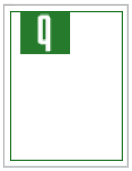
Development
Vol. 11 No. 10 – October 2013



Bugs and Bragging Rights
It's not always size that matters.

Dear KV, I've been dealing with a large program written in Java that seems to spend most of its time asking me to restart it because it has run out of memory.

by George Neville-Neil

Making the Web Faster with HTTP 2.0
HTTP continues to evolve

HTTP (Hypertext Transfer Protocol) is one of the most widely used application protocols on the Internet. Since its publication, RFC 2616 (HTTP 1.1) has served as a foundation for the unprecedented growth of the Internet: billions of devices of all shapes and sizes, from desktop computers to the tiny Web devices in our pockets, speak HTTP every day to deliver news, video, and millions of other Web applications we have all come to depend on in our everyday lives.

by Ilya Grigorik

The Challenge of Cross-language Interoperability
Interfacing between languages is increasingly important.

Interoperability between languages has been a problem since the second programming language was invented. Solutions have ranged from language-independent object models such as COM (Component Object Model) and CORBA (Common Object Request Broker Architecture) to VMs (virtual machines) designed to integrate languages, such as JVM (Java Virtual Machine) and CLR (Common Language Runtime). With software becoming ever more complex and hardware less homogeneous, the likelihood of a single language being the correct tool for an entire program is lower than ever. As modern compilers become more modular, there is potential for a new generation of interesting solutions.

by David Chisnall

Intermediate Representation
The increasing significance of intermediate representations in compilers

Program compilation is a complicated process. A compiler is a software program that translates a high-level source language program into a form ready to execute on a computer. Early in the evolution of compilers, designers introduced IRs (intermediate representations, also commonly called intermediate languages) to manage the complexity of the compilation process. The use of an IR as the compiler's internal representation of the program enables the compiler to be broken up into multiple phases and components, thus benefiting from modularity.

by Fred Chow

acmqueue

Bugs and Bragging Rights

It's not always size that matters

Dear KV,
I've been dealing with a large program written in Java that seems to spend most of its time asking me to restart it because it has run out of memory. I'm not sure if this is an issue in the JVM (Java Virtual Machine) I'm using or in the program itself, but during these frequent restarts, I keep wondering why this program is so incredibly bloated. I would have thought Java's garbage collector would prevent programs from running out of memory, especially when my desktop has quite a lot of it. It seems that eight gigabytes just isn't enough to handle a modern IDE anymore.

Lack of RAM

Dear Lack,
Eight gigabytes?! Is that all you have? Are you writing me from the desert wasteland where PCs go to die? No one in his or her right mind runs a machine with less than 48 GB in our modern era, at least no one who wants to run certain, very special, pieces of Java code.

While I would love to spend several hundred words bashing Java—for, like all languages, it has many sins—the problem you're seeing is probably not related to a bug in the garbage collector. It has to do with bugs in the code you're running, and with a certain, fundamental bug in the human mind. I'll address both of these in turn.

The bug in the code is easy enough to describe. Any computer language that takes the management of memory out of the hands of the programmer and puts it into an automatic garbage-collection system has one fatal flaw: the programmer can easily prevent the garbage collector from doing its work. Any object that continues to have a reference cannot be garbage collected, and therefore freed back into the system's memory.

Sloppy programmers who do not free their references cause memory leaks. In systems with many objects (and almost everything in a Java program is an object) a few small leaks can lead to out-of-memory errors quite quickly. These memory leaks are hard to find. Sometimes they reside in the code you, yourself, are working on, but often they reside in libraries that your code depends on. Without access to the library code, the bugs are impossible to fix, and even with access to the source, who wants to spend their lives fixing memory leaks in other people's code? I certainly don't. Moore's law often protects fools and little children from these problems, because while frequency scaling has stopped, memory density continues to increase. Why bother trying to find that small leak in your code when your boss is screaming to ship the next version of whatever it is you're working on? "The system stayed up for a whole day, ship it!"

The second bug is far more pernicious. One thing you didn't ask was, "Why do we have a garbage collector in our system?" The reason we have a garbage collector is because some time in the past, someone—well, really, a group of someones—wanted to remedy another problem: programmers who couldn't manage their own memory. C++, another object-oriented language, also has lots of objects floating around when its programs execute. In C++, as we all know, objects must be created

or destroyed using new and delete. If they're not destroyed, then we have a memory leak. Not only must the programmer manage objects, but in C++, the programmer can also get direct access to the memory that underlies the object, which leads naughty programmers to touch things they ought not to. The C++ runtime doesn't really say, "Bad touch, call an adult," but that is what a segmentation fault really means. Depending on your point of view, garbage collection was promulgated either to free programmers from the tedium of managing memory by hand or to prevent them from doing naughty things.

The problem is that we traded one set of problems for another. Before garbage collection, we would forget to delete an object, or double delete it by mistake; and after garbage collection, we had to manage our references to objects, which, in all honesty, is the exact same problem as forgetting to delete an object. We traded pointers for references and are none the wiser for it.

Longtime readers of KV know that silver bullets never work, and that one has to be very careful about protecting programmers from themselves. A side effect of creating a garbage-collected language was that the overhead of having the virtual machine manage memory was too high for many workloads. The performance penalty has led to people building huge Java libraries that do not use garbage collection and in which the objects have to be managed manually, just as they did with languages such as C++. When one of your key features has such high overhead that your own users create huge frameworks that avoid that feature, something has gone terribly wrong.

The situation as it stands is this: with a C++ (or C) program, you're more likely to see segmentation faults and memory-smashing bugs than you are to see out-of-memory errors on a modern system with a lot of RAM. If you're running something written in Java, then you had better pony up the cash for all the memory sticks you can manage because you're going to need them.

KV

Dear KV,
I cannot help but notice that a lot of large systems call themselves "Operating Systems" when they really don't bear much resemblance to one. Has the definition of *operating system* changed to the point where any large piece of software can call itself one?

OS or not OS

Dear OS,
Certainly my definition of operating system has not changed to the point where any large piece of software can call itself one, but I have also spotted the trend. An old joke is that every program grows in size until it can be used to read e-mail, which, if you can believe Wikipedia, is attributed to Jamie Zawinski, based on an earlier joke by Greg Kuperberg, "Every program in development at MIT expands until it can read mail." Now, it seems, mail is not enough. Every large program expands until it gets "OS" appended to its name.

An operating system is a program that is used to give efficient access to an underlying piece of hardware, hopefully in a portable manner, though that is not a strict requirement. The purpose of the software is to provide a consistent set of APIs to programmers such that they do not need to rewrite low-level code every time they want to run their programs on a new computer model. That may not be what Oxford defines as an OS, but as it recently added *selfie* to its dictionary, I'm starting to think a bit less of the quality of their output, anyway.

I think the propensity for programmers to label their larger creations as operating systems comes from the need to secure bragging rights. Programmers never stop comparing their code with the code of their peers. The same can be seen even within actual operating-system projects. Everyone seems to want to (re)write the scheduler. Why? Because to many programmers, it's the most important piece of code in the system, and if they do a great job, and the scheduler runs really well, they'll give their peers a good dose of coder envy. Never mind that the scheduler really ought to be incredibly small, and very, very simple, but that's not the point. The point is the bragging rights one gets from having rewritten it, often for the umpteenth time.

None of this is meant to belittle those programmers or teams of programmers who have slaved long and hard to produce elegant pieces of complex code that make our lives better. If you look closely, though, you'll find that those pieces of code are appropriately named, and they don't need to tack on an OS to make them look bigger.

KV

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.

acmqueue

Making the Web Faster with HTTP 2.0

HTTP continues to evolve

Ilya Grigorik

HTTP (Hypertext Transfer Protocol) is one of the most widely used application protocols on the Internet. Since its publication, RFC 2616 (HTTP 1.1) has served as a foundation for the unprecedented growth of the Internet: billions of devices of all shapes and sizes, from desktop computers to the tiny Web devices in our pockets, speak HTTP every day to deliver news, video, and millions of other Web applications we have all come to depend on in our everyday lives.

What began as a simple one-line protocol for retrieving hypertext (i.e., `"GET /resource"`—Telnet to google.com on port 80, take HTTP 1.0 for a spin) quickly evolved into a generic hypermedia transport protocol. Now a decade later it is used to power just about any use case imaginable.

Under the weight of its own success, however, and as more and more everyday interactions continue to migrate to the Web—social, e-mail, news and video, and, increasingly, our personal and job workspaces—HTTP has begun to show signs of stress. Users and developers alike are now demanding near-realtime responsiveness and protocol performance from HTTP 1.1, which it simply cannot provide without some modifications.

To meet these new challenges, HTTP must continue to evolve, which is where HTTP 2.0 enters the picture. HTTP 2.0 will make applications faster, simpler, and more robust by enabling efficient multiplexing and low-latency delivery over a single connection and allowing Web developers to undo many of the application "hacks" used today to work around the limitations of HTTP 1.1.

PERFORMANCE CHALLENGES OF MODERN WEB APPLICATIONS

A lot has changed in the decade since the HTTP 1.1 RFC was published: browsers have evolved at an accelerating rate, user connectivity profiles have changed, with the mobile Web now at an inflection point, and Web applications have grown in their scope, ambition, and complexity. Some of these factors help performance while others hinder. On balance, Web performance remains a large and unsolved problem.

TABLE 1 **Global broadband adoption**

Rank	Country	Average Mbps	Year over Year change
-	Global	3.1	17%
1	South Korea	14.2	-10%
2	Japan	11.7	6.8%
3	Hong Kong	10.9	16%
4	Switzerland	10.1	24%
5	Netherlands	9.9	12%
...			
9	United States	8.6	27%

First, the good news: modern browsers have put significant effort into performance. JavaScript execution speed continues its steady climb (e.g., the launch of the Chrome browser in 2008 delivered a 20x improvement, and in 2012 alone, the performance was further improved by more than 50 percent on mobile[11]). And it's not just JavaScript where the improvement is occurring; modern browsers also leverage GPU acceleration for drawing and animation (e.g., CSS3 animations and WebGL), provide direct access to native device APIs, and leverage numerous speculative optimization techniques[5] to help hide and reduce various sources of network latency.

Similarly, broadband adoption (table 1) has continued its steady climb over the past decade. According to Akamai, while the global average is now at 3.1 Mbps, many users have access to far higher throughput, especially with the rollout of residential fiber solutions.[1] Bandwidth is only half the equation, however. Latency is the oft-forgotten factor, and unfortunately, it is now often *the limiting factor* when it comes to browsing the Web.[4]

In practice, once the user has more than 5 Mbps of bandwidth, further improvements deliver minimal increase in the loading speed of the average Web application: streaming HD video from the Web is bandwidth-bound; loading the page hosting the HD video, with all of its assets, is latency-bound.

