Introducing the B3 JIT Compiler

Feb 15, 2016

by Filip Pizlo

@filpizlo

As of r195562, WebKit's FTL JIT (Faster Than Light Just In Time compiler) now uses a new backend on OS X. The Bare Bones Backend, or B3 for short, replaces LLVM as the low-level optimizer in the FTL JIT. While LLVM is an excellent optimizer, it isn't specifically designed for the optimization challenges of dynamic languages like JavaScript. We felt that we could get a bigger speed-up if we combined the best of the FTL architecture with a new compiler backend that incorporated what we learned from using LLVM but was tuned specifically for our needs. This post reviews how the FTL JIT works, describes the motivation and design of B3, and shows how B3 fits into WebKit's compiler infrastructure.

The Philosophy Behind the FTL JIT

The FTL JIT was designed as a marriage between a high-level optimizing compiler that inferred types and optimized away type checks, and a low-level optimizing compiler that handled traditional optimizations like register allocation. We used our existing DFG (Data Flow Graph) compiler for high level optimizations and we used LLVM for the low-level compiler. This architecture change worked out great by providing an additional performance boost to optimize hot code paths.

The high-level compiler and the low-level compiler were glued together using a lowering phase called `FTL::LowerDFGToLLVM`. This phase abstracted the low-

level compiler's intermediate representation behind an interface called `FTL::Output`. Abstracting the low-level compiler meant that we had some flexibility in choosing what kind of backend to use. We chose LLVM for the initial FTL implementation because it was freely available and did a great job at generating efficient code. It meant that WebKit could enjoy the benefits of a mature optimizing compiler without requiring WebKit engineers to implement it.

The original role of the FTL JIT was to provide additional low-level optimizations on code that already had sufficient high-level optimizations by one of our more latency-focused lower-tier JITs, like the DFG JIT. But as we optimized the FTL JIT, we found ourselves adding powerful high-level optimizations to it, without also implementing equivalent optimizations in the lower-tier JITs. This made sense, because the FTL's use of a low-level optimizing backend meant that high-level optimizations could be sloppier, and therefore faster to run and quicker to write. For example, transformations written specifically for the FTL can leave behind dead code, unreachable code, unfused load/compare/branch sequences, and suboptimal SSA graphs because LLVM will clean these things up.

The FTL JIT became more than just the sum of its parts, and a large fraction of its throughput advantage had nothing to do with LLVM. Here is an incomplete list of optimizations that the FTL JIT does but that are absent from our other JITs and from LLVM:

- sophisticated must-points-to analysis to eliminate or sink object allocations.
- algebraic analysis to prove bounds on ranges of integers.

Sometimes, the FTL implements an optimization even though LLVM also has it. LLVM has the following optimizations, but we use our own FTL implementation of them because they need awareness of JavaScript semantics to be completely effective:

- global common subexpression elimination.
- loop-invariant code motion.

The FTL has grown so powerful that the basic strategy for improving WebKit JavaScript performance on any test is to first ensure that the test benefits from the FTL. But on some platforms, and on some workloads that either don't run for enough total time or that have a large amount of slow-to-compile code, the bottleneck is simply FTL's compile time. In some programs, we pay the penalty of running FTL compiler threads but by the time the compiler finishes, the program is already done running those functions. The time has come to tackle compile times, and because the overwhelming majority – often three quarters or more – of the FTL compile time is LLVM, that meant reconsidering whether LLVM should be the FTL backend.

Designing a New Compiler Backend

We knew that we could get a speed-up on lots of real-world JavaScript code if we could get all of FTL's high level optimizations without paying the full price of LLVM's compile time. But we also knew that LLVM's low-level optimizations were essential for benchmarks that ran long enough to hide compile time. This was a huge opportunity and a huge risk: beating LLVM's compile times seemed easy, but it would only be a net win if the new backend could compete on throughput.

Competing against LLVM is a tall order. LLVM has years of work behind it and includes comprehensive optimizations for C-like languages. As the FTL JIT project previously showed, LLVM did a great job of optimizing JavaScript programs and its compile times were good enough that using LLVM was a net win for WebKit.

We projected that if we could build a backend that compiled about ten times faster than LLVM while still generating comparable code, then we would get speed-ups on code that currently blocks on LLVM compilation. The opportunity

outweighed the risk. We started working on a prototype in late October 2015. Our intuition was that we needed a compiler that operated at the same level of granularity as LLVM and so allowed the same kinds of optimizations. To make the compiler faster to run, we used a more compact IR (intermediate representation). Our plan was to match LLVM's throughput not only by implementing the same optimizations that LLVM had but also by doing WebKit-specific tuning.

The rest of this section describes the architecture of the Bare Bones Backend with a specific focus on how we planned to make it achieve great throughput while reducing overall FTL compile time.

B3 Intermediate Representations

We knew that we wanted a low-level intermediate representation that exposed raw memory access, raw function calls, basic control flow, and numerical operations that closely matched the target processor's capabilities. But low-level representations mean that lots of memory is needed to represent each function. Using lots of memory also means that the compiler must scan a lot of memory when analyzing the function, which causes the compiler to take lots of cache misses. B3 has two intermediate representations – a slightly higher-level representation called B3 IR and a machine-level one called Assembly IR, or Air for short. The goal of both IRs is to represent low-level operations while minimizing the number of expensive memory accesses needed to analyze the code. This implies four strategies: reduce the overall size of IR objects, reduce the number of IR objects needed to represent typical operations, reduce the need for pointer chasing when walking the IR, and reduce the total number of intermediate representations.

