

JUST-IN-TIME COMPILATION

A lecture on the last 60 years

JF BASTIEN

Software architect



Presented at CppCon 2020.

Just-in-Time compilers... we've all heard of them! What are they really? Why would anyone want them, are they actually a good idea, and how do they fit in with C++ since we all use Ahead-of-Time compilers?

In this talk I'll tell you about C++ AoT compiler, JiTs for dynamic language, JiTs for binary translation, and dive back 20, 30, 40, 50, 60 years, way back into compiler history and read wonderful academic papers about compilers. I'll illustrate how our view of compilers is really monolithic, and how compilers through time, and still today, are actually a continuum.

PAPERS IN THIS TALK

[GITHUB.COM/JFBASTIEN/JIT-TALK](https://github.com/jfbastien/jit-talk)

CPPCON SLACK CHANNEL

[#SIG_JIT](#)

I'll cover 20-some papers in this talk, and have collected them on a GitHub repo.

I'll be going through them in chronological order, covering 60 years.

We've also got a CppCon Slack channel: SIG_JIT (Special Interest Group).

In a way this talk isn't my usual talk because it's more of a lecture on JIT compilers, where I'll outline the papers that spoke the most to me, and which I want to share with you. I'll have more text on the screen than usual, from these papers.

First, some definitions.



With some artistic liberty, folks usually think of JIT as:

The executable code changes after the program is loaded into memory and the linker/loader are done doing their work.

On modern systems: pages mapped X at some point in time are modified.



Is AoT the opposite of JiT?

C and C++ are pretty much AoT these days: compile code to a target machine, then run it.

X mapping never changes.

INTERPRETER



Is an interpreter a JiT, or AoT? What if it modifies its bytecode?

A CPU executes machine code... an interpreter executes bytecode...

That's the same thing, a CPU is an interpreter for machine code.

A compiler can perform partial evaluation of a program, thereby interpreting it... The compiler itself can be compiled. The compiler compiler is then a compiler interpreter... but I'm getting ahead of myself with Futamura projections.

We'll dive further into this, but this gets us to this talk's first goal...

GOAL #1

JiT / AoT is a continuum

Hopefully my explanation of interpreter whet your appetite on this: computation and compilation can occur in a bunch of places, at different points in time.

“THOSE WHO CANNOT REMEMBER THE
PAST ARE CONDEMNED TO REPEAT IT.”

Jorge Agustín Nicolás Ruiz de Santayana y Borrás, 1863–1952

Bartlett, J. 1992. *Familiar Quotations* (16th ed.)

Aycock, J. 2003. *A Brief History of Just-In-Time*

Bastien, JF. 2020. *CppCon—Just-in-Time compilation*

Van Eerd, T. 2021. *CppCon—SOLID, Revisited*

While researching this talk I came upon this most wonderful aphorism.

GOAL #2

What *have* JiTs done?

Since computation and compilation can occur in different places and time, we ought to look at where those places and times have historically been, and *why*.

This is why we'll look at published papers in the field of compilers.

Not all of them are directly applicable to C++, but many of the ideas are relevant.

GOAL #3

What *could* JiTs do?

Many folks in the C++ community have the following perspective:

> I understand C++, and I kinda get assembly because of compiler explorer.

Our typical model of AoT is “what C and C++ do”, and I want to expand the understanding for what other computation models exist.

A warning: I’ll mostly avoid diving into the shortcomings and pitfalls of JiTs in this talk, but keep in mind that there are many! It would take a second talk to cover these.

I’m not trying to get everyone to JiT everything.



ALICE'S ADVENTURES IN *JiT* LAND

To that end—what could JiTs do—an alternate title for the talk is:

Alice's adventures in JiT land

Just like Alice challenges the reader's views,
my goal is to expand our minds regarding what's possible with compilers.
Let's look at our first paper...

A BRIEF HISTORY OF JUST-IN-TIME

— 2003

Let's start with our first paper.

A BRIEF HISTORY OF JUST-IN-TIME

— 2003

Software systems have been using “just-in-time” compilation (JiT) techniques since the 1960s.

(read)

A BRIEF HISTORY OF JUST-IN-TIME

— 2003

Software systems have been using “just-in-time” compilation (JiT) techniques since the 1960s. Broadly, JiT compilation includes any translation performed dynamically, after a program has started execution.

(read)

A BRIEF HISTORY OF JUST-IN-TIME

— 2003

Software systems have been using “just-in-time” compilation (JiT) techniques since the 1960s. Broadly, JiT compilation includes any translation performed dynamically, after a program has started execution. We examine the **motivation** behind JiT compilation and **constraints** imposed on JiT compilation systems, and present a **classification scheme** for such systems.

This is a great paper. If you read one paper, this is the paper.

A big chunk of this presentation borrows from it.

Notice, however, that it’s almost 20 years old!

HOW LONG IS
FOREVER?
SOMETIMES, JUST
ONE SECOND.

Carroll, L. 1865. *Alice's Adventures in Wonderland*



Lewis Carroll, on JiT compilers.

Imagine if my JiT compiler is in a foreground thread, blocking interaction, for an entire *second*!

Now contrast this with your usual C++ project build time... One second sounds wonderful!

Clearly, JiT authors wouldn't want to block your interaction this way. The design space is therefore worth considering.

Strictly speaking, JIT compilation systems are *completely unnecessary*.

(read)

Strictly speaking, JIT compilation systems are *completely unnecessary*. They are only a means to improve the time and space efficiency of programs.

(read)

Strictly speaking, JIT compilation systems are *completely unnecessary*. They are only a means to improve the time and space efficiency of programs. After all, the central problem JIT systems address is a solved one: translating programming languages into a form that is executable on a target platform.

Time and space efficiency are where it's at. How do these benefits break down?



Again, Alice knows about trading off space and time, what with all the bottles labeled “drink me” and rabbits running around.

To root this into C++: imagine templating all versions of a function based on all valid integer parameters which it could receive. That’s a whole lot of space, and a whole lot of time. Clearly we want to do some tradeoffs.

A large, bold, red number '4' is centered on a solid black rectangular background.

KEY BENEFITS

There are 4 key benefits to JIT systems, gained from AoT and interpreters.
Let's look at them...

1.

Compiled programs run faster, especially if they are compiled into a form that is directly executable on the underlying hardware.

(read)

1.

Compiled programs run faster, especially if they are compiled into a form that is directly executable on the underlying hardware. Static compilation can also devote an arbitrary amount of time to program analysis and optimization.

(read)

1.

Compiled programs run faster, especially if they are compiled into a form that is directly executable on the underlying hardware. Static compilation can also devote an arbitrary amount of time to program analysis and optimization.

This brings us to the primary constraint on JiT systems:

SPEED

(read)

1.

Compiled programs run faster, especially if they are compiled into a form that is directly executable on the underlying hardware. Static compilation can also devote an arbitrary amount of time to program analysis and optimization.

This brings us to the primary constraint on JiT systems:

SPEED

A JiT system must not cause untoward pauses in normal program execution as a result of its operation.

What constitutes an “untoward pause” depends on the system!

2.

Interpreted programs are typically smaller, if only because the representation chosen is at a higher level than machine code, and can carry much more semantic information implicitly.

Here, think of examples where a single bytecode instruction might do a full matrix multiplication, or change the prototype of a class. There are much higher level primitives than even what a CISC processor does.

JiTs can benefit from this size saving by only compiling the code that matters, and leaving the rest in a compressed form.

3.

Interpreted programs tend to be more portable.

(read)

3.

Interpreted programs tend to be more portable. Assuming a machine-independent representation, such as high-level source code or virtual machine code, only the interpreter need be supplied to run the program on a different machine.

