Speculation in JavaScriptCore

This post is all about speculative compilation, or just *speculation* for short, in the context of the JavaScriptCore virtual machine. Speculative compilation is ideal for making dynamic languages, or any language with enough dynamic features, run faster. In this post, we will look at speculation for JavaScript. Historically, this technique or closely related variants has been applied successfully to Smalltalk, Self, Java, .NET, Python, and Ruby, among others. Starting in the 90's, intense benchmark-driven competition between many Java implementations helped to create an understanding of how to build speculative compilers for languages with small amounts of dynamism. Despite being a lot more dynamic than Java, the JavaScript performance war that started in the naughts has generally favored increasingly aggressive applications of the same speculative compilation tricks that worked great for Java. It seems like speculation can be applied to any language implementation that uses runtime checks that are hard to reason about statically.

This is a long post that tries to demystify a complex topic. It's based on a two hour compiler lecture (slides also available in PDF). We assume some familiarity with compiler concepts like intermediate representations (especially Static Single Assignment Form, or SSA for short), static analysis, and code generation. The intended audience is anyone wanting to understand JavaScriptCore better, or anyone thinking about using these techniques to speed up their own language implementation. Most of the concepts described in this post are not specific to JavaScript and this post doesn't assume prior knowledge about JavaScriptCore.

Before going into the details of speculation, we'll provide an overview of speculation and an overview of JavaScriptCore. This will help provide context for the main part of this post, which describes speculation by breaking it down into five parts: *bytecode* (the common IR), *control*, *profiling*, *compilation*, and *OSR* (on stack replacement). We conclude with a small review of related work.

Overview of Speculation

The intuition behind speculation is to leverage traditional compiler technology to make dynamic languages as fast as possible. Construction of high-performance compilers is a well-understood art, so we want to reuse as much of that as we can. But we cannot do this directly for a language like JavaScript because the lack of type information means that the compiler can't do meaningful optimizations for any of the fundamental operations (even things like `+` or `==`). Speculative compilers use profiling to infer types dynamically. The generated code uses dynamic type checks to validate the profiled types. If the program uses a type that is different from what we profiled, we throw out the optimized code and try again. This lets the optimizing compiler work with a statically typed representation of the dynamically typed program.

Types are a major theme of this post even though the techniques we are describing are for implementing dynamically typed languages. When languages include static types, it can be to provide safety properties for the programmer or to help give an optimizing compiler leverage. We are only interested in types for performance and the speculation strategy in JavaScriptCore can be thought of in broad strokes as inferring the kinds of types that a C program would have, but using an internal type system purpose built for our optimizing compiler. More generally, the techniques described in this post can be used to enable any kind of profile-guided optimizations, including ones that aren't related to types. But both this post and JavaScriptCore focus on the kind of profiling and speculation that is most natural to think if as being about type (whether a variable is an integer, what object shapes a pointer points to, whether an operation has effects, etc).

To dive into this a bit deeper, we first consider the impact of types. Then we look at how speculation gives us types.

Impact of Types

We want to give dynamically typed languages the kind of optimizing compiler pipeline that would usually be found in ahead-of-time compilers for high-performance statically typed languages like C. The input to such an optimizer is typically some kind of internal representation (IR) that is precise about the type of each operation, or at least a representation from which the type of each operation can be inferred.

To understand the impact of types and how speculative compilers deal with them, consider this C function:

```
int foo(int a, int b)
{
    return a + b;
}
```

In C, types like `int` are used to describe variables, arguments, return values, etc. Before the optimizing compiler has a chance to take a crack at the above function, a type checker fills in the blanks so that the `+` operation will be represented using an IR instruction that knows that it is adding 32-bit signed integers (i.e. `int`s). This knowledge is essential:

- Type information tells the compiler's code generator how to emit code for this instruction. We know to use integer addition instructions (not double addition or something else) because of the `int` type.
- Type information tells the optimizer how to allocate registers for the inputs and outputs. Integers mean using general purpose registers. Floating point means using floating point registers.
- Type information tells the optimizer what optimizations are possible for this instruction. Knowing exactly what it does allows us to know what other operations can be used in place of it, allows us to do some algebraic reasoning about the math the program is doing, and allows us to fold the instruction to a constant if the inputs are constants. If there are types for which `+` has effects (like in C++), then the fact that this is an integer `+` means that it's pure. Lots of compiler optimizations that work for `+` would not work if it wasn't pure.

Now consider the same program in JavaScript:

```
function foo(a, b)
{
    return a + b;
}
```

We no longer have the luxury of types. The program doesn't tell us the types of `a` or `b`. There is no way that a type checker can label the `+` operation as being anything specific. It can do a bunch of different things based on the runtime types of `a` and `b`:

- It might be a 32-bit integer addition.
- It might be a double addition.
- It might be a string concatenation.
- It might be a loop with method calls. Those methods can be user-defined and may perform arbitrary effects. This'll happen if `a` or `b` are objects.

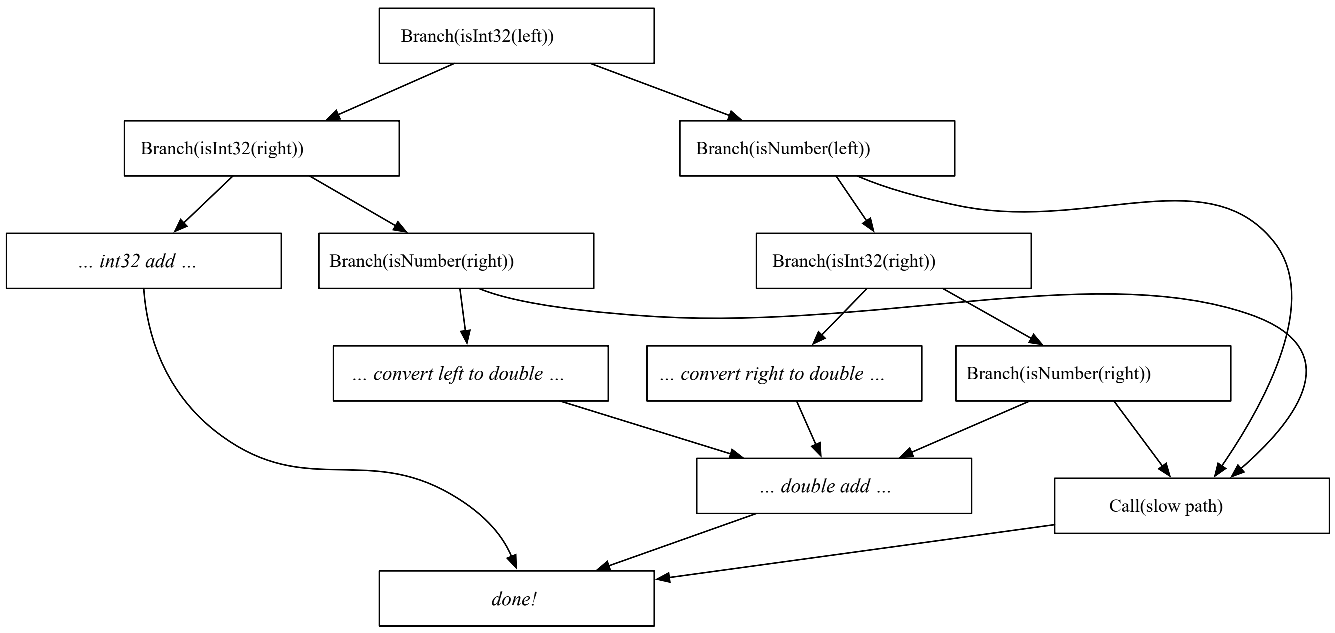


Figure 1. The best that a nonspeculative compiler can do if given a JavaScript plus operation. This figure depicts a control flow graph as a compiler like JavaScriptCore's DFG might see. The `Branch` operation is like an `if` and has outgoing edges for the then/else outcomes of the condition.

Based on this, it's not possible for an optimizer to know what to do. Instruction selection means emitting either a function call for the whole thing or an expensive control flow subgraph to handle all of the various cases (Figure 1). We won't know which register file is best for the inputs or results; we're likely to go with general purpose registers and then do additional move instructions to get the data into floating point registers in case we have to do a double addition. It's not possible to know if one addition produces the same results as another, since they have loops with effectful method calls. Anytime a + happens we have to allow for the the possibility that the whole heap might have been mutated.

In short, it's not practical to use optimizing compilers for JavaScript unless we can somehow provide types for all of the values and operations. For those types to be useful, they need to help us avoid basic operations like + seeming like they require control flow or effects. They also need to help us understand which instructions or register files to use. Speculative compilers get speed-ups by applying this kind of reasoning to all of the dynamic operations in a language — ranging from those represented as fundamental operations (like + or memory accesses like `o.f` and `o[i]`) to those that involve intrinsics or recognizable code patterns (like calling `Function.prototype.apply`).

Speculated Types

This post focuses on those speculations where the collected information can be most naturally understood as type information, like whether or not a variable is an integer and what properties a pointed-to object has (and in what order). Let's appreciate two aspects of this more deeply: when and how the profiling and optimization happen and what it means to speculate on type.

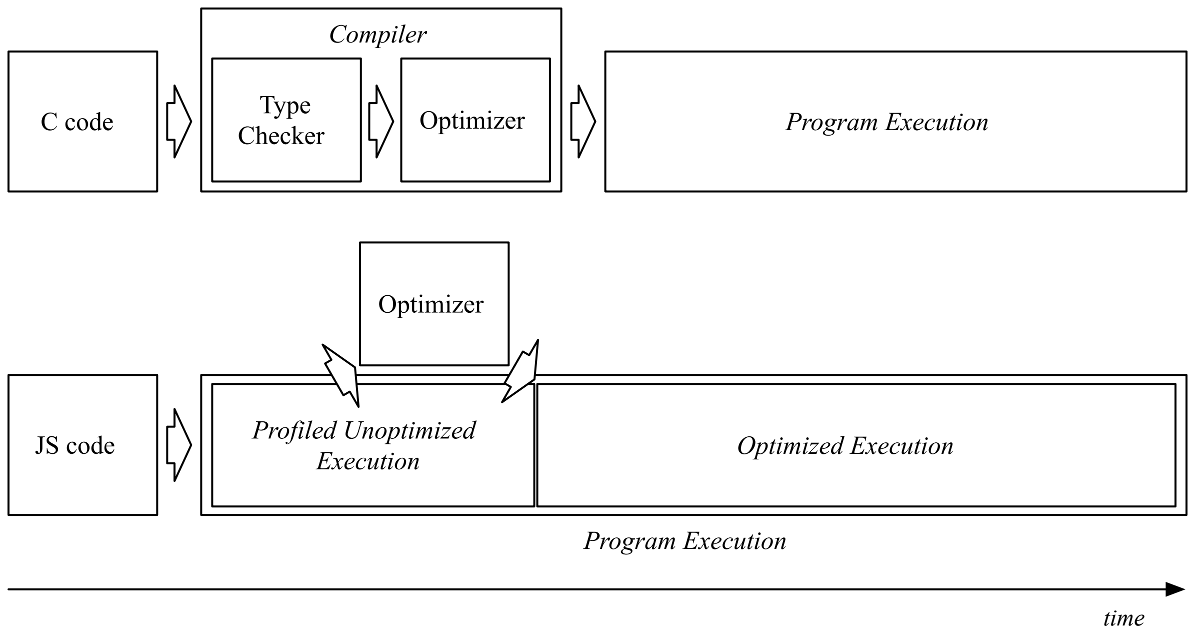


Figure 2. Optimizing compilers for C and JavaScript.

Let's consider what we mean by speculative compilation for JavaScript. JavaScript implementations pretend to be interpreters; they accept JS source as input. But internally, these implementations use a combination of interpreters and compilers. Initially, code starts out running in an execution engine that does no speculative type-based optimizations but collects profiling about types. This is usually an interpreter, but not always. Once a function has a satisfactory amount of profiling, the engine will start an optimizing compiler for that function. The optimizing compiler is based on the same fundamentals as the one found in a C compiler, but instead of accepting types from a type checker and running as a command-line tool, here it accepts types from a profiler and runs in a thread in the same process as the program it's compiling. Once that compiler finishes emitting optimized machine code, we switch execution of that function from the profiling tier to the optimized tier. Running JavaScript code has no way of observing this happening to itself except if it measures execution time. (However, the environment we use for testing JavaScriptCore includes many hooks for introspecting what has been compiled.) Figure 2 illustrates how and when profiling and optimization happens when running JavaScript.

Roughly, speculative compilation means that our example function will be transformed to look something like this:
```
function foo(a, b)
{
    speculate(isInt32(a));
    speculate(isInt32(b));
    return a + b;
}
```

The tricky thing is what exactly it means to *speculate*. One simple option is what we call *diamond speculation*. This means that every time that we perform an operation, we have a fast path specialized for what the profiler told us and a slow path to handle the generic case:
```
if (is int)
    int add
else
    Call(slow path)
```

To see how that plays out, let's consider a slightly different example:
```
var tmp1 = x + 42;
... // things
var tmp2 = x + 100;
```

Here, we use x twice, both times adding it to a known integer. Let's say that the profiler tells us that x is an integer but that we have no way of proving this statically. Let's also say that x's value does not change between the two uses and we have proved that statically.

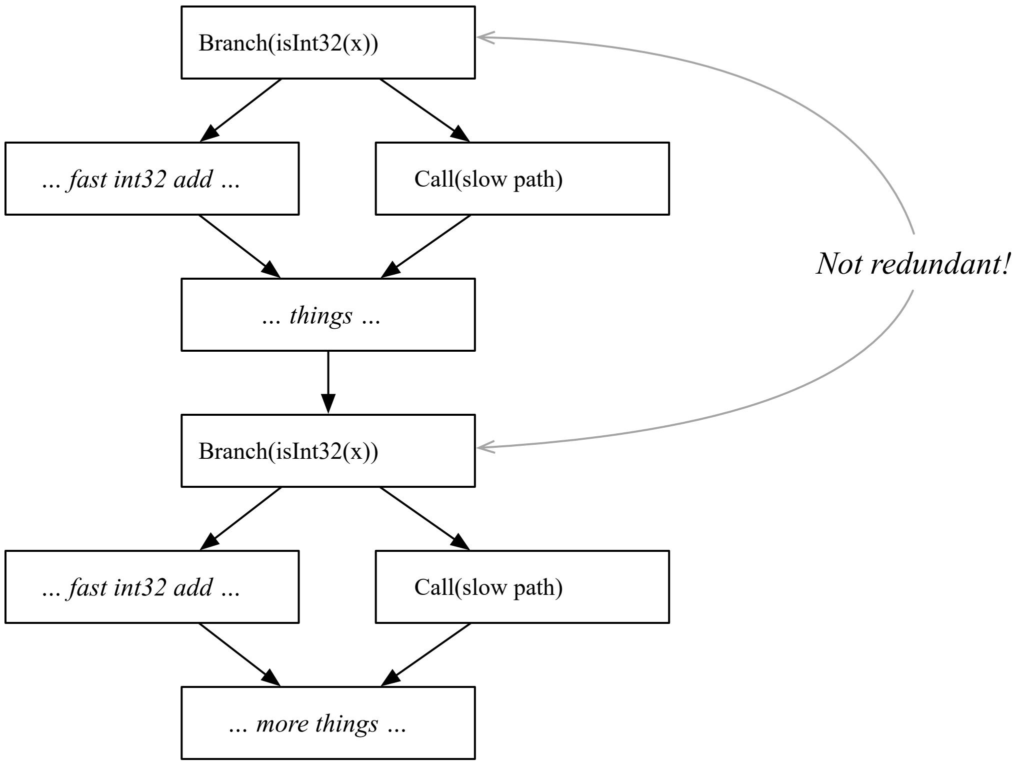


Figure 3. Diamond speculation that x is an integer.

Figure 3 shows what happens if we speculate on the fact that x is an integer using a diamond speculation: we get a fast path that does the integer addition and a slow path that bails out to a helper function. Speculations like this can produce modest speed-ups at modest cost. The cost is modest because if the speculation is wrong, only the operations on x pay the price. The trouble with this approach is that repeated uses of x must recheck whether it is an integer. The rechecking is necessary because of the control flow merge that happens at the *things* block and again at *more things*.

The original solution to this problem was splitting, where the region of the program between *things* and *more things* would get duplicated to avoid the branch. An extreme version of this is tracing, where the entire remainder of a function is duplicated after any branch. The trouble with these techniques is that duplicating code is expensive. We want to minimize the number of times that the same piece of code is compiled so that we can compile a lot of code quickly. The closest thing to splitting that JavaScriptCore does is tail duplication, which optimizes diamond speculations by duplicating the code between them if that code is tiny.

A better alternative to diamond speculations or splitting is *OSR* (on stack replacement). When using OSR, a failing type check *exits* out of the optimized function back to the equivalent point in the unoptimized code (i.e. the profiling tier's version of the function).