Reducing the Sizes of IR Objects

B3 and Air are designed to use small objects to describe operations in the program. B3 reduces the sizes of IR objects by dropping redundant information that can be lazily recomputed. Air is much more aggressive and reduces the

sizes of objects by having a carefully tuned layout of all objects even when it means making the IR somewhat harder to manipulate.

B3 IR uses SSA (static single assignment), which implies that each variable is only assigned to once in the text of the program. This creates a one-to-one mapping between variables and the statements that assign to them. For compactness, SSA IRs usually use one object to encapsulate a statement that assigns to a variable and the variable itself. This makes SSA look much like data flow, though B3 is not really a data flow IR since each statement is anchored in a specific place in control flow.

In B3 the class used to mean a variable and the statement that assigns to it is the `Value`. Each value is owned by some basic block. Each basic block contains a sequence of values to execute. Each value will have a list of references to other values (which we call *children* in B3-speak) that represent inputs as well as any meta-data needed for that opcode.

It's tempting to have the child edge be a bidirectional edge. This way, it's cheap to find all of the uses of any value. This is great if you want to replace a value with some other value — an operation so common that LLVM uses the acronym "RAUW" (Replace All Uses With) and defines it in its lexicon. But bidirectional edges are expensive. The edge itself must be represented as an object that contains pointers to both the child and the *parent*, or user of the child. The list of child values becomes an array of Use objects. This is one example of memory use that B3 IR avoids. In B3, you cannot replace all uses of a value with another value in one standalone operation, because values do not know their parents. This saves a substantial amount of memory per value and makes reading the whole IR much cheaper. Instead of "replacing all uses with", B3 optimizations replace values with identities using the `Value::replaceWithIdentity()` method. An identity is a value that has one child and just returns the child. This ends up being cheap. We can change a value into an `Identity` value in-place. Most transformations that perform such

replacements already walk the entire function, or `Procedure` in B3-speak. So, we just arrange for the existing walk over the procedure to also call `Value::performSubstitution()`, which replaces all uses of `Identity` with uses of the `Identity`'s child. This lets us have our cake and eat it, too: B3 phases become faster because fewer memory accesses are required to process a value, and those phases don't gain any asymptotic complexity from their inability to "replace all uses with" because the work is just incorporated into the processing they are already doing.

Air takes an even more extreme approach to reducing size. The basic unit of operation in Air is an `Inst`. An `Inst` is passed by-value. Even though an `Inst` can take a variable number of arguments, it has inline capacity for three of them, which is the common case. So a basic block usually contains a single contiguous slab of memory that completely describes all of its instructions. The Arg object, used to represent arguments, is entirely in-place. It's just a tightly packed bag of bits for describing all of the kinds of arguments that an assembly instruction will see.

Minimizing the Number of IR Objects

It's also important to minimize the number of objects needed to represent a typical operation. We looked at the common operations used in FTL. It's common to branch to a basic block that exits. It's common to add a constant to an integer, convert the integer to a pointer, and use the pointer for loading or storing. It's common to mask the shift amount before shifting. It's common to do integer math with an overflow check. B3 IR and Air make common operations use just one `Value` or `Inst` in the common case. The branch-and-exit operation doesn't require multiple basic blocks because B3 encapsulates it in the `Check` opcode and Air encapsulates it in a special `Branch` instruction. Pointers in B3 IR are just integers, and all load/store opcodes take an optional offset immediate, so that a typical FTL memory access only requires one object. Arithmetic operations in B3 IR carefully balance the semantics of modern hardware and modern languages. Masking the shift amount is a vestige of C semantics and ancient disagreements over what it means

to shift an N-bit integer by more than N bits; none of this makes sense anymore since X86 and ARM both mask the amount for you and modern languages all assume that this is how shifting works. B3 IR saves memory by not requiring you to emit those pointless masks. Math with overflow checks is represented using `CheckAdd`, `CheckSub`, and `CheckMul` opcodes in B3 and `BranchAdd`, `BranchSub`, and `BranchMul` opcodes in Air. Those opcodes all understand that the slow path is a sudden termination of the optimized code. This obviates the need for creating control flow every time the JavaScript program does math.

Reducing Pointer Chasing

Most compiler phases have to walk the entire procedure. It's rare that you write a compiler phase that only does things for a single kind of opcode. Common subexpression elimination, constant propagation, and strength reduction — the most common kinds of optimizations — all handle virtually all opcodes in the IR and so will have to visit all values or instructions in the procedure. B3 IR and Air are optimized to reduce memory bottlenecks associated with such linear scans by using arrays as much as possible and reducing the need to chase pointers. Arrays tend to be faster than linked lists if the most common operation is reading the whole thing. Accessing two values that are next to each other in an array is cheap because they are likely to fall on the same cache line. This means that memory latency is only encountered every N loads from the array, where N is the ratio between cache line size and the size of the value. Usually, cache lines are big: 32 bytes, or enough room for 4 pointers on a 64-bit system, tends to be the minimum these days. Linked lists don't get this benefit. For this reason, B3 and Air represent the procedure as an array of basic blocks, and they represent each basic block as an array of operations. In B3, it's an array (specifically, a `WTF::Vector`) of pointers to value objects. In Air, it's an array of `Inst`s. This means that Air requires no pointer chasing in the common case of walking a basic block. Air is necessarily most extreme about optimizing accesses because Air is lower-level and so will inherently see more code. For example, you can get an immediate's value, a temporary or register's type, and the meta-attributes of

each argument (does it read a value? write a value? how many bits does it access? what type is it?) all without having to read anything other than what is contained inside an `Inst`.

