches. This, again, adds complexity; it also means that an incorrect guess results in work being done and then discarded, which is not ideal for power consumption. This discarded work has visible side effects, which the Spectre and Meltdown attacks could exploit.

On a modern high-end core, the register rename engine is one of the largest consumers of die area and power. To make matters worse, it cannot be turned off or power gated while any instructions are running, which makes it inconvenient in a dark silicon era when transistors are cheap but powered transistors are an expensive resource. This unit is conspicuously absent on GPUs, where parallelism again comes from multiple threads rather than trying to extract instruction-level parallelism from intrinsically scalar code. If instructions do not have dependencies that need to be reordered, then register renaming is not necessary.

Consider another core part of the C abstract machine's

memory model: flat memory. This hasn't been true for more than two decades. A modern processor often has three levels of cache in between registers and main memory, which attempt to hide latency.

The cache is, as its name implies, hidden from the programmer and so is not visible to C. Efficient use of the cache is one of the most important ways of making code run quickly on a modern processor, yet this is completely hidden by the abstract machine, and programmers must rely on knowing implementation details of the cache (for example, two values that are 64-byte aligned may end up in the same cache line) to write efficient code.

OPTIMIZING C

One of the common attributes ascribed to low-level languages is that they're fast. In particular, they should be easy to translate into fast code without requiring a particularly complex compiler. The argument that a sufficiently smart compiler can make a language fast is one that C proponents often dismiss when talking about other languages.

Unfortunately, simple translation providing fast code is not true for C. In spite of the heroic efforts that processor architects invest in trying to design chips that can run C code fast, the levels of performance expected by C programmers are achieved only as a result of incredibly complex compiler transforms. The Clang compiler, including the relevant parts of LLVM, is around 2 million lines of code. Even just counting the analysis and transform passes required to make C run quickly adds up to almost 200,000 lines (excluding comments and blank lines).

For example, in C, processing a large amount of data means writing a loop that processes each element sequentially. To run this optimally on a modern CPU, the compiler must first determine that the loop iterations are independent. The C `restrict` keyword can help here. It guarantees that writes through one pointer do not interfere with reads via another (or if they do, that the programmer is happy for the program to give unexpected results). This information is far more limited than in a language such as Fortran, which is a big part of the reason that C has failed to displace Fortran in high-performance computing.

Once the compiler has determined that loop iterations are independent, then the next step is to attempt to vectorize the result, because modern processors get four to eight times the throughput in vector code that they achieve in scalar code. A low-level language for such processors would have native vector types of arbitrary lengths. LLVM IR (intermediate representation) has precisely this, because it is always easier to split a large vector operation into smaller ones than to construct larger vector operations.

Optimizers at this point must fight the C memory layout guarantees. C guarantees that structures with the same prefix can be used interchangeably, and it exposes the offset of structure fields into the language. This means that a compiler is not free to reorder fields or insert padding to improve vectorization (for example, transforming a structure of arrays into an array of structures or vice versa). That's not necessarily a problem for a low-level language, where fine-grained control over

data structure layout is a feature, but it does make it harder to make C fast.

C also requires padding at the end of a structure because it guarantees no padding in arrays. Padding is a particularly complex part of the C specification and interacts poorly with other parts of the language. For example, you must be able to compare two `struct`s using a type-oblivious comparison (e.g., `memcmp`), so a copy of a `struct` must retain its padding. In some experimentation, a noticeable amount of total runtime on some workloads was found to be spent in copying padding (which is often awkwardly sized and aligned).

Consider two of the core optimizations that a C compiler performs: SROA (scalar replacement of aggregates) and loop unswitching. SROA attempts to replace `struct`s (and arrays with fixed lengths) with individual variables. This then allows the compiler to treat accesses as independent and elide operations entirely if it can prove that the results are never visible. This has the side effect of deleting padding in some cases but not others.

The second optimization, loop unswitching, transforms a loop containing a conditional into a conditional with a loop in both paths. This changes flow control, contradicting the idea that a programmer knows what code will execute when low-level language code runs. It can also cause significant problems with C's notions of unspecified values and undefined behavior.