A modern Web application looks significantly different from a decade ago. According to HTTP Archive,[6] an average Web application is now composed of more than 90 resources, which are fetched from more than 15 distinct hosts, totaling more than 1,300 KB of (compressed) transferred data. As a result, a large fraction of HTTP data flows consist of small (less than 15 KB), bursty data transfers over dozens of distinct TCP connections. Therein lies the problem. TCP is optimized for long-lived connections and bulk data transfers. Network RTT (round-trip time) is the limiting factor in throughput of new TCP connections (a result of TCP congestion control), and consequently, latency is also the performance bottleneck for most Web applications.

How do you address this mismatch? First, you could try to reduce the round-trip latency by positioning the servers and bits closer to the user, as well as using lower-latency links. Unfortunately, while these are necessary optimizations—there is now an entire CDN (content delivery network) industry focused on exactly this problem—they are not sufficient. As an example, the global average RTT to google.com, which employs all of these techniques, was approximately 100 milliseconds in 2012, and unfortunately this number has not budged in the past few years.

Many existing links are already within a small constant factor (1.2~1.5) of the speed-of-light limit, and while there is still room for improvement, especially with respect to "last-mile latency," the relative gains are modest. Worse, with the rise of mobile networks, the impact of the latency bottleneck has only gotten worse. While the latest 4G mobile networks are specifically targeting low-latency data delivery, the advertised and real-world performance is still often measured in hundreds of milliseconds of overhead (see table 2 for advertised latencies in the AT&T core radio networks[2,3]).

TABLE 2 **Advertised latencies**

	LTE	HSPA+	HSPA	EDGE	GPRS
Latency	40-50 ms	100-200 ms	150-400 ms	600-750 ms	600-750 ms

If you can't get the performance-step function needed from improving the underlying links—if anything, with the rise of mobile traffic, there is a regression—then you must turn your attention to how you construct applications and tune the performance of the underlying transport protocols responsible for their delivery.

PERFORMANCE LIMITATIONS OF HTTP 1.1

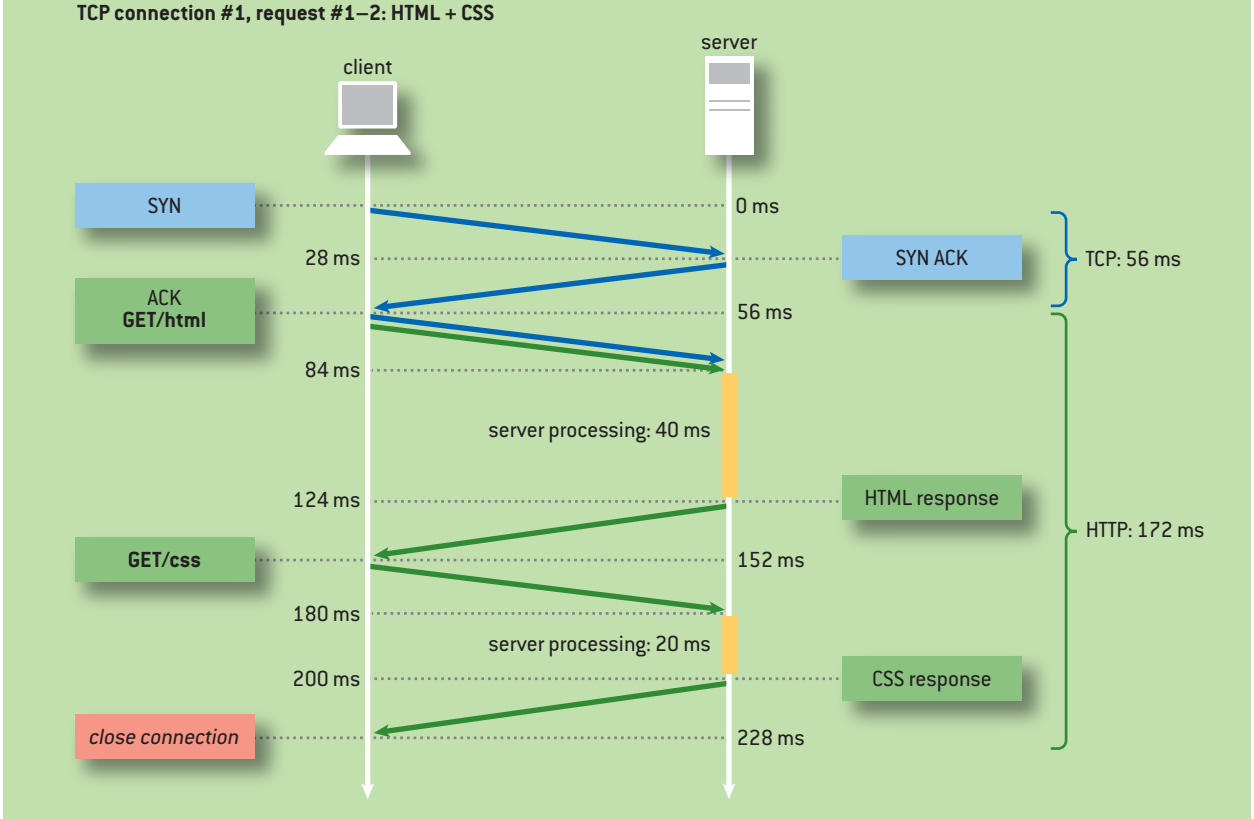
Improving the performance of HTTP was one of the key design goals for the HTTP 1.1 Working Group, and the standard introduced many critical performance enhancements. A few of the best-known include:

- Persistent connections to allow connection reuse.
- Chunked transfer encoding to allow response streaming.
- Request pipelining to allow parallel request processing.
- Byte serving to allow range-based resource requests.
- Improved and much better-specified caching mechanisms.

Unfortunately, some HTTP 1.1 features such as request pipelining have effectively failed due to lack of support and deployment challenges; while some browsers today support pipelining as an optional feature, few if any have it enabled by default. As a result, HTTP 1.1 forces strict request queuing on the client (figure 1): the client dispatches the request and must wait until the response

FIGURE 1

HTTP 1.1 Forces Strict Request Queuing on the Client



is returned by the server, which means that a single large transfer or a slow dynamic resource can block the entire connection. Worse, the browser has no way of reliably predicting this behavior and, as a result, is often forced to rely on heuristics to guess whether it should wait and attempt to reuse the existing connection or open another one.

In light of the limitations of HTTP 1.1, the Web developer community—always an inventive lot—has created and popularized a number of homebrew application workarounds (calling them *optimizations* would give them too much credit):

• Modern browsers allow up to six parallel connections per origin, which effectively allows up to six parallel resource transfers. Not satisfied with the limit of six connections, many developers decided to apply *domain sharding*, which splits site resources across different origins, thereby allowing more TCP connections. Recall that an average page now talks to 15 distinct hosts, each of which might use as many as six TCP connections.

• Small files with the same file type are often concatenated together, creating larger bundles to minimize HTTP request overhead. In effect, this is a form of multiplexing, but it is applied at the application layer—for example, CSS (Cascading Style Sheets) and JavaScript files are combined into larger bundles, small images are merged into image sprites, and so on.

• Some files are inlined directly into the HTML document to avoid the HTTP request entirely.

For many Web developers all of these are matter-of-fact optimizations—familiar, necessary, and universally accepted. Each of these workarounds, however, also often carries many negative implications for both the complexity and the performance of applications:

• Aggressive sharding often causes network congestion and is counterproductive, leading to: additional and unnecessary DNS (Domain Name Service) lookups and TCP handshakes; higher resource load caused by more sockets on client, server, and intermediaries; more network contention between parallel streams; and so on.

• Concatenation breaks the modularity of application code and has a negative impact on caching (e.g., a common practice is to concatenate all JavaScript or CSS files into large bundles, which forces download and invalidation of the entire bundle on a single byte change). Similarly, JavaScript and CSS files are parsed and executed only when the entire file is downloaded, which adds processing delays; large image sprites also occupy more memory on the client and require more resources to decode and process.

• Inlined assets cannot be cached individually and inflate the parent document. A common practice of inlining small images also inflates their size by more than 30 percent via base64 encoding and breaks request prioritization in the browser—typically, images are fetched with lower priority by the browser to accelerate page construction.

In short, many of the workarounds have serious negative performance implications. Web developers shouldn't have to worry about concatenating files, spriting images, inlining assets, or domain sharding. All of these techniques are stopgap workarounds for limitations of the HTTP 1.1 protocol. Hence, HTTP 2.0.

HTTP 2.0 DESIGN AND TECHNICAL GOALS

Developing a major revision of a protocol underlying all Web communication is a nontrivial task requiring a lot of careful thought, experimentation, and coordination. As such, it is important to define a clear technical charter and, arguably even more important, to define the boundaries of the

project. The intent is not to overhaul every detail of the protocol but to make meaningful though incremental progress to improve Web performance.

With that, the HTTPbis Working Group charter[7,8] for HTTP 2.0 is scoped as follows:

- Substantially and measurably improve end-user-perceived latency in most cases over HTTP 1.1 using TCP.
- Address the HOL (head-of-line) blocking problem in HTTP.
- Do not require multiple connections to a server to enable parallelism, thus improving its use of TCP, especially regarding congestion control.
- Retain the semantics of HTTP 1.1, leveraging existing documentation, including (but not limited to) HTTP methods, status codes, URIs, and where appropriate, header fields.
- Clearly define how HTTP 2.0 interacts with HTTP 1.x, especially in intermediaries.
- Clearly identify any new extensibility points and policy for their appropriate use.

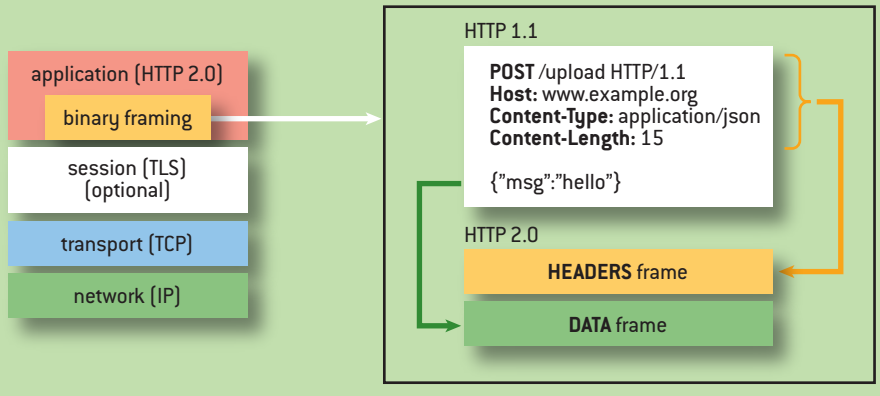
To deliver on these goals HTTP 2.0 introduces a new layering mechanism onto TCP, which addresses the well-known performance limitations of HTTP 1.x. The application semantics of HTTP remain untouched, and no changes are being made to the core concepts such as HTTP methods, status codes, URIs, and header fields—these changes are explicitly out of scope. With that in mind, let's now take a look "under the hood" of HTTP 2.0.

REQUEST AND RESPONSE MULTIPLEXING

At the core of all HTTP 2.0's performance enhancements is the new binary framing layer (figure 2), which dictates how HTTP messages are encapsulated and transferred between the client and server. HTTP semantics such as verbs, methods, and headers are unaffected, but the way they are encoded while in transit is different.