(read)

3.

Interpreted programs tend to be more portable. Assuming a machine-independent representation, such as high-level source code or virtual machine code, only the interpreter need be supplied to run the program on a different machine. (Of course, the program still may be doing nonportable operations, but that's a different matter.)

Portability is a huge upside for programs that are entirely JiT'ed, even though the JiT itself has to be compiled to the target architecture.

4.

Interpreters have access to run-time information, such as input parameters, control flow, and target machine specifics.

(read)

4.

Interpreters have access to run-time information, such as input parameters, control flow, and target machine specifics. This information may change from run to run or be unobtainable prior to run-time.

(read)

4.

Interpreters have access to run-time information, such as input parameters, control flow, and target machine specifics. This information may change from run to run or be unobtainable prior to run-time. Additionally, gathering some types of information about a program before it runs may involve algorithms which are undecidable using static analysis.

Interpreters simply know more about the program, and a JiT can collect such information and optimize accordingly. Think about LTO and PGO in C++: it gives more information to the compiler. It can also speculate on certain facts being true, optimize optimistically, and roll back if it turns out that it was wrong.

THE FIRST JiT SYSTEMS

Alright, now that we have a rough idea of *why* you'd want a JiT, lets go chronologically and see how JiTs came into being 60 years ago.

RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS AND THEIR COMPUTATION BY MACHINE

—1960

LISP

Values of compiled functions are computed about **60 times as fast** as they would if interpreted. Compilation is fast enough so that it is **not necessary to punch compiled program** for future use.

This is the first published papers which talk about JiT compilation.

There were a few like this in the 60s.

They're quaint!

Of course, when they speak of punching compiled programs, they mean “punch cards”!

REGULAR EXPRESSION

SEARCH ALGORITHM

—1968

The compiler consists of three concurrently running stages:

1. Syntax sieve that allows only syntactically correct regular expressions to pass.
2. Convert the regular expression to reverse Polish form.
3. Object code producer, which expects a syntactically correct, reverse Polish regular expression.

Here's another quaint paper from the 60s, from some "K Thompson" person.
That's a pretty simple 3-step compiler!

EFFICIENT IMPLEMENTATION OF THE **SMALLTALK-80 SYSTEM**

—1984

The Smalltalk-80 programming language includes dynamic storage allocation, full upward funargs, and universally polymorphic procedures.

SMALLTALK

(read)

EFFICIENT IMPLEMENTATION OF THE **SMALLTALK-80 SYSTEM**

—1984

The Smalltalk-80 programming language includes dynamic storage allocation, full upward funargs, and universally polymorphic procedures. The Smalltalk-80 programming system features interactive execution with incremental compilation, and implementation portability.

SMALLTALK

(read)

EFFICIENT IMPLEMENTATION OF THE SMALLTALK-80 SYSTEM

—1984

The Smalltalk-80 programming language includes dynamic storage allocation, full upward funargs, and universally polymorphic procedures. The Smalltalk-80 programming system features interactive execution with incremental compilation, and implementation portability. These features of modern programming systems are among the most difficult to implement efficiently, even individually.

It's interesting to see what authors call out about their work.

In particular, “full upward funargs” allow passing a function with its closure.

The paper dives into important optimizations for Smalltalk.

It discussed “macro-expansion of v-code into n-code, with caching”, in other words there's an IR before machine code.

OPTIMIZING DYNAMICALLY-TYPED OBJECT-ORIENTED LANGUAGES WITH POLYMORPHIC INLINE CACHES

—1991

PICs achieve a median speedup of 11%.

SELF

(read)

OPTIMIZING DYNAMICALLY-TYPED OBJECT-ORIENTED LANGUAGES WITH POLYMORPHIC INLINE CACHES

—1991

SELF

PICs achieve a median speedup of 11%.

As an important side effect, PICs collect type information by recording all of the receiver types actually used at a given call site.

(read)

OPTIMIZING DYNAMICALLY-TYPED OBJECT-ORIENTED LANGUAGES WITH POLYMORPHIC INLINE CACHES

—1991

PICs achieve a median speedup of 11%.

As an important side effect, PICs collect type information by recording all of the receiver types actually used at a given call site. The compiler can exploit this type information to generate better code when recompiling a method.

(read)

OPTIMIZING DYNAMICALLY-TYPED OBJECT-ORIENTED LANGUAGES WITH POLYMORPHIC INLINE CACHES

—1991

SELF

PICs achieve a median speedup of 11%.

As an important side effect, PICs collect type information by recording all of the receiver types actually used at a given call site. The compiler can exploit this type information to generate better code when recompiling a method. An experimental version of such a system achieves a median **speedup of 27%** for our set of SELF programs, reducing the number of non-inlined message sends by a **factor of two**.

Self is a precursor of JavaScript and has prototypical inheritance. Types can change the equivalent of their v-table at runtime, but that rarely happens in practice.

Polymorphic Inline Caches record what actual types are seen, and check against the common cases at runtime.

It's a cute optimization which tries to make the language's dynamism more static when it actually is static.



ALICE: WOULD YOU TELL ME, PLEASE,
WHICH WAY I OUGHT TO GO FROM HERE?

THE CHESHIRE CAT: THAT DEPENDS A
GOOD DEAL ON **WHERE** YOU WANT TO GET TO.

ALICE: I DON'T MUCH CARE WHERE.

THE CHESHIRE CAT: THEN IT DOESN'T
MUCH MATTER WHICH WAY YOU GO.

Carroll, L. 1865. Alice's Adventures in Wonderland

Lewis Carroll, on JIT compilers. The key insight: figure out what you're trying to do, and why it matters.
There's a more "grown up" way to say this...

“IN THE COURSE OF OUR EXPERIMENTS WE
DISCOVERED THAT THE TRIGGER
MECHANISM (“WHEN”) IS MUCH LESS
IMPORTANT FOR GOOD RECOMPILATION
RESULTS THAN THE SELECTION
MECHANISM (“WHAT”).”

Hölzle 1994

Same as the Cheshire cat, Urs figured out that what you do matters more than when you do it.

Maybe you can take a bit more time and optimize the important functions, instead of optimizing the first thing you see.

DPF

FAST, FLEXIBLE MESSAGE DEMULTIPLEXING USING DYNAMIC CODE GENERATION

—1994

A new packet-filter system, DPF (Dynamic Packet Filters), that provides both the traditional flexibility of packet filters and the speed of hand-crafted demultiplexing routines.

(read)

DPF

FAST, FLEXIBLE MESSAGE DEMULTIPLEXING USING DYNAMIC CODE GENERATION

—1994

A new packet-filter system, DPF (Dynamic Packet Filters), that provides both the traditional flexibility of packet filters and the speed of hand-crafted demultiplexing routines.

DPF filters run 10–50 times faster than the fastest packet filters reported in the literature.

(read)

DPF

FAST, FLEXIBLE MESSAGE DEMULTIPLEXING USING DYNAMIC CODE GENERATION

—1994

A new packet-filter system, DPF (Dynamic Packet Filters), that provides both the traditional flexibility of packet filters and the speed of hand-crafted demultiplexing routines.

DPF filters run 10–50 times faster than the fastest packet filters reported in the literature.

DPF's performance is either equivalent to or, when it can exploit runtime information, superior to hand-coded demultiplexors.

(read)

DPF

FAST, FLEXIBLE MESSAGE DEMULTIPLEXING USING DYNAMIC CODE GENERATION

—1994

A new packet-filter system, DPF (Dynamic Packet Filters), that provides both the traditional flexibility of packet filters and the speed of hand-crafted demultiplexing routines.