In C, a read from an uninitialized variable is an unspecified value and is allowed to be any value each time it is read. This is important, because it allows behavior such as lazy recycling of pages: for example, on FreeBSD the

`malloc` implementation informs the operating system that pages are currently unused, and the operating system uses the first write to a page as the hint that this is no longer true. A read to newly `malloc`ed memory may initially read the old value; then the operating system may reuse the underlying physical page; and then on the next write to a different location in the page replace it with a newly zeroed page. The second read from the same location will then give a zero value.

If an unspecified value for flow control is used (for example, using it as the condition in an `if` statement), then the result is undefined behavior: anything is allowed to happen. Consider the loop-unswitching optimization, this time in the case where the loop ends up being executed zero times. In the original version, the entire body of the loop is dead code. In the unswitched version, there is now a branch on the variable, which may be uninitialized. Some dead code has now been transformed into undefined behavior. This is just one of many optimizations that a close investigation of the C semantics shows to be unsound.

In summary, it is possible to make C code run quickly but only by spending thousands of person-years building a sufficiently smart compiler—and even then, only if you violate some of the language rules. Compiler writers let C programmers pretend that they are writing code that is "close to the metal" but must then generate machine code that has very different behavior if they want C programmers to keep believing that they are using a fast language.

UNDERSTANDING C

One of the key attributes of a low-level language is that

programmers can easily understand how the language's abstract machine maps to the underlying physical machine. This was certainly true on the PDP-11, where each C expression mapped trivially to one or two instructions. Similarly, the compiler performed a straightforward lowering of local variables to stack slots and mapped primitive types to things that the PDP-11 could operate on natively.

Since then, implementations of C have had to become increasingly complex to maintain the illusion that C maps easily to the underlying hardware and gives fast code. A 2015 survey of C programmers, compiler writers, and standards committee members raised several issues about the comprehensibility of C.[3] For example, C permits an implementation to insert padding into structures (but not into arrays) to ensure that all fields have a useful alignment for the target. If you zero a structure and then set some of the fields, will the padding bits all be zero? According to the results of the survey, 36 percent were sure that they would be, and 29 percent didn't know. Depending on the compiler (and optimization level), it may or may not be.

Implementations of C have had to become increasingly complex to maintain the illusion that C maps easily to the underlying hardware and gives fast code.

This is a fairly trivial example, yet a significant proportion of programmers either believe the wrong thing or are not sure. When you introduce pointers, the semantics of C become a lot more confusing. The BCPL model was fairly simple: values are words. Each word is either some data or the address of some data. Memory is a flat array of storage cells indexed by address.

The C model, in contrast, was intended to allow implementation on a variety of targets, including segmented architectures (where a pointer might be a segment ID and an offset) and even garbage-collected

virtual machines. The C specification is careful to restrict valid operations on pointers to avoid problems for such systems. The response to Defect Report 260[1] included the notion of *pointer provenance* in the definition of pointer:

> *Implementations are permitted to track the origins of a bit pattern and treat those representing an indeterminate value as distinct from those representing a determined value. They may also treat pointers based on different origins as distinct even though they are bitwise identical.*

Unfortunately, the word *provenance* does not appear in the C11 specification at all, so it is up to compiler writes to decide what it means. GCC (GNU Compiler Collection) and Clang, for example, differ on whether a pointer that is converted to an integer and back retains its provenance through the casts. Compilers are free to determine that two pointers to different `malloc` results or stack allocations always compare as not-equal, even when a bitwise comparison of the pointers may show them to describe the same address.

These misunderstandings are not purely academic in nature. For example, security vulnerabilities have been observed from signed integer overflow (undefined behavior in C) and from code that dereferenced a pointer before a null check, indicating to the compiler that the pointer could not be null because dereferencing a null pointer is undefined behavior in C and therefore can be assumed not to happen (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897).

In light of such issues, it is difficult to argue that a programmer can be expected to understand exactly how a

C program will map to an underlying architecture.

IMAGINING A NON-C PROCESSOR

The proposed fixes for Spectre and Meltdown impose significant performance penalties, largely offsetting the advances in microarchitecure in the past decade. Perhaps it's time to stop trying to make C code fast and instead think about what programming models would look like on a processor designed to be fast.