With HTTP 1.x, if the client wants to make multiple parallel requests to improve performance, multiple TCP connections are required. This behavior is a direct consequence of the newline-delimited plaintext HTTP 1.x protocol, which ensures that only one response at a time can be delivered per connection—worse, this also results in HOL blocking and inefficient use of the

FIGURE 2 **HTTP 2.0 Binary Framing**



underlying TCP connection.

The new binary framing layer in HTTP 2.0 removes these limitations and enables full request and response multiplexing. The following HTTP 2.0 terminology will help in understanding this process:

• *Stream*—a bidirectional flow of bytes, or a virtual channel, within a connection. Each stream has a relative priority value and a unique integer identifier.
• *Message*—a complete sequence of frames that maps to a logical message such as an HTTP request or a response.
• *Frame*—the smallest unit of communication in HTTP 2.0, each containing a consistent frame header, which at minimum identifies the stream to which the frame belongs, and carries a specific type of data (e.g., HTTP headers, payload, and so on).

All HTTP 2.0 communication can be performed within a single connection that can carry any number of bidirectional *streams*. In turn, each stream communicates in *messages*, which consist of one or multiple *frames*, each of which may be interleaved (figure 3) and then reassembled via the embedded stream identifier in the header of each individual frame.

The ability to break down an HTTP message into independent frames, prioritize and interleave them within a shared connection, and then reassemble them on the other end is the single most important enhancement of HTTP 2.0. By itself, this change is entirely unremarkable, since many protocols below HTTP already implement similar mechanisms. This "small" change, however, introduces a ripple effect of numerous performance benefits across the entire stack of all Web technologies, allowing developers to do the following:

• Interleave multiple requests in parallel without blocking on any one.
• Interleave multiple responses in parallel without blocking on any one.
• Use a single connection to deliver many requests and responses in parallel.
• Reduce page-load times by eliminating unnecessary latency.
• Remove unnecessary HTTP 1.x workarounds from application code.
• And much more…

BINARY FRAMING

HTTP 2.0 uses a binary, length-prefixed framing layer, which offers more compact representation than the newline-delimited plaintext HTTP 1.x protocol and is both easier and more efficient to

FIGURE 3 **Interleaved Frames from Multiple In-Flight HTTP 2.0 Streams Within a Single Connection**

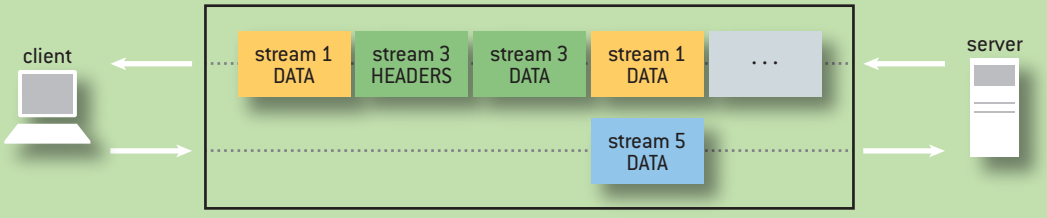
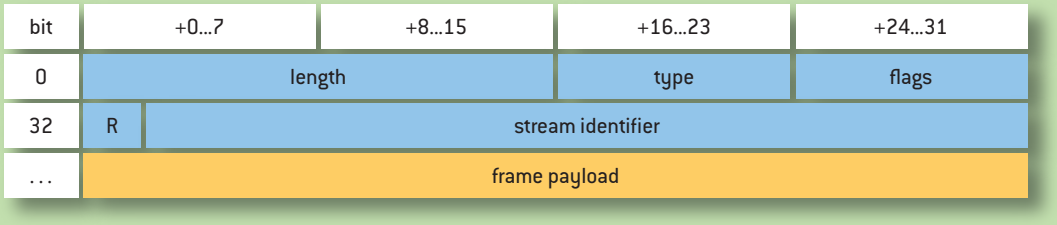


FIGURE 4 **Common Eight-Byte Frame Header**



process. All HTTP 2.0 frames share a common eight-byte header (figure 4), which contains the length of the frame, its type, a bit field for flags, and a 31-bit stream identifier.

- The 16-bit length prefix reveals that a single frame can carry 2^{16}–1 bytes of data - ~64 KB, which excludes the eight-byte header size.
- The eight-bit type field determines how the rest of the frame is interpreted.
- The eight-bit flags field allows different frame types to define frame-specific messaging flags.
- A one-bit reserved field is always set to 0.
- The 31-bit stream identifier uniquely identifies the HTTP 2.0 stream.

Given this knowledge of the shared HTTP 2.0 frame header, you can write a simple parser that can examine any HTTP 2.0 bytestream, identify different frame types, and report their flags and the length of each by examining the first eight bytes of every frame. Further, because each frame is length prefixed, the parser can skip ahead to the beginning of the next frame quickly and efficiently. This is a big performance improvement over HTTP 1.x.

Once the frame type is known, the parser can interpret the remainder of the frame. The HTTP 2.0 standard defines the types listed in table 3.

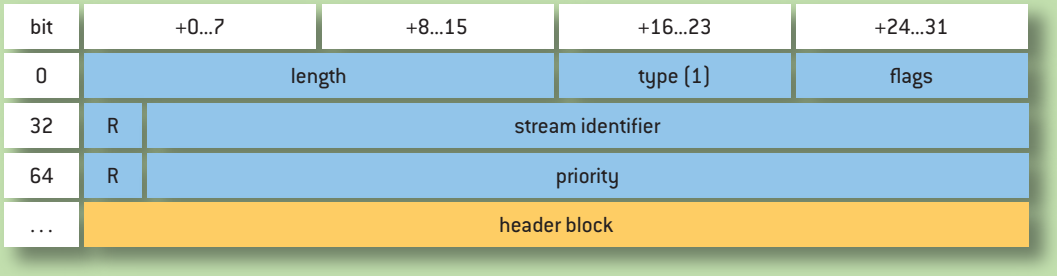
Full analysis of this taxonomy of frames is outside the scope of this discussion—after all, that's

TABLE 3 **HTTP 2.0 frame types**

DATA	used to transport HTTP message bodies
HEADERS	used to communicate additional header fields for a stream
PRIORITY	used to assign, or reassign, priority of referenced resource
RST_STREAM	used to signal abnormal termination of a stream
SETTINGS	used to signal configuration data about how two endpoints may communicate
PUSH_PROMISE	used to signal a promise to create a stream and serve referenced resource
PING	used to measure the round-trip time and perform "liveness" checks
GOAWAY	used to inform the peer to stop creating streams for current connection
WINDOW_UPDATE	used to implement flow control on per-stream or per-connection basis
CONTINUATION	used to continue a sequence of header block fragments

FIGURE 5

Headers Frame with Stream Priority and Header Payload



what the spec[7] is for (and it does a great job!). Having said that, let's go just one step further and look at the two most common workflows: initiating a new stream and exchanging application data.

INITIATING NEW HTTP 2.0 STREAMS

Before any application data can be sent, a new stream must be created and the appropriate metadata such as HTTP headers must be sent. That's what the HEADERS frame (figure 5) is for.

Notice that in addition to the common header, an optional 31-bit stream priority has been added. As a result, whenever the client initiates a new stream, it can signal to the server the relative priority of that request, and even reprioritize it later by sending another PRIORITY frame.

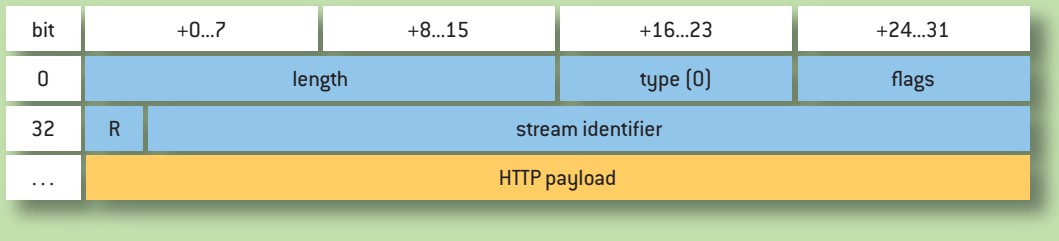
How do the client and server negotiate the unique stream IDs? They don't. Client-initiated streams have even-numbered stream IDs and server-initiated streams have odd-numbered stream IDs. This offset eliminates collisions in stream IDs between the client and server.

Finally, the HTTP header key-value pairs are encoded via a custom header compression algorithm (more on this later) to minimize the size of the payload, and they are appended to the end of the frame.

Notice that the HEADERS frames are used to communicate only the metadata about each stream. The actual application payload is delivered independently within the DATA frames (figure 6) that follow them (i.e., there is a separation between "data" and "control" messaging).

FIGURE 6

DATA Frame



The DATA frame is trivial: it's the common eight-byte header followed by the actual payload. To reduce HOL blocking, the HTTP 2.0 standard requires that each DATA frame not exceed 2^{14}-1 (16,383) bytes, which means that larger messages have to be broken up into smaller chunks. The last message in a sequence sets the END_STREAM flag to mark the end of data transfer.

There are a few more implementation details, but this information is enough to build a very basic HTTP 2.0 parser—emphasis on *very basic*. Features such as stream prioritization, server push, header compression, and flow control (not yet mentioned) warrant a bit more discussion, as they are critical to getting the best performance out of HTTP 2.0.

STREAM PRIORITIZATION

Once an HTTP message can be split into many individual frames, the exact order in which the frames are interleaved and delivered within a connection can be optimized to improve the performance of an application further. Hence, the optional 31-bit priority value: 0 represents the highest-priority stream; 2^{31}–1 represents the lowest-priority stream.

Not all resources have equal priority when rendering a page in the browser: the HTML document is, of course, critical, as it contains the structure and references to other resources; CSS is required to create the visual rendering tree (you can't paint pixels until you have the style-sheet rules); increasingly, JavaScript is also required to bootstrap the page; remaining resources such as images can be fetched with lower priority.

The good news is that all modern browsers already perform this sort of internal optimization by prioritizing different resource requests based on type of asset, their location on the page, and even learned priority from previous visits[4] (e.g., if the rendering was blocked on a certain asset in a previous visit, the same asset may be given a higher priority in the future).

With the introduction of explicit stream prioritization in HTTP 2.0, the browser can communicate these inferred priorities to the server to improve performance: the server can prioritize stream processing by controlling the allocation of resources (CPU, memory, bandwidth); and once the response data is available, the server can prioritize delivery of high-priority frames to the client. Even better, the client is now able to dispatch all of the requests as soon as they are discovered (i.e., eliminate client-side request queueing latency) instead of relying on request prioritization heuristics in light of the limited parallelism provided by HTTP 1.x.

SERVER PUSH

A powerful new feature of HTTP 2.0 is the ability of the server to send multiple replies for a single client request—that is, in addition to the response to the original request, the server can push additional resources to the client without having the client explicitly request each one.

Why would such a mechanism be needed? A typical Web application consists of dozens of resources, all of which the client discovers by examining the document provided by the server. As a result, why not eliminate the extra latency and let the server push the associated resources to the client ahead of time? The server already knows which resources the client will require—that's *server push*.