DPF filters run 10–50 times faster than the fastest packet filters reported in the literature.

DPF's performance is either equivalent to or, when it can exploit runtime information, superior to hand-coded demultiplexors. DPF achieves high performance by using a carefully-designed declarative packet-filter language that is aggressively optimized using dynamic code generation.

Userspace can dynamically generate packet filter code, to steer network packets, into the kernel, safely.

EXOKERNEL

AN OPERATING SYSTEM ARCHITECTURE FOR APPLICATION-LEVEL RESOURCE MANAGEMENT

—1995

In the exokernel architecture, a small kernel securely exports all hardware resources through a low-level interface to untrusted library operating systems. Hardcoding the implementations of these abstractions is inappropriate for three main reasons:

1. it denies applications the advantages of domain-specific optimizations,
2. it discourages changes to the implementations of existing abstractions,
3. and it restricts the flexibility of application builders, since new abstractions can only be added by awkward emulation on top of existing ones (if they can be added at all).

DPF is part of exokernel.

The rationale for exposing hardware interfaces, instead of abstracting them away, are very similar to the rationale we've discussed for using a JiT.

ATOM

—1994

```
void InstrumentInit(int p1, char **p2) {
    AddCallProto("OpenFile()");
    AddCallProto("DataLoad(VALUE)");
    AddCallProto("DataStore(VALUE)");
    AddCallProto("CloseFile()");
    AddCallProgram(ProgramBefore, "OpenFile");
    AddCallProgram(ProgramAfter, "CloseFile");
}

void Instrument(int argc, char **argv, Obj *obj) {
    Proc *p; Block *b; Inst *i;
    for (p = GetFirstObjProc(obj); p != NULL; p = GetNextProc(p))
        for (b = GetFirstBlock(p); b != NULL; b = GetNextBlock(b))
            for (i = GetFirstInst(b); i != NULL; i = GetNextInst(i)) {
                if (IsInstType(i, InstTypeLoad))
                    AddCallInst(i, InstBefore, "DataLoad", EffAddrValue);
                if (IsInstType(i, InstTypeStore))
                    AddCallInst(i, InstBefore, "DataStore", EffAddrValue);
            }
}
```

A system for building customized program analysis tools, called “dynamic binary instrumentation” elsewhere.

This simple ATOM program instruments all of an (already compiled) program’s load and store instructions, adding a call to a function before each. Therefore, ATOM disassembles a program, instruments it, then re-assembles it. Very cute.

There’s another paper called PIN, from 2005, which goes way further than ATOM and is worth reading.

EMBRA

FAST AND FLEXIBLE MACHINE SIMULATION

—1996

A simulator for the processors, caches, and memory systems of uniprocessors and cache-coherent multiprocessors.

Uses dynamic binary translation to generate code sequences which simulate the workload. Embra can simulate real workloads at speeds **only 3 to 9 times slower** than native execution of the workload.

Can customize its generated code to include a processor cache model which allows it to compute the cache misses and memory stall time of a workload, at slowdowns of **only a factor of 7 to 20**.

(read 3 paragraphs)

Simulation is the process of running native executable machine code for one architecture on another architecture.

Dynamic binary translation: execute the program from one Instruction Set Architecture in another (or the same) ISA, performing the translation dynamically. In other words, disassemble the binary as you try to execute it, and reassemble another binary. Embra simulates MIPS R3000/R4000 on SGI IRIX. (cont'd)

EMBRA

FAST AND FLEXIBLE MACHINE SIMULATION

—1996

A simulator for the processors, caches, and memory systems of uniprocessors and cache-coherent multiprocessors.

Uses dynamic binary translation to generate code sequences which simulate the workload. Embra can simulate real workloads at speeds **only 3 to 9 times slower** than native execution of the workload.

Can customize its generated code to include a processor cache model which allows it to compute the cache misses and memory stall time of a workload, at slowdowns of **only a factor of 7 to 20**.

How do you develop hardware that doesn't exist yet? When I started working on a CPU, it wasn't obvious to me that you do so with simulators—multiple simulators—which simulate what the target hardware does with varying accuracy. You might want to only simulate architectural state (registers and memory), or simulate details such as non-architectural state (shadow registers, etc), timing of instructions, caches, memory, etc. This is very slow to interpret. Embra is the first machine simulator to use DBT. The speed numbers quoted here seem slow, but they're actually quite good! Embra can change the functionality of the simulation too (e.g. caches on / off), it doesn't need multiple implementations to do that work.

DYNAMIC OPTIMIZATION

—1999

Kistler looked at cache optimizations—rearranging fields in a structure dynamically to optimize a program's data-access patterns—and a dynamic version of trace scheduling, which optimizes based on information about a program's control flow during execution.

The continuous optimizer itself executes in the background, as a separate low-priority thread which executes only during a program's idle time. Kistler used a more sophisticated metric than straightforward counters to determine when to optimize, and observed that deciding *what* to optimize is highly optimization-specific.

(read 2 paragraphs)

C++ programmers often do Array-of-Structs / Structs-of-Arrays optimizations manually, but there's so much more that could theoretically be done if we could figure out ABI concerns.

Trace scheduling reorders code based on what's likely / unlikely, and can inline based on this.

This therefore changes the structure of data as well as code dynamically, at runtime.

DYNAMO

A TRANSPARENT DYNAMIC OPTIMIZATION SYSTEM

—2000

A software dynamic optimization system that is capable of transparently improving the performance of a native instruction stream as it executes on the processor.

Focus its efforts on optimization opportunities that tend to manifest only at runtime, and hence opportunities that might be difficult for a static compiler to exploit.

(read 2 paragraphs)

Dynamo translate from the same instruction set architecture, to the same one, but optimize the code. It can also handle optimization of JIT-generated code.