It would seem that structuring a basic block as an array precludes insertion of operations into the middle of the basic block. But again, we are saved by the fact that most transformations must already process the whole procedure since it's so rare to write a transformation that will only affect a small subset of operations. When a transformation wishes to insert instructions, it builds up an `InsertionSet` (here it is in B3 and in Air) that lists the indices at which those values should be inserted. Once a transformation is done with a basic block, it executes the set. This is a linear-time operation, as a function of basic block size, which performs all insertions and all necessary memory reallocation all at once.

Reducing the Number of Intermediate Representations

B3 IR covers all of our use cases of LLVM IR and, to our knowledge, does not preclude us from doing any optimizations that LLVM would do. It does this while using less memory and being cheaper to process, which reduces compile time.

Air covers all of our use cases of LLVM's low-level IRs like selection DAG, machine SSA, and the MC layer. It does this in a single compact IR. This section discusses some of the architectural choices that made it possible for Air to be so simple.

Air is meant to represent the final result of all instruction selection decisions. While Air does provide enough information that you could transform it, it's not an efficient IR for doing anything but specific kinds of transformations like register allocation and calling convention lowering. Therefore, the phase that converts B3 to Air, called `B3::LowerToAir`, must be clever about picking the right Air sequences for each B3 operation. It does this by

performing a backwards greedy pattern matching. It proceeds through each basic block in reverse. When it sees a value, it attempts to traverse both the value and its children, and possibly even its children's children (transitively, many levels deep) to construct an `Inst`. The B3 pattern that matches the resulting `Inst` will involve a root value and some number of *internal* values that are only used by the root and can be computed as part of the resulting `Inst`. The instruction selector *locks* those internal values, which prevents them from emitting their own instruction — emitting them again would be unnecessary since the `Inst` we selected already does what those internal values would have done. This is the reason for selecting in reverse: we want to see the users of a value before seing the value itself, in case there is only one user and that user can incorporate the value's computation into a single instruction.

The instruction selector needs to know which Air instructions are available. This may vary by platform. Air provides a fast query API for determining if a particular instruction can be efficiently handled on the current target CPU. So, the B3 instruction selector cascades through all possible instructions and the patterns of B3 values that match them until it finds one that Air approves of. This is quite fast, since the Air queries are usually folded into constants. It's also quite powerful. We can fuse patterns involving loads, stores, immediates, address computations, compares, branches, and basic math operations to emit efficient instructions on x86 and ARM. This in turn reduces the total number of instructions and reduces the number of registers we need for temporaries, which reduces the number of accesses to the stack.

B3 Optimizations

The design of B3 IR and Air gives us a good foundation for building advanced optimizations. B3's optimizer already includes a lot of optimizations:

- Strength reduction, which encompasses:
  - Control flow graph simplification.

    - Constant folding.
    - Aggressive dead code elimination.
    - Integer overflow check elimination.
    - Lots of miscellaneous simplification rules.
- Flow-sensitive constant folding.
- Global common subexpression elimination.
- Tail duplication.
- SSA fix-up.
- Optimal placement of constant materializations.

We also do optimizations in Air, like dead code elimination, control flow graph simplification, and fix-ups for partial register stalls and spill code pathologies. The most significant optimization done in Air is register allocation; this is discussed in detail in a later section.

Having B3 also gives us the flexibility to tune any of our optimizations specifically for the FTL. The next section goes into the details of some of our FTL-specific features.

Tuning B3 for the FTL

B3 contains some features that were inspired by the FTL. Since the FTL is a heavy user of custom calling conventions, self-modifying code snippets, and on-stack replacement, we knew that we had to make these first-class concepts in B3. B3 handles these using the `Patchpoint` and `Check` family of opcodes. We knew that the FTL generates a lot of repeated branches on the same property in some cases, so we made sure that B3 can specialize these paths. Finally, we knew that we needed an excellent register allocator that worked well out of the box and didn't require a lot of tuning. Our register allocation strategy deviates significantly from LLVM. So, we have a section that describes our thinking.

B3 Patchpoints

The FTL is a heavy user of custom calling conventions, self-modifying code snippets, and on-stack replacement. Although LLVM does support these capabilities, it treats them as experimental intrinsics, which limits optimization. We designed B3 from the ground up to support these

concepts, so they get the same treatment as any other kind of operation.

LLVM supports these features using the patchpoint intrinsic. It allows the client of LLVM to emit custom machine code for implementing an operation that LLVM models like a call. B3 supports these features with a native `Patchpoint` opcode. B3's native support for patching provides important optimizations:

- It's possible to pin an arbitrary argument to an arbitrary register or stack slot. It's also possible to request that an argument gets any register, or that it gets represented using whatever is most convenient to the compiler (register, stack slot, or constant).
- It's also possible to constrain how the result is returned. It's possible to pin the result to an arbitrary register or stack slot.
- The patchpoint does not need to have a predetermined machine code size, which obviates the need to pad the code inside the patchpoint with no-ops. LLVM's patchpoints require a predetermined machine code size. LLVM's backend emits that many bytes of no-ops in place of the patchpoint and reports the location of those no-ops to the client. B3's patchpoints allow the client to emit code in tandem with its own code generator, obviating the need for no-ops and predetermined sizes.
- B3's patchpoints can provide details about effects, like whether the patchpoint can read or write memory, cause abrupt termination, etc. If the patchpoint does read or write memory, it's possible to provide a bound on the memory that is accessed using the same aliasing language that B3 uses to put a bound on the effects of loads, stores, and calls. LLVM has a language for describing abstract regions of the heap (called TBAA), but you cannot use it to describe a bound on the memory read or written by a patchpoint.