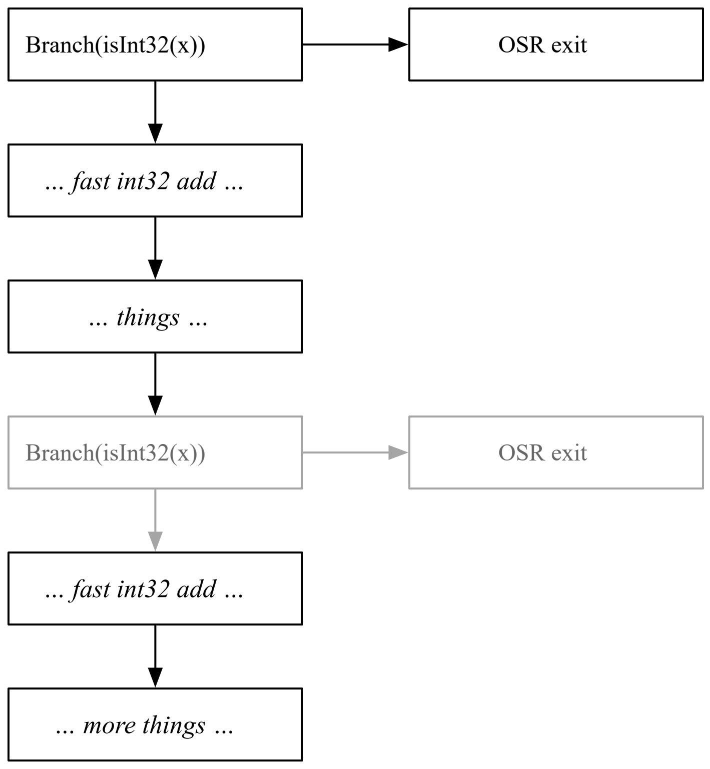


Figure 4. OSR speculation that `x` *is an integer.*

Figure 4 shows what happens when we speculate that `x` is an integer using OSR. Because there is no control flow merge between the case where `x` is an int and the case where it isn't, the second check becomes redundant and can be eliminated. The lack of a merge means that the only way to reach the second check is if the first check passed.

OSR speculations are what gives our traditional optimizing compiler its static types. After any OSR-based type check, the compiler can assume that the property that was checked is now fact. Moreover, because OSR check failure does not affect semantics (we exit to the same point in the same code, just with fewer optimizations), we can hoist those checks as high as we want and infer that a variable always has some type simply by guarding all assignments to it with the corresponding type check.

Note that what we call *OSR exit* in this post and in JavaScriptCore is usually called *deoptimization* elsewhere. We prefer to use the term *OSR exit* in our codebase because it emphasizes that the point is to *exit* an optimized function using an exotic technique (*OSR*). The term *deoptimization* makes it seem like we are undoing optimization, which is only true in the narrow sense that a particular execution jumps from optimized code to unoptimized code. For this post we will follow the JavaScriptCore jargon.

JavaScriptCore uses OSR or diamond speculations depending on our confidence that the speculation will be right. OSR speculation has higher benefit and higher cost: the benefit is higher because repeated checks can be eliminated but the cost is also higher because OSR is more expensive than calling a helper function. However, the cost is only paid if the exit actually happens. The benefits of OSR speculation are so superior that we focus on that as our main speculation strategy, with diamond speculation being the fallback if our profiling indicates lack of confidence in the speculation.

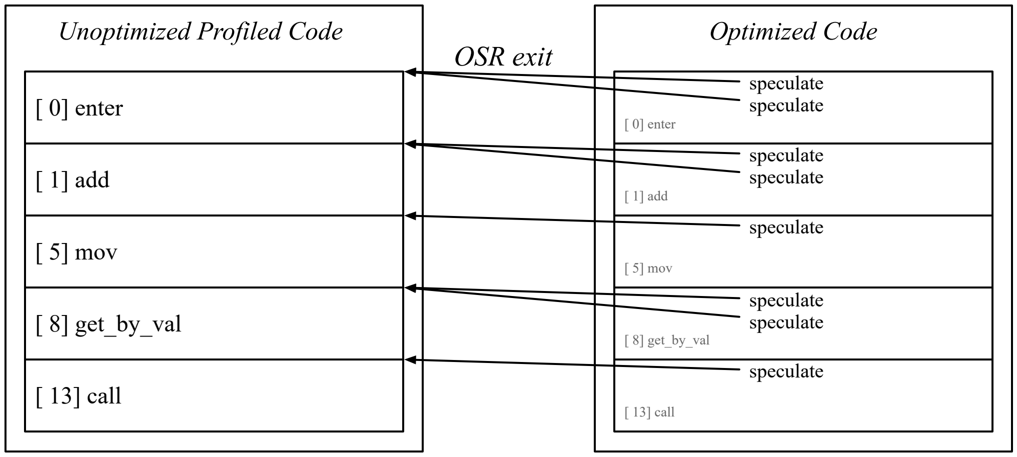


Figure 5. Speculating with OSR and exiting to bytecode.

OSR-based speculation relies on the fact that traditional compilers are already good at reasoning about side exits. Trapping instructions (like for null check optimization in Java virtual machines), exceptions, and multiple return statements are all examples of how compilers already support exiting from a function.

Assuming that we use bytecode as the common language shared between the unoptimizing profiled tier of execution and the optimizing tier, the exit destinations can just be bytecode instruction boundaries. Figure 5 shows how this might work. The machine code generated by the optimizing compiler contains speculation checks against unlikely conditions. The idea is to do *lots* of speculations. For example, the prologue (the `enter` instruction in the figure) may speculate about the types of the arguments — that's one speculation per argument. An `add` instruction may speculate about the types of its inputs and about the result not overflowing. Our type profiling may tell us that some variable tends to always have some type, so a `mov` instruction whose source is not proved to have that type may speculate that the value has that type at runtime. Accessing an array element (what we call `get_by_val`) may speculate that the array is really an array, that the index is an integer, that the index is in bounds, and that the value at the index is not a hole (in JavaScript, loading from a never assigned array element means walking the array's prototype chain to see if the element can be found there — something we avoid doing most of the time by speculating that we don't have to). Calling a function may speculate that the callee is the one we expected or at least that it has the appropriate type (that it's something we can call).

While exiting out of a function is straightforward without breaking fundamental assumptions in optimizing compilers, entering turns out to be super hard. Entering into a function somewhere other than at its primary entrypoint pessimises optimizations at any merge points between entrypoints. If we allowed entering at every bytecode instruction boundary, this would negate the benefits of OSR exit by forcing every instruction boundary to make worst-case assumptions about type. Even allowing OSR entry just at loop headers would break lots of loop optimizations. This means that it's generally not possible to reenter optimized execution after exiting. We only support entry in cases where the reward is high, like when our profiler tells us that a loop has not yet terminated at the time of compilation. Put simply, the fact that traditional compilers are designed for single-entry multiple-exit procedures means that OSR entry is hard but OSR exit is easy.

JavaScriptCore and most speculative compilers support OSR entry at hot loops, but since it's not an essential feature for most applications, we'll leave understanding how we do it as an exercise for the reader.

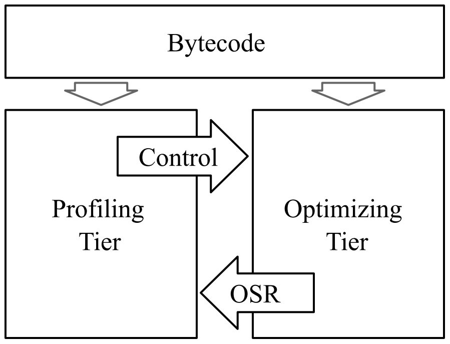


Figure 6. Speculation broken into the five topics of this post.

The main part of this post describes speculation in terms of its five components (Figure 6): the *bytecode*, or common IR, of the virtual machine that allows for a shared understanding about the meaning of profiling and exit sites between the unoptimized profiling tier and the optimizing tier; the unoptimized *profiling tier* that is used to execute functions at start-up, collect profiling about them, and to serve as an exit destination; the *control* system for deciding when to invoke the optimizing compiler; the *optimizing tier* that combines a traditional optimizing compiler with enhancements to support speculation based on profiling; and the *OSR exit* technology that allows the optimizing compiler to use the profiling tier as an exit destination when speculation checks fail.

Overview of JavaScriptCore

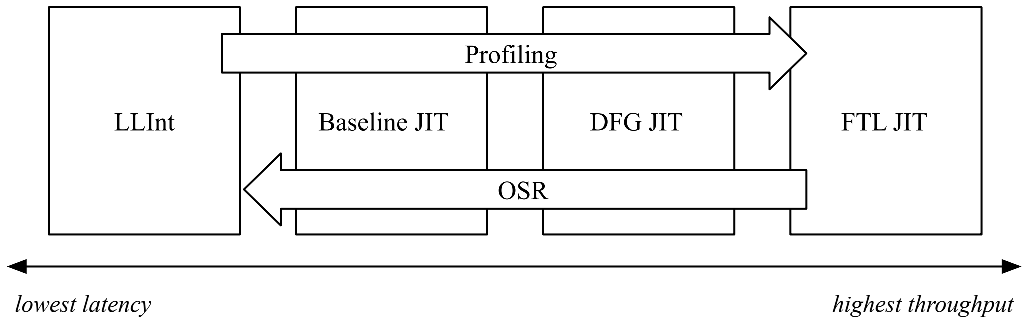


Figure 7. The tiers of JavaScriptCore.

JavaScriptCore embraces the idea of tiering and has four tiers for JavaScript (and three tiers for WebAssembly, but that's outside the scope of this post). Tiering has two benefits: the primary benefit, described in the previous section, of enabling speculation; and a secondary benefit of allowing us to fine-tune the throughput-latency tradeoff on a per-function basis. Some functions run for so short — like straight-line run-once initialization code — that running any compiler on those functions would be more expensive than interpreting them. Some functions get invoked so frequently, or have such long loops, that their total execution time far exceeds the time to compile them with an aggressive optimizing compiler. But there are also lots of functions in the grey area in between: they run for not enough time to make an aggressive compiler profitable, but long enough that some intermediate compiler designs can provide speed-ups. JavaScriptCore has four tiers as shown in Figure 7:

- The *LLInt*, or low-level interpreter, which is an interpreter that obeys JIT compiler ABI. It runs on the same stack as the JITs and uses a known set of registers and stack locations for its internal state.
- The *Baseline JIT*, also known as a bytecode template JIT, which emits a *template* of machine code for each bytecode instruction without trying to reason about relationships between multiple instructions in the function. It compiles whole functions, which makes it a

method JIT. Baseline does no OSR speculations but does have a handful of diamond speculations based on profiling from the LLInt.

- The *DFG JIT*, or data flow graph JIT, which does OSR speculation based on profiling from the LLInt, Baseline, and in some rare cases even using profiling data collected by the DFG JIT and FTL JIT. It may OSR exit to either baseline or LLInt. The DFG has a compiler IR called DFG IR, which allows for sophisticated reasoning about speculation. The DFG avoids doing expensive optimizations and makes many compromises to enable fast code generation.
- The *FTL JIT*, or faster than light JIT, which does comprehensive compiler optimizations. It's designed for peak throughput. The FTL never compromises on throughput to improve compile times. This JIT reuses most of the DFG JIT's optimizations and adds lots more. The FTL JIT uses multiple IRs (DFG IR, DFG SSA IR, B3 IR, and Assembly IR).

An ideal example of this in action is this program:

```
"use strict";

let result = 0;
for (let i = 0; i < 10000000; ++i) {
    let o = {f: i};
    result += o.f;
}

print(result);
```

Thanks to the object allocation inside the loop, it will run for a long time until the FTL JIT can compile it. The FTL JIT will kill that allocation, so then the loop finishes quickly. The long running time before optimization virtually guarantees that the FTL JIT will take a stab at this program's global function. Additionally, because the function is clean and simple, all of our speculations are right and there are no OSR exits.

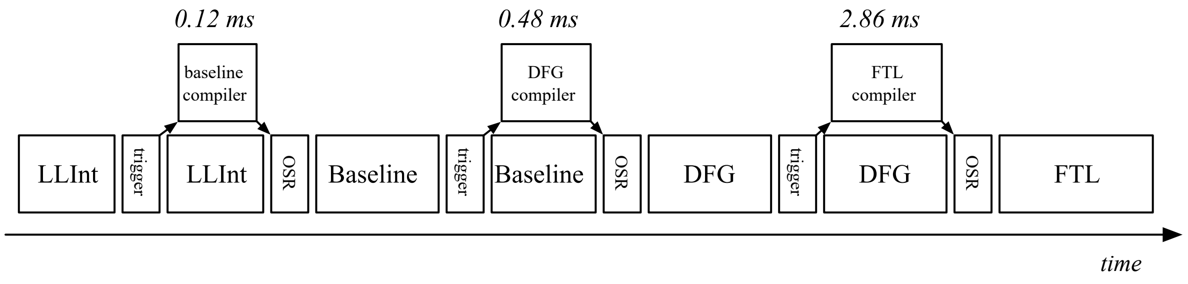


Figure 8. Example timeline of a simple long loop executing in JavaScriptCore. Execution times recorded on my computer one day.

Figure 8 shows the timeline of this benchmark executing in JavaScriptCore. The program starts executing in the LLInt. After about a thousand loop iterations, the loop trigger causes us to start a baseline compiler thread for this code. Once that finishes, we do an OSR entry into the baseline JITed code at the `for` loop's header. The baseline JIT also counts loop iterations, and after about a thousand more, we spawn the DFG compiler. The process repeats until we are in the FTL. When I measured this, I found that the DFG compiler needs about 4× the time of the baseline compiler, and the FTL needs about 6× the time of the DFG. While this example is contrived and ideal, the basic idea holds for any JavaScript program that runs long enough since all tiers of JavaScriptCore support the full JavaScript language.

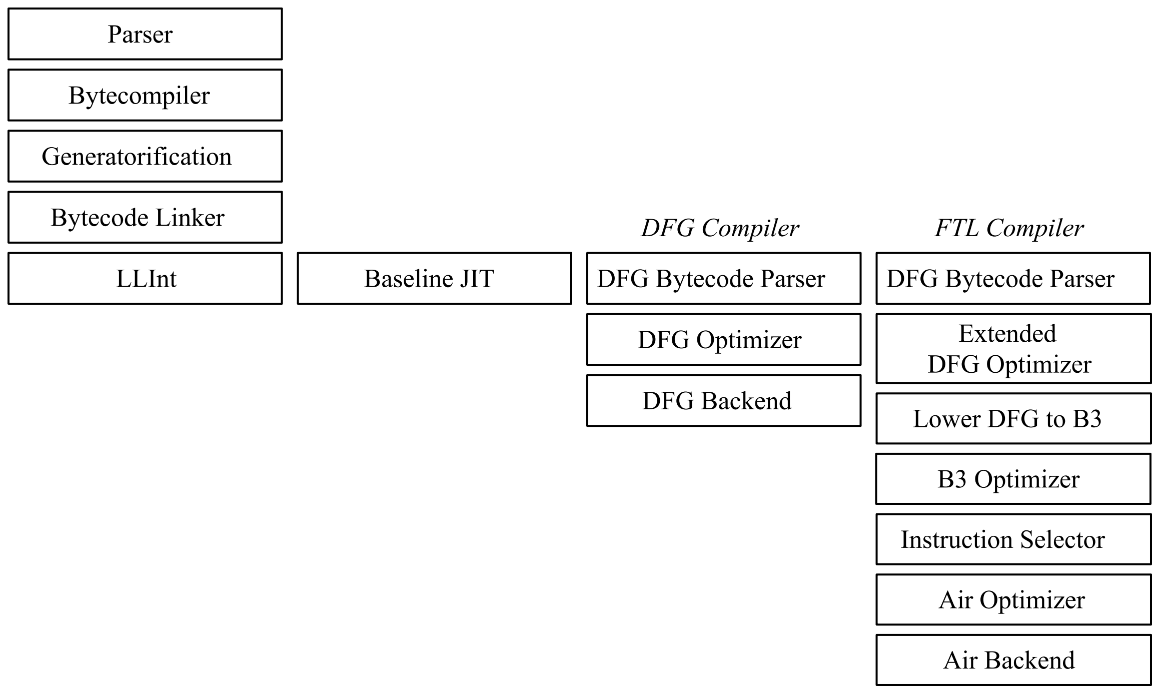


Figure 9. JavaScriptCore tier architecture.