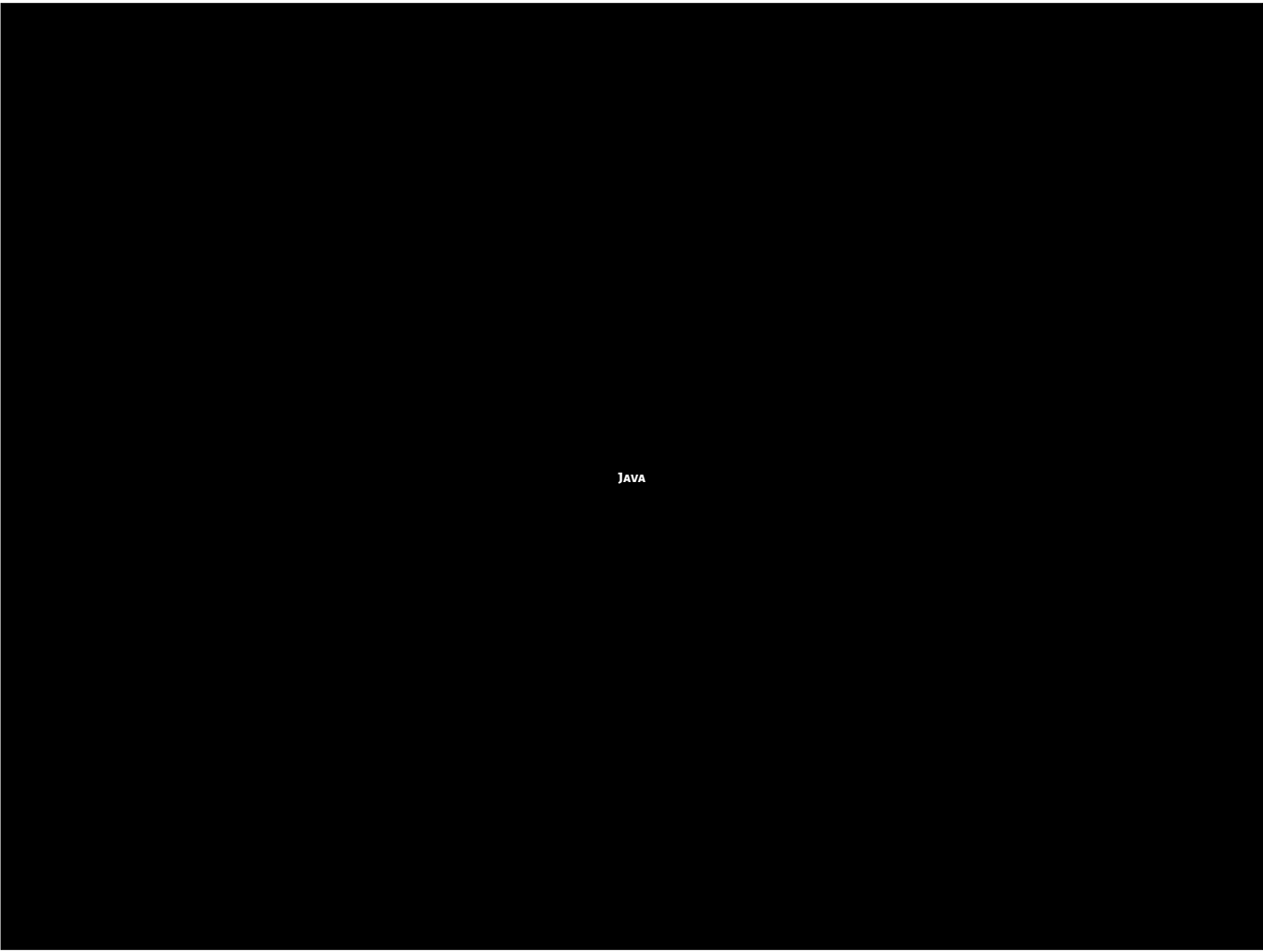
Doesn't require user guidance to optimize, nor multiple runs. You just run the program under Dynamo. They can bring SpecInt95 at -O level to the performance level of -O4.

“BY FAR THE **FASTEST** SIMULATOR OF THE
CPU, MMU, AND MEMORY SYSTEM OF AN
SGI MULTIPROCESSOR IS AN SGI
MULTIPROCESSOR.”

Rosenblum et al. 1995

In other words, when the source and target architectures are the same, as in the case where the goal is dynamic optimization of a source program, the source program can be executed directly by the CPU.
Remember the slowdowns quoted by Embra? If you simulate on the target architecture, then you can erase many of the performance costs.

Now let's change topics a bit and use a swear word...



“Java”

That’s right, I said it, at a C++ conference...

JAVA

At least I didn't say "Rust"...

COMPILING JAVA JUST IN TIME

—1997

Avoiding unnecessary overhead is crucial for fast compilation.

(read)

COMPILING JAVA JUST IN TIME

—1997

Avoiding unnecessary overhead is crucial for fast compilation. In many compilers, constructing an intermediate representation (IR) of a method is a standard process.

(read)

COMPILING JAVA JUST IN TIME

—1997

Avoiding unnecessary overhead is crucial for fast compilation. In many compilers, constructing an intermediate representation (IR) of a method is a standard process. When compiling from Java bytecode, however, we can eliminate that overhead.

(read)

COMPILING JAVA JUST IN TIME

—1997

Avoiding unnecessary overhead is crucial for fast compilation. In many compilers, constructing an intermediate representation (IR) of a method is a standard process. When compiling from Java bytecode, however, we can eliminate that overhead. The bytecodes themselves are an IR.

(read)

COMPILING JAVA JUST IN TIME

—1997

Avoiding unnecessary overhead is crucial for fast compilation. In many compilers, constructing an intermediate representation (IR) of a method is a standard process. When compiling from Java bytecode, however, we can eliminate that overhead. The bytecodes themselves are an IR. Because they are primarily designed to be compact and to facilitate interpretation, they are not the ideal IR for compilation, but they can easily be used for that purpose.

Early Java was interpreted, and was quite slow.

Fundamental design criteria: portable and secure.

The bytecode can't be trusted, and invariants must be checked on top of interpretation of semantics.

Classes can be loaded at runtime, making dynamic dispatch near-impossible to determine statically.

(cont'd)

COMPILING JAVA JUST IN TIME

—1997

Avoiding unnecessary overhead is crucial for fast compilation. In many compilers, constructing an intermediate representation (IR) of a method is a standard process. When compiling from Java bytecode, however, we can eliminate that overhead. The bytecodes themselves are an IR. Because they are primarily designed to be compact and to facilitate interpretation, they are not the ideal IR for compilation, but they can easily be used for that purpose.

A key to JiT design is IR design.

This has deep effects on which optimizations are feasible.

In particular, some information is lost when translating from original source to IR.

Also, some IRs have easier to analyze structure

Such as:

- * “Which instructions use this result?”

- * “What is the control flow that leads here?”

DESIGN OF THE
JAVA HotSpot
CLIENT COMPILER FOR JAVA 6

— 2006

The paper talks about many interesting aspects of a JIT, referring back to the publications which originally pioneered them. The paper itself adds fast algorithms for escape analysis, automatic object inlining, and array bounds check elimination. Where HotSpot is really amazing is in putting all of these things together in a very complex JIT and runtime. Let's look at a few interesting bits...

DESIGN OF THE
JAVA HotSPOT
CLIENT COMPILER FOR JAVA 6

— 2006

If a method contains a long-running loop, it may be compiled regardless of its invocation frequency.

(read)

DESIGN OF THE
JAVA HotSpot
CLIENT COMPILER FOR JAVA 6

— 2006

If a method contains a long-running loop, it may be compiled regardless of its invocation frequency. The VM counts the number of backward branches taken and, when a threshold is reached, it suspends interpretation and compiles the running method.

(read)

DESIGN OF THE
JAVA HotSPOT
CLIENT COMPILER FOR JAVA 6

— 2006

If a method contains a long-running loop, it may be compiled regardless of its invocation frequency. The VM counts the number of backward branches taken and, when a threshold is reached, it suspends interpretation and compiles the running method. A new stack frame for the native method is set up and initialized to match the interpreter's stack frame.

(read)

DESIGN OF THE
JAVA HotSPOT
CLIENT COMPILER FOR JAVA 6

— 2006

If a method contains a long-running loop, it may be compiled regardless of its invocation frequency. The VM counts the number of backward branches taken and, when a threshold is reached, it suspends interpretation and compiles the running method. A new stack frame for the native method is set up and initialized to match the interpreter's stack frame. Execution of the method then continues using the machine code of the native method.

(read)

DESIGN OF THE
JAVA HOTSPOT
CLIENT COMPILER FOR JAVA 6

— 2006

If a method contains a long-running loop, it may be compiled regardless of its invocation frequency. The VM counts the number of backward branches taken and, when a threshold is reached, it suspends interpretation and compiles the running method. A new stack frame for the native method is set up and initialized to match the interpreter's stack frame. Execution of the method then continues using the machine code of the native method. Switching from interpreted to compiled code in the middle of a running method is called *on-stack-replacement* (OSR).

Most of the time we think of JITs jumping to the entry of a function, but if you're stuck in a hot loop then you can't do that. This is where OSR comes in.

OSR effectively turns a function "inside out" so that the interpreter can jump into the middle of the function while it executes, while preserving the semantics.