Patchpoints can be used to generate arbitrarily complex snippets of potentially self-modifying code. Other WebKit JITs have access to a wealth of utilities for generating code that will be patched later. It's exposed through our internal

`MacroAssembler` API. When creating a patchpoint, you give it a code generation callback in the form of a C++ lamda that the patchpoint carries with it. The lambda is invoked once the patchpoint is emitted by Air. It's given a MacroAssembler reference (specifically, a CCallHelpers reference, which augments the assembler with some higher-level utilities) and an object that describes the mapping between the patchpoint's high-level operands in B3 and machine locations, like registers and stack slots. This object, called the `StackmapGenerationParams`, also makes available other data about the code generation state, like the set of used registers. The contract between B3 and the patchpoint's generator is that:

- The generator must generate code that either terminates abrubtly or falls through. It cannot jump to other code that B3 emitted. For example, it would be wrong to have one patchpoint jump to another.
- The patchpoint must account for the kinds of effects that the generator can do using an `Effects` object. For example, the patchpoint cannot access memory or terminate abruptly without this being noted in the effects. The default effects of any patchpoint claims that it can access any memory and possibly terminate abruptly.
- The patchpoint must not clobber any registers that the StackmapGenerationParams claims are in use. That is, the patchpoint may use them so long as it saves them first and restores them prior to falling through. Patchpoints can request any number of scratch GPRs and FPRs, so scavenging for registers is not necessary.

Because B3 patchpoints never require the machine code snippet to be presized, they are cheaper than LLVM's patchpoints and can be used more widely. Because B3 makes it possible to request scratch registers, the FTL is less likely to spill registers in code it emits into its patchpoints.

Here is an example of how a client of B3 would emit a patchpoint. This is a classic kind of patchpoint for dynamic languages: it loads a value from memory at an offset that

can be patched after the compiler is done. This example illustrates how natural it is to introduce patchable code into a B3 procedure.

```
// Emit a load from basePtr at an offset that can be
// compiler is done. Returns a box that holds the lo
// that patchable offset and the value that was load
std::pair<Box<CodeLocationDataLabel32>, Value*>
emitPatchableLoad(Value* basePtr) {
    // Create a patchpoint that returns Int32 and ta
    // argument. Using SomeRegister means that the v
    // some register chosen by the register allocato
    PatchpointValue* patchpoint =
        block->appendNew<PatchpointValue>(procedure,
    patchpoint->appendSomeRegister(basePtr);

    // A box is a reference-counted memory location.
    // of the offset immediate in memory once all co
    auto offsetPtr = Box<CodeLocationDataLabel32>::c

    // The generator is where the magic happens. It'
    // during final code generation from Air to mach
    patchpoint->setGenerator(
        [=] (CCallHelpers& jit, const StackmapGenera
            // The 'jit' will let us emit code inlin
            // is generating. The 'params' give us t
            // result (params[0]) and the register u
            // (params[1]). This emits a load instru
            // 32-bit offset. The 'offsetLabel' is t
            // immediate in the current assembly buf
            CCallHelpers::DataLabel32 offsetLabel =
                jit.load32WithAddressOffsetPatch(
                    CCallHelpers::Address(params[1].

            // We can add a link task to populate th
            // very end.
            jit.addLinkTask(
                [=] (LinkBuffer& linkBuffer) {
                    // At link time, we can get the
                    // label, since at this point we
                    // code into its final resting p
```

```
                    *offsetPtr = linkBuffer.location
                });
        });

    // Tell the compiler about the things we won't d
    // the worst. But, we know that this won't write
    // that this will never return or trap.
    patchpoint->effects.writes = HeapRange();
    patchpoint->effects.exitsSideways = false;

    // Return the offset box and the patchpoint, sin
    // is the result of the load.
    return std::make_pair(offsetPtr, patchpoint);
}
```

Patchpoints are just half of B3's dynamic language support. The other half is the Check family of opcodes, used to implement on-stack replacement, or OSR for short.

OSR in B3

JavaScript is a dynamic language. It doesn't have the kinds of types and operations that are fast to execute natively. The FTL JIT turns those operations into fast code by speculating about their behavior. Speculation implies bailing from the FTL JIT code if some undesirable condition is encountered. This ensures that subsequent code does not have to check this condition again. Speculation uses OSR to handle bailing from the FTL JIT. OSR encompasses two strategies:

- Arranging for the runtime to switch all future execution of a function to a version compiled by a non-optimizing (i.e. baseline) JIT. This happens when the runtime detects that some condition has arisen that the FTL JIT did not expect, and so the optimized code is no longer valid. WebKit does this by rewriting all call linking data structures to point to the baseline entrypoint. If the function is currently on the stack, WebKit scribbles over all *invalidation points* inside the optimized function. The invalidation points are invisible in the generated code – they are simply places where it's safe to overwrite the

machine code with a jump to an OSR handler, which figures out how to reshape the stack frame to make it look like the baseline stack frame and then jump to the appropriate place in baseline code.
- Emitting code that performs some check and jumping to an OSR handler if the check fails. This allows subsequent code to assume that the check does not need to be repeated. For example, the FTL speculates that integer math doesn't overflow, which enables it to use integers to represent many of JavaScript's numbers, which semantically behave like doubles. The FTL JIT also emits a lot of speculative type checks, like quickly checking if a value is really an object with some set of fields or checking if a value is really an integer.