We have a number of examples of designs that have not focused on traditional C code to provide some inspiration. For example, highly multithreaded chips, such as Sun/Oracle's UltraSPARC Tx series, don't require as much cache to keep their execution units full. Research processors[2] have extended this concept to very large numbers of hardware-scheduled threads. The key idea behind these designs is that with enough high-level parallelism, you can suspend the threads that are waiting for data from memory and fill your execution units with instructions from others. The problem with such designs is that C programs tend to have few busy threads.

ARM's SVE (Scalar Vector Extensions)—and similar work from Berkeley[4]—provides another glimpse at a better interface between program and hardware. Conventional vector units expose fixed-sized vector operations and expect the compiler to try to map the algorithm to the available unit size. In contrast, the SVE interface expects the programmer to describe the degree of parallelism available and relies on the hardware to map it down to the available number of execution units. Using this from C is complex, because the autovectorizer must infer the available parallelism from loop structures. Generating code for it

from a functional-style map operation is trivial: the length of the mapped array is the degree of available parallelism.

Caches are large, but their size isn't the only reason for their complexity. The *cache coherency protocol* is one of the hardest parts of a modern CPU to make both fast and correct. Most of the complexity involved comes from supporting a language in which data is expected to be both shared and mutable as a matter of course. Consider in contrast an Erlang-style abstract machine, where every object is either thread-local or immutable (Erlang has a simplification of even this, where there is only one mutable object per thread). A cache coherency protocol for such a system would have two cases: mutable or shared. A software thread being migrated to a different processor would need its cache explicitly invalidated, but that's a relatively uncommon operation.

Immutable objects can simplify caches even more, as well as making several operations even cheaper. Sun Labs' Project Maxwell noted that the objects in the cache and the objects that would be allocated in a young generation are almost the same set. If objects are dead before they need to be evicted from the cache, then never writing them back to main memory can save a lot of power. Project Maxwell proposed a young-generation garbage collector (and allocator) that would run in the cache and allow memory to be recycled quickly. With immutable objects on the heap and a mutable stack, a garbage collector becomes a very simple state machine that is trivial to implement in hardware and allows for more efficient use of a relatively small cache.

A processor designed purely for speed, not for a

compromise between speed and C support, would likely support large numbers of threads, have wide vector units, and have a much simpler memory model. Running C code on such a system would be problematic, so, given the large amount of legacy C code in the world, it would not likely be a commercial success.

There is a common myth in software development that parallel programming is hard. This would come as a surprise to Alan Kay, who was able to teach an actor-model language to young children, with which they wrote working programs with more than 200 threads. It comes as a surprise to Erlang programmers, who commonly write programs with thousands of parallel components. It's more accurate to say that parallel programming in a language with a C-like abstract machine is difficult, and given the prevalence of parallel hardware, from multicore CPUs to many core GPUs, that's just another way of saying that C doesn't map to modern hardware very well.

Related articles

The Challenge of Cross-language Interoperability
David Chisnall
Interfacing between languages is increasingly important.
https://queue.acm.org/detail.cfm?id=2543971

Finding More than One Worm in the Apple
Mike Bland
If you see something, say something.
https://queue.acm.org/detail.cfm?id=2620662

Coding for the Code
Friedrich Steimann, Thomas Kühne
Can models provide the DNA for software development?
https://queue.acm.org/detail.cfm?id=1113336

References

1. C Defect Report 260. 2004; http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

2. Chadwick, G. A. 2013. Communication centric, multi-core, fine-grained processor architecture. Technical Report 832. University of Cambridge, Computer Laboratory; http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-832.pdf.
3. Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D. Watson, R. N. M., Sewell, P. 2016. Into the depths of C: elaborating the de facto standards. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*: 1-15; http://dl.acm.org/authorize?N04455.
4. Ou, A., Nguyen, Q., Lee, Y., Asanović, K. 2014. A case for MVPs: mixed-precision vector processors. Second International Workshop on Parallelism in Mobile Platforms at the 41st International Symposium on Computer Architecture.
5. Perlis, A. 1982. Epigrams on programming. *ACM SIGPLAN Notices* 17(9).

David Chisnall *is a researcher at the University of Cambridge, where he works on programming language design and implementation. He spent several years consulting in between finishing his Ph.D. and arriving at Cambridge, during which time he also wrote books on Xen and the Objective-C and Go programming languages, as well as numerous articles. He also contributes to the LLVM, Clang, FreeBSD, GNUstep, and Étoilé open-source projects, and he dances the Argentine tango.*