DESIGN OF THE
JAVA HotSpot
CLIENT COMPILER FOR JAVA 6

— 2006

The compiler creates *debugging information* that maps the state of a compiled method back to the state of the interpreter. This enables aggressive compiler optimizations, because the VM can *deoptimize* back to a safe state when the assumptions under which an optimization was performed are invalidated.

Garbage collection and deoptimization are allowed to occur only at some discrete points in the program, called *safepoints*, such as backward branches, method calls, return instructions, and operations that may throw an exception.

(read 2 paragraphs)

I've always found the concept of optimizers de-optimizing amusing.

An example of deoptimization is when a class is loaded dynamically. Some functions might have been inlined and this has to be undone, but it's exceedingly rare.

Deoptimization kicks the execution back to the interpreter, but in some cases you might have multiple *tiers* in a compiler. Execution could “go back” from -O3 with speculation to -O1, instead of a slower interpreter.

JAVA BROUGHT THE TERM
JUST-IN-TIME
INTO COMMON USE IN COMPUTING LITERATURE

*Gosling borrowed from manufacturing terminology,
traced his own use to about 1993*

Thank you, Java 🙏

And thank you Toyota, for the borrowed word.

FX!32

A PROFILE-DIRECTED BINARY TRANSLATOR

—1998

Because Digital's Alpha architecture provides the world's fastest processors, many applications, especially those requiring high processor performance, have been ported to it. However,

(read)

FX!32

A PROFILE-DIRECTED BINARY TRANSLATOR

—1998

Because Digital's Alpha architecture provides the world's fastest processors, many applications, especially those requiring high processor performance, have been ported to it. However,

😞 many other applications are available only under the x86 architecture. 😞

(I added the frownies)

FX!32

A PROFILE-DIRECTED BINARY TRANSLATOR

—1998

Because Digital's Alpha architecture provides the world's fastest processors, many applications, especially those requiring high processor performance, have been ported to it. However,

😞 many other applications are available only under the x86 architecture. 😞

We designed Digital FX!32 to make the complete set of applications, both native and x86, available to Alpha.

(read)

FX!32

A PROFILE-DIRECTED BINARY TRANSLATOR

—1998

Because Digital's Alpha architecture provides the world's fastest processors, many applications, especially those requiring high processor performance, have been ported to it. However,

😞 many other applications are available only under the x86 architecture. 😞

We designed Digital FX!32 to make the complete set of applications, both native and x86, available to Alpha. The goal for the software is to provide fast and transparent execution of x86 Win32 applications on Alpha systems.

(read)

FX!32

A PROFILE-DIRECTED BINARY TRANSLATOR

—1998

Because Digital's Alpha architecture provides the world's fastest processors, many applications, especially those requiring high processor performance, have been ported to it. However,

😞 many other applications are available only under the x86 architecture. 😞

We designed Digital FX!32 to make the complete set of applications, both native and x86, available to Alpha. The goal for the software is to provide fast and transparent execution of x86 Win32 applications on Alpha systems. FX!32 achieves its goal by transparently running those applications at speeds comparable to high-performance x86 platforms.

(read)

FX!32

A PROFILE-DIRECTED BINARY TRANSLATOR

—1998

Because Digital's Alpha architecture provides the world's fastest processors, many applications, especially those requiring high processor performance, have been ported to it. However,

😞 many other applications are available only under the x86 architecture. 😞

We designed Digital FX!32 to make the complete set of applications, both native and x86, available to Alpha. The goal for the software is to provide fast and transparent execution of x86 Win32 applications on Alpha systems. FX!32 achieves its goal by transparently running those applications at speeds comparable to high-performance x86 platforms. Digital FX!32 is a software utility that enables x86 Win32 applications to be run on Windows NT/Alpha platforms.

(read)

FX!32

A PROFILE-DIRECTED BINARY TRANSLATOR

—1998

Because Digital's Alpha architecture provides the world's fastest processors, many applications, especially those requiring high processor performance, have been ported to it. However,

😞 many other applications are available only under the x86 architecture. 😞

We designed Digital FX!32 to make the complete set of applications, both native and x86, available to Alpha. The goal for the software is to provide fast and transparent execution of x86 Win32 applications on Alpha systems. FX!32 achieves its goal by transparently running those applications at speeds comparable to high-performance x86 platforms. Digital FX!32 is a software utility that enables x86 Win32 applications to be run on Windows NT/Alpha platforms. Once FX!32 has been installed, almost all x86 applications can be run on Alpha without special commands and with excellent performance.

(read)

FX!32

A PROFILE-DIRECTED BINARY TRANSLATOR

—1998

Because Digital's Alpha architecture provides the world's fastest processors, many applications, especially those requiring high processor performance, have been ported to it. However,

😞 many other applications are available only under the x86 architecture. 😞

We designed Digital FX!32 to make the complete set of applications, both native and x86, available to Alpha. The goal for the software is to provide fast and transparent execution of x86 Win32 applications on Alpha systems. FX!32 achieves its goal by transparently running those applications at speeds comparable to high-performance x86 platforms. Digital FX!32 is a software utility that enables x86 Win32 applications to be run on Windows NT/Alpha platforms. Once FX!32 has been installed, almost all x86 applications can be run on Alpha without special commands and with excellent performance.

Three significant innovations of Digital FX!32 include transparent operation, interface to native APIs, and, most importantly, profile-directed binary translation.

Claim: the first system to exploit this combination of emulation, profile generation, and binary translation.

What's really cool about this is that applications run unmodified on a different architecture, and effectively a different operating system. This makes transitioning architectures much easier, especially if the destination is more powerful.

Were one to launch such a system nowadays, it would still be a Big Deal.

WHY, SOMETIMES
I'VE BELIEVED AS
MANY AS SIX
IMPOSSIBLE THINGS
BEFORE BREAKFAST.

Carroll, L. 1865. *Alice's Adventures in Wonderland*



Lewis Carroll, on speculation.

We talked about speculation a bit in the context of HotSpot...

Let's look at some deeper speculation, and much, *much* cooler emulation.

TRANSMETA

THE TECHNOLOGY BEHIND CRUSOE PROCESSORS

— 2000

Low-power x86-Compatible Processors Implemented with

CODE MORPHING SOFTWARE

The new technology is fundamentally software-based: the power savings come from replacing large numbers of transistors with *software*.

(read 3 paragraphs) This was the hottest startup at the end of the 90's.

For those of you who remember Slashdot: Slashdot was abuzz with it.

“Code morphing” is a fancy way to say “dynamic binary translation”.

Imagine:

- * Hardware that presents as x86, but isn't actually x86.
- * Hardware that can get faster through firmware updates.
- * A stable-seeming ISA, with hardware that can radically change at each generation.

THE VLIW'S NATIVE INSTRUCTION SET BEARS NO RESEMBLANCE TO THE X86 INSTRUCTION SET; IT HAS BEEN DESIGNED PURELY FOR FAST LOW-POWER IMPLEMENTATION.

(read)

THE VLIW'S NATIVE INSTRUCTION SET BEARS NO RESEMBLANCE TO THE X86 INSTRUCTION SET; IT HAS BEEN DESIGNED PURELY FOR FAST LOW-POWER IMPLEMENTATION. THE SURROUNDING SOFTWARE LAYER GIVES X86 PROGRAMS THE IMPRESSION THAT THEY ARE RUNNING ON X86 HARDWARE.