An invalidation point can be implemented using `Patchpoint`. The patchpoint's generator emits no code other than to warn our MacroAssembler that we want a label at which we might scribble a jump later. Our MacroAssembler has long supported this. It will emit a jump-sized no-op at this label only if the label sits too close to other control flow. So long as there is no other control flow, nothing will be emitted. The patchpoint takes all of the live state that the OSR handler will need. When the OSR handler runs, it will query the data left behind by the StackmapGenerationParams to figure out which registers or stack locations contain which part of the state. Note that this may happen long after B3 has finished compiling, which is why B3 represents this data in by-value objects called `B3::ValueRep`. This is not unlike what we did in LLVM, though because LLVM did not emit instructions using our MacroAssembler, we had to use a separate mechanism to warn that we repatch with a jump.

For OSR that involves an explicit check, the `Patchpoint` opcode would be sufficient but not necessarily efficient. In LLVM, we modeled each of these OSR exits as a basic block that called a patchpoint and passed all live state as arguments. This required representing each speculative check as:

- A branch on the predicate being checked.

- A basic block to represent the code to execute after the check succeeds.
- A basic block to represent the OSR exit. This basic block included a patchpoint that takes all of the state needed to reconstruct the baseline stack frame as its arguments.

This accurately represents what it means to speculate, but it does so by breaking basic blocks. Each check breaks its basic block in two and then introduces another basic block for the exit. This doesn't just increase compile times. It weakens any optimizations that become conservative around control flow. We found that this affected many things in LLVM, including instruction selection and dead store elimination.

B3 is designed to handle OSR without breaking basic blocks. We do this using the `Check` opcode. A `Check` is like a marriage between a `Branch` and a `Patchpoint`. Like a `Branch`, it takes a predicate as input. Just like it does for `Branch`, the instruction selector supports compare/branch fusion for `Check`. Like a `Patchpoint`, a `Check` also takes additional state and a generator. The `Check` is not a basic block terminal. It encapsulates branching on a predicate, emitting an OSR exit handler for when the predicate is true, and falling through to the next instruction when the predicate is false. The generator passed to a `Check` emits code on an out-of-line path linked to from the `Check`'s branch. It's not allowed to fall through or jump back into B3 code. B3 is allowed to assume that falling through a `Check` proves that the predicate was false.

We have found that `Check` is an essential optimization. It dramatically reduces the amount of control flow required to represent speculative operations. Like LLVM, B3 also has some optimizations that become conservative around actual control flow, but they handle `Check`s just fine.

Tail Duplication

Code generated by FTL often has many repeated branches on the same property. These can be eliminated using

optimizations like jump threading and tail duplication. Tail duplication is a bit more powerful, since it allows for a bounded amount of code specialization even before the compiler realizes what kinds of optimizations would be revealed by that specialization. LLVM doesn't do tail duplication until the backend, and so misses many high-level optimization opportunities that it would reveal. B3 has tail duplication, and adding this optimization resulted in a >2% speed-up in the Octane geomean score. LLVM used to have tail duplication, but it was removed in 2011. The discussions around the time of its removal seem to indicate that it adversely affected compile times and was not profitable for clang. We also find that it hurts compile times a bit — but unlike clang, we see an overall speed-up. We are fine with losing some compile time if it's a net win on the benchmarks we track. This is an example of web-specific tuning that we can do freely in B3.

Register Allocation

Register allocation is still a thriving area of exploration in computer science. LLVM has overhauled their register allocator in the last five years. GCC overhauled their allocator eight years ago and is currently still in the process of another overhaul.

B3 does register allocation over Air. Air is like assembly code, but it allows temoraries to be used anywhere that a register would have been allowed by the target processor. It's then the job of the register allocator to transform the program so that it refers to registers and spill slots instead of temporaries. Later phases transform spill slots into concrete stack addresses.

Choosing a register allocator is hard. The history of LLVM and GCC seem to suggest that a compiler's original choice of register allocator is rarely the final choice. We decided to use the classic graph-coloring register allocator called Iterated Register Coalescing, or IRC for short. This algorithm is very concise, and we had past experience implementing and tuning it. Other allocators, like the ones used in LLVM and GCC, don't have such detailed documentation, and so

would have taken longer to integrate into Air. Our initial implementation directly turned the pseudocode in the IRC paper into real code. Our current performance results do not show a significant difference in the quality of generated code between LLVM and B3, which seems to imply that this register allocator does about as good of a job as LLVM's Greedy register allocator. Besides adding a very basic fix-up phase and tuning the implementation to reduce reliance on hashtables, we have not changed the algorithm. We like that IRC makes it easy to model the kinds of register constraints that we encounter in patchpoints and in crazy X86 instructions, and it's great that IRC has a smart solution for eliminating pointless variable assignments by coalescing variables. IRC is also fast enough that it doesn't prevent us from achieving our goal of reducing compile times by an order of magnitude relative to LLVM. The performance evaluation section at the end of this post goes into greater detail about the trade-offs between our IRC implementation and LLVM's Greedy register allocator. See here for our IRC implementation.

We plan to revisit the choice of register allocator. We should pick whatever register allocator results in the best performance on real-world web workloads. Based on our performance results so far, we have an open bug about trying LLVM's Greedy register allocator in Air.

Summary of B3's Design

B3 is designed to enable the same kinds of optimizations as we previously got from LLVM while generating code in less time. It also enables optimizations specific to the FTL JIT, like precise modeling of the effects of self-modifying code. The next section describes how B3 fits into the FTL JIT.