JavaScriptCore is architected so that having many tiers is practical. Figure 9 illustrates this architecture. All tiers share the same bytecode as input. That bytecode is generated by a compiler pipeline that desugars many language features, such as generators and classes, among others. In many cases, it's possible to add new language features just by modifying the bytecode generation frontend. Once linked, the bytecode can be understood by any of the tiers. The bytecode can be interpreted by the LLInt directly or compiled with the baseline JIT, which mostly just converts each bytecode instruction into a preset template of machine code. The LLInt and Baseline JIT share a lot of code, mostly in the slow paths of bytecode instruction execution. The DFG JIT converts bytecode to its own IR, the DFG IR, and optimizes it before emitting code. In many cases, operations that the DFG chooses not to speculate on are emitted using the same code generation helpers as the Baseline JIT. Even operations that the DFG does speculate on often share slow paths with the Baseline JIT. The FTL JIT reuses the DFG's compiler pipeline and adds new optimizations to it, including multiple new IRs that have their own optimization pipelines. Despite being more sophisticated than the DFG or Baseline, the FTL JIT shares slow path implementations with those JITs and in some cases even shares code generation for operations that we choose not to speculate on. Even though the various tiers try to share code whenever possible, they aren't required to. Take the `get_by_val` (access an array element) instruction in bytecode. This has duplicate definitions in the bytecode liveness analysis (which knows the liveness rules for `get_by_val`), the LLInt (which has a very large implementation that switches on a bunch of the common array types and has good code for all of them), the Baseline (which uses a polymorphic inline cache), and the DFG bytecode parser. The DFG bytecode parser converts `get_by_val` to the DFG IR GetByVal operation, which has separate definitions in the DFG and FTL backends as well as in a bunch of phases that know how to optimize and model GetByVal. The only thing that keeps those implementations in agreement is good convention and extensive testing.

To give a feeling for the relative throughput of the various tiers, I'll share some informal performance data that I've gathered over the years out of curiosity.

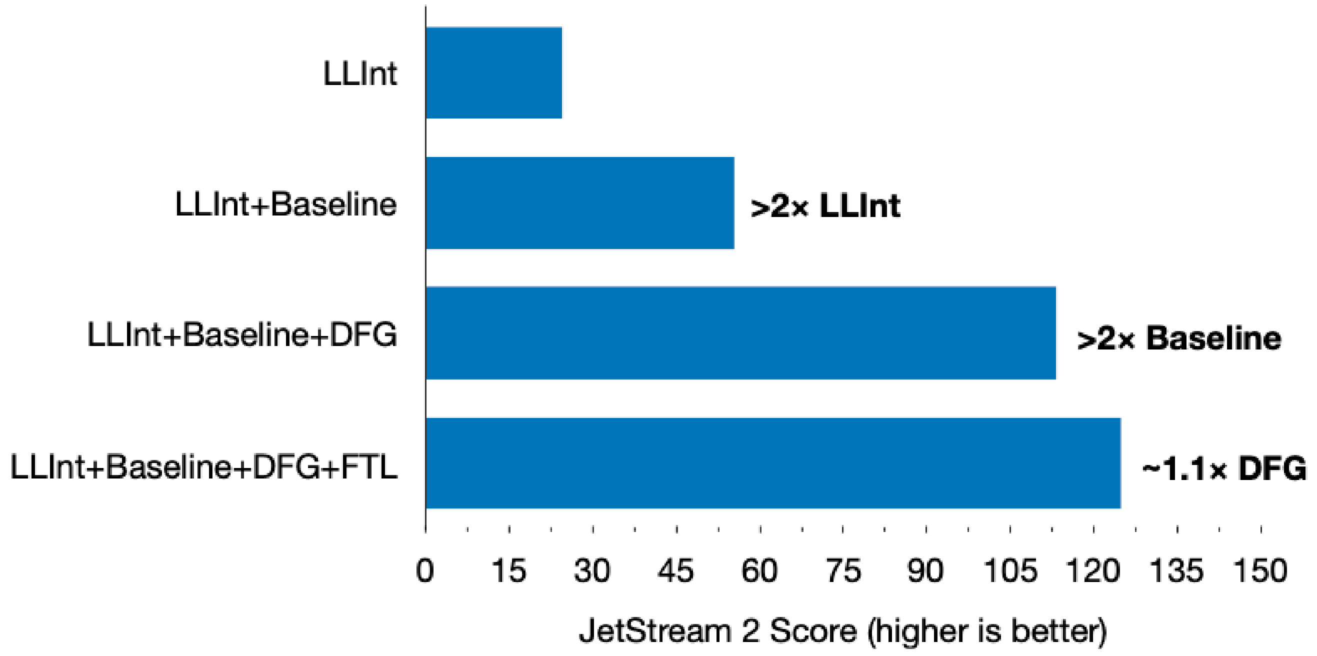


Figure 10. Relative performance of the four tiers on JetStream 2 on my computer at the time of that benchmark's introduction.

We're going to use the JetStream 2 benchmark suite since that's the main suite that JavaScriptCore is tuned for. Let's first consider an experiment where we run JetStream 2 with the tiers progressively enabled starting with the LLInt. Figure 10 shows the results: the Baseline and DFG are more than 2× better than the tier below them and the FTL is 1.1× better than the DFG.

The FTL's benefits may be modest but they are unique. If we did not have the FTL, we would have no way of achieving the same peak throughput. A great example is the *gaussian-blur* subtest. This is the kind of compute test that the FTL is built for. I managed to measure the benchmark's performance when we first introduced it and did not yet have a chance to tune for it. So, this gives a glimpse of the speed-ups that we expect to see from our tiers for code that hasn't yet been through the benchmark tuning grind. Figure 11 shows the results. All of the JITs achieve spectacular speed-ups: Baseline is 3× faster than LLInt, DFG is 6× faster than Baseline, and FTL is 1.6× faster than DFG.

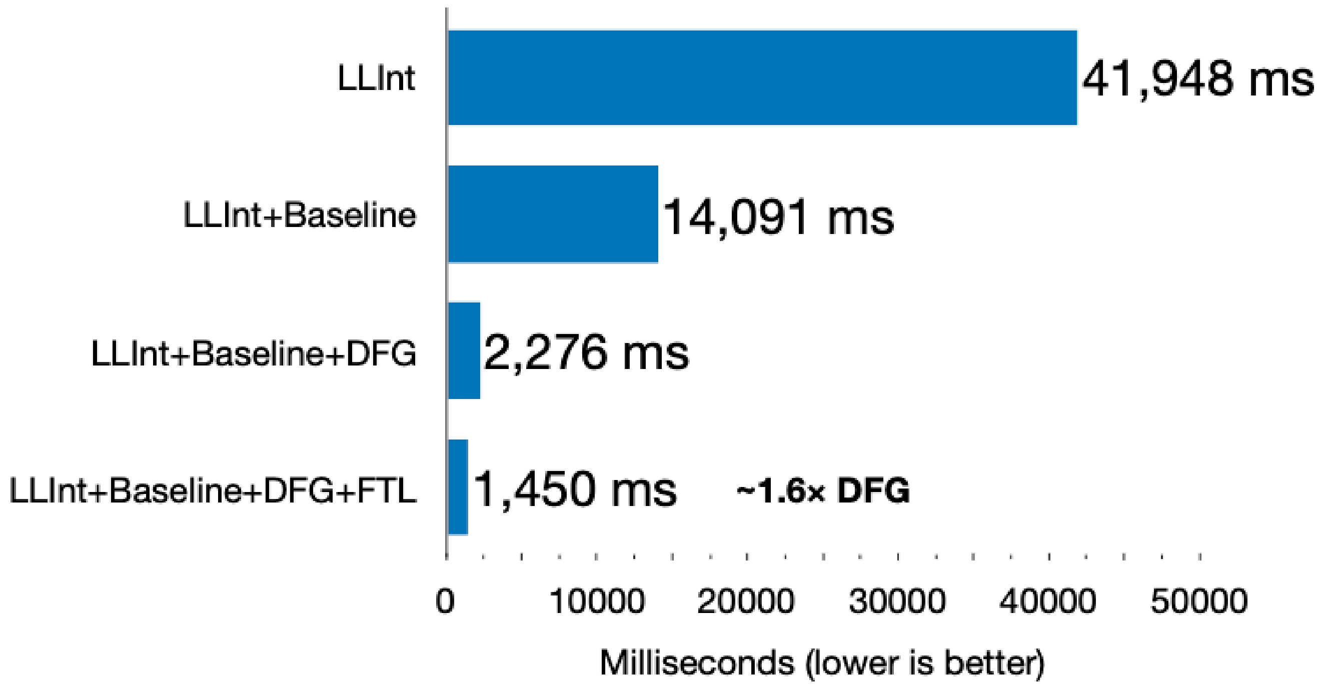


Figure 11. Relative performance of the four tiers on the guassian-blur subtest of JetStream 2.

The DFG and FTL complement one another. The DFG is designed to be a fast-running compiler and it achieves this by excluding the most aggressive optimizations, like global register allocation, escape analysis, loop optimizations, or anything that needs SSA. This means that the DFG will always get crushed on peak throughput by compilers that have those features. It's the FTL's job to provide those optimizations if a function runs long enough to warrant it. This ensures that there is no scenario where a hypothetical competing implementation could outperform us unless they had the same number of tiers. If you wanted to make a compiler that compiles faster than the FTL then you'd lose on peak throughput, but if you wanted to make a compiler that generates better code than the DFG then you'd get crushed on start-up times. You need both to stay in the game.

Another way of looking at the performance of these tiers is to ask: how much time does a bytecode instruction take to execute in each of the tiers on average? This tells us just about the throughput that a tier achieves without considering start-up at all. This can be hard to estimate, but I made an attempt at it by repeatedly running each JetStream 2 benchmark and having it limit the maximum tier of each function at random. Then I employed a stochastic counting mechanism to get an estimate of the number of bytecode instructions executed at each tier in each run. Combined with the execution times of those runs, this gave a simple linear regression problem of the form:

```
ExecutionTime = (Latency of LLInt) * (Bytecodes in LLInt)
              + (Latency of Baseline) * (Bytecodes in Baseline)
              + (Latency of DFG) * (Bytecodes in DFG)
              + (Latency of FTL) * (Bytecodes in FTL)
```

Where the `Latency of LLInt` means the average amount of time it takes to execute a bytecode instruction in LLInt.

After excluding benchmarks that spent most of their time outside JavaScript execution (like regexp and wasm benchmarks) and fiddling with how to weight benchmarks (I settled on solving each benchmarks separately and computing geomean of the coefficients since this matches JetStream 2 weighting), the solution I arrived at was:

```
Execution Time = (3.97 ns) * (Bytecodes in LLInt)
               + (1.71 ns) * (Bytecodes in Baseline)
               + (.349 ns) * (Bytecodes in DFG)
               + (.225 ns) * (Bytecodes in FTL)
```

In other words, Baseline executes code about 2× faster than LLInt, DFG executes code about 5× faster than Baseline, and the FTL executes code about 1.5× faster than DFG. Note how this data is in the same ballpark as what we saw for *gaussian-blur*. That makes sense since that was a peak throughput benchmark.

Although this isn't a garbage collection blog post, it's worth understanding a bit about how the garbage collector works. JavaScriptCore picks a garbage collection strategy that makes the rest of the virtual machine, including all of the support for speculation, easier to implement. The garbage collector has the following features that make speculation easier:

- The collector scans the stack conservatively. This means that compilers don't have to worry about how to report pointers to the collector.
- The collector doesn't move objects. This means that if a data structure (like the compiler IR) has many possible ways of referencing some object, we only have to report one of them to the collector.
- The collector runs to fixpoint. This makes it possible to invent precise rules for whether objects created by speculation should be kept alive.
- The collector's object model is expressed in C++. JavaScript objects look like C++ objects, and JS object pointers look like C++ pointers.

These features make the compiler and runtime easier to write, which is great, since speculation requires us to write a lot of compiler and runtime code. JavaScript is a slow enough language even with the optimizations we describe in this post that garbage collector performance is rarely the longest pole in the tent. Therefore, our garbage collector makes many tradeoffs to make it easier to work on the performance-critical parts of our engine (like speculation). It would be unwise, for example, to make it harder to implement some compiler optimization as a way of getting a small garbage collector optimization, since the compiler has a bigger impact on performance for typical JavaScript programs.

To summarize: JavaScriptCore has four tiers, two of which do speculative optimizations, and all of which participate in the collection of profiling. The first two tiers are an interpreter and bytecode template JIT while the last two are optimizing compilers tuned for different throughput-latency trade-offs.

Speculative Compilation

Now that we've established some basic background about speculation and JavaScriptCore, this section goes into the details. First we will discuss JavaScriptCore's bytecode. Then we show the control system for launching the optimizing compiler. Next will be a detailed section about how JavaScriptCore's profiling tiers work, which focuses mostly on how they collect profiling. Finally we discuss JavaScriptCore's optimizing compilers and their approach to OSR.

Bytecode

Speculation requires having a profiling tier and an optimizing tier. When the profiling tier reports profiling, it needs to be able to say what part of the code that profiling is for. When the optimizing compiler wishes to compile an OSR exit, it needs to be able to identify the exit site in a way that both tiers understand. To solve both issues, we need a common IR that is:

- Used by all tiers as input.
- Persistent for as long as the function that it represents is still live.
- Immutable (at least for those parts that all tiers are interested in).

In this post, we will use bytecode as the common IR. This isn't required; abstract syntax trees or even SSA could work as a common IR. We offer some insights into how we designed our bytecode for JavaScriptCore. JavaScriptCore's bytecode is register-based, compact, untyped, high-level, directly interpretable, and transformable.

Our bytecode is *register-based* in the sense that operations tend to be written as:
```
add result, left, right
```

Which is taken to mean:
```
result = left + right
```

Where `result`, `left`, and `right` are *virtual registers*. Virtual registers may refer to locals, arguments, or constants in the constant pool. Functions declare how many locals they need. Locals are used both for named variables (like `var`, `let`, or `const` variables) and temporaries arising from expression tree evaluation.

Our bytecode is *compact*: each opcode and operand is usually encoded as one byte. We have wide prefixes to allow 16-bit or 32-bit operands. This is important since JavaScript programs can be large and the bytecode must persist for as long as the function it represents is still live.

Our bytecode is *untyped*. Virtual registers never have static type. Opcodes generally don't have static type except for the few opcodes that have a meaningful type guarantee on their output (for example, the `|` operator always returns int32, so our `bitor` opcode returns int32). This is important since the bytecode is meant to be a common source of truth for all tiers. The profiling tier runs before we have done type inference, so the bytecode can't have any more types than the JavaScript language.

Our bytecode is almost as *high-level* as JavaScript. While we use desugaring for many JavaScript features, we only do that when implementation by desugaring isn't thought to cost performance. So, even the "fundamental" features of our bytecode are high level. For example, the `add` opcode has all of the power of the JavaScript `+` operator, including that it might mean a loop with effects.

Our bytecode is *directly interpretable*. The same bytecode stream that the interpreter executes is the bytecode stream that we will save in the cache (to skip parsing later) and feed to the compiler tiers.

Finally, our bytecode is *transformable*. Normally, intermediate representations use a control flow graph and make it easy to insert and remove instructions. That's not how bytecode works: it's an array of instructions encoded using a nontrivial variable-width encoding. But we do have a bytecode editing API and we use it for generatorification (our generator desugaring bytecode-to-bytecode pass). We can imagine this facility also being useful for other desugarings or for experimenting with bytecode instrumentation.

Compared to non-bytecode IRs, the main advantages of bytecode are that it's easy to:

- *Identify targets for OSR exit.* OSR exit in JavaScriptCore requires entering into an unoptimized bytecode execution engine (like an interpreter) at some arbitrary bytecode instruction. Using bytecode instruction index as a way of naming an exit target is intuitive since it's just an integer.
- *Compute live state at exit.* Register-based bytecode tends to have dense register numberings so it's straightforward to analyze liveness using bitvectors. That tends to be fast and doesn't require a lot of memory. It's practical to cache the results of bytecode liveness analysis, for example.

JavaScriptCore's bytecode format is independently implemented by the execution tiers. For example, the baseline JIT doesn't try to use the LLInt to create its machine code templates; it just emits those templates itself and doesn't try to match the LLInt exactly (the behavior is identical but the implementation isn't). The tiers do share a lot of code – particularly for inline caches and slow paths – but they aren't required to. It's common for bytecode instructions to have algorithmically different implementations in the four tiers. For example the LLInt might implement some instruction with a large switch that handles all possible types, the Baseline might implement the same instruction with an inline cache that repatches based on type, and the DFG and FTL might try to do some combination of inline speculations, inline caches, and emitting a switch on all types. This exact scenario happens for `add` and other arithmetic ops as well as `get_by_val`/`put_by_val`. Allowing this independence allows each tier to take advantage of its unique properties to make things run faster. Of course, this approach also means that adding new bytecodes or changing bytecode semantics requires changing all of the tiers. For that reason, we try to implement new language features by desugaring them to existing bytecode constructs.

It's possible to use any sensible IR as the common IR for a speculative compiler, including abstract syntax trees or SSA, but JavaScriptCore uses bytecode so that's what we'll talk about in the rest of this post.