(read)

THE VLIW'S NATIVE INSTRUCTION SET BEARS NO RESEMBLANCE TO THE X86 INSTRUCTION SET; IT HAS BEEN DESIGNED PURELY FOR FAST LOW-POWER IMPLEMENTATION. THE SURROUNDING SOFTWARE LAYER GIVES X86 PROGRAMS THE IMPRESSION THAT THEY ARE RUNNING ON X86 HARDWARE. THE SOFTWARE LAYER IS CALLED CODE MORPHING SOFTWARE BECAUSE IT DYNAMICALLY **"MORPHS"** X86 INSTRUCTIONS INTO VLIW INSTRUCTIONS.

(read)

THE VLIW'S NATIVE INSTRUCTION SET BEARS NO RESEMBLANCE TO THE X86 INSTRUCTION SET; IT HAS BEEN DESIGNED PURELY FOR FAST LOW-POWER IMPLEMENTATION. THE SURROUNDING SOFTWARE LAYER GIVES X86 PROGRAMS THE IMPRESSION THAT THEY ARE RUNNING ON X86 HARDWARE. THE SOFTWARE LAYER IS CALLED CODE MORPHING SOFTWARE BECAUSE IT DYNAMICALLY "MORPHS" X86 INSTRUCTIONS INTO VLIW INSTRUCTIONS. THE CODE MORPHING SOFTWARE INCLUDES A NUMBER OF ADVANCED FEATURES TO ACHIEVE GOOD SYSTEM-LEVEL PERFORMANCE.

(read)

THE VLIW'S NATIVE INSTRUCTION SET BEARS NO RESEMBLANCE TO THE X86 INSTRUCTION SET; IT HAS BEEN DESIGNED PURELY FOR FAST LOW-POWER IMPLEMENTATION. THE SURROUNDING SOFTWARE LAYER GIVES X86 PROGRAMS THE IMPRESSION THAT THEY ARE RUNNING ON X86 HARDWARE. THE SOFTWARE LAYER IS CALLED CODE MORPHING SOFTWARE BECAUSE IT DYNAMICALLY "MORPHS" X86 INSTRUCTIONS INTO VLIW INSTRUCTIONS. THE CODE MORPHING SOFTWARE INCLUDES A NUMBER OF ADVANCED FEATURES TO ACHIEVE GOOD SYSTEM-LEVEL PERFORMANCE. CODE MORPHING SUPPORT FACILITIES ARE ALSO BUILT INTO THE UNDERLYING CPUs.

(read)

THE VLIW'S NATIVE INSTRUCTION SET BEARS NO RESEMBLANCE TO THE X86 INSTRUCTION SET; IT HAS BEEN DESIGNED PURELY FOR FAST LOW-POWER IMPLEMENTATION. THE SURROUNDING SOFTWARE LAYER GIVES X86 PROGRAMS THE IMPRESSION THAT THEY ARE RUNNING ON X86 HARDWARE. THE SOFTWARE LAYER IS CALLED CODE MORPHING SOFTWARE BECAUSE IT DYNAMICALLY "MORPHS" X86 INSTRUCTIONS INTO VLIW INSTRUCTIONS. THE CODE MORPHING SOFTWARE INCLUDES A NUMBER OF ADVANCED FEATURES TO ACHIEVE GOOD SYSTEM-LEVEL PERFORMANCE. CODE MORPHING SUPPORT FACILITIES ARE ALSO BUILT INTO THE UNDERLYING CPUs. IN OTHER WORDS, THE TRANSMETA DESIGNERS HAVE JUDICIOUSLY RENDERED SOME FUNCTIONS IN HARDWARE AND SOME IN SOFTWARE, ACCORDING TO THE PRODUCT DESIGN GOALS AND CONSTRAINTS.

(read)

THE VLIW'S NATIVE INSTRUCTION SET BEARS NO RESEMBLANCE TO THE X86 INSTRUCTION SET; IT HAS BEEN DESIGNED PURELY FOR FAST LOW-POWER IMPLEMENTATION. THE SURROUNDING SOFTWARE LAYER GIVES X86 PROGRAMS THE IMPRESSION THAT THEY ARE RUNNING ON X86 HARDWARE. THE SOFTWARE LAYER IS CALLED CODE MORPHING SOFTWARE BECAUSE IT DYNAMICALLY "MORPHS" X86 INSTRUCTIONS INTO VLIW INSTRUCTIONS. THE CODE MORPHING SOFTWARE INCLUDES A NUMBER OF ADVANCED FEATURES TO ACHIEVE GOOD SYSTEM-LEVEL PERFORMANCE. CODE MORPHING SUPPORT FACILITIES ARE ALSO BUILT INTO THE UNDERLYING CPUs. IN OTHER WORDS, THE TRANSMETA DESIGNERS HAVE JUDICIOUSLY RENDERED SOME FUNCTIONS IN HARDWARE AND SOME IN SOFTWARE, ACCORDING TO THE PRODUCT DESIGN GOALS AND CONSTRAINTS. DIFFERENT GOALS AND CONSTRAINTS IN FUTURE PRODUCTS MAY RESULT IN DIFFERENT HARDWARE-SOFTWARE PARTITIONING.

Let's unpack this.

VLIW: very long instruction word. On Crusoe, each actual instruction is composed of 4 different simple instructions.

The hardware executes VLIW "molecules" in-order (*not* superscalar), but that's 4 instructions at a time.

It's statically out of order from what the original x86 program contained, because of binary translation. (cont'd)

THE VLIW'S NATIVE INSTRUCTION SET BEARS NO RESEMBLANCE TO THE X86 INSTRUCTION SET; IT HAS BEEN DESIGNED PURELY FOR FAST LOW-POWER IMPLEMENTATION. THE SURROUNDING SOFTWARE LAYER GIVES X86 PROGRAMS THE IMPRESSION THAT THEY ARE RUNNING ON X86 HARDWARE. THE SOFTWARE LAYER IS CALLED CODE MORPHING SOFTWARE BECAUSE IT DYNAMICALLY "MORPHS" X86 INSTRUCTIONS INTO VLIW INSTRUCTIONS. THE CODE MORPHING SOFTWARE INCLUDES A NUMBER OF ADVANCED FEATURES TO ACHIEVE GOOD SYSTEM-LEVEL PERFORMANCE. CODE MORPHING SUPPORT FACILITIES ARE ALSO BUILT INTO THE UNDERLYING CPUs. IN OTHER WORDS, THE TRANSMETA DESIGNERS HAVE JUDICIOUSLY RENDERED SOME FUNCTIONS IN HARDWARE AND SOME IN SOFTWARE, ACCORDING TO THE PRODUCT DESIGN GOALS AND CONSTRAINTS. DIFFERENT GOALS AND CONSTRAINTS IN FUTURE PRODUCTS MAY RESULT IN DIFFERENT HARDWARE-SOFTWARE PARTITIONING.

The hardware supports transactional memory to enable speculative optimizations, including speculative reordering of loads and stores.

x86 instructions are initially interpreted, and if deemed "hot" they're JiT-compiled in a hidden part of the hardware, effectively ring -1.

I don't know about you, but this wall of text gets me super excited!

At this point nothing can impress you anymore, right?

QEMU

A FAST AND PORTABLE DYNAMIC TRANSLATOR

— 2005

We present the internals of QEMU, a fast machine emulator using an original portable dynamic translator.

(read)

QEMU

A FAST AND PORTABLE DYNAMIC TRANSLATOR

— 2005

We present the internals of QEMU, a fast machine emulator using an original portable dynamic translator. It emulates several CPUs (x86, PowerPC, ARM and Sparc) on several hosts (x86, PowerPC, ARM, Sparc, Alpha and MIPS).