Fitting the B3 JIT Into the FTL JIT

The FTL JIT was already designed to abstract over the intermediate representation of its backend. We did this because LLVM's C API is verbose and often requires many instructions for what the FTL thinks should be one instruction, like loading a value from a pointer and offset.

Using our abstraction meant that we could make it more concise. This same abstraction allowed us to “swap in” the B3 JIT without having to rewrite the entire FTL JIT.

Most of the changes to the FTL JIT were in the form of beneficial refactorings that took advantage of B3’s patchpoints. LLVM patchpoints require the FTL to intercept LLVM’s memory allocations in order to get a pointer to a blob of memory called a “stackmap”. The way you get the blob differs by platform because LLVM matches its section naming strategy to the dynamic linker that the platform uses. Then the FTL must parse the binary stackmap blob to produce an object-oriented description of the moral equivalent of what B3 calls the StackmapGenerationParams: the list of in-use registers at each patchpoint, the mapping from high-level operands to registers, and the information necessary to emit each patchpoint’s machine code. We could have written a layer around B3 that generates LLVM-like stackmaps using B3 `Patchpoint`s, but we instead opted for a rewrite of those parts of the FTL that use patchpoints. Using B3’s patchpoints is a lot easier. For example, here’s all of the code that is needed to use a patchpoint to emit a double-to-int instruction. B3 doesn’t support it natively because the specific behavior of such an instruction differs greatly between different targets.

```
LValue Output::doubleToInt(LValue value)
{
    PatchpointValue* result = patchpoint(Int32);
    result->append(value, ValueRep::SomeRegister);
    result->setGenerator(
        [] (CCallHelpers& jit, const StackmapGenerat
            jit.truncateDoubleToInt32(params[1].fpr(
        });
    result->effects = Effects::none();
    return result;
}
```

Notice how easy it is to query the register used for the input (params[1].fpr()) and the register used for the output (params[0].gpr()). Most patchpoints are much more sophisticated than this one, so having a clear API makes these patchpoints much easier to manage.

Performance Results

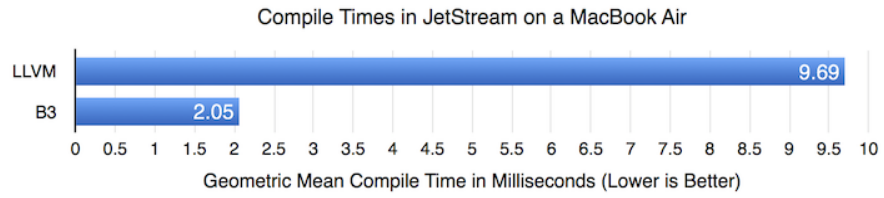
At the time of writing, it's still possible to switch between LLVM and B3 using a compile-time flag. This enables head-to-head comparisons of LLVM and B3. This section summarizes the results of performance tests done on two machines using three benchmark suites.

We tested on two different Macs: an old Mac Pro with 12 hyperthreaded cores (two 6-core Xeons) and 32 GB or RAM, and somewhat newer MacBook Air with a 1.3 GHz Core i5 and 8 GB of RAM. The Mac Pro has 24 logical CPUs while the MacBook Air only has 4. It's important to test different kinds of machines because the number of available CPUs, the relative speed of memory, and the kinds of things that the particular CPU model is good at all affect the bottom line. We have seen WebKit changes that improve performance on one kind of machine while regressing performance on another.

We used three different benchmark suites: JetStream, Octane, and Kraken. JetStream and Kraken have no overlap, and test different kinds of code. Kraken has a lot of numerical code that runs for a relatively short amount of time — something that is missing from JetStream. Octane tests similar things to JetStream, but with a different focus. We believe that WebKit should perform well on all benchmarks, so we always use a wide range of benchmarks as part of our experimental methodology.

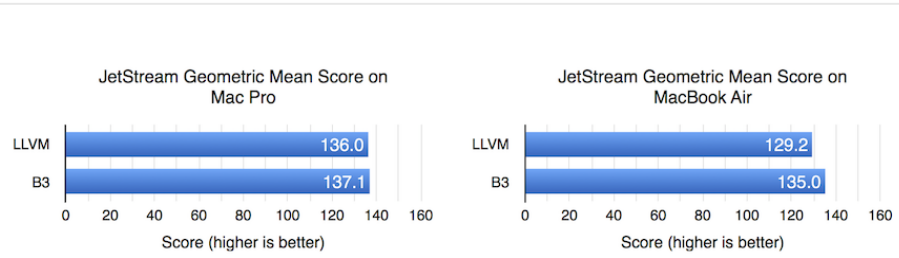
All tests compare LLVM against B3 in the same revision of WebKit, r195946. The tests are run in a browser to make it as real as possible.

Compile Times in JetStream



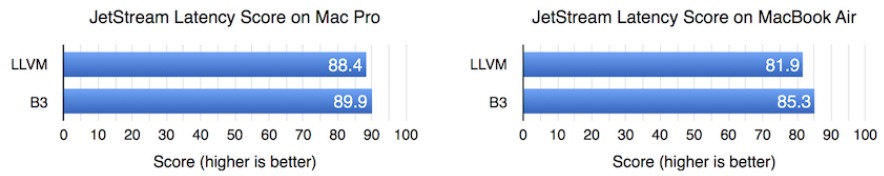
Our hypothesis with B3 is that we could build a compiler that reduces compile times, which would enable the FTL JIT to benefit a wider set of applications. The chart above shows the geometric mean compile time for hot functions in JetStream. LLVM takes 4.7 times longer to compile than B3. While this is off from our original hypothetical goal of 10x reduction in compile times, it's a great improvement. The sections that follow show how this compile time reduction coupled with overall good code generation quality of B3 combine to produce speed-ups on JetStream and other benchmarks.