In fact, while support for server push as an HTTP protocol feature is new, many Web applications are already using it, just under a different name: *inlining*. Whenever the developer inlines an asset—CSS, JavaScript, or any other asset via a data URI—they are, in effect, pushing that resource to the

client instead of waiting for the client to request it. The only difference with HTTP 2.0 is that this workflow can now move out of the application and into the HTTP protocol itself, which offers important benefits: pushed resources can be cached by the client, declined by the client, reused across different pages, and prioritized by the server.

In effect, server push makes obsolete most of the cases where inlining is used in HTTP 1.x.

HEADER COMPRESSION

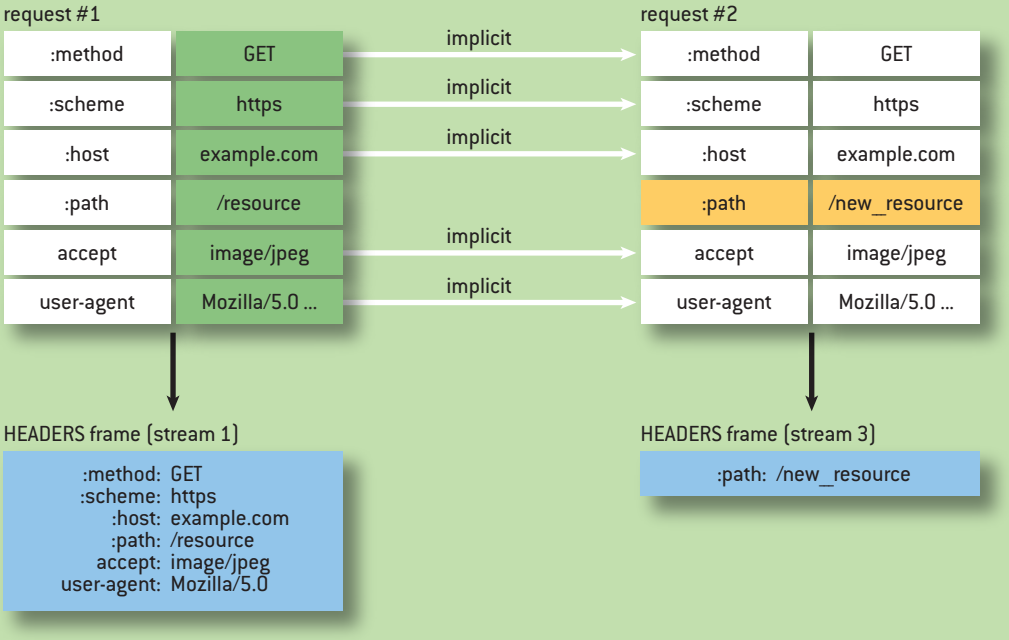
Each HTTP transfer carries a set of headers that are used to describe the transferred resource. In HTTP 1.x, this metadata is always sent as plaintext and typically adds anywhere from 500 to 800 bytes of overhead per request, and often much more if HTTP cookies are required. To reduce this overhead and improve performance, HTTP 2.0 compresses header metadata[9]:

- Instead of retransmitting the same data on each request and response, HTTP 2.0 uses header tables on both the client and server to track and store previously sent header key-value pairs.
- Header tables persist for the entire HTTP 2.0 connection and are incrementally updated by both the client and the server.
- Each new header key-value pair is either appended to the existing table or replaces a previous value in the table.

As a result, both sides of the HTTP 2.0 connection know which headers have been sent, and their previous values, which allows a new set of headers to be coded as a simple difference (figure 7) from the previous set.

Common key-value pairs that rarely change throughout the lifetime of a connection (e.g., user-

FIGURE 7 **Differential Encoding of HTTP 2.0 Headers**



agent, accept header, and so on), need to be transmitted only once. In fact, if no headers change between requests (e.g., a polling request for the same resource), then the header-encoding overhead is zero bytes—all headers are automatically inherited from the previous request.

FLOW CONTROL

Multiplexing multiple streams over the same TCP connection introduces contention for shared bandwidth resources. Stream priorities can help determine the relative order of delivery, but priorities alone are insufficient to control how resources are allocated between multiple streams. To address this, HTTP 2.0 provides a simple mechanism for stream and connection flow control:

- Flow control is hop-by-hop, not end-to-end.
- Flow control is based on window update frames: the receiver advertises how many bytes of DATA-frame payload it is prepared to receive on a stream and for the entire connection.
- Flow-control window size is updated via a WINDOW_UPDATE frame that specifies the stream ID and the window increment value.
- Flow control is directional—the receiver may choose to set any window size it desires for each stream and for the entire connection.
- Flow control can be disabled by a receiver.

As experience with TCP shows, flow control is both an art and a science. Research on better algorithms and implementation improvements are continuing to this day. With that in mind, HTTP 2.0 does not mandate any specific approach. Instead, it simply provides the necessary tools to implement such an algorithm—a great area for further research and optimization.

EFFICIENT HTTP 2.0 UPGRADE AND DISCOVERY

Though there are a lot more technical and implementation details, this whirlwind tour of HTTP 2.0 has covered the highlights: binary framing, multiplexing, prioritization, server push, header compression, and flow control. Combined, these features will deliver significant performance improvements on both the client and server.

Having said that, there is one more minor detail: how does one deploy a major revision of the HTTP protocol? The switch to HTTP 2.0 cannot happen overnight. Millions of servers must be updated to use the new binary framing protocol, and billions of clients must similarly update their browsers and networking libraries.

The good news is that most modern browsers use efficient background update mechanisms, which will enable HTTP 2.0 support quickly and with minimal intervention for a large portion of existing users. Despite this, some users will be stuck with older browsers, and servers and intermediaries will also have to be updated to support HTTP 2.0, which is a much longer labor- and capital-intensive process.

HTTP 1.x will be around for at least another decade, and most servers and clients will have to support both 1.x and 2.0 standards. As a result, an HTTP 2.0-capable client must be able to discover whether the server—and any and all intermediaries—support the HTTP 2.0 protocol when initiating a new HTTP session. There are two cases to consider:

- Initiating a new (secure) HTTPS connection via TLS.
- Initiating a new (unencrypted) HTTP connection.

In the case of a secure HTTPS connection, the new ALPN (Application Layer Protocol

FIGURE 8

HTTP Upgrade mechanism

```
GET /page HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: HTTP/2.0
HTTP2-Settings: (SETTINGS payload)

HTTP/1.1 200 OK
Content-length: 243
Content-type: text/html

(... HTTP 1.1 response ...)

          (or)

HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: HTTP/2.0

(... HTTP 2.0 response ...)
```

Negotiation[10]) extension to the TLS protocol allows users to negotiate HTTP 2.0 support as part of the regular TLS handshake: the client sends the list of protocols it supports (e.g., http/2.0); the server selects one of the advertised protocols and confirms its choice by sending the protocol name back to the client as part of the regular TLS handshake.

Establishing an HTTP 2.0 connection over a regular, nonencrypted channel requires a bit more work. Because both HTTP 1.0 and HTTP 2.0 run on the same port (80), in the absence of any other information about the server's support for HTTP 2.0, the client will have to use the HTTP Upgrade mechanism to negotiate the appropriate protocol, as shown in figure 8.

Using the Upgrade flow, if the server does not support HTTP 2.0, then it can immediately respond to the request with an HTTP 1.1 response. Alternatively, it can confirm the HTTP 2.0 upgrade by returning the "101 Switching Protocols" response in HTTP 1.1 format, and then immediately switch to HTTP 2.0 and return the response using the new binary framing protocol. In either case, no extra round-trips are incurred.

CRYSTAL GAZING

Developing a major revision of a protocol underlying all Web communication is a nontrivial task requiring a lot of careful thought, experimentation, and coordination. As such, crystal gazing for HTTP 2.0 timelines is a dangerous business—it will be ready when it's ready. Having said that, the HTTP Working Group is making rapid progress. Its past and projected milestones are as follows:

• November 2009—SPDY protocol announced by Google.
• March 2012—call for proposals for HTTP 2.0.
• September 2012—first draft of HTTP 2.0.
• July 2013—first implementation draft of HTTP 2.0.
• April 2014—Working Group last call for HTTP 2.0.
• November 2014—submit HTTP 2.0 to IESG (Internet Engineering Steering Group) as a Proposed Standard.

SPDY was an experimental protocol developed at Google and announced in mid-2009, which later formed the basis of early HTTP 2.0 drafts. Many revisions and improvements later, as of late 2013, there is now an implementation draft of the protocol, and interoperability work is in full swing—recent Interop events featured client and server implementations from Microsoft Open Technologies, Mozilla, Google, Akamai, and other contributors. In short, all signs indicate that the projected schedule is (for once) on track: 2014 should be the year for HTTP 2.0.

MAKING THE WEB (EVEN) FASTER

With HTTP 2.0 deployed far and wide, can we kick back and declare victory? The Web will be fast, right? Well, as with any performance optimization, the moment one bottleneck is removed, the next one is unlocked. There is plenty of room for further optimization:

• HTTP 2.0 eliminates HOL blocking at the application layer, but it still exists at the transport (TCP) layer. Further, now that all of the streams can be multiplexed over a single connection, tuning congestion control, mitigating bufferbloat, and all other TCP optimizations become even more critical.
• TLS is a critical and largely unoptimized frontier: we need to reduce the number of handshake round-trips, upgrade outdated clients to get wider adoption, and improve client and server performance in general.
• HTTP 2.0 opens up a new world of research opportunities for optimal implementations of header-compression strategies, prioritization, and flow-control logic both on the client and on the server, as well as the use of server push.
• All existing Web applications will continue to work over HTTP 2.0—the servers will have to be upgraded, but otherwise the transport switch is transparent. That is not to say, however, that existing and new applications can't be tuned to perform better over HTTP 2.0 by leveraging new functionality such as server push, prioritization, and so on. Web developers will have to develop new best practices, and revert and unlearn the numerous HTTP 1.1 workarounds they are using today.

In short, there is a lot more work to be done. HTTP 2.0 is a significant milestone that will help make the Web faster, but it is not the end of the journey.

REFERENCES

1. Akamai. 2013. State of the Internet; http://www.akamai.com/stateoftheinternet/.
2. AT&T. 2013. Average speeds for AT&T LaptopConnect Devices; http://www.att.com/esupport/article.jsp?sid=64785&cv=820&_requestid=733221#fbid=vttq9CyA2iG.
3. AT&T. 2012. Best Practices for 3G and 4G App Development; http://developer.att.com/home/develop/referencesandtutorials/whitepapers/BestPracticesFor3Gand4GAppDevelopment.pdf.