Control

Speculative compilation needs a control system to decide when to run the optimizing compiler. The control system has to balance competing concerns: compiling functions as soon as it's profitable, avoiding compiling functions that aren't going to run long enough to benefit from it, avoiding compiling functions that have inadequate type profiling, and recompiling functions if a prior compilation did speculations that turned out to be wrong. This section describes JavaScriptCore's control system. Most of the heuristics we describe were necessary, in our experience, to make speculative compilation profitable. Otherwise the optimizing compiler would kick in too often, not often enough, or not at the right rate for the right functions. This section describes the full details of JavaScriptCore's tier-up heuristics because we suspect that to reproduce our performance, one would need all of these heuristics.

JavaScriptCore *counts executions* of functions and loops to decide when to compile. Once a function is compiled, we *count exits* to decide when to throw away compiled functions. Finally, we *count recompilations* to decide how much to back off from recompiling a function in the future.

Execution Counting

JavaScriptCore maintains an execution counter for each function. This counter gets incremented as follows:

- Each call to the function *adds 15 points* to the execution counter.
- Each loop execution *adds 1 point* to the execution counter.

We trigger tier-up once the counter reaches some threshold. Thresholds are determined dynamically. To understand our thresholds, first consider their *static* versions and then let's look at how we modulate these thresholds based on other information.

- LLInt→Baseline tier-up requires 500 points.
- Baseline→DFG tier-up requires 1000 points.
- DFG→FTL tier-up requires 100000 points.

Over the years we've found ways to dynamically adjust these thresholds based on other sources of information, like:

- Whether the function got JITed the last time we encountered it (according to our cache). Let's call this `wasJITed`.
- How big the function is. Let's call this `S`. We use the number of bytecode opcodes plus operands as the size.
- How many times it has been recompiled. Let's call this `R`.
- How much executable memory is available. Let's use `M` to say how much executable memory we have total, and `U` is the amount we estimate that we would use (total) if we compiled this function.
- Whether profiling is "full" enough.

We select the LLInt→Baseline threshold based on `wasJITed`. If we don't know (the function wasn't in the cache) then we use the basic threshold, 500. Otherwise, if the function `wasJITed` then we use 250 (to accelerate tier-up) otherwise we use 2000. This optimization is especially useful for improving page load times.

Baseline→DFG and DFG→FTL use the same scaling factor based on `S`, `R`, `M`, and `U`. The scaling factor is defined as follows:

```
(0.825914 + 0.061504 * sqrt(S + 1.02406)) * pow(2, R) * M / (M - U)
```

We multiply this by 1000 for Baseline→DFG and by 100000 for DFG→FTL. Let's break down what this scaling factor does:

First we scale by the square root of the size. The expression `0.825914 + 0.061504 * sqrt(S + 1.02406)` gives a scaling factor that is between 1 and 2 for functions smaller than about 350 bytecodes, which we consider to be "easy" functions to compile. The scaling factor uses square root so it grows somewhat gently. We've also tried having the staling factor be linear, but that's much worse. It is worth it to delay compilations of large functions a bit, but it's not worth it to delay it too much. Note that the ideal delay doesn't just have to do with the cost of compilation. It's also about running long enough to get good profiling. Maybe there is some deep reason why square root works well here, but all we really care about is that scaling by this amount makes programs run faster.

Then we introduce exponential backoff based on the number of times that the function has been recompiled. The `pow(2, R)` expression means that each recompilation doubles the thresholds.

After that we introduce a hyperbolic scaling factor, `M / (M - U)`, to help avoid cases where we run out of executable memory altogether. This is important since some configurations of JavaScriptCore run with a small available pool of executable memory. This expression means that if we use half of executable memory then the thresholds are doubled. If we use 3/4 of executable memory then the thresholds are quadrupled. This makes filling up executable memory a bit like going at the speed of light: the math makes it so that as you get closer to filling it up the thresholds get closer to infinity. However, it's worth noting that this is imperfect for truly large programs, since those might have other reasons to allocate executable memory not covered by this heuristic. The heuristic is also imperfect in cases of multiple things being compiled in parallel. Using this factor increases the maximum program size we can handle with small pools of executable memory, but it's not a silver bullet.

Finally, if the execution count does reach this dynamically computed threshold, we check that some kinds of profiling (specifically, value and array profiling, discussed in detail in the upcoming profiling section) are full enough. We say that profiling is full enough if more than 3/4 of the profiling sites in the function have data. If this threshold is not met, we reset the execution counters. We let this process repeat five times. The optimizing compilers tend to speculate that unprofiled code is unreachable. This is profitable if that code really won't ever run, but we want to be extra sure before doing that, hence we give functions with partial profiling 5× the time to warm up.

This is an exciting combination of heuristics! These heuristics were added early in the development of tiering in JSC. They were all added before we built the FTL, and the FTL inherited those heuristics just with a 100× multiplier. Each heuristic was added because it produced either a speed-up or a memory usage reduction or both. We try to remove heuristics that are not known to be speed-ups anymore, and to our knowledge, all of these still contribute to better performance on benchmarks we track.

Exit Counting

After we compile a function with the DFG or FTL, it's possible that one of the speculations we made is wrong. This will cause the function to OSR exit back to LLInt or Baseline (we prefer Baseline, but may throw away Baseline code during GC, in which case exits from DFG and FTL will go to LLInt). We've found that the best way of dealing with a wrong speculation is to throw away the optimized code and try optimizing again later with better profiling. We detect if a DFG or FTL function should be recompiled by counting exits. The exit count thresholds are:

- For a normal exit, we require `100 * pow(2, R)` exits to recompile.
- If the exit causes the Baseline JIT to enter its loop trigger (i.e. we got stuck in a hot loop after exit), then it's counted specially. We only allow `5 * pow(2, R)` of those kinds of exits before we recompile. Note that this can mean exiting five times and tripping the loop

optimization trigger each time or it can mean exiting once and tripping the loop optimization trigger five times.

The first step to recompilation is to *jettison* the DFG or FTL function. That means that all future calls to the function will call the Baseline or LLInt function instead.

Recompilation

If a function is jettisoned, we increment the recompilation counter (`R` in our notation) and reset the tier-up functionality in the Baseline JIT. This means that the function will keep running in Baseline for a while (twice as long as it did before it was optimized last time). It will gather new profiling, which we will be able to combine with the profiling we collected before to get an even more accurate picture of how types behave in the function.

It's worth looking at an example of this in action. We already showed an idealized case of tier-up in Figure 8, where a function gets compiled by each compiler exactly once and there are no OSR exits or recompilations. We will now show an example where things don't go so well. This example is picked because it's a particularly awful outlier. This isn't how we expect our engine to behave normally. We expect amusingly bad cases like the following to happen occasionally since the success or failure of speculation is random and random behavior means having bad outliers.

```
_handlePropertyAccessExpression = function (result, node)
{
    result.possibleGetOverloads = node.possibleGetOverloads;
    result.possibleSetOverloads = node.possibleSetOverloads;
    result.possibleAndOverloads = node.possibleAndOverloads;
    result.baseType = Node.visit(node.baseType, this);
    result.callForGet = Node.visit(node.callForGet, this);
    result.resultTypeForGet = Node.visit(node.resultTypeForGet, this);
    result.callForAnd = Node.visit(node.callForAnd, this);
    result.resultTypeForAnd = Node.visit(node.resultTypeForAnd, this);
    result.callForSet = Node.visit(node.callForSet, this);
    result.errorForSet = node.errorForSet;
    result.updateCalls();
}
```

This function belongs to the WSL subtest of JetStream 2. It's part of the WSL compiler's AST walk. It ends up being a large function after inlining Node.visit. When I ran this on my computer, I found that JSC did 8 compilations before hitting equilibrium for this function:

1. After running the function in LLInt for a bit, we compile this with Baseline. This is the easy part since Baseline doesn't need to be recompiled.
2. We compile with DFG. Unfortunately, the DFG compilation exits 101 times and gets jettisoned. The exit is due to a bad type check that the DFG emitted on `this`.
3. We again compile with the DFG. This time, we exit twice due to a check on `result`. This isn't enough times to trigger jettison and it doesn't prevent tier-up to the FTL.
4. We compile with the FTL. Unfortunately, this compilation gets jettisoned due to a failing watchpoint. Watchpoints (discussed in greater detail in later sections) are a way for the compiler to ask the runtime to notify it when bad things happen rather than emitting a check. Failing watchpoints cause immediate jettison. This puts us back in Baseline.
5. We try the DFG again. We exit seven times due to a bad check on `result`, just like in step 3. This still isn't enough times to trigger jettison and it doesn't prevent tier-up to the FTL.
6. We compile with the FTL. This time we exit 402 times due to a bad type check on `node`. We jettison and go back to Baseline.
7. We compile with the DFG again. This time there are no exits.
8. We compile with the FTL again. There are no further exits or recompilations.

This sequence of events has some intriguing quirks in addition to the number of compilations. Notice how in steps 3 and 5, we encounter exits due to a bad check on `result`, but none of the FTL compilations encounter those exits. This seems implausible since the FTL will do at least all of the speculations that the DFG did and a speculation that doesn't cause jettison also cannot pessimise future speculations. It's also surprising that the speculation that jettisons the FTL in step 6 wasn't encountered by the DFG. It is possible that the FTL does more speculations than the DFG, but that usually only happens in inlined functions, and this speculation on `node` doesn't seem to be in inlined code. A possible explanation for all of these surprising quirks is that the function is undergoing phase changes: during some parts of execution, it sees one set of types, and during another part of execution, it sees a somewhat different set. This is a common issue. Types are not random and they are often a function of time.

JavaScriptCore's compiler control system is designed to get good outcomes both for functions where speculation "just works" and for functions like the one in this example that need some extra time. To summarize, control is all about counting executions, exits, and recompilations, and either launching a higher tier compiler ("tiering up") or jettisoning optimized code and returning to Baseline.

Profiling

This section describes the profiling tiers of JavaScriptCore. The profiling tiers have the following responsibilities:

- To provide a non-speculative execution engine. This is important for start-up (before we do any speculation) and for OSR exits. OSR exit needs to exit to something that does no speculation so that we don't have chains of exits for the same operation.
- To record useful profiling. Profiling is useful if it enables us to make profitable speculations. Speculations are profitable if doing them makes programs run faster.

In JavaScriptCore, the LLInt and Baseline are the profiling tiers while DFG and FTL are the optimizing tiers. However, DFG and FTL also collect some profiling, usually only when it's free to do so and for the purpose of refining profiling collected by the profiling tiers.

This section is organized as follows. First we explain how JavaScriptCore's profiling tiers execute code. Then we explain the philosophy of how to profile. Finally we go into the details of JavaScriptCore's profiling implementation.

How Profiled Execution Works

JavaScriptCore profiles using the LLInt and Baseline tiers. LLInt interprets bytecode while Baseline compiles it. The two tiers share a nearly identical ABI so that it's possible to jump from one to the other at any bytecode instruction boundary.

LLInt: The Low Level Interpreter

The LLInt is an interpreter that obeys JIT ABI (in the style of HotSpot's interpreter). To that end, it is written in a portable assembly language called *offlineasm*. Offlineasm has a functional macro language (you can pass macro closures around) embedded in it. The offlineasm compiler is written in Ruby and can compile to multiple CPUs as well as C++. This section tells the story of why this crazy design produces a good outcome.

The LLInt simultaneously achieves multiple goals for JavaScriptCore:

- LLInt is JIT-friendly. The LLInt runs on the same stack that the JITs run on (which happens to be the C stack). The LLInt even agrees on register conventions with the JITs. This makes it cheap for LLInt to call JITed functions and vice versa. It makes LLInt→Baseline and Baseline→LLInt OSR trivial and it makes *any JIT*→LLInt OSR possible.
- LLInt allows us to execute JavaScript code even if we can't JIT. JavaScriptCore in no-JIT mode (we call it "mini mode") has some advantages: it's harder to exploit and uses less memory. Some JavaScriptCore clients prefer the mini mode. JSC is also used on CPUs that we don't have JIT support for. LLInt works great on those CPUs.
- LLInt reduces memory usage. Any machine code you generate from JavaScript is going to be big. Remember, there's a reason why they call JavaScript "high level" and machine code "low level": it refers to the fact that when you lower JavaScript to machine code, you're going to get many instructions for each JavaScript expression. Having the LLInt means that we don't have to generate machine code for all JavaScript code, which saves us memory.
- LLInt starts quickly. LLInt interprets our bytecode format directly. It's designed so that we could map bytecode from disk and point the interpreter at it. The LLInt is essential for achieving great page load time in the browser.
- LLInt is portable. It can be compiled to C++.

It would have been natural to write the LLInt in C++, since that's what most of JavaScriptCore is written in. But that would have meant that the interpreter would have a C++ stack frame constructed and controlled by the C++ compiler. This would have introduced two big problems:

1. It would be unclear how to OSR from the LLInt to the Baseline JIT or vice versa, since OSR would have to know how to decode and reencode a C++ stack frame. We don't doubt that it's possible to do this with enough cleverness, but it would create constraints on exactly how OSR works and it's not an easy piece of machinery to maintain.
2. JS functions running in the LLInt would have two stack frames instead of one. One of those stack frames would have to go onto the C++ stack (because it's a C++ stack frame). We have multiple choices of how to manage the JS stack frame (we could try to `alloca` it on top of the C++ frame, or allocate it somewhere else) but this inevitably increases cost: calls into the interpreter would have to do twice the work. A common optimization to this approach is to have interpreter→interpreter calls reuse the same C++ stack frame by managing a separate JS stack on the side. Then you can have the JITs use that separate JS stack. This still leaves cost when calling out of interpreter to JIT or vice versa.

A natural way to avoid these problems is to write the interpreter in assembly. That's basically what we did. But a JavaScript interpreter is a complex beast. It would be awful if porting JavaScriptCore to a new CPU meant rewriting the interpreter in another assembly language. Also, we want to use abstraction to write it. If we wrote it in C++, we'd probably have multiple functions, templates, and lambdas, and we would want all of them to be inlined. So we designed a new language, *offlineasm*, which has the following features:

- Portable assembly with our own mnemonics and register names that match the way we do portable assembly in our JIT. Some high-level mnemonics require lowering. Offlineasm reserves some scratch registers to use for lowering.
- The `macro` construct. It's best to think of this as a lambda that takes some arguments and returns void. Then think of the portable assembly statements as print statements that output that assembly. So, the macros are executed for effect and that effect is to produce an assembly program. These are the execution semantics of offlineasm at compile time.

Macros allow us to write code with rich abstractions. Consider this example from the LLInt:

```
macro llintJumpTrueOrFalseOp(name, op, conditionOp)
    llintOpWithJump(op_%name%, op, macro (size, get, jump, dispatch)
        get(condition, t1)
        loadConstantOrVariable(size, t1, t0)
        btqnz t0, ~0xf, .slow
        conditionOp(t0, .target)
        dispatch()

    .target:
        jump(target)

    .slow:
        callSlowPath(_llint_slow_path_%name%)
        nextInstruction()
    end)
end
```

This is a macro that we use for implementing both `jtrue` and `jfalse` and opcodes. There are only three lines of actual assembly in this listing: the `btqnz` (branch test quad not zero) and the two labels (`.target` and `.slow`). This also shows the use of first-class macros: on the second line, we call `llintOpWithJump` and pass it a macro closure as the third argument. The great thing about having a lambda-like construct like `macro` is that we don't need much else to have a pleasant programming experience. The LLInt is written in about 5000 lines of offlineasm (if you only count the 64-bit version).

To summarize, LLInt is an interpreter written in offlineasm. LLInt understands JIT ABI so calls and OSR between LLInt and JIT are cheap. The LLInt allows JavaScriptCore to load code more quickly, use less memory, and run on more platforms.

Baseline: The Bytecode Template JIT

The Baseline JIT achieves a speed-up over the LLInt at the cost of some memory and the time it takes to generate machine code. Baseline's speed-up is thanks to two factors:

- Removal of interpreter dispatch. Interpreter dispatch is the costliest part of interpretation, since the indirect branches used for selecting the implementation of an opcode are hard for the CPU to predict. This is the primary reason why Baseline is faster than LLInt.
- Comprehensive support for polymorphic inline caching. It is possible to do sophisticated inline caching in an interpreter, but currently our best inline caching implementation is the one shared by the JITs.

The Baseline JIT compiles bytecode by turning each bytecode instruction into a template of machine code. For example, a bytecode instruction like:
```
add loc6, arg1, arg2
```

Is turned into something like:
```
0x2f8084601a65: mov 0x30(%rbp), %rsi
0x2f8084601a69: mov 0x38(%rbp), %rdx
0x2f8084601a6d: cmp %r14, %rsi
0x2f8084601a70: jb 0x2f8084601af2
0x2f8084601a76: cmp %r14, %rdx
0x2f8084601a79: jb 0x2f8084601af2
0x2f8084601a7f: mov %esi, %eax
0x2f8084601a81: add %edx, %eax
0x2f8084601a83: jo 0x2f8084601af2
0x2f8084601a89: or %r14, %rax
0x2f8084601a8c: mov %rax, -0x38(%rbp)
```

The only parts of this code that would vary from one `add` instruction to another are the references to the operands. For example, `0x30(%rbp)` (that's x86 for the memory location at frame pointer plus 0x30) is the machine code representation of `arg1` in bytecode.