JetStream Performance



On the Mac Pro, there is hardly any difference in performance between LLVM and B3. But on the MacBook Air, B3 gives a 4.5% overall speed-up. We can break this down further by JetStream's Latency and Throughput components. This section also explores register allocation performance and considers JetStream's performance with various LLVM configurations.

JetStream Latency

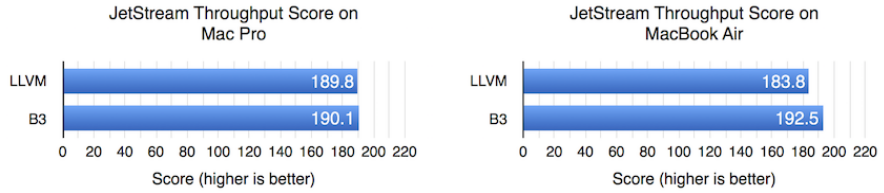


The latency component of JetStream has many shorter-running tests. A JIT compiler that takes too long impacts the latency score in multiple ways:

- It delays the availability of optimized code, so the program spends extra time running slow code.
- It puts pressure on the memory bus and slows down memory accesses on the thread running the benchmark.
- Compiler threads are not perfectly concurrent. They can sometimes be scheduled on the same CPU as the main thread. They can sometimes take locks that cause the main thread to block.

B3 appears to improve the JetStream latency score on both the Mac Pro and the MacBook Air, though the effect on the MacBook Air is considerably larger.

JetStream Throughput



JetStream's throughput tests run long enough that the time it takes to compile them should not matter much. But to the extent that the tests provide a warm-up cycle, the duration is fixed regardless of hardware. So, on slower hardware, we expect that not all of the hot code will be compiled by the time the warm-up cycle is done.

On the Mac Pro, the throughput score appears identical regardless of which backend we pick. But on the MacBook Air, B3 shows a 4.7% win.

The Mac Pro performance results seem to imply that LLVM and B3 are generating code of roughly equivalent quality, but we cannot be sure. It is possible, though unlikely, that the B3 is creating an X% slow-down in steady-state performance against an X% speed-up due to reduction in compiler latency, and they perfectly cancel each other out. We think that it's more likely that LLVM and B3 are generating code of roughly equivalent quality.

We suspect that on the MacBook Air, compile times matter a great deal. Because a lot of JetStream tests have a 1 second warm-up where performance is not recorded, a computer that is fast enough (like the Mac Pro) will be able to completely hide the effects of compile time. So long as all relevant compiling is done in one second, nobody will know if it took the full second or just a fraction of it. But on a slow enough computer, or a computer like a MacBook Air or iPhone that doesn't have 24 logical CPUs, the compiling of hot functions might still be happening after the 1 second of warm-up. At this point, every moment spent compiling deducts from the score because of the compounded effects of running slower code on the main thread and having to compete with compiler threads for memory bandwidth. The 4.7x reduction in compile times combined with the low parallelism of the MacBook Air probably explains why it gets a JetStream throughput speed-up from B3 even though the scores are identical on the Mac Pro.

Register Allocation Performance in JetStream

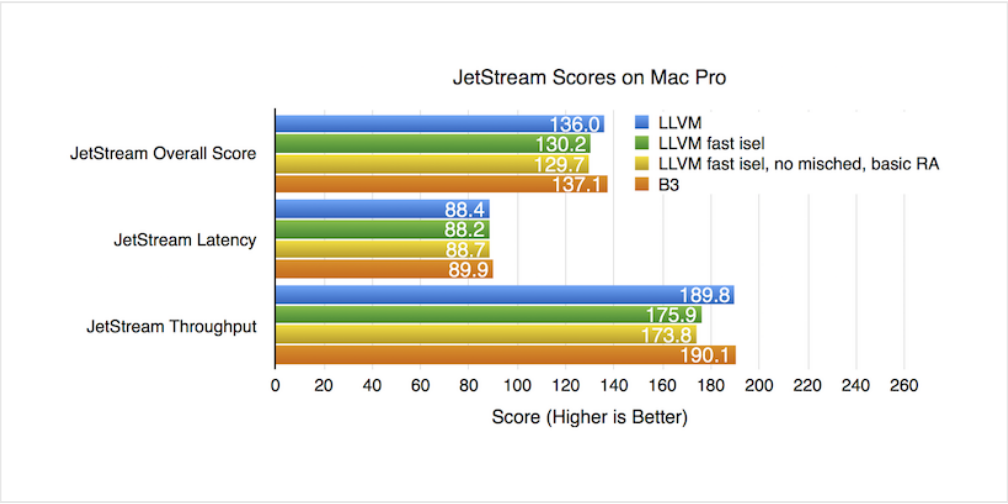
Our throughput results seem to indicate that there isn't a significant difference in the quality of code produced by B3's and LLVM's register allocators. This begs the question: which register allocator is quickest? We measured time spent in register allocation in JetStream. Like we did for compile times, we correlated the register allocation times in all of JetStream's hot functions, and computed the

geometric mean of the ratios of time spent in LLVM's Greedy register allocator versus B3's IRC.