4. Belshe, M. 2010. More bandwidth doesn't matter (much); https://docs.google.com/a/chromium.org/viewer?a=v&pid=sites&srcid=Y2hyb21pdW0ub3JnfGRldnxneDoxMzcyOWI1N2I4YzI3NzE2.
5. Grigorik, I. 2013. High-performance networking in Google Chrome; http://www.igvita.com/posa/high-performance-networking-in-google-chrome/.
6. HTTP Archive; http://www.httparchive.org/.
7. IETF HTTPbis Working Group. 2012. Charter; http://datatracker.ietf.org/doc/charter-ietf-httpbis/.
8. IETF HTTPbis Working Group. 2013. HTTP 2.0 specifications; http://tools.ietf.org/html/draft-ietf-httpbis-http2.
9. IETF HTTPbis Working Group. 2013. HPACK-Header Compression for HTTP/2.0; http://tools.ietf.org/html/draft-ietf-httpbis-header-compression.
10. IETF Network Working Group. 2013. Transport Layer Security (TLS) Application Layer Protocol Negotiation Extension; http://tools.ietf.org/html/draft-friedl-tls-applayerprotoneg.
11. Upson, L. 2013. Google I/O 2013 keynote address; http://www.youtube.com/watch?v=9pmPa_KxsAM.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ILYA GRIGORIK is a web performance engineer and developer advocate at Google, where he works to make the Web faster by building and driving adoption of performance best practices at Google and beyond.

acmqueue

The Challenge of Cross-language Interoperability

Interfacing between languages is increasingly important

David Chisnall

Interoperability between languages has been a problem since the second programming language was invented. Solutions have ranged from language-independent object models such as COM (Component Object Model) and CORBA (Common Object Request Broker Architecture) to VMs (virtual machines) designed to integrate languages, such as the JVM (Java Virtual Machine) and CLR (Common Language Runtime). With software becoming ever more complex and hardware less homogeneous, the likelihood of a single language being the correct tool for an entire program is lower than ever. As modern compilers become more modular, there is potential for a new generation of interesting solutions.

In 1961 the British company Stantec released a computer called the ZEBRA, which was interesting for a number of reasons, not least of which was its data flow-based instruction set. The ZEBRA was quite difficult to program with the full form of its native instruction set, so it also included a more conventional version, called Simple Code. This form came with some restrictions, including a limit of 150 instructions per program. The manual helpfully informs users that this is not a severe limitation, as it is impossible that someone would write a working program so complex that it would need more than 150 instructions.

Today, this claim seems ludicrous. Even simple functions in a relatively low-level language such as C have more than 150 instructions once they are compiled, and most programs are far more than a single function. The shift from writing assembly code to writing in a higher-level language dramatically increased the complexity of programs that were possible, as did various software engineering practices.

The trend toward increased complexity in software shows no sign of abating, and modern hardware creates new challenges. Programmers in the late 1990s had to target PCs at the low end that had an abstract model a lot like a fast PDP-11. At the high end, they would have encountered an abstract model like a very fast PDP-11, possibly with two to four (identical) processors. Now, mobile phones are starting to appear with eight cores with the same ISA (instruction set architecture) but different speeds, some other streaming processors optimized for different workloads (DSPs, GPUs), and other specialized cores.

The traditional division between high-level languages representing the class that is similar to a human's understanding of the problem domain and low-level languages representing the class similar to the hardware no longer applies. No low-level language has semantics that are close to a programmable data-flow processor, an x86 CPU, a massively multithreaded GPU, and a VLIW (very long instruction word) DSP (digital signal processor). Programmers wanting to get the last bit of performance out of the available hardware no longer have a single language they can use for all probable targets.

Similarly, at the other end of the abstraction spectrum, domain-specific languages are growing

more prevalent. High-level languages typically trade generality for the ability to represent a subset of algorithms efficiently. More general-purpose high-level languages such as Java sacrifice the ability to manipulate pointers directly in exchange for providing the programmer with a more abstract memory model. Specialized languages such as SQL make certain categories of algorithms impossible to implement but make common tasks within their domain possible to express in a few lines.

You can no longer expect a nontrivial application to be written in a single language. High-level languages typically call code written in lower-level languages as part of their standard libraries (for example, GUI rendering), but adding calls can be difficult.

In particular, interfaces between two languages that are not C are often difficult to construct. Even relatively simple examples, such as bridging between C++ and Java, are not typically handled automatically and require a C interface. The Kaffe Native Interface[4] did provide a mechanism for doing this, but it was not widely adopted and had limitations.

The problem of interfacing between languages is going to become increasingly important to compiler writers over the coming years. It presents a number of challenges, detailed here.

OBJECT MODEL DIFFERENCES

Object-oriented languages bind some notion of code and data together. Alan Kay, who helped develop object-oriented programming while at Xerox PARC, described objects as "simple computers that communicate via message passing." This definition leaves a lot of leeway for different languages to fill in the details:

- Should there be factory objects (classes) as first-class constructs in the language?
- If there are classes, are they also objects?
- Should there be zero (e.g., Go), one (e.g., Smalltalk, Java, JavaScript, Objective-C), or many (e.g., C++, Self, Simula) superclasses or prototypes for an object?
- Is method lookup tied to the static type system (if there is one)?
- Is the data contained within an object of static or dynamic layout?
- Is it possible to modify method lookup at runtime?

The question of multiple inheritance is one of the most common areas of focus. Single inheritance is convenient, because it simplifies many aspects of the implementation. Objects can be extended just by appending fields; a cast to the supertype just involves ignoring the end, and a cast to a subtype just involves a check—the pointer values remain the same. Downcasting in C++ requires a complex search of the inheritance graph in the run-time type information via a runtime library function.

In isolation, both types of inheritance are possible to implement, but what happens if you want, for example, to expose a C++ object into Java? You could perhaps follow the .NET or Kaffe approach, and support direct interoperability with only a subset of C++ (Managed C++ or C++/CLI) that supports single inheritance only for classes that will be exposed on the Java side of the barrier.

This is a good solution in general: define a subset of one language that maps cleanly to the other but can understand the full power of the other. This is the approach taken in Pragmatic Smalltalk:[5] allow Objective-C++ objects (which can have C++ objects as instance variables and invoke their methods) to be exposed directly as if they were Smalltalk objects, sharing the same underlying representation.

This approach still provides a cognitive barrier, however. If you want to use a C++ framework

directly, such as LLVM from Pragmatic Smalltalk or .NET, then you will need to write single-inheritance classes that encapsulate the multiple-inheritance classes that the library uses for most of its core types.

Another possible approach would be to avoid exposing any fields within the objects and just expose each C++ class as an interface. This would, however, make it impossible to inherit from the bridged classes without special compiler support to understand that some interfaces came along with implementation.

Although complex, this is a simpler system than interfacing between languages that differ on what method lookup means. For example, Java and Smalltalk have almost identical object and memory models, but Java ties the notion of method dispatch to the class hierarchy, whereas in Smalltalk two objects can be used interchangeably if they implement methods with the same names.

This is a problem encountered by RedLine Smalltalk,[1] which compiles Smalltalk to run on JVM. Its mechanism for implementing Smalltalk method dispatch involves generating a Java interface for each method and then performing a cast of the receiver to the relevant interface type before dispatch. Sending messages to Java classes requires extra information, because existing Java classes don't implement this; thus, RedLine Smalltalk must fall back to using Java's Reflection APIs.

The method lookup for Smalltalk (and Objective-C) is more complex, because there are a number of second-chance dispatch mechanisms that are either missing or limited in other languages. When compiling Objective-C to JavaScript, rather than using the JavaScript method invocation, you must wrap each Objective-C message send in a small function that first checks if the method actually exists and, if it doesn't, calls some lookup code.

This is relatively simple in JavaScript because it handles variadic functions in a convenient way: if a function or method is called with more arguments than it expects, then it receives the remainder as an array that it can expect. Go does something similar. C-like languages just put them on the stack and expect the programmer to do the write with no error checking.

MEMORY MODELS

The obvious dichotomy in memory models is between automatic and manual deallocation. A slightly more important concern is the difference between deterministic and nondeterministic destruction.

It is possible to run C with the Boehm-Demers-Weiser garbage collector[3] without problems in many cases (unless you run out of memory and have a lot of integers that look like pointers). It is much harder to do the same for C++, because object deallocation is an observable event. Consider the following code:

```
{
      LockHolder l( mutex );
      /* Do stuff that requires mutex to be locked */
}
```

The `LockHolder` class defines a very simple object; a mutex passes into the object, which then locks the mutex in its constructor and unlocks it in the destructor. Now, imagine running this same code in a fully garbage-collected environment—the time at which the destructor runs is not defined.

This example is relatively simple to get right. A garbage-collected C++ implementation is required

to run the destructor at this point but not to deallocate the object. This idiom is not available in languages that were designed to support garbage collection from the start. The fundamental problem with mixing them is not determining who is responsible for releasing memory; rather, it is that code written for one model expects deterministic operation, whereas code written for the other does not.

There are two trivial approaches to implementing garbage collection for C++: the first is to make the `delete` operator invoke destructors but not reclaim the underlying storage; the other is to make `delete` a no-op and call destructors when the object is detected as unreachable.

Destructors that call only `delete` are the same in both cases: they are effectively no-ops. Destructors that release other resources are different. In the first case, they run deterministically but will fail to run if the programmer does not explicitly delete the relevant object. In the second case, they are guaranteed to run eventually but not necessarily by the time the underlying resource is exhausted.

Additionally, a fairly common idiom in many languages is a self-owned object that waits for some event or performs a long-running task and then fires a callback. The receiver of the callback is then responsible for cleaning up the notifier. While it's live, it is disconnected from the rest of the object graph and so appears to be garbage. The collector must be explicitly told that it is not. This is the opposite of the pattern in languages without automatic garbage collection, where objects are assumed to be live unless the system is told otherwise. (Hans Boehm discussed some of these issues in more detail in a 1996 paper.[2])

All of these problems were present with Apple's ill-fated (and, thankfully, no longer supported) attempt to add garbage collection to Objective-C. A lot of Objective-C code relies on running code in the `-dealloc` method. Another issue was closely related to the problem of interoperability. The implementation supported both traced and untraced memory but did not expose this information in the type system. Consider the following snippet:

```
void allocateSomeObjects (id * buffer, int count)
{
       for (int i=0 ; i<count ; i++)
       {
              buffer [i] = [SomeClass new];
       }
}
```

In garbage-collected mode, it is impossible to tell if this code is correct. Whether it is correct or not depends on the caller. If the caller passes a buffer allocated with `NSAllocateCollectable(),` with `NSScannedOption` as the second parameter, or with a buffer allocated on the stack or in a global in a compilation unit compiled with garbage-collection support, then the objects will last (at least) as long as the buffer. If the caller passes a buffer that was allocated with `malloc()` or as a global in a C or C++ compilation unit, then the objects will (potentially) be deallocated before the buffer. The *potentially* in this sentence makes this a bigger problem: because it's nondeterministic, it is hard to debug.

The ARC (Automatic Reference Counting) extensions to Objective-C do not provide complete garbage collection (they still allow garbage cycles to leak), but they do extend the type system to

define the ownership type for such buffers. Copying object pointers to C requires the insertion of an explicit cast containing an ownership transfer.

Reference counting also solves the determinism problem for acyclic data. In addition, it provides an interesting way of interoperating with manual memory management: by making `free()` decrement the reference count. Cyclic (or potentially cyclic) data structures require the addition of a cycle detector. David F. Bacon's team at IBM has produced a number of designs for cycle detectors[8] that allow reference counting to be a full garbage-collection mechanism, as long as pointers can be accurately identified.