The Baseline JIT does few optimizations beyond just emitting code templates. It does no register allocation between instruction boundaries, for example. The Baseline JIT does some local optimizations, like if an operand to a math operation is a constant, or by using profiling information collected by the LLInt. Baseline also has good support for code repatching, which is essential for implementing inline caching. We discuss inline caching in detail later in this section.

To summarize, the Baseline JIT is a mostly unoptimized JIT compiler that focuses on removing interpreter dispatch overhead. This is enough to make it a ~2× speed-up over the LLInt.

Profiling Philosophy

Profiling in JSC is designed to be *cheap* and *useful*.

JavaScriptCore's profiling aims to incur little or no cost in the common case. Running with profiling turned on but never using the results to do optimizations should result in throughput that is about as good as if all of the profiling was disabled. We want profiling to be cheap because even in a long running program, lots of functions will only run once or for too short to make an optimizing JIT profitable. Some functions might finish running in less time than it takes to optimize them. The profiling can't be so expensive that it makes functions like that run slower.

Profiling is meant to help the compiler make the kinds of speculations that cause the program to run faster when we factor in both the speed-ups from speculations that are right and the slow-downs from speculations that are wrong. It's possible to understand this formally by thinking of speculation as a bet. We say that profiling is useful if it turns the speculation into a value bet. A value bet is one where the expected value (EV) is positive. That's another way of saying that the average outcome is profitable, so if we repeated the bet an infinite number of times, we'd be richer. Formally the expected value of a bet is:

```
p * B - (1 - p) * C
```

Where `p` is the probability of winning, `B` is the benefit of winning, and `C` is the cost of losing (both `B` and `C` are positive). A bet is a value bet iff:

```
p * B - (1 - p) * C > 0
```

Let's view speculation using this formula. The scenario in which we have the choice to make a bet or not is that we are compiling a bytecode instruction, we have some profiling that implies that we should speculate, and we have to choose whether to speculate or not. Let's say that `B` and `C` both have to do with the latency, in nanoseconds, of executing a bytecode instruction once. `B` is the improvement to that latency if we do some speculation and it turns out to be right. `C` is the regression to that latency if the speculation we make is wrong. Of course, after we have made a speculation, it will run many times and may be right sometimes and wrong sometimes. But `B` is just about the speed-up in the right cases, and `C` is just about the slow-down in the wrong cases. The baseline relative to which `B` and `C` are measured is the latency of the bytecode instruction if it was compiled with an optimizing JIT but without that particular OSR-exit-based speculation.

For example, we may have a less-than operation, and we are considering whether to speculate that neither input is double. We can of course compile less-than without making that speculation, so that's the baseline. If we do choose to speculate, then `B` is the speed-up to the average execution latency of that bytecode in those cases when neither input is double. Meanwhile, `C` is the slow-down to the average execution latency of that bytecode in those cases when at least one input is a double.

For `B`, let's just compute some bounds. The lower bound is zero, since some speculations are not profitable. A pretty good first order upper bound for `B` is the difference in per-bytecode-instruction latency between the baseline JIT and the FTL. Usually, the full speed-up of a bytecode instruction between baseline to FTL is the result of multiple speculations as well as nonspeculative compiler optimizations. So, a single speculation being responsible for the full difference in performance between baseline and FTL is a fairly conservative upper bound for `B`. Previously, we said that on average in the JetStream 2 benchmark on my computer, a bytecode instruction takes 1.71 ns to execute in Baseline and .225 ns to execute in FTL. So we can say:

```
B <= 1.71 ns - .225 ns = 1.48 ns
```

Now let's estimate `C`. `C` is how many more nanoseconds it takes to execute the bytecode instruction if we have speculated and we experience speculation failure. Failure means executing an OSR exit stub and then reexecuting the same bytecode instruction in baseline or LLInt. Then, all subsequent bytecodes in the function will execute in baseline or LLInt rather than DFG or FTL. Every 100 exits or so, we jettison and eventually recompile. Compiling is concurrent, but running a concurrent compiler is sure to slow down the main thread even if there is no lock contention. To fully capture `C`, we have to account for the cost of the OSR exit itself and then amortize the cost of reduced execution speed of the remainder of the function and the cost of eventual recompilation. Fortunately, it's pretty easy to measure this directly by hacking the DFG frontend to randomly insert pointless OSR exits with low probability and by having JSC report a count of the number of exits. I did an experiment with this hack for every JetStream 2 benchmark. Running without the synthetic exits, we get an execution time and a count of the number of exits. Running with synthetic exits, we get a longer execution time and a larger number of exits. The slope between these two points is an estimate of `C`. This is what I found, on the same machine that I used for running the experiments to compute `B`:

```
[DFG] C = 2499 ns
[FTL] C = 9998 ns
```

Notice how `C` is way bigger than `B`! This isn't some slight difference. We are talking about three orders of magnitude for the DFG and four orders of magnitude for the FTL. This paints a clear picture: speculation is a bet with tiny benefit and enormous cost.

For the DFG, this means that we need:

```
p > 0.9994
```

For speculation to be a value bet. `p` has to be even closer to 1 for FTL. Based on this, our philosophy for speculation is we won't do it unless we think that:

```
p ~ 1
```

Since the cost of speculation failure is so enormous, we only want to speculate when we know that we won't fail. The speed-up of speculation happens because we make lots of sure bets and only a tiny fraction of them ever fail.

It's pretty clear what this means for profiling:

- Profiling needs to focus on noting counterexamples to whatever speculations we want to do. We don't want to speculate if profiling tells us that the counterexample ever happened, since if it ever happened, then the EV of this speculation is probably negative. This means that we are not interested in collecting probability distributions. We just want to know if the bad thing ever happened.
- Profiling needs to run for a long time. It's common to wish for JIT compilers to compile hot functions sooner. One reason why we don't is that we need about 3-4 "nines" of confidence that that the counterexamples didn't happen. Recall that our threshold for tiering up into the DFG is about 1000 executions. That's probably not a coincidence.

Finally, since profiling is a bet, it's important to approach it with a healthy gambler's philosophy: the fact that a speculation succeeded or failed in a particular program does not tell us if the speculation is good or bad. Speculations are good or bad only based on their average behavior. Focusing too much on whether profiling does a good job for a particular program may result in approaches that cause it to perform badly on average.

Profiling Sources in JavaScriptCore

JavaScriptCore gathers profiling from multiple different sources. These profiling sources use different designs. Sometimes, a profiling source is a unique source of data, but other times, profiling sources are able to provide some redundant data. We only speculate when all profiling sources concur that the speculation would always succeed. The following sections describe our profiling sources in detail.

Case Flags

Case flags are used for branch speculation. This applies anytime the best way to implement a JS operation involves branches and multiple paths, like a math operation having to handle either integers or doubles. The easiest way to profile and speculate is to have the profiling tiers implement both sides of the branch and set a different flag on each side. That way, the optimizing tier knows that it can profitably speculate that only one path is needed if the flags for the other paths are not set. In cases where there is clearly a preferred speculation — for example, speculating that an integer add did not overflow is clearly preferred overspeculating that it did overflow — we only need flags on the paths that we don't like (like the overflow path).

Let's consider two examples of case flags in more detail: integer overflow and property accesses on non-object values.

Say that we are compiling an add operation that is known to take integers as inputs. Usually the way that the LLInt interpreter or Baseline compiler would "know" this is that the add operation we'll talk about is actually the part of a larger add implementation after we've already checked that the inputs are integers. Here's the logic that the profiling tier would use written as if it was C++ code to make it easy to parse:

```
int32_t left = ...;
int32_t right = ...;
ArithProfile* profile = ...; // This is the thing with the case flags.
int32_t intResult;
JSValue result; // This is a tagged JavaScript value that carries type.
if (UNLIKELY(addOverflowed(left, right, &intResult))) {
    result = jsNumber(static_cast<double>(left) +
                      static_cast<double>(right));

    // Set the case flag indicating that overflow happened.
    profile->setObservedInt32Overflow();
} else
    result = jsNumber(intResult);
```

When optimizing the code, we will inspect the `ArithProfile` object for this instruction. If `!profile->didObserveInt32Overflow()`, we will emit something like:

```
int32_t left = ...;
int32_t right = ...;
int32_t result;
speculate(!addOverflowed(left, right, &result));
```

I.e. we will add and branch to an exit on overflow. Otherwise we will just emit the double path:

```
double left = ...;
double right = ...;
double result = left + right;
```

Unconditionally doing double math is not that expensive; in fact on benchmarks that I've tried, it's cheaper than doing integer math and checking overflow. The only reason why integers are profitable is that they are cheaper to use for bit operations and pointer arithmetic. Since CPUs don't accept floats or doubles for bit and pointer math, we need to convert the double to an integer first if the JavaScript program uses it that way (pointer math arises when a number is used as an array index). Such conversions are relatively expensive even on CPUs that support them natively. Usually it's hard to tell, using profiling or any static analysis, whether a number that a program computed will be used for bit or pointer math in the future. Therefore, it's better to use integer math with overflow checks so that if the number ever flows into an operation that requires integers, we won't have to pay for expensive conversions. But if we learn that any such operation overflows — even occasionally — we've found that it's more efficient overall to unconditionally switch to double math. Perhaps the presence of overflows is strongly correlated with the result of those operations not being fed into bit math or pointer math.

A simpler example is how case flags are used in property accesses. As we will discuss in the inline caches section, property accesses have associated metadata that we use to track details about their behavior. That metadata also has flags, like the `sawNonCell` bit, which we set to true if the property access ever sees a non-object as the base. If the flag is set, the optimizing compilers know not to speculate that the property access will see objects. This typically forces all kinds of conservatism for that property access, but that's better than speculating wrong and exiting in this case. Lots of case flags look like `sawNonCell`: they are casually added as a bit in some existing data structure to help the optimizing compiler know which paths were taken.

To summarize, case flags are used to record counterexamples to the speculations that we want to do. They are a natural way to implement profiling in those cases where the profiling tiers would have had to branch anyway.

Case Counts

A predecessor to case flags in JavaScriptCore is case counts. It's the same idea as flags, but instead of just setting a bit to indicate that a bad thing happened, we would count. If the count never got above some threshold, we would speculate.

Case counts were written before we realized that the EV of speculation is awful unless the probability of success is basically 1. We thought that we could speculate in cases where we knew we'd be right a majority of the time, for example. Initial versions of case counts had variable thresholds — we would compute a ratio with the execution count to get a case rate. That didn't work as well as fixed thresholds, so we switched to a fixed count threshold of 100. Over time, we lowered the threshold to 20 or 10, and then eventually found that the threshold should really be 1, at which point we switched to case flags.

Some functionality still uses case counts. We still have case counts for determining if the `this` argument is exotic (some values of `this` require the function to perform a possibly-effectful conversion in the prologue). We still have case counts as a backup for math operations overflowing, though that is almost certainly redundant with our case flags for math overflow. It's likely that we will remove case counts from JavaScriptCore eventually.

Value Profiling

Value profiling is all about inferring the types of JavaScript values (JSValues). Since JS is a dynamic language, JSValues have a runtime type. We use a 64-bit JSValue representation that uses bit encoding tricks to hold either doubles, integers, booleans, null, undefined, or pointers to cell, which may be JavaScript objects, symbols, or strings. We refer to the act of encoding a value in a JSValue as *boxing* it and the act of decoding as *unboxing* (note that *boxing* is a term used in other engines to refer specifically to the act of allocating a box object in the heap to hold a value; our use of the term *boxing* is more like what others call *tagging*). In order to effectively optimize JavaScript, we need to have some way of inferring the type so that the compiler can assume things about it statically. Value profiling tracks the set of values that a particular program point saw so that we can predict what types that program point will see in the future.

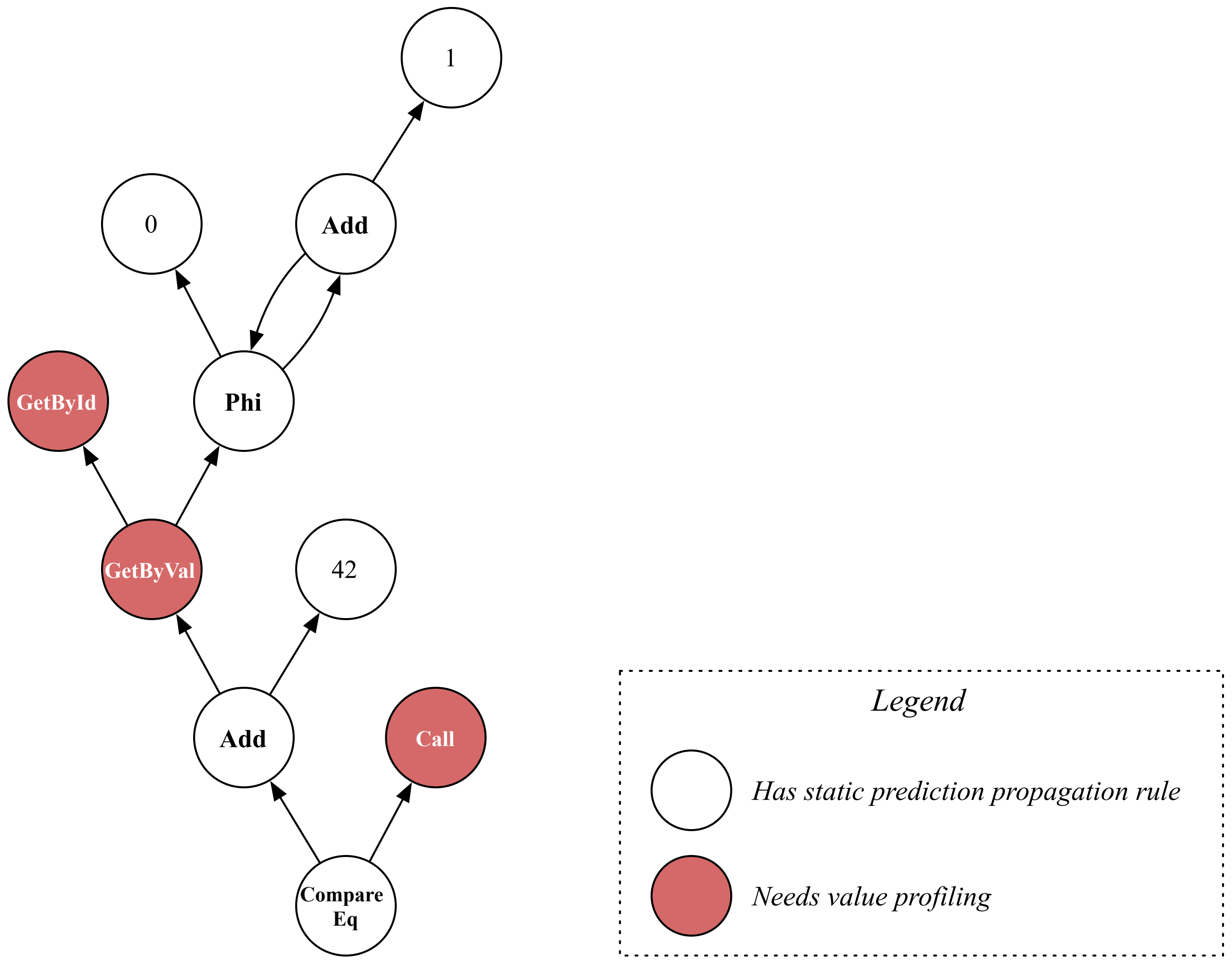


Figure 12. Value profiling and prediction propagation for a sample data flow graph.

We combine value profiling with a static analysis called *prediction propagation*. The key insight is that prediction propagation can infer good guesses for the types for most operations if it is given a starting point for certain opaque operations:

- Arguments incoming to the function.
- Results of most load operations.
- Results of most calls.

There's no way that a static analysis running just on some function could guess what types loads from plain JavaScript arrays or calls to plain JavaScript functions could have. Value profiling is about trying to help the static analysis guess the types of those opaque operations. Figure 12 shows how this plays out for a sample data flow graph. There's no way static analysis can tell the type of most GetByVal and GetById oerations, since those are loads from dynamically typed locations in the heap. But if we did know what those operations return then we can infer types for this entire graph by using simple type rules for Add (like that if it takes integers as inputs and the case flags tell us there was no overflow then it will produce integers).

Let’s break down value profiling into the details of how exactly values are profiled, how prediction propagation works, and how the results of prediction propagation are used.

Recording value profiles. At its core, value profiling is all about having some program point (either a point in the interpreter or something emitted by the Baseline JIT) log the value that it saw. We log values into a single bucket so that each time the profiling point runs, it overwrites the last seen value. The code looks like this in the LLInt:

```
macro valueProfile(op, metadata, value)
    storeq value, %op%::Metadata::profile.m_buckets[metadata]
end
```

Let’s look at how value profiling works for the `get_by_val` bytecode instruction. Here’s part of the code for `get_by_val` in LLInt:

```
llintOpWithMetadata(
    op_get_by_val, OpGetByVal,
    macro (size, get, dispatch, metadata, return)
        macro finishGetByVal(result, scratch)
            get(dst, scratch)
            storeq result, [cfr, scratch, 8]
            valueProfile(OpGetByVal, t5, result)
            dispatch()
        end

        ... // more code for get_by_val
```

The implementation of `get_by_val` includes a `finishGetByVal` helper macro that stores the result in the right place on the stack and then dispatches to the next instruction. Note that it also calls `valueProfile` to log the result just before finishing.