(read)

QEMU

A FAST AND PORTABLE DYNAMIC TRANSLATOR

— 2005

We present the internals of QEMU, a fast machine emulator using an original portable dynamic translator. It emulates several CPUs (x86, PowerPC, ARM and Sparc) on several hosts (x86, PowerPC, ARM, Sparc, Alpha and MIPS). QEMU supports full system emulation in which a complete and unmodified operating system is run in a virtual machine and Linux user mode emulation where a Linux process compiled for one target CPU can be run on another CPU.

QEMU sounds less grandiose than Transmeta, but it still impresses me.

It's extremely malleable, does interesting optimizations, and is very portable.

The paper is super short, but what this project does is close to magic, which is why it's used everywhere nowadays. I won't go into virtual machines, but there's also plenty of interesting work there, particularly when using special hardware to help virtualize the guest operating system. In particular, the virtualization extensions are quite powerful, and so are paravirtualization techniques.

VALGRIND

A FRAMEWORK FOR HEAVYWEIGHT DYNAMIC BINARY INSTRUMENTATION

— 2007

We focus on Valgrind's unique support for *shadow values*—a powerful but previously little-studied and difficult-to-implement dynamic binary analysis technique, which requires a tool to shadow every register and memory value with another value that describes it.

Let's build on the instrumentation capabilities of tools such as ATOM, and go extremely far in the analysis capabilities. In particular, Valgrind uses *shadow values* to track extra facts about registers and memory to unlock new superpowers.

Folks are used to Valgrind as a use-after-free or out-of-bounds tool, but that's just one of Valgrind's many capabilities.

The true feat of Valgrind is its tooling infrastructure: it lets you write JIT manipulations without having to write a JIT compiler.

TRACE-BASED
JUST-IN-TIME TYPE SPECIALIZATION
FOR DYNAMIC LANGUAGES

— 2009

Identifies frequently executed loop traces at run-time and then generates machine code on the fly that is specialized for the actual dynamic types occurring on each path through the loop.

We have implemented a dynamic compiler for JavaScript based on our technique and we have measured speedups of **10× and more** for certain benchmark programs.

(read 2 paragraphs)

There's a lot to say about the evolution of JIT compilers once browsers started JIT-compiling JavaScript. Many person-decades having gone into optimizing JavaScript, and your mobile devices' batteries thank them for their efforts. I'll only mention this one paper because the use of trace compilation is neat and goes directly back to Oberon. (cont'd)

TRACE-BASED
JUST-IN-TIME TYPE SPECIALIZATION
FOR DYNAMIC LANGUAGES

— 2009

Identifies frequently executed loop traces at run-time and then generates machine code on the fly that is specialized for the actual dynamic types occurring on each path through the loop.

We have implemented a dynamic compiler for JavaScript based on our technique and we have measured speedups of **10× and more** for certain benchmark programs.

PIN and Transmeta also used these techniques (“traces” and “regions”), where instead of dealing with each function at a time and sometime inlining, you instead ignore functions and trace the execution flow itself. What’s neat about this is that it follows particular flow of execution, which matches up nicely with, for example, types remaining the same over a particular call sequence.

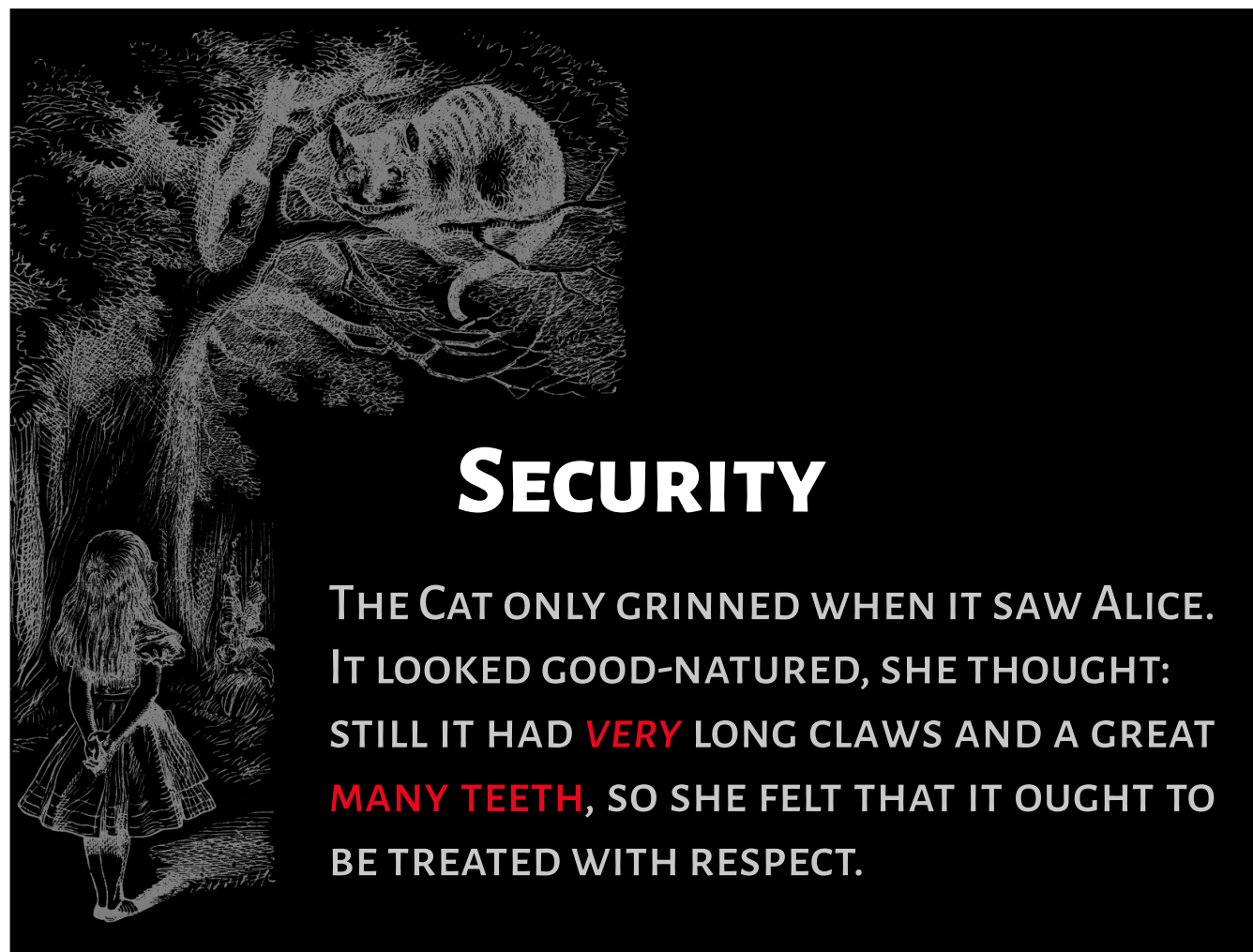
**“DYNAMIC, OR JUST-IN-TIME, COMPILATION IS
AN OLD IMPLEMENTATION TECHNIQUE
WITH A FRAGMENTED HISTORY. BY
COLLECTING THIS HISTORICAL
INFORMATION TOGETHER, WE HOPE TO
SHORTEN THE VOYAGE OF REDISCOVERY.”**

Aycock, J. 2003. A Brief History of Just-In-Time

Bastien, JF. 2020. CppCon—Just-in-Time compilation

This completes our Brief History...

But there's one more thing I want to mention.



SECURITY

THE CAT ONLY GRINNED WHEN IT SAW ALICE. IT LOOKED GOOD-NATURED, SHE THOUGHT: STILL IT HAD **VERY** LONG CLAWS AND A GREAT **MANY TEETH**, SO SHE FELT THAT IT OUGHT TO BE TREATED WITH RESPECT.