Both LLVM and B3 have expensive register allocators, but LLVM's Greedy register allocator tends to run faster on average – the ratio of LLVM time to B3 time is around 0.6 on the MacBook Air. This isn't enough to overcome LLVM's performance deficits elsewhere, but it does suggest that we should implement LLVM's Greedy allocator in WebKit to see if that gives a speed-up. We are also likely to continue tuning our IRC allocator, since there are still many improvements that can still be made, like removing the remaining uses of hashtables and switching to better data structures for representing interference.

These results suggest that IRC is still a great choice for new compilers, because it's easy to implement, doesn't require a lot of tuning, and delivers respectable performance results. It's worth noting that while IRC is around 1300 lines of code, LLVM's Greedy is close to 5000 lines if you consider the allocator itself (over 2000 lines) and the code for its supporting data structures.

Comparison with LLVM's Fast Instruction Selector

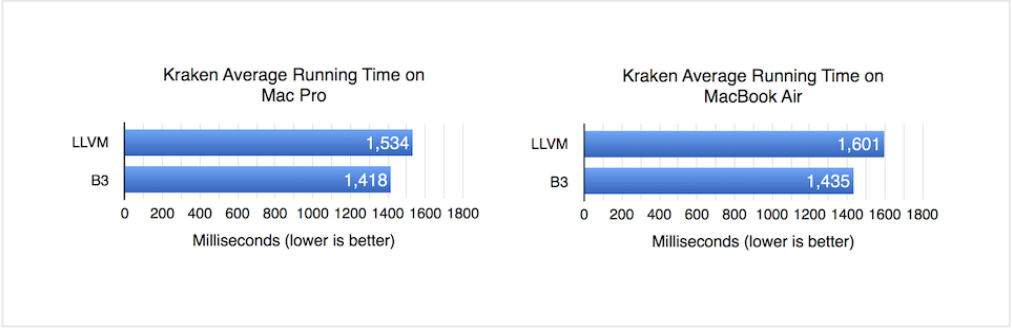


Our final JetStream comparison considers the FTL JIT's ability to run LLVM in the "fast isel" configuration, which we use on ARM64. This configuration reduces compile times by disabling some LLVM optimizations:

- The Fast instruction selector is used instead of the Selection DAG instruction selector. When we switch to fast isel, we also always enable the LowerSwitch pass, since fast isel cannot handle switch statements.
- The misched pass is disabled.
- The Basic register allocator is used instead of the Greedy register allocator.

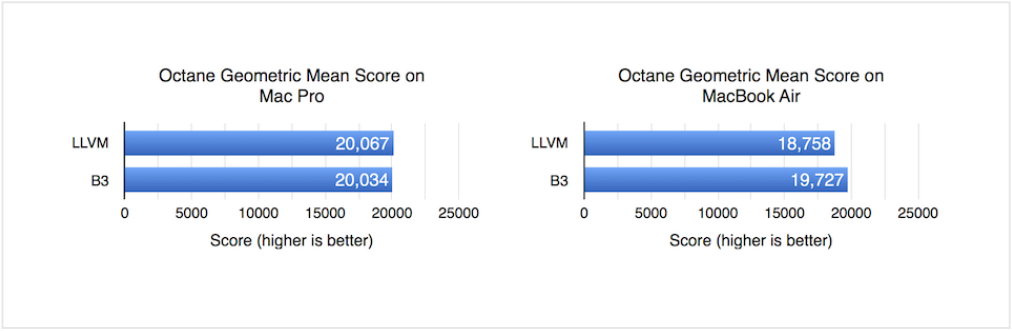
We tested just switching to fast isel, as well as switching to fast isel and doing the two other changes as well. As the chart above shows, using fast isel hurts throughput: the JetStream throughput score goes down by 7.9%, causing the overall score to go down by 4.5%. Also disabling the other optimizations seems to hurt throughput even more: the throughput score goes down by 9.1% and the overall score goes down by 4.9%, though these differences are close to the margin of error. B3 is able to reduce compile times without hurting performance: on this machine, B3's and LLVM's throughput scores are too close to call.

Kraken Performance



The Kraken benchmark has been challenging for the FTL JIT, because its tests run for a relatively short amount of time yet it contains a lot of code that requires the kinds of optimizations that WebKit only does in the FTL JIT. In fact, we had Kraken in mind when we first thought of doing B3. So it's not surprising that B3 provides a speed-up: 8.2% on the Mac Pro and 11.6% on the MacBook Air.

Octane Performance



Octane has a lot of overlap with JetStream: roughly half of JetStream's throughput component is taken from Octane. The Octane results, which show no difference on Mac Pro and a 5.2% B3 speed-up on MacBook Air, mirror the results we saw in JetStream throughput.

Conclusion

B3 generates code that is as good as LLVM on the benchmarks we tried, and makes WebKit faster overall by reducing the amount of time spent compiling. We're happy to have enabled the new B3 compiler in the FTL JIT. Having our own low-level compiler backend gives us an immediate boost on some benchmarks. We hope that in the future it will allow us to better tune our compiler infrastructure for the web.

B3 is not yet complete. We still need to finish porting B3 to ARM64. B3 passes all tests, but we haven't finished optimizing performance on ARM. Once all platforms that used the FTL switch to B3, we plan to remove LLVM support from the FTL JIT.

If you're interested in more information about B3, we plan to maintain up-to-date documentation — every commit that changes how B3 works must change the documentation accordingly. The documentation focuses on B3 IR and we hope to add Air documentation soon. ■

Editing CSS with the Visual Styles Sidebar

Learn more

Previously

CSS Font Features

Learn more