Each ValueProfile object has a pair of buckets and a *predicted type*. One bucket is for normal execution. The `valueProfile` macro in the LLInt uses this bucket. The other bucket is for OSR exit: if we exit due to a speculation on a type that we got from value profiling, we feed the value that caused OSR exit back into the second bucket of the ValueProfile.

Each time that our execution counters (used for controlling when to invoke the next tier) count about 1000 points, the execution counting slow path updates all predicted types for the value profiles in that function. Updating value profiles means computing a predicted type for the value in the bucket and merging that type with the previously predicted type. Therefore, after repeated predicted type updates, the type will be broad enough to be valid for multiple different values that the code saw.

Predicted types use the *SpeculatedType* type system. A SpeculatedType is a 64-bit integer in which we use the low 40 bits to represent a set of 40 fundamental types. The fundamental types, shown in Figure 13, represent non-overlapping set of possible JSValues. 2^{40} SpeculatedTypes are possible by setting any combination of bits.

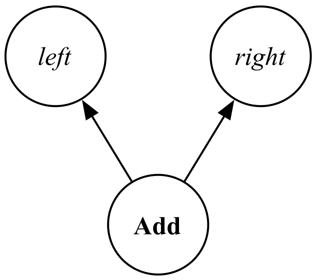
FinalObject	Array	FunctionWithDefaultHasInstance	FunctionWithNonDefaultHasInstance	Int8Array
Int16Array	Int32Array	Uint8Array	Uint8ClampedArray	Uint16Array
Uint32Array	Float32Array	Float64Array	DirectArguments	ScopedArguments
StringObject	RegExpObject	MapObject	SetObject	WeakMapObject
WeakSetObject	ProxyObject	DerivedArray	ObjectOther	StringIdent
StringVar	Symbol	CellOther	BoolInt32	NonBoolInt32
Int52Only	AnyIntAsDouble	NotIntAsDouble	DoublePureNaN	DoubleImpureNaN
Boolean	Other	Empty	BigInt	DataViewObject

Figure 13. All of the fundamental SpeculatedTypes.

This allows us to invent whatever types are useful for optimization. For example, we distinguish between 32-bit integers whose value is either 0 or 1 (BoolInt32) versus whose value is anything else (NonBoolInt32). Together these form the Int32Only type, which just has both bits set. BoolInt32 is useful for cases there integers are converted to booleans.

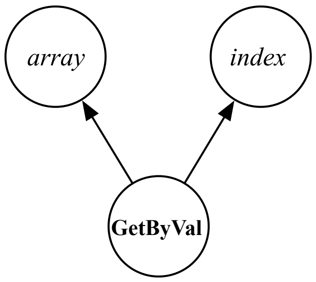
Prediction propagation. We use value profiling to fill in the blanks for the prediction propagation pass of the DFG compiler pipeline. Prediction propagation is an abstract interpreter that tracks the set of types that each variable in the program can have. It's unsound since the types it produces are just predictions (it can produce any combination of types and at worst we will just OSR exit too much). However, it can be said that we optimize it to be sound; the more sound it is, the fewer OSR exits we have. Prediction propagation fills in the things that the abstract interpreter can't reason about (loads from the heap, results returned by calls, arguments to the function, etc.) using the results of value profiling. On the topic of soundness, we would consider it to be a bug if the prediction propagation was unsound in a world where value profiling is never wrong. Of course, in reality, we know that value profiling will be wrong, so we know that prediction propagation is unsound.

Let's consider some of the cases where prediction propagation can be sure about the result type of an operation based on the types of its inputs.



type of *left*	type of *right*	type of **Add**
Int32	Int32	Int32
Int32	Double	Double
Double	Int32	Double
Double	Double	Double

Figure 14. Some of the prediction propagation rules for Add. This figure doesn't show the rules for string concatenation and objects.



type of *array*	type of *index*	type of **GetByVal**
Int32Array	Int32	Int32
Float64Array	Int32	Double
Object	Int32	?
Object	String	?

Figure 15. Some of the prediction propagation rules for GetByVal (the DFG opcode for subscript access like `array[index]`). This figure only shows a small sample of the GetByVal rules.

Figure 14 shows some of the rules for the Add operation in DFG IR. Prediction propagation and case flags tell us everything we want to know about the output of Add. If the inputs are integers and the overflow flag isn't set, the output is an integer. If the inputs are any other kinds of numbers or there are overflows, the output is a double. We don't need anything else (like value profiling) to understand the output type of Add.

Figure 15 shows some of the rules for GetByVal, which is the DFG representation of `array[index]`. In this case, there are types of arrays that could hold any type of value. So, even knowing that it is a JSArray isn't enough to know the types of values inside the array. Also, if the `index` is a string, then this could be accessing some named property on the `array` object or one of its prototypes and those could have any type. It's in cases like GetByVal that we leverage value profiling to guess what the result type is.

Prediction propagation combined with value profiling allows the DFG to infer a predicted type at every point in the program where a variable is used. This allows operations that don't do any profiling on their own to still perform type-based speculations. It's of course possible to also have bytecode instructions that can speculate on type collect case flags (or use some other mechanism) to drive those speculations — and that approach can be more precise — but value profiling means that we don't have to do this for every operation that wants type-based speculation.

Using predicted types. Consider the CompareEq operation in DFG IR, which is used for the DFG lowering of the `eq`, `eq_null`, `neq`, `neq_null`, `jeq`, `jeq_null`, `jneq`, and `jneq_null` bytecodes. These bytecodes do no profiling of their own. But CompareEq is one of the most aggressive type speculators in all of the DFG. CompareEq can speculate on the types it sees without doing any profiling of its own because the values it uses will either have value profiling or will have a predicted type filled in by prediction propagation.

Type speculations in the DFG are written like:
```
CompareEq(Int32:@left, Int32:@right)
```

This example means that the CompareEq will specuate that both operands are Int32. CompareEq supports the following speculations, plus others we don't list here:
```
CompareEq(Boolean:@left, Boolean:@right)
CompareEq(Int32:@left, Int32:@right)
CompareEq(Int32:BooleanToNumber(Boolean:@left), Int32:@right)
CompareEq(Int32:BooleanToNumber(Untyped:@left), Int32:@right)
CompareEq(Int32:@left, Int32:BooleanToNumber(Boolean:@right))
CompareEq(Int32:@left, Int32:BooleanToNumber(Untyped:@right))
CompareEq(Int52Rep:@left, Int52Rep:@right)
CompareEq(DoubleRep:DoubleRep(Int52:@left), DoubleRep:DoubleRep(Int52:@right))
CompareEq(DoubleRep:DoubleRep(Int52:@left), DoubleRep:DoubleRep(RealNumber:@right))
CompareEq(DoubleRep:DoubleRep(Int52:@left), DoubleRep:DoubleRep(Number:@right))
CompareEq(DoubleRep:DoubleRep(Int52:@left), DoubleRep:DoubleRep(NotCell:@right))
CompareEq(DoubleRep:DoubleRep(RealNumber:@left), DoubleRep:DoubleRep(RealNumber:@right))
CompareEq(DoubleRep:..., DoubleRep:...)
CompareEq(StringIdent:@left, StringIdent:@right)
CompareEq(String:@left, String:@right)
CompareEq(Symbol:@left, Symbol:@right)
CompareEq(Object:@left, Object:@right)
CompareEq(Other:@left, Untyped:@right)
CompareEq(Untyped:@left, Other:@right)
CompareEq(Object:@left, ObjectOrOther:@right)
CompareEq(ObjectOrOther:@left, Object:@right)
CompareEq(Untyped:@left, Untyped:@right)
```

Some of these speculations, like `CompareEq(Int32:, Int32:)` or `CompareEq(Object:, Object:)`, allow the compiler to just emit an integer compare instruction. Others, like `CompareEq(String:, String:)`, emit a string compare loop. We have lots of variants to optimally handle bizarre comparisons that are not only possible in JS but that we have seen happen frequently in the wild, like comparisons between numbers and booleans and comparisons between one value that is always a number and another that is either a number or a boolean. We provide additional optimizations for comparisons between doubles, comparisons between strings that have been hash-consed (so-called StringIdent, which can be compared using comparison of the string pointer), and comparisons where we don't know how to speculate (`CompareEq(Untyped:, Untyped:)`).

The basic idea of value profiling — storing a last-seen value into a bucket and then using that to bootstrap a static analysis — is something that we also use for profiling the behavior of array accesses. *Array profiles* and *array allocation profiles* are like value profiles in that they save the last result in a bucket. Like value profiling, data from those profiles is incorporated into prediction propagation.

To summarize, value profiling allows us to predict the types of variables at all of their use sites by just collecting profiling at those bytecode instructions whose output cannot be predicted with abstract interpretation. This serves as the foundation for how the DFG (and FTL, since it reuses the DFG's frontend) speculates on the types of JSValues.

Inline Caches

Property accesses and function calls are particularly difficult parts of JavaScript to optimize:

- Objects behave as if they were just ordered mappings from strings to JSValues. Lookup, insertion, deletion, replacement, and iteration are possible. Programs do these operations a lot, so they have to be fast. In some cases, programs use objects the same way that programs in other languages would use hashtables. In other cases, programs use objects the same way that they would in Java or some sensibly-typed object-oriented language. Most programs do both.
- Function calls are polymorphic. You can't make static promises about what function will be called.

Both of these dynamic features are amenable to optimization with Deutsch and Schiffman's inline caches (ICs). For dynamic property access, we combine this with *structures*, based on the idea of *maps* in the Chambers, Ungar, and Lee's Self implementation. We also follow Hölzle, Chambers, and Ungar: our inline caches are *polymorphic* and we use data from these caches as profiling of the types observed at a particular property access or call site.

It's worth dwelling a bit on the power of inline caching. Inline caches are great optimizations separately from speculative compilation. They make the LLInt and Baseline run faster. Inline caches are our most powerful profiling source, since they can precisely collect information about every type encountered by an access or call. Note that we previously said that good profiling has to be cheap. We think of inline caches as negative cost profiling since inline caches make the LLInt and Baseline faster. It doesn't get cheaper than that!

This section focuses on inline caching for dynamic property access, since it's strictly more complex than for calls (accesses use structures, polymorphic inline caches (PICs), and speculative compilation; calls only use polymorphic inline caches and speculative compilation). We organize our discussion of inline caching for dynamic property access as follows. First we describe how structures work. Then we show the JavaScriptCore object model and how it incorporates structures. Next we show how inline caches work. Then we show how profiling from inline caches is used by the optimizing compilers. After that we show how inline caches support polymorphism and polyvariance. Finally we talk about how inline caches are integrated with the garbage collector.

Structures. Objects in JavaScript are just mappings from strings to JSValues. Lookup, insertion, deletion, replacement, and iteration are all possible. We want to optimize those uses of objects that would have had a type if the language had given the programmer a way to say it.

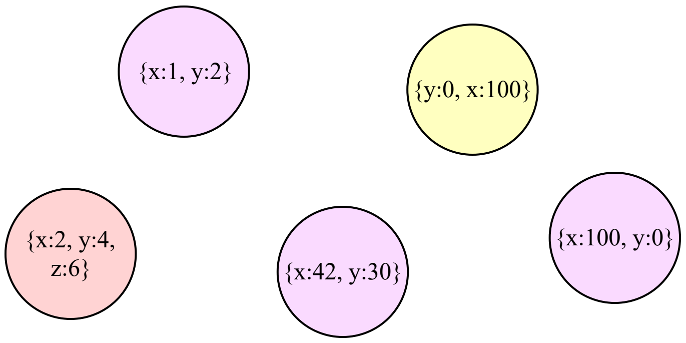


Figure 16. Some JavaScript objects that have x and y properties. Some of them have exactly the same shape (only x and y in the same order).

Consider how to implement a property access like:
```
var tmp = o.x;
```

Or:
```
o.x = tmp;
```

One way to make this fast is to use hashtables. That's certainly a necessary fallback mechanism when the JavaScript program uses objects more like hashtables than like objects (i.e. it frequently inserts and deletes properties). But we can do better.

This problem frequently arises in dynamic programming languages and it has a well-understood solution. The key insight of Chambers, Ungar, and Lee's Self implementation is that property access sites in the program will typically only see objects of the same shape. Consider the objects in Figure 16 that have x and y properties. Of course it's possible to insert x and y in two possible orders, but folks will tend to pick some order and stick to it (like x first). And of course it's possible to also have objects that have a z property, but it's less likely that a property access written as part of the part of the program that works with {x, y} objects will be reused for the part that uses {x, y, z}. It's possible to have shared code for many different kinds of objects but unshared code is more common. Therefore, we split the object representation into two parts:

- The object itself, which only contains the property values and a structure pointer.
- The *structure*, which is a hashtable that maps property names (strings) to indices in the objects that have that structure.

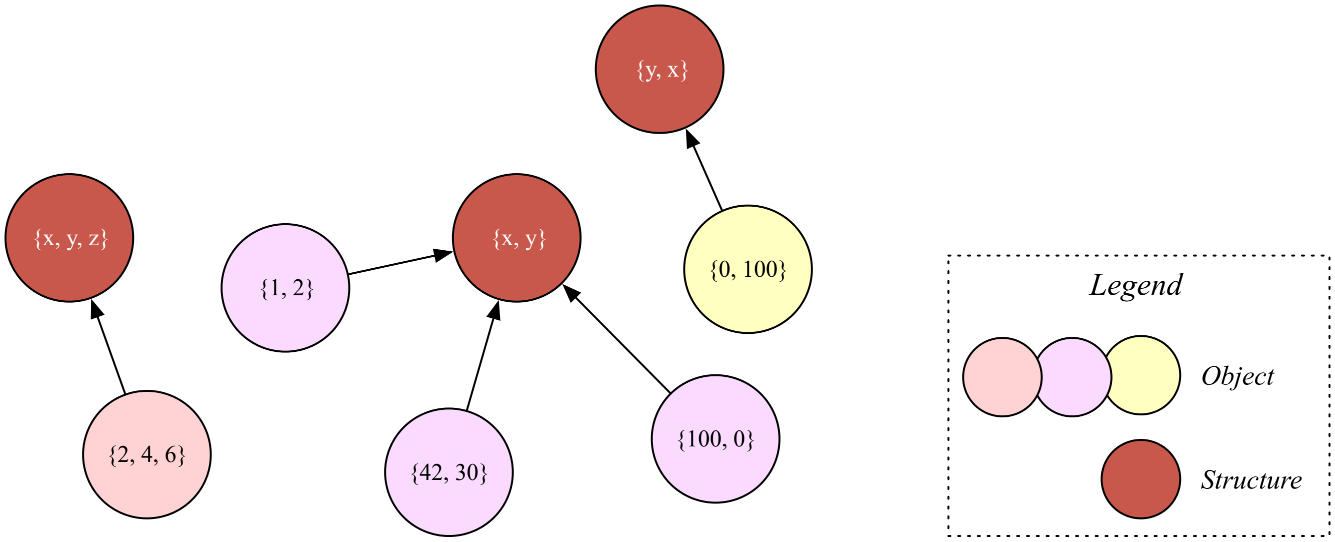


Figure 17. The same objects as in Figure 16, but using structures.

Figure 17 shows objects represented using structures. Objects only contain object property values and a pointer to a structure. The structure tells the property names and their order. For example, if we wanted to ask the {1, 2} object in Figure 17 for the value of property x, we would load the pointer to its structure, {x, y}, and ask that structure for the index of x. The index is 0, and the value at index 0 in the {1, 2} object is 1.

A key feature of structures is that they are hash consed. If two objects have the same properties in the same order, they are likely to have the same structure. This means that checking if an object has a certain structure is O(1): just load the structure pointer from the object header and compare the pointer to a known value.

Structures can also indicate that objects are in *dictionary* or *uncacheable dictionary* mode, which are basically two levels of hashtable badness. In both cases, the structure stops being hash consed and is instead paired 1:1 with its object. Dictionary objects can have new properties added to them without the structure changing (the property is added to the structure in-place). Uncacheable dictionary objects can have properties deleted from them without the structure changing. We won't go into these modes in too much detail in this post.

To summarize, structures are hashtables that map property names to indices in the object. Object property lookup uses the object's structure to find the index of the property. Structures are hash consed to allow for fast structure checks.