I said I wouldn't go into downsides of JiT compilation too much, but one I want to dig int a bit is security.

Good news about JiTs: you're now shipping a compiler!

Bad news about JiTs: you're now shipping a compiler!

Here are a few more publications...

NATIVE CLIENT

A SANDBOX FOR PORTABLE, UNTRUSTED X86 NATIVE CODE

— 2009

1. Once loaded into the memory, the binary is not writable.
2. The binary is statically linked at a start address of zero, with the first byte of text at 64K.
3. All indirect control transfers use a `nacl jmp` pseudo-instruction.
4. The binary is padded up to the nearest page with at least one `hlt` instruction.
5. The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
6. All valid instruction addresses are reachable by a fallthrough disassembly that starts at the load (base) address.
7. All direct control transfers target valid instructions.

On top of x86-32, variants for x86-64, 32-bit ARMv7 ARM, and MIPS32.

Not technically a JiT, but interesting because of the proof aspect:

7 rules for verifiable sandboxing at the instruction level, little reliance on the OS to enforce invariants of the inner sandbox. What's cute for x86-32 is that NaCl uses segmentation to implement its Harvard architecture and separate data from code.

Similar ruleset on other ISAs, but data / code separation is enforced through address masking for all memory operations. (cont'd)

NATIVE CLIENT

A SANDBOX FOR PORTABLE, UNTRUSTED X86 NATIVE CODE

— 2009

1. Once loaded into the memory, the binary is not writable.
2. The binary is statically linked at a start address of zero, with the first byte of text at 64K.
3. All indirect control transfers use a `nacl jmp` pseudo-instruction.
4. The binary is padded up to the nearest page with at least one `hlt` instruction.
5. The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
6. All valid instruction addresses are reachable by a fallthrough disassembly that starts at the load (base) address.
7. All direct control transfers target valid instructions.

To reiterate: NaCl knows nothing of the source code for the application it runs, neither does it trust the compiler which compiled it. It only looks at the object code, and if the 7 rules are followed then the binary is known to not escape its sandbox.

This technique can be used in jITs as well.

Portable Native Client made NaCl portable, by running LLVM inside an NaCl sandbox, having it generate NaCl code, and validating the generated code before executing it, therefore it doesn't trust that LLVM is correct to be secure.

ATTACKING CLIENT SIDE JIT COMPILERS

— 2011

While the concept of a compiler producing incorrect code is not new, JIT engines raise the stakes by performing this compilation at runtime while potentially under the influence of untrusted inputs.

(read)

ATTACKING CLIENT SIDE JIT COMPILERS

— 2011

While the concept of a compiler producing incorrect code is not new, JIT engines raise the stakes by performing this compilation at runtime while potentially under the influence of untrusted inputs. The Common Weakness Enumeration guide only contains one mention of compilation related vulnerabilities.

(read)

ATTACKING CLIENT SIDE JIT COMPILERS

— 2011

While the concept of a compiler producing incorrect code is not new, JIT engines raise the stakes by performing this compilation at runtime while potentially under the influence of untrusted inputs. The Common Weakness Enumeration guide only contains one mention of compilation related vulnerabilities. This CWE entry concerns a compiler optimizing away a security check inserted by a developer.

(read)

ATTACKING CLIENT SIDE JiT COMPILERS

— 2011

While the concept of a compiler producing incorrect code is not new, JiT engines raise the stakes by performing this compilation at runtime while potentially under the influence of untrusted inputs. The Common Weakness Enumeration guide only contains one mention of compilation related vulnerabilities. This CWE entry concerns a compiler optimizing away a security check inserted by a developer. One concern with complex JiT engines is a compiler producing incorrect code at runtime through either a miscalculation of code locations, mishandled register states or a bad pointer dereference, to name a few.

This paper had a pretty big impact on JiT design for untrusted inputs, particularly in browsers.

What's nice is that it also proposed concrete mitigations for the flaws it outlined.

Since then, JiTs have changed significantly, in some cases thanks to hardware support.

Google Project Zero has many great deep-dives into various JiT compilers' vulnerabilities.

EMSCRIPTEN

AN LLVM-TO-JAVASCRIPT COMPILER

— 2011

We presented Emscripten, an LLVM-to-JavaScript compiler, which opens up numerous opportunities for running code written in languages other than JavaScript on the web, including some not previously possible.

(read)

EMSCRIPTEN

AN LLVM-TO-JAVASCRIPT COMPILER

— 2011

We presented Emscripten, an LLVM-to-JavaScript compiler, which opens up numerous opportunities for running code written in languages other than JavaScript on the web, including some not previously possible. Emscripten can be used to, among other things, compile real-world C and C++ code and run that on the web. In addition, by compiling the runtimes of languages which are implemented in C and C++, we can run them on the web as well, for example Python and Lua.

What's particularly neat about Emscripten is the asm.js approach that followed it.

It has a clever use of JavaScript's type system to efficiently represent static languages, despite JavaScript nominally being a dynamic language.

None of these features had originally been intended for the use asm.js made of them.

BRINGING THE WEB UP TO SPEED WITH

WEBASSEMBLY

— 2017

Engineers from the four major browser vendors have risen to the challenge and collaboratively designed a portable low-level bytecode called WebAssembly.

(read)

BRINGING THE WEB UP TO SPEED WITH

WEBASSEMBLY

— 2017

Engineers from the four major browser vendors have risen to the challenge and collaboratively designed a portable low-level bytecode called WebAssembly. It offers compact representation, efficient validation and compilation, and safe low to no-overhead execution.

(read)

BRINGING THE WEB UP TO SPEED WITH

WEBASSEMBLY

— 2017

Engineers from the four major browser vendors have risen to the challenge and collaboratively designed a portable low-level bytecode called WebAssembly. It offers compact representation, efficient validation and compilation, and safe low to no-overhead execution. Rather than committing to a specific programming model, WebAssembly is an abstraction over modern hardware, making it language-, hardware-, and platform-independent, with use cases beyond just the Web.

(read)

BRINGING THE WEB UP TO SPEED WITH

WEBASSEMBLY

— 2017

Engineers from the four major browser vendors have risen to the challenge and collaboratively designed a portable low-level bytecode called WebAssembly. It offers compact representation, efficient validation and compilation, and safe low to no-overhead execution. Rather than committing to a specific programming model, WebAssembly is an abstraction over modern hardware, making it language-, hardware-, and platform-independent, with use cases beyond just the Web. WebAssembly has been designed with a formal semantics from the start.

(read)

BRINGING THE WEB UP TO SPEED WITH

WEBASSEMBLY

— 2017

Engineers from the four major browser vendors have risen to the challenge and collaboratively designed a portable low-level bytecode called WebAssembly. It offers compact representation, efficient validation and compilation, and safe low to no-overhead execution. Rather than committing to a specific programming model, WebAssembly is an abstraction over modern hardware, making it language-, hardware-, and platform-independent, with use cases beyond just the Web. WebAssembly has been designed with a formal semantics from the start. We describe the motivation, design and formal semantics of WebAssembly and provide some preliminary experience with implementations.

The marriage of PNaCl and Emscripten / asm.js.

With a strong execution model: the virtual ISA is well defined. It pretends to be a modern CPU, but is actually portable.

There's still much more work needed in the space of secure virtual machines, but we're getting there, one publication at a time...

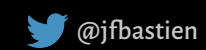
And this concludes the talk!

JUST-IN-TIME COMPILATION

A lecture on the last 60 years

JF BASTIEN


Software architect





TRI-AD

JF BASTIEN

 [@jfbastien](https://twitter.com/jfbastien)