Unfortunately, cycle detection involves walking the entire object graph from a potentially cyclic object. Some simple steps can be taken to lessen this cost. The obvious one is to defer it. An object is only potentially part of a cycle if its reference count is decremented but not deallocated. If it is later incremented, then it is not part of a garbage cycle (it may still be part of a cycle, but you don't care yet). If it is later deallocated, then it is acyclic.

The longer you defer cycle detection, the more nondeterminism you get, but the less work the cycle detector has to do.

EXCEPTIONS AND UNWINDING

These days, most people think of exceptions in the sense popularized by C++: something that is roughly equivalent to `setjmp()` and `longjmp()` in C, although possibly with a different mechanism.

A number of other mechanisms for exceptions have been proposed. In Smalltalk-80, exceptions are implemented entirely in the library. The only primitive that the language provides is that when you explicitly return from a closure, you return from the scope in which the closure was declared. If you pass a closure down the stack, then a return will implicitly unwind the stack.

When a Smalltalk exception occurs, it invokes a handler block on the top of the stack. This may then return, forcing the stack to unwind, or it may do some cleanup. The stack itself is a list of activation records (which are objects) and therefore may do something more complex. Common Lisp provides a rich set of exceptions too, including those that support resuming or restarting immediately afterward.

Exception interoperability is difficult even within languages with similar exception models. For example, C++ and Objective-C both have similar notions of an exception, but what should a C++ catch block that expects to catch a `void*` do when it encounters an Objective-C object pointer? In the GNUstep Objective-C runtime[6], we chose not to catch it after deciding not to emulate Apple's behavior of a segmentation fault. Recent versions of OS X have adopted this behavior, but the decision is somewhat arbitrary.

Even if you do catch the object pointer from C++, that doesn't mean that you can do anything with it. By the time it's caught, you've lost all of the type information and have no way of determining that it is an Objective-C object.

Subtler issues creep in when you start to think about performance. Early versions of VMKit[7] (which implements Java and CLR VMs on top of LLVM) used the zero-cost exception model designed for C++. This is *zero cost* because entering a try block costs nothing. When throwing an exception, however, you must parse some tables that describe how to unwind the stack, then call into a personality function for each stack frame to decide whether (and where) the exception should be caught.

This mechanism works very well for C++, where exceptions are rare, but Java uses exceptions to report lots of fairly common error conditions. In benchmarks, the performance of the unwinder was a limiting factor. To avoid this, the calling convention was modified for methods that were likely to throw an exception. These functions returned the exception as a second return value (typically in a different register), and every call just had to check that this register contained 0 or jump to the exception handling block if it did not.

This is fine when you control the code generator for every caller, but this is not the case in a cross-language scenario. You might address the issue by adding another calling convention to C that mirrors this behavior or that provides something like the multiple-return-values mechanism commonly used in Go for returning error conditions, but that would require every C caller to be aware of the foreign language semantics.

MUTABILITY AND SIDE EFFECTS

When you start to include functional languages in the set with which you wish to interoperate, the notion of mutability becomes important. A language such as Haskell has no mutable types. Modifying a data structure in place is something that the compiler may do as an optimization, but it's not something exposed in the language.

This is a problem encountered by F#, which is sold as a dialect of OCaml and can integrate with other .NET languages, use classes written in C#, and so on. C# already has a notion of mutable and immutable types. This is a very powerful abstraction, but an immutable class is simply one that doesn't expose any fields that are not read only, and a read-only field may contain references to objects that (via an arbitrary chain of references) refer to mutable objects whose state may be changed out from under the functional code. In other languages, such as C++ or Objective-C, mutability is typically implemented within the class system by defining some classes that are immutable, but there is no language support and no easy way of determining whether an object is mutable.

C and C++ have a very different concept of mutability in the type system provided by the language: a particular reference to an object may or may not modify it, but this doesn't mean that the object itself won't change. This, combined with the deep copying problem, makes interfacing functional and object-oriented languages a difficult problem.

Monads provide some tantalizing possibilities for the interface. A monad is an ordered sequence of computational steps. In an object-oriented world, this is a series of message sends or method invocations. Methods that have the C++ notion of const (i.e., do not modify the state of the object) may be invoked outside of the monad, and so are amenable to speculative execution and backtracking, whereas other methods should be invoked in a strict sequence defined by the monad.

MODELS OF PARALLELISM

Mutability and parallelism are closely related. The cardinal rule for writing maintainable, scalable, parallel code is that no object may be both mutable and aliased. This is trivial to enforce in a purely functional language: no object is mutable at all. Erlang makes one concession to mutability, in the form of a process dictionary—a mutable dictionary that is accessible only from the current Erlang process and so can never be shared.

Interfacing languages with different notions of what can be shared presents some unique

problems. This is interesting for languages intended to target massively parallel systems or GPUs, where the model for the language is intimately tied to the underlying hardware.

This is the issue encountered when trying to extract portions of C/C++/Fortran programs to turn into OpenCL. The source languages typically have in-place modification as the fastest way of implementing an algorithm, whereas OpenCL encourages a model where a source buffer is processed to generate an output buffer. This is important because each kernel runs in parallel on many inputs; thus, for maximum throughput they should be independent.

In C++, however, ensuring that two pointers do not alias is nontrivial. The `restrict` keyword exists to allow programmers to provide this annotation, but it's impossible in the general case for a compiler to check that it is correctly used.

Efficient interoperability is very important for heterogeneous multicore systems. On a traditional single-core or SMP (symmetric multiprocessing) computer, there is a one-dimensional spectrum between high-level languages that are close to the problem domain and low-level languages that are close to the architecture. On a heterogeneous system, no one language is close to the underlying architecture, as the difficulty of running arbitrary C/C++ and Fortran code on GPUs has shown.

Current interfaces—for example, OpenCL—are a long way from ideal. The programmer must write C code to manage the creation of a device context and the movement of data to and from the device, and then write the kernel in OpenCL C. The ability to express the part that runs on the device in another language is useful, but when most of the code for simple operations is related to the boundary between the two processing elements rather than the work done on either side, then something is wrong.

How to expose multiple processing units with very different abstract machine models to the programmer is an interesting research problem. It is very difficult to provide a single language that efficiently captures the semantics. Thus, this problem becomes one of interoperability between specialized languages. This is an interesting shift in that domain-specific languages, which are traditionally at the high-level end of the spectrum, now have an increasing role to play as low-level languages.

THE VM DELUSION

The virtual machine is often touted as a way of addressing the language interoperability problem. When Java was introduced, one of the promises was that you would soon be able to compile all of your legacy C or C++ code and run it in JVM alongside Java, providing a clean migration path. Today, Ohloh.net (which tracks the number of lines of code available in public open source repositories) reports 4 billion lines of C code, and around 1.5 billion each of C++ and Java. While other languages such as Scala (almost 6 million lines of code tracked by Ohloh.net) run in JVM, legacy low-level languages do not.

Worse, calling native code from Java is so cumbersome (in terms of both cognitive and runtime overhead) that developers end up writing applications in C++ rather than face calling into a C++ library from Java. Microsoft's CLR did a little better, allowing code written in a subset of C++ to run; it makes calling out to native libraries easier but still provides a wall.

This approach has been a disaster for languages such as Smalltalk that don't have large companies backing them. The Smalltalk VM provides some advantages that neither CLR nor JVM provides in the form of a persistence model and reflective development environment, but it also forms a very

large PLIB (programming language interoperability barrier) by dividing the world into things that are inside and things that are outside the box.

This gets even more complex once you have two or more VMs and now have the problem of source-language interoperability and the (very similar) problem of interoperability between the two VMs, which are typically very low-level programming languages.

THE PATH FORWARD

Many years ago the big interoperability question of the day was C and Pascal—two languages with an almost identical abstract machine model. The problem was that Pascal compilers pushed their parameters onto the stack left to right (because that required fewer temporaries), whereas C compilers pushed them right to left (to ensure that the first ones were at the top of the stack for variadic functions).

This interoperability problem was largely solved by the simple expedient of defining calling conventions as part of the platform ABI (application binary interface). No virtual machine or intermediate target was required, nor was any source-to-source translation. The equivalent of the virtual machine is defined by the ABI and the target machine's ISA.

Objective-C provides another useful case study. Methods in Objective-C use the C calling convention, with two hidden parameters (the object and the selector, which is an abstract form of the method name) passed first. All parts of the language that don't trivially map to the target ABI or ISA are factored out into library calls. A method invocation is implemented as a call to the `objc_msgSend()` function, which is implemented as a short assembly routine. All of the introspection works via the mechanism of calls to the runtime library.

We've used GNUstep's Objective-C runtime to implement front ends for dialects of Smalltalk and JavaScript in LanguageKit. This uses LLVM, but only because having a low-level intermediate representation permits optimizations to be reused between compilers: the interoperability happens in the native code. This runtime also supports the blocks ABI defined by Apple; therefore, closures can be passed between Smalltalk and C code.

Boehm GC (garbage collector) and Apple AutoZone both aimed to provide garbage collection in a library form, with different requirements. Can concurrent compacting collectors be exposed as libraries, with objects individually marked as nonmovable when they are passed out to low-level code? Is it possible to enforce mutability and concurrency guarantees in an ABI or library? These are open problems, and the availability of mature libraries for compiler design makes them interesting research questions.

Perhaps more interesting is the question of how many of these can be sunk down into the hardware. In CTSRD (Crash-worthy Trustworthy Systems R&D), a joint project between SRI International and the University of Cambridge Computer Laboratory, researchers have been experimenting with putting fine-grained memory protection into the hardware, which they hope will provide more efficient ways of expressing certain language memory models. This is a start, but there is a lot more potential for providing richer feature sets for high-level languages in silicon, something that was avoided in the 1980s because transistors were scarce and expensive resources. Now transistors are plentiful but power is scarce, so the tradeoffs in CPU design are very different.

The industry has spent the past 30 years building CPUs optimized for running languages such as C, because people who needed fast code used C (because people who designed processors optimized

them for C, because...). Maybe the time has come to start exploring better built-in support for common operations in other languages. The RISC project was born from looking at the instructions that a primitive compiler generated from compiling C code. What would we end up with if we started by looking at what a native JavaScript or Haskell compiler would emit?