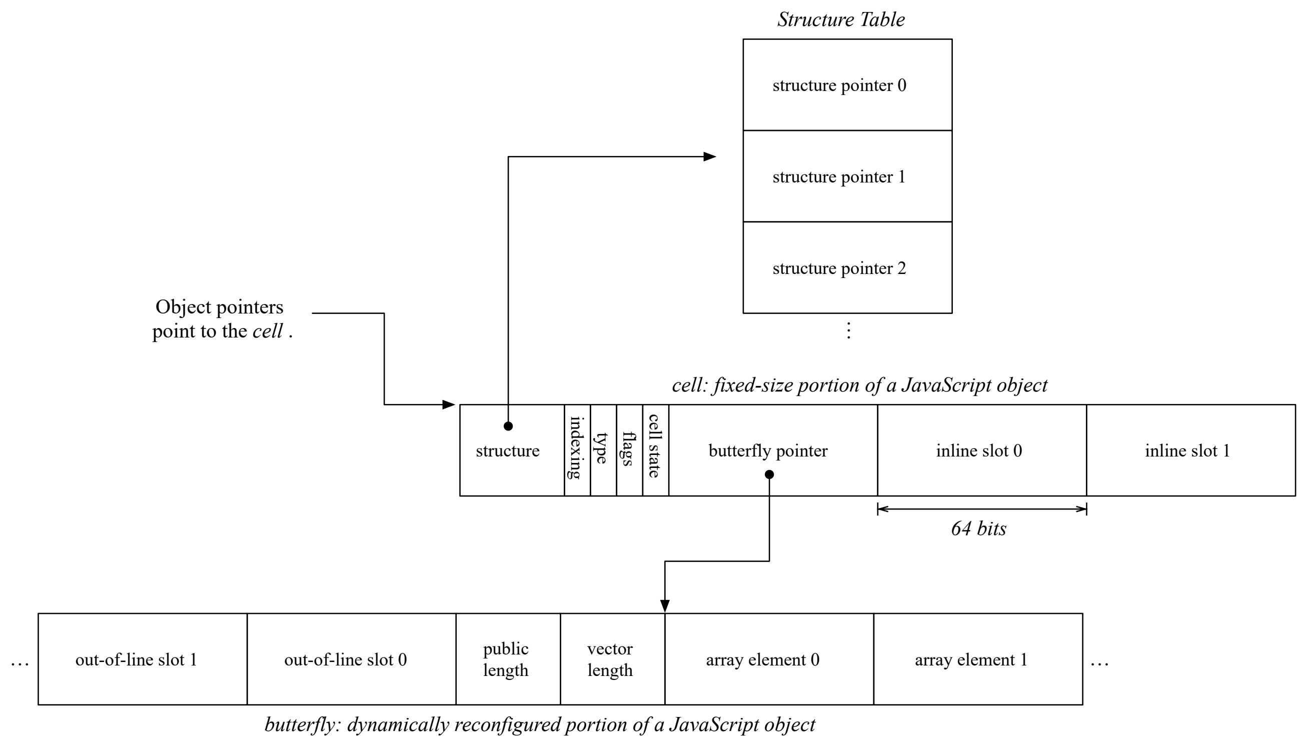


Figure 18. The JavaScriptCode object model.

JavaScriptCore object model. JavaScriptCore uses objects with a 64-bit header that includes a 32-bit structure ID and 32 bits worth of extra state for GC, type checks, and arrays. Figure 18 shows the object model. Named object properties may end up either in the inline slots or the out-of-line slots. Objects get some number of inline slots based on simple static analysis around the allocation site. If a property is added that doesn't fit in the inline slots, we allocate a *butterfly* to store additional properties out-of-line. Accessing out-of-line properties in the butterfly costs one extra load.

Figure 19 shows an example object that only has two inline properties. This is the kind of object you would get if you used the object literal `{f:5, g:6}` or if you assigned to the `f` and `g` properties reasonably close to the allocation.

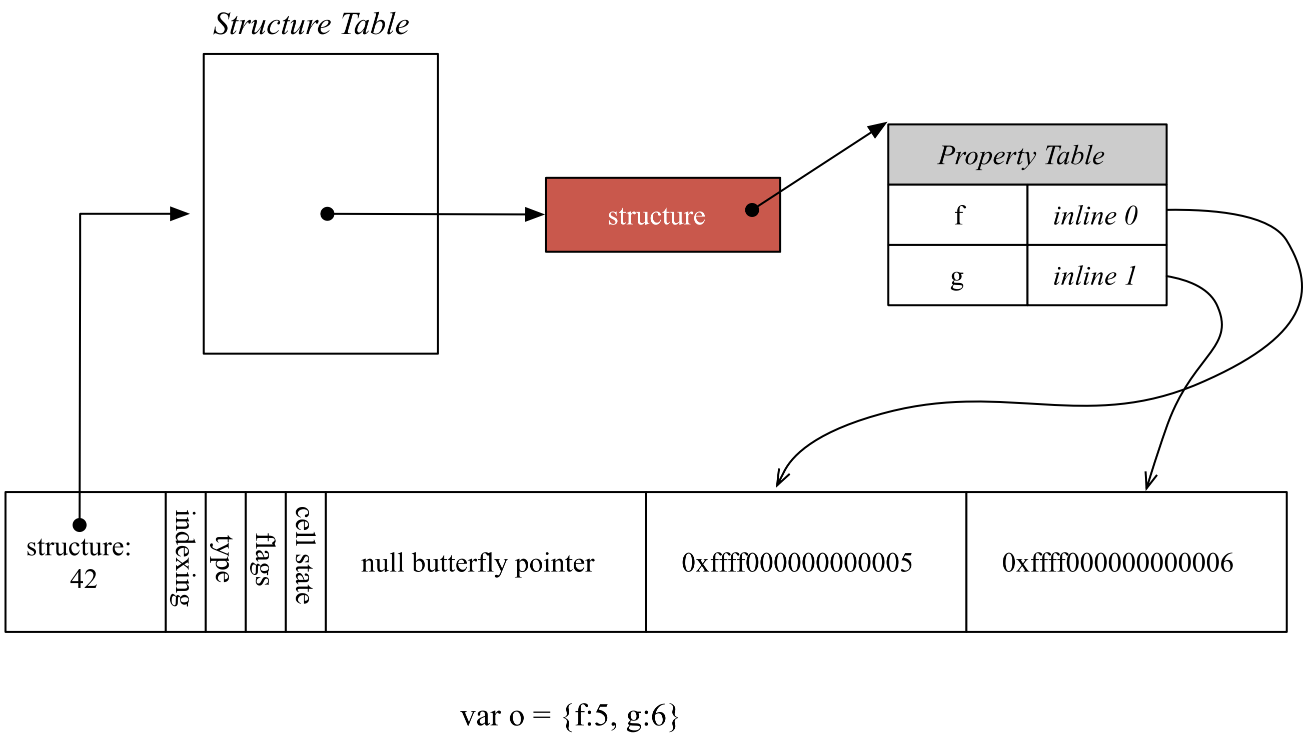


Figure 19. Example JavaScriptCore object together with its structure.

Simple inline caches. Let's consider the code:
```
var v = o.f;
```

Let's assume that all of the objects that flow into this have structure 42 like the object in Figure 19. Inline caching this property access is all about emitting code like the following:
```
if (o->structureID == 42)
    v = o->inlineStorage[0]
else
    v = slowGet(o, "f")
```

But how do we know that `o` will have structure 42? JavaScript does not give us this information statically. Inline caches get this information by filling it in once the code runs. There are a number of techniques for this, all of which come down to self-modifying code. Let's look at how the LLInt and Baseline do it.

In the LLInt, the metadata for `get_by_id` contains a cached structure ID and a cached offset. The cached structure ID is initialized to an absurd value that no structure can have. The fast path of `get_by_id` loads the property at the cached offset if the object has the cached structure. Otherwise, we take a slow path that does the full lookup. If that full lookup is cacheable, it stores the structure ID and offset in the metadata.

The Baseline JIT does something more sophisticated. When emitting a `get_by_id`, it reserves a slab of machine code space that the inline caches will later fill in with real code. The only code in this slab initially is an unconditional jump to a slow path. The slow path does the fully dynamic lookup. If that is deemed cacheable, the reserved slab is replaced with code that does the right structure check and loads at the right offset. Here's an example of a `get_by_id` initially compiled with Baseline:
```
0x46f8c30b9b0: mov 0x30(%rbp), %rax
0x46f8c30b9b4: test %rax, %r15
```

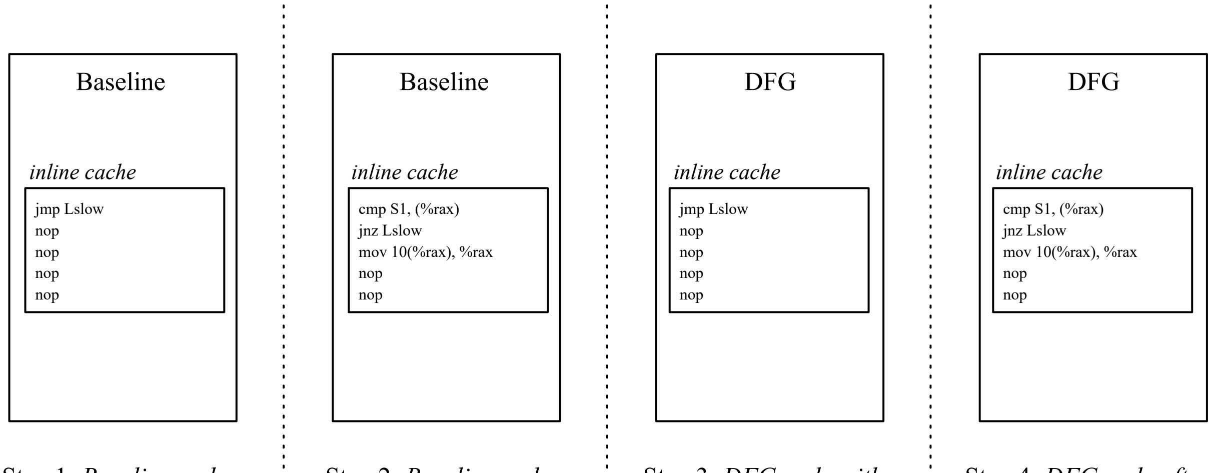
```
0x46f8c30b9b7: jnz 0x46f8c30ba2c
0x46f8c30b9bd: jmp 0x46f8c30ba2c
0x46f8c30b9c2: o16 nop %cs:0x200(%rax,%rax)
0x46f8c30b9d1: nop (%rax)
0x46f8c30b9d4: mov %rax, -0x38(%rbp)
```

The first thing that this code does is check that `o` (stored in `%rax`) is really an object (using a `test` and `jnz`). Then notice the unconditional `jmp` followed by two long nop instructions. This jump goes to the same slow path that we would have branched to if `o` was not an object. After the slow path runs, this is repatched to:

```
0x46f8c30b9b0: mov 0x30(%rbp), %rax
0x46f8c30b9b4: test %rax, %r15
0x46f8c30b9b7: jnz 0x46f8c30ba2c
0x46f8c30b9bd: cmp $0x125, (%rax)
0x46f8c30b9c3: jnz 0x46f8c30ba2c
0x46f8c30b9c9: mov 0x18(%rax), %rax
0x46f8c30b9cd: nop 0x200(%rax)
0x46f8c30b9d4: mov %rax, -0x38(%rbp)
```

Now, the is-object check is followed by a structure check (using `cmp` to check that the structure is 0x125) and a load at offset 0x18.

Inline caches as a profiling source. The metadata we use to maintain inline caches makes for a fantastic profiling source. Let's look closely at what this means.



```
jmp Lslow
nop
nop
nop
nop
```

```
cmp S1, (%rax)
jnz Lslow
mov 10(%rax), %rax
nop
nop
```

```
jmp Lslow
nop
nop
nop
nop
```

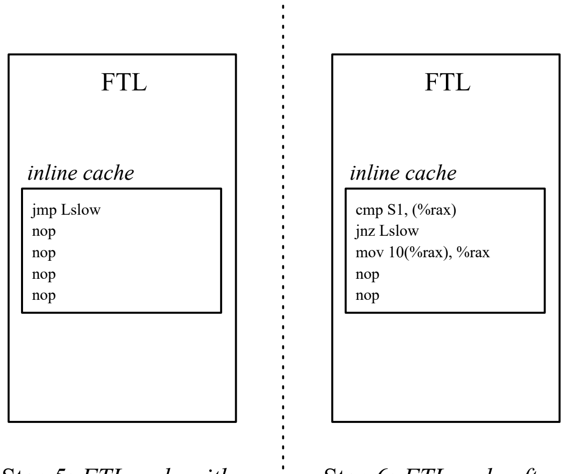
```
cmp S1, (%rax)
jnz Lslow
mov 10(%rax), %rax
nop
nop
```

Step 1: *Baseline code with uninitialized IC.*

Step 2: *Baseline code after IC repartching.*

Step 3: *DFG code with uninitialized IC.*

Step 4: *DFG code after IC repatching.*



```
jmp Lslow
nop
nop
nop
nop
```

```
cmp S1, (%rax)
jnz Lslow
mov 10(%rax), %rax
nop
nop
```

Step 5: *FTL code with uninitialized IC.*

Step 6: *FTL code after IC repatching.*

*Figure 20. Timeline of using an inline cache at each JIT tier. Note that we end up having to generate code for this `get_by_id` *six times* in the berst case that each tier compiles this only once.*

Figure 20 shows a naive use of inline caches in a multi-tier engine, where the DFG JIT forgets everything that we learned from the Baseline inline cache and just compiles a blank inline cache. This is reasonably efficient and we fall back on this approach when the inline caches from the LLInt and Baseline tell us that there is unmanagable polymorphism. Before we go into how polymorphism is profiled, let's look at how a speculative compiler really wants to handle simple monomorphic inline caches like the one in Figure 20, where we only see one structure (S1) and the code that the IC emits is trivial (load at offset 10 from `%rax`).

When the DFG frontend (shared by DFG and FTL) sees an operation like `get_by_id` that can be implemented with ICs, it reads the state of all ICs generated for that `get_by_id`. By "all ICs" we mean all ICs that are currently in the heap. This usually just means reading the LLInt and Baseline ICs, but if there exists a DFG or FTL function that generated an IC for this `get_by_id` then we will also read that IC. This can happen if a function gets compiled multiple times due to inlining — we may be compiling function `bar` that inlines a call to function `foo` and `foo` already got compiled with FTL and the FTL emitted an IC for our `get_by_id`.

If all ICs for a `get_by_id` concur that the operation is monomorphic and they tell us the structure to use, then the DFG frontend converts the `get_by_id` into inline code that does not get repatched. This is shown in Figure 21. Note that this simple `get_by_id` is lowered to two DFG operations: CheckStructure, which OSR exits if the given object does not have the required structure, and GetByOffset, which is just a load with known offset and field name.

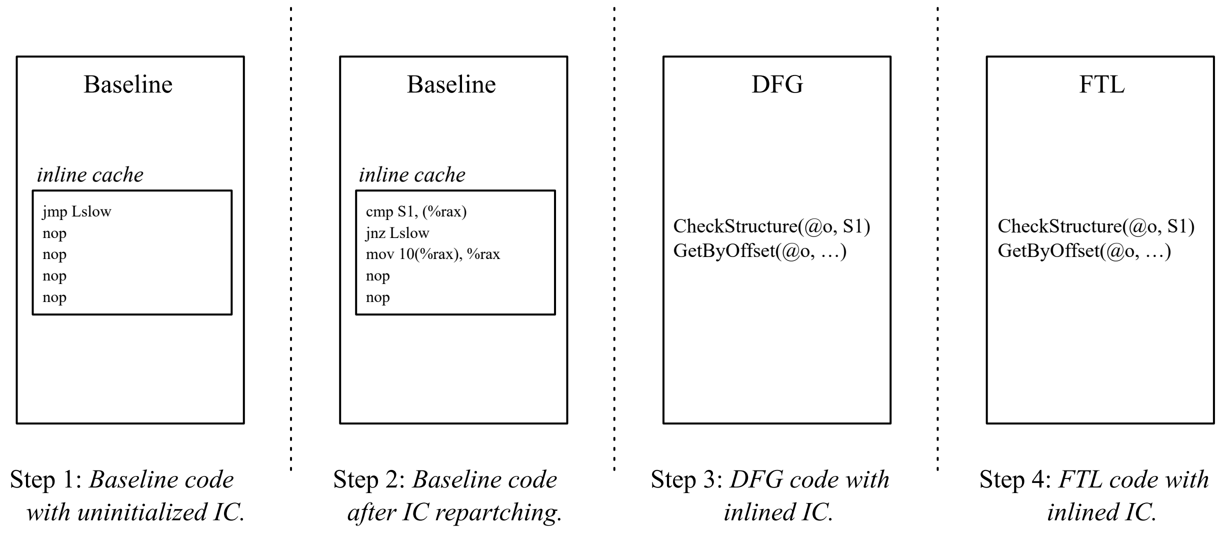


Figure 21. Inlining a simple momomorphic inline cache in DFG and FTL.