REFERENCES

1. Allen, S. 2011. RedLine Smalltalk. Presented at the International Smalltalk Conference.
2. Boehm, H.-J. 1996. Simple garbage-collector-safety. *ACM SIGPLAN Notices* 31(5):89-98.
3. Boehm, H.-J., Weiser, M. 1988. Garbage collection in an uncooperative environment. *Software Practice and Experience* 18(9): 807-820.
4. Bothner, P., Tromey, T. 2001. Java/C++ integration; http://per.bothner.com/papers/UsenixJVM01/CNI01.pdf. http://gcc.gnu.org/java/papers/native++.html
5. Chisnall, D. 2012. Smalltalk in a C world. In *Proceedings of the International Workshop on Smalltalk Technologies:* 4:1-4:12.
6. Chisnall, D. 2012. A New Objective-C Runtime: from Research to Production. *ACM Queue*. http://queue.acm.org/detail.cfm?id=2331170
7. Geoffray, N., Thomas, G., Lawall, J., Muller, G., Folliot, B. 2010. VMKit: a substrate for managed runtime environments. *ACM SIGPLAN Notices* 45(7): 51-62.
8. Paz, H., Bacon, D. F., Kolodner, E. K., Petrank, E., Rajan, V. T. 2005. An efficient on-the-fly cycle collection. In *Proceedings of the 14th International Conference on Compiler Construction*: 156-171. Berlin, Heidelberg: Springer-Verlag.

Portions of this work were sponsored by DARPA (Defense Advanced Research Projects Agency) and AFRL (Air Force Research Laboratory), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the author and should not be interpreted as representing the official views or policies, either expressed or implied, of DARPA or the Department of Defense.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

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.

acmqueue

Intermediate Representation

The increasing significance of intermediate representations in compilers

Fred Chow

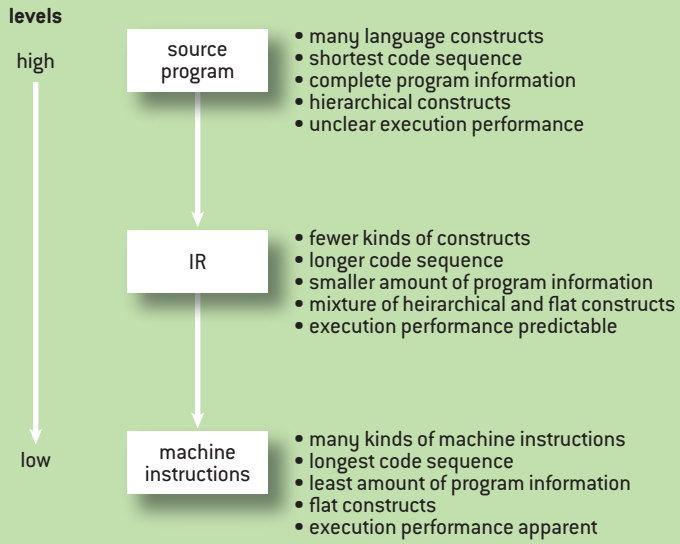
Program compilation is a complicated process. A compiler is a software program that translates a high-level source language program into a form ready to execute on a computer. Early in the evolution of compilers, designers introduced IRs (intermediate representations, also commonly called intermediate languages) to manage the complexity of the compilation process. The use of an IR as the compiler's internal representation of the program enables the compiler to be broken up into multiple phases and components, thus benefiting from modularity.

An IR is any data structure that can represent the program without loss of information so that its execution can be conducted accurately. It serves as the common interface among the compiler components. Since its use is internal to a compiler, each compiler is free to define the form and details of its IR, and its specification needs to be known only to the compiler writers. Its existence can be transient during the compilation process, or it can be output and handled as text or binary files.

THE IMPORTANCE OF IRS TO COMPILERS

An IR should be general so that it is capable of representing programs translated from multiple languages. Compiler writers traditionally refer to the semantic content of programming languages

FIGURE 1 **The Different Levels of Program Representations**



as being high. The semantic content of machine-executable code is considered low because it has retained only enough information from the original program to allow its correct execution. It would be difficult (if not impossible) to re-create the source program from its lower form. The compilation process entails the gradual lowering of the program representation from high-level human programming constructs to low-level real or virtual machine instructions (figure 1). In order for an IR to be capable of representing multiple languages, it needs to be closer to the machine level to represent the execution behavior of all the languages. Machine-executable code is usually longer because it reflects the details of the machines on which execution takes place.

A well-designed IR should be translatable into different forms for execution on multiple platforms. For execution on a target processor or CPU, it needs to be translated into the assembly language of that processor, which usually is a one-to-one mapping to the processor's machine instructions. Since there are different processors with different ISAs (instruction set architectures), the IR needs to be at a higher level than typical machine instructions, and not assume any special machine characteristic.

Using an IR enables a compiler to support multiple front ends that translate from different programming languages and multiple back ends to generate code for different processor targets (figure 2). The execution platform can also be interpretive in the sense that its execution is conducted by a software program or virtual machine. In such cases, the medium of execution can be at a level higher than assembly code, while being lower or at the same level as the IR.

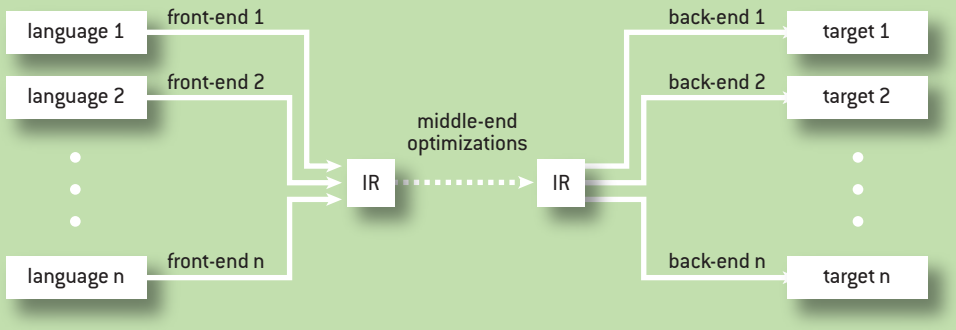
The adoption of IRs enables the modularization of the compilation process into the front end, the middle end, and the back end. The front end specializes in handling the programming language aspects of the compiler. A programming language implementer only needs to realize the accurate translation of the language to an IR before declaring the work complete.

The back end takes into account the particulars of the target machine and translates the IR into the machine instructions to be executed on the hardware. It also transforms the code to take advantage of any hardware features that benefit performance. By starting its translation from the IR, the back end in effect supports the different languages that produce the IR.

The middle end is where target-independent optimizations take place. The middle-end phases perform different transformations on the IR so the program can run more efficiently. Because optimizations on the IR usually benefit all targets, the IR has significant role of performing target-

FIGURE 2

A Compiler System Supporting Multiple Languages and Multiple Targets



independent optimizing transformations. In this role, its design and specification become even more important. It needs to encode any source program information that is helpful to the optimization tasks. The IR's design has a bearing on optimization efficiency, and optimization is the most time-consuming part of the compilation process. In modern-day compilers, the IR dictates the infrastructure and overall engineering of the compiler. Any major change to the IR could imply a substantial overhaul in the compiler implementation.

THE DIFFERENT IR FORMS

The minimal requirement of an IR is to provide enough information for the correct execution of the original program. Each instruction in an IR typically represents one simple operation. An IR should have fewer kinds of constructs than any typical programming language, because it does not need to be feature-rich to facilitate programming use by humans. Compilers like to see the same programming constructs or idioms being translated to uniform code sequences in the IR, regardless of their source languages, programming styles, or the ways the programmers choose to code them. Imposing canonical forms in IRs reduces the variety of code patterns that the compiler has to deal with in performing code generation and optimization. Because of its finer-grained representation of the program, an IR's instructions may map many-to-one to a machine instruction because one machine instruction may perform multiple operations, as in multiply-add or indexed addressing.

The form of an IR can be classified as either hierarchical or flat. A hierarchical IR allows nested structures. In a typical programming language, both program control flows (e.g., if-then-else, do-loops) and arithmetic expressions are nested structures. A hierarchical IR is thus closer in form to the typical programming language, and is regarded as being at a higher level. A hierarchical IR can be represented internally in the form of trees (the data structure preferred by compilers) without loss of accuracy.

A flat IR is often viewed as the instructions of an abstract or virtual machine. The instructions are executed sequentially, as in a typical processor, and control flows are specified by branch or jump instructions. Each instruction takes a number of operands and produces a result. Such IRs are often specified as compilation targets in the teaching of compiler construction.

Lying somewhere between hierarchical and flat IRs is the language of an abstract stack machine. In a stack machine, each operand for arithmetic computation is specified by an instruction that pushes the operand onto the stack. Each arithmetic expression evaluation is done on the operands that are popped off the top of the stack, and the subsequent result is pushed back onto the stack. The form of the IR is flat, with control flow represented by explicit branch instructions, but the instruction sequence for arithmetic computation can be regarded as corresponding to the reverse Polish notation, which can be easily represented internally in a tree data structure. Using the language of a stack machine as the IR has been a common practice from the first IR defined for Pascal, called p-code,[1] to the current-day Java bytecode[6] or CIL (Common Intermediate Language[2]).

There is information complementary to the IR that serves purposes other than representing code execution. The compiler compiles the namespace in the original program into a collection of symbol names. Variables, functions, and type information belong to these symbol tables, and they can encode information that governs the legality of certain optimizing transformations. They also provide information needed by various tools such as debuggers and program analyzers. The symbol tables can be considered adjuncts to the IRs.

C has been used as the translation target of many programming languages because of its widespread use as a system programming language and its ability to represent any machine operation. C can be regarded as an IR because of its lower level relative to most languages, but it was not designed for easy manipulation by compilers or to be directly interpreted. In spite of this, many IRs have been designed by closely modeling the C language semantics. In fact, a good IR can be constructed by carefully stripping away C's high-level control-flow constructs and structured data types, leaving behind only its primitives. Many IRs can also be translated to C-like output for easy perusal by compiler developers. Such C-like IRs, however, usually cannot be translated to C programs that can be recompiled because of C's deficiencies in representing certain programming concepts such as exception handling, overflow checking, or multiple entry points to a function.

IRS FOR PROGRAM DELIVERY

With the widespread use of networked computers, people soon understood the advantage of an execution medium that is processor- and operating-system-neutral. The distribution and delivery process is easier with programs that can run on any machine. This write-once, run-anywhere approach can be realized with the virtual machine execution model to accommodate the diversity of system hardware.

Interpretive execution contributes to some loss of performance compared with compiled execution, and initially it made sense only for applications that are not computation-intensive. As machines become faster and faster, however, the advantages of the write-once, run-anywhere approach outweigh potential performance loss in many applications. This gave rise to the popularity of languages such as Java that can be universally deployed. The Java language defines the Java bytecode, which is a form of IR, as its distribution medium. Java bytecode can be run on any platform as long as the JVM (Java virtual machine) software is installed. Another example is CIL, which is the IR of the CLI (Common Language Infrastructure) runtime environment used by the .NET Framework.

With the growth of the mobile Internet, applications are often downloaded to handheld devices to be run instantly. Since IRs take up less storage than machine executables, they reduce network transmission overhead, as well as enabling hardware-independent program distribution.

JUST-IN-TIME COMPILATION

As the virtual machine execution model gained widespread acceptance, it became important to find ways of speeding up the execution. One method is JIT (just-in-time) compilation, also known as dynamic compilation, which improves the performance of interpreted programs by compiling them during execution into native code to speed up execution on the underlying machine. Since compilation at runtime incurs overhead that slows down the program execution, it would be prudent to take the JIT route only if there is a high likelihood that the resultant reduction in execution time more than offsets the additional compilation time. In addition, the dynamic compiler cannot spend too much time optimizing the code, as optimization incurs much greater overhead than translation to native code. To restrain the overhead caused by dynamic compilation, most JIT compilers compile only the code paths that are most frequently taken during execution.

Dynamic compilation does have a few advantages over static compilation. First, dynamic compilation can use realtime profiling data to optimize the generated code more effectively.

Second, if the program behavior changes during execution, the dynamic compiler can recompile to adjust the code to the new profile. Finally, with the prevalent use of shared (or dynamic) libraries, dynamic compilation has become the only safe means of performing whole program analysis and optimization, in which the scope of compilation spans both user and library code. JIT compilation has become an indispensable component of the execution engines of many virtual machines that take IRs as input. The goal is to make the performance of programs built for machine-independent distribution approach that of native code generated by static compilers.

In recent years, computer manufacturers have come to the realization that further increases in computing performance can no longer rely on increases in clock frequency. This has given rise to special-purpose processors and coprocessors, which can be DSPs (digital signal processors), GPUs, or accelerators implemented in ASICs (application-specific integrated circuits) or FPGAs (field-programmable gate arrays). The computing platform can even be heterogeneous, where different types of computation are handed off to different types of processors, each having different instruction sets. Special languages or language extensions such as CUDA,[3] OpenCL,[8] and HMPP (Hybrid Multicore Parallel Programming),[4] with their underlying compilers, have been designed to make it easier for programmers to derive maximum performance in a heterogeneous setting.

Because these special processors are designed to increase performance, programs must be compiled to execute in their native instructions. As the proliferation of special-purpose hardware gathered speed, it became impossible for a compiler supplier to provide customized support for the variety of processors that exist in the market or are about to emerge. In this setting, the custom hardware manufacturer is responsible for providing the back-end compiler that compiles the IR to the custom machine instructions, and platform-independent program delivery has become all the more important. In practice, the IR can be compiled earlier, at installation time or at program loading, instead of during execution. Nowadays, the term *AOT* (ahead-of-time), in contrast with JIT, characterizes the compilation of IRs into machine code before its execution. Whether it's JIT or AOT, however, IRs obviously play an enabling role in this new approach to providing high-performance computing platforms.

STANDARDIZING IRS

So far, IRs have been linked to individual compiler implementations because most compilers are distinguished by the IRs they use. IRs are translatable, however, and it is possible to translate the IR of compiler A to that of compiler B, so compiler B can benefit from the work in compiler A. With the trend toward open source software in the past two decades, more and more compilers have been open sourced.[9] When a compiler becomes open source, it exposes its IR definition to the world. As the compiler's developer community grows, it has the effect of promoting its IR. Using an IR, however, is subject to the terms of its compiler's open source license, which often prohibits mixing it with other types of open source licenses. In case of licensing conflicts, special agreements need to be worked out with the license providers before such IR translations can be realized. When realized, IR translation enables collaboration between compilers.

Java bytecode is the first example of an IR with an open standard definition that is independent of compilers, because JVM is so widely accepted that it has spawned numerous compiler and VM implementations. The prevalence of JVM has led to many other languages being translated to Java bytecode,[7] but because it was originally defined to serve only the Java language, support for high-

level abstractions not present in Java is either not straightforward or absent. This lack of generality limits the use of Java bytecode as a universal IR.

Because IRs can solve the object-code compatibility issue among different processors by simplifying program delivery while enabling maximum compiled-code performance on each processor, standardizing on an IR would serve the computing industry well. Experience tells us that it takes time for all involved parties to agree on a standard; most existing standards have taken years to develop, and sometimes, competing standards take time to consolidate into one. The time is ripe to start developing an IR standard. Once such a standard is in place, it will not stifle innovation as long as it is being continuously extended to capture the latest technological trends.

A standard IR will solve two different issues that have persisted in the computing industry:

• **Software compatibility.** Two pieces of software are not compatible when they are in different native code of different ISAs. Even if their ISAs are the same, they can still be incompatible if they have been built using different ABIs (application binary interfaces) or under different operating systems with different object file formats. As a result, many different incompatible software ecosystems exist today. The computing industry would be well served by defining a standard software distribution medium that is acceptable by most if not all computing platforms. Such a distribution medium can be based on the IR of an abstract machine. It will be rendered executable on a particular platform through AOT or JIT compilation. A set of compliance tests can be specified. Software vendors will need to distribute their software products only in this medium. Computing devices supporting this standard will be able to run all software distributed in this form. This standardized software ecosystem will create a level playing field for manufacturers of different types of processors, thus encouraging innovation in hardware.

• **Compiler interoperability.** The field of compilation with optimization is a conundrum. No single compiler can claim to excel in everything. The algorithm that a compiler uses may work well for one program but not so well for another. Thus, developing a compiler requires a huge effort. Even for a finished compiler, there may still be endless enhancements deemed desirable. Until now, each production-quality compiler has been operating on its own. This article has discussed IR translation as a way of allowing compilers to work together. A standard IR, if adopted by compiler creators, would make it possible to combine the strengths of the different compilers that use it. These compilers will no longer need to incorporate the full compilation functionalities. They can be developed and deployed as compilation modules, and their creators can choose to make the modules either proprietary or open source. If a compiler module wants to use its own unique internal program representation, it can choose to use the standard IR only as an interchange format. A standard IR would lower the entry barrier for compiler writers, because their projects could be conceived at smaller scales, allowing each compiler writer to focus on his or her specialties. An IR standard would also make it easier to do comparisons among the compilers because they would produce the same IR as output, which will lead to more fine-tuning. An IR standard could revolutionize today's compiler industry and would serve the interests of compiler writers very well.

Two visions for an IR standard are outlined here: the first is centered on the computing industry, the second on the compiler industry. The first emphasizes the virtual machine aspect, and the second focuses on providing good support to the different aspects of compilation. Because execution requires less program information than compilation, the second goal will require greater content in the IR definition compared with the first goal. In other words, an IR standard that addresses the

first goal may not fulfill the needs of the second. It is also hard to say at this point whether one well-defined IR standard can fulfill both purposes at the same time.

The HSA (Heterogeneous System Architecture) Foundation was formed in 2012 with the charter of making programming heterogeneous devices dramatically easier by putting forth royalty-free specifications and open source software.[5] Its members intend to build a heterogeneous software ecosystem rooted in open royalty-free industry standards.

Recently, the foundation put forth a specification for HSAIL (HSA Intermediate Language), which is positioned as the ISA of an HSAIL virtual machine for any computing device that plans to adhere to the standard. HSAIL is quite low level, somewhat analogous to the assembly language of a RISC machine. It assumes a specific program and memory model catering to heterogeneous platforms where multiple ISAs exist, with one specified as the host. It also specifies a model of parallel processing as part of the virtual machine.

Although HSAIL is aligned with the vision of enabling a software ecosystem based on a virtual machine, its requirements are too strong and lack generality, and thus will limit its applicability to the specific segment of the computing industry that it targets. Though HSAIL is meant as the compilation target for compiler developers, it is unlikely that any compiler will adopt HSAIL as an IR during compilation because of the lack of simplicity in the HSAIL virtual machine. It is a step in the right direction, however.

IR DESIGN ATTRIBUTES

In conclusion, here is a summary of the important design attributes of IRs and how they pertain to the two visions discussed here. The first five attributes are shared by both visions.

- **Completeness.** The IR must provide clean representation of all programming language constructs, concepts, and abstractions for accurate execution on computing devices. A good test of this attribute is whether it is easily translatable both to and from popular IRs in use today for various programming languages.
- **Semantic gap.** The semantic gap between the source languages and the IR must be large enough that it is not possible to recover the original source program, in order to protect intellectual property rights. This implies the level of the IR must be low.
- **Hardware neutrality.** The IR must not have built-in assumptions of any special hardware characteristic. Any execution model apparent in the IR should be a reflection of the programming language and not the hardware platform. This will ensure it can be compiled to the widest range of machines, and implies that the level of the IR cannot be too low.
- **Manually programmable.** Programming in IRs is similar to assembly programming. This gives programmers the choice to hand-optimize their code. It is also a convenient feature that helps compiler writers during compiler development. A higher-level IR is usually easier to program.
- **Extensibility.** As programming languages continue to evolve, there will be demands to support new programming paradigms. The IR definition should provide room for extensions without breaking compatibility with earlier versions.

From the compiler's perspective, there are three more attributes that are important considerations for the IR to be used as a program representation during compilation:

- **Simplicity.** The IR should have as few constructs as possible while remaining capable of representing all computations translated from programming languages. Compilers often perform

a process called canonicalization that massages the input program into canonical forms before performing various optimizations. Having the fewest possible ways of representing a computation is actually good for the compiler, because there are fewer code variations for the compiler to cover.

• **Program information.** The most complete program information exists in the source form in which the program was originally written, some of which is derived from programming language rules. Translation out of the programming language will contribute to information loss, unless the IR provides ways of encoding the escaped information. Examples are high-level types and pointer aliasing information, which are not needed for program execution but affect whether certain transformations can be safely performed during optimization. A good IR should preserve any information in the source program that is helpful to compiler optimization.

• **Analysis information.** Apart from information readily available at the program level, program transformations and optimizations rely on additional information generated by the compiler's analysis of the program. Examples are data dependency, use-def, and alias analysis information. Encoding such information in the IR makes it usable by other compiler components, but such information can be invalidated by program transformations. If the IR encodes such analysis information, it needs to be maintained throughout the compilation, which puts additional burdens on the transformation phases. Thus, whether or not to encode information that can be gathered via program analysis is a judgment call. For the sake of simplicity, it can be left out or made optional.

A standard for a universal IR that enables target-independent program binary distribution and is usable internally by all compilers may sound idealistic, but it is a good cause that holds promise for the entire computing industry.

REFERENCES

1. Barron, D. W. (Ed.). 1981. *Pascal–The Language and its Implementation*. John Wiley.
2. CIL (Common Intermediate Language); http://en.wikipedia.org/wiki/Common_Intermediate_Language.
3. CUDA; http://www.nvidia.com/object/cuda_home_new.html.
4. HMPP; http://www.caps-entreprise.com/openhmpp-directives/.
5. HSA Foundation; http://www.hsafoundation.com/.
6. Java bytecode; http://www.javaworld.com/jw-09-1996/jw-09-bytecodes.html.
7. JVM languages; http://en.wikipedia.org/wiki/List_of_JVM_languages.
8. OpenCL; http://www.khronos.org/opencl/.
9. Open source compilers; http://en.wikipedia.org/wiki/List_of_compilers#Open_source_compilers.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

FRED CHOW (chowfred@icubecorp.com) pioneered the first optimizing compiler for RISC processors, the MIPS Ucode compiler. He was the chief architect behind the Pro64 compiler at SGI, later open sourced as the Open64 compiler. He later created the widely accepted PathScale version of the Open64 compiler. Algorithms he developed have been widely adopted in today's compilers. He is currently leading the compiler effort for a new processor at ICube Corp. He received a B.S. degree from the University of Toronto and M.S. and Ph.D. degrees from Stanford University.