CheckStructure and GetByOffset are understood precisely in DFG IR:

- CheckStructure is a load to get the structure ID of an object and a branch to compare that structure ID to a constant. The compiler knows what structures are. After a CheckStructcure, the compiler knows that it's safe to execute loads to any of the properties that the structure says that the object has.
- GetByOffset is a load from either an inline or out-of-line property of a JavaScript object. The compiler knows what kind of property is being loaded, what its offset is, and what the name of the property would have been.

The DFG knows all about how to model these operations and the dependency between them:

- The DFG knows that neither operation causes a side effect, but that the CheckStructure represents a conditional side exit, and both operations read the heap.
- The DFG knows that two CheckStructures on the same structure are redundant unless some operation between them could have changed object structure. The DFG knows a lot about how to optimize away redundant structure checks, even in cases where there is a function call between two of them (more on this later).
- The DFG knows that two GetByOffsets that speak of the same property and object are loading from the same memory location. The DFG knows how to do alias analaysis on those properties, so it can precisely know when a GetByOffset's memory location got clobbered.
- The DFG knows that if it wants to hoist a GetByOffset then it has to ensure that the corresponding CheckStructure gets hoisted first. It does this using abstract interpretation, so there is no need to have a dependency edge between these operations.
- The DFG knows how to generate either machine code (in the DFG tier) or B3 IR (in the FTL tier) for CheckStructure and GetByOffset. In B3, CheckStructure becomes a Load, NotEqual, and Check, while GetByOffset usually just becomes a Load.

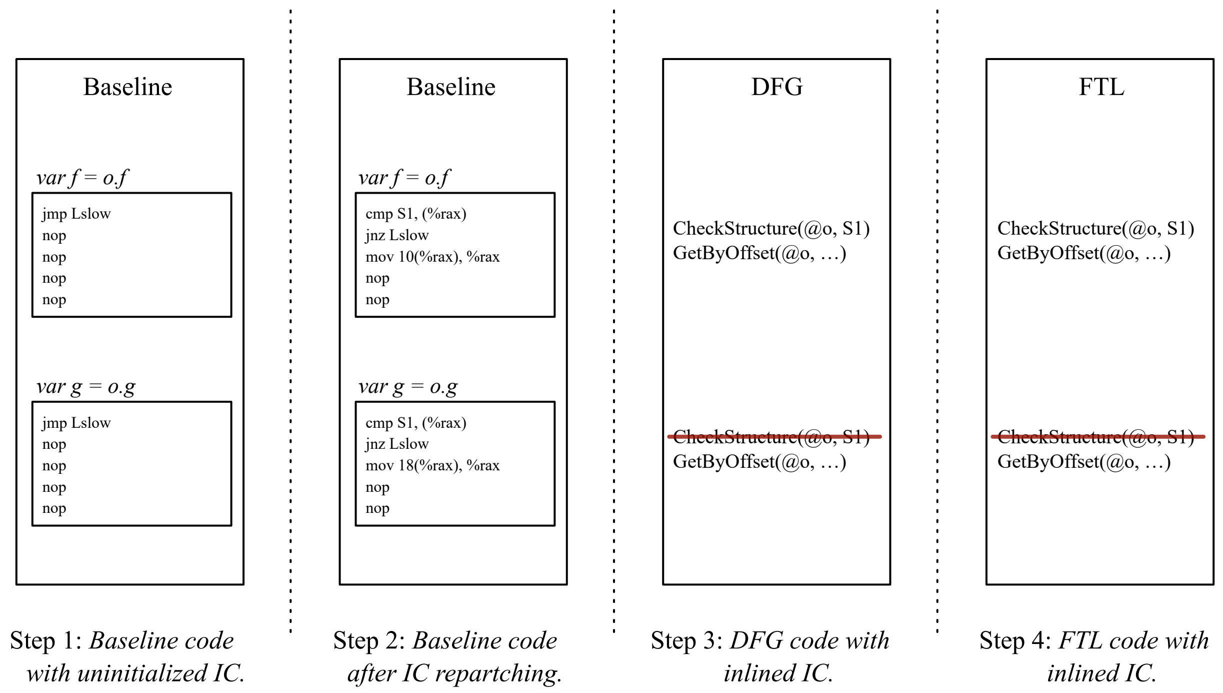


Figure 22. Inlining two momomorphic inline caches, for different properties on the same object, in DFG and FTL. The DFG and FTL are able to eliminate the CheckStructure for the second IC.

The biggest upshot of lowering ICs to CheckStructure and GetByOffset is the redundancy elimination. The most common redundancy we eliminate is multiple CheckStrutures. Lots of code will do multiple loads from the same object, like:

```
var f = o.f;
var g = o.g;
```

With ICs, we would check the structure twice. Figure 22 shows what happens when the speculative compilers inline these ICs. We are left with just a single CheckStructure instead of two thanks to the fact that:

- CheckStructure is an OSR speculation.
- CheckStructure is not an IC. The compiler knows exactly what it does, so that it can model it, so that it can eliminate it.

Let's pause to appreciate what this technique gives us so far. We started out with a language in which property accesses seem to need hashtable lookups. A `o.f` operation requires calling some procedure that is doing hashing and so forth. But by combining inline caches, structures, and speculative compilation we have landed on something where some `o.f` operations are nothing more than load-at-offset like they would have been in C++ or Java. But this assumes that the `o.f` operation was monomorphic. The rest of this section considers *minimorphism*, *polymorphism*, and *polyvariance*.

Minimorphism. Certain kinds of polymorphic accesses are easier to handle than others. Sometimes an access will see two or more structures but all of those structures have the property at the same offset. Other times an access will see multiple structures and those structures do not agree on the offset of the property. We say that an access is *minimorphic* if it sees more than one structure and all structures agree on the offset of the property.

Our inline caches handle all forms of polymorphism by generating a stub that switches on the structure. But in the DFG, minimorphic accesses are special because they still qualify for full inlining. Consider an access `o.f` that sees structures S1 and S2, and both agree that `f` is at offset 0. Then we would have:

```
CheckStructure(@o, S1, S2)
GetByOffset(@o, 0)
```

This minimorphic CheckStructure will OSR exit if @o has none of the listed structures. Our optimizations for CheckStructure generally work for both monomorphic and minimorphic variants. So, minimorphism usually doesn't hurt performance much compared to monomorphism.

Polymorphism. But what about some access sees different structures, and those structures have the property at different offsets? Consider an access to `o.f` that sees structures S1 = {f, g}, S2 = {f, g, h}, and S3 = {g, f}. This would be a minimorphic access if it was just S1 or S2, but S3 has `f` at a different offset. In this case, the FTL will convert this to:

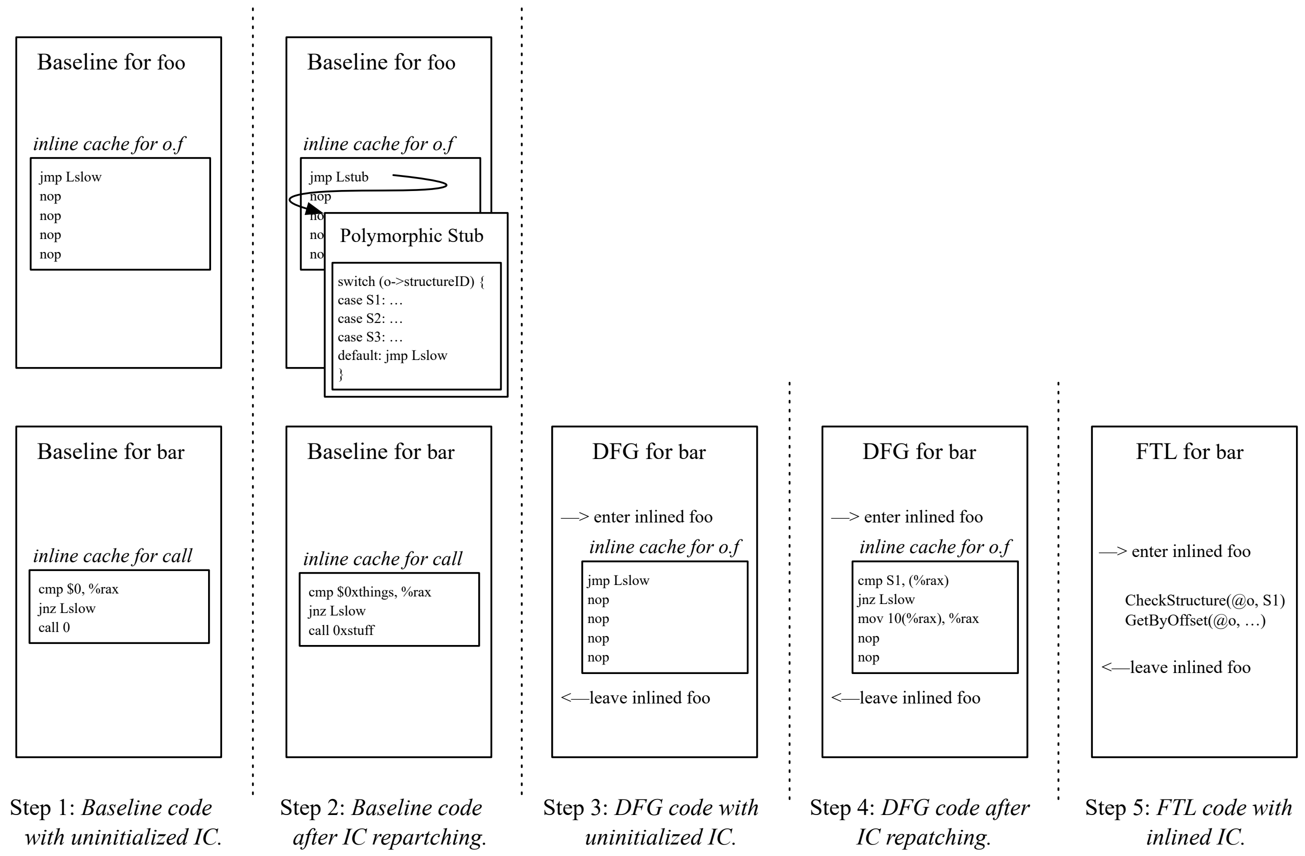
```
MultiGetByOffset(@o, [S1, S2] => 0, [S3] => 1)
```

in DFG IR and then lower it to something like:

```
if (o->structureID == S1 || o->structureID == S2)
    result = o->inlineStorage[0]
else
    result = o->inlineStorage[1]
```

in B3 IR. In fact, we would use B3's Switch since that's the canonical form for this code pattern in B3.

Note that we only do this optimization in the FTL. The reason is that we want polymorphic accesses to remain ICs in the DFG so that we can use them to collect refined profiling.



```
jmp Lslow
nop
nop
nop
nop
```

```
jmp Lstub
nop
```

```
switch (o->structureID) {
case S1: …
case S2: …
case S3: …
default: jmp Lslow
}
```

```
cmp $0, %rax
jnz Lslow
call 0
```

```
cmp $0xthings, %rax
jnz Lslow
call 0xstuff
```

```
jmp Lslow
nop
nop
nop
nop
```

```
cmp S1, (%rax)
jnz Lslow
mov 10(%rax), %rax
nop
nop
```

```
CheckStructure(@o, S1)
GetByOffset(@o, …)
```

Figure 23. Polyvariant inlining of an inline cache. The FTL can inline the inline cache in `foo`-inlined-into-`bar` after DFG compiles `bar` and uses an IC to collect polyvariant profiling about the `get_by_id`.

Polyvariance. Polyvariance is when an analysis is able to reason about a function differently depending on where it is called from. We achieve this by inlining in the DFG tier and keeping polymorphic ICs as ICs. Consider the following example. Function `foo` has an access to `o.f` that is polymorphic and sees structures S1 = {f, g}, S2 = {f, g, h}, and S3 = {g, f}:

```
function foo(o)
{
    // o can have structure S1, S2, or S3.
    return o.f;
}
```

This function is small, so it will be inlined anytime our profiling tells us that we are calling it (or may be calling it, since call inlining supports inlining polymorphic calls). Say that we have another function `bar` that always passes objects with structure S1 = {f, g} to `foo`:

```
function bar(p)
{
    // p.g always happens to have structure S1.
    return foo(p.g);
}
```

Figure 23 shows what happens. When the DFG compiles `bar` (step 3), it will inline `foo` based on the profiling of its `call` opcode (in step 2). But it will leave `foo`'s `get_by_id` as an IC because `foo`'s Baseline version told us that it's polymorphic (also step 2). But then, since the DFG's IC for `foo`'s `get_by_id` is the context of that call from `bar`, it only ever sees S1 (step 4). So, when the FTL compiles `bar` and inlines `foo`, it knows that this `get_by_id` can be inlined with a monomorphic structure check for just S1 (step 5).

Inline caches also support more exotic forms of property access, like loading from objects in the prototype chain, calling accessors, adding/replacing properties, and even deleting properties.

Inline caches, structures, and garbage collection. Inline caches results in objects that are allocated and referenced only for inline caching. Structures are the most notorious example of these kinds of objects. Structures are particularly problematic because they need strong references to both the object's prototype and its global object. In some cases, a structure will only be reachable from some inline cache, that inline cache will never run again (but we can't prove it), and there is a large global object only referenced by that structure. It can be difficult to determine if that means that the structure has to be deleted or not. If it should be deleted, then the inline cache must be reset. If any optimized code inlined that inline cache, then that code must be jettisoned and recompiled. Fortunately, our garbage collector allows us to describe this case precisely. Since the garbage collector runs to fixpoint, we simply add the constraint that the pointer from an inline cache to a structure only marks the structure if the structure's global object and prototype are already marked. Otherwise, the pointer behaves like a weak pointer. So, an inline cache will only be reset if the only way to reach the structure is through inline caches and the corresponding global object and prototype are dead. This is an example of how our garbage collector is engineered to make speculation easy.

To summarize, inline caching is an optimization employed by all of our tiers. In addition to making code run faster, inline caching is a high-precision profiling source that can tell us about the type cases that an operation saw. Combined with structures, inline caches allow us to turn dynamic property accesses into easy-to-optimize instructions.

Watchpoints

We allow inline caches and speculative compilers to set *watchpoints* on the heap. A watchpoint in JavaScriptCore is nothing more than a mechanism for registering for notification that something happened. Most watchpoints are engineered to trigger only the first time that something bad happens; after that, the watchpoint just remembers that the bad thing had ever happened. So, if an optimizing compiler wants to do something that is valid only if some bad thing never happened, and the bad thing has a watchpoint, the compiler just checks if the watchpoint is still valid (i.e. the bad thing hasn't happened yet) and then associates its generated code with the watchpoint (so the code will only get installed if the watchpoint is still valid when the code is done getting compiled, and will be jettisoned as soon as the watchpoint is fired). The runtime allows for setting watchpoints on a large number of activities. The following stick out:

- It's possible to set a watchpoint on structures to get a notification whenever any object switches from that structure to another one. This only works for structures whose objects have never transitioned to any other structure. This is called a *structure transition watchpoint*. It establishes a structure as a leaf in the structure transition tree.
- It's possible to set a watchpoint on properties in a structure to get a notification whenever the property is overwritten. Overwriting a property is easy to detect because the first time this happens, it usually involves repatching a `put_by_id` inline cache so that it's in the property replacement mode. This is called a *property replacement watchpoint*.
- It's possible to set a watchpoint on the mutability of global variables.

Putting these watchpoints together gives the speculative compiler the ability to constant-fold object properties that happen to be immutable. Let's consider a simple example:

```
Math.pow(42, 2)
```

Here, `Math` is a global property lookup. The base object is known to the compiler: it's the global object that the calling code belongs to. Then, `Math.pow` is a lookup of the `pow` propery on the Math object. It's extremely unlikely that the `Math` property of the global object or the `pow` property of the Math object had ever been overwritten. Both the global object and the Math object have structures that are unique to them (both because those structures have special magic since those are special objects and because those objects have what is usually a globally unique set of properties), which guarantees that they have leaf structures, so the structure transition watchpoint can be set. Therefore, except for pathological programs, the expression `Math.pow` is compiled to a constant by the speculative compiler. This makes lots of stuff fast:

- It's common to have named and scoped enumerations using objects and object properties, like `TypeScript.NodeType.Error` in the typescript compiler benchmark in JetStream 2. Watchpoints make those look like a constant to the speculative compiler.
- Method calls like `o.foo(things)` are usually turned just into a structure check on `o` and a direct call. Once the structure is checked, watchpoints establish that the object's prototype has a property called `foo` and that this property has some constant value.
- Inline caches use watchpoints to remove some checks in their generated stubs.
- The DFG can use watchpoints to remove redundant CheckStructures even when there is a side effect between them. If we set the structure transition watchpoint then we know that no effect can change the structure of any object that has this structure.
- Watchpoints are used for lots of miscellaneous corner cases of JavaScript, like having a bad time.

To summarize, watchpoints let inline caches and the speculative compilers fold certain parts of the heap's state to constants by getting a notification when things change.

Exit Flags

All of the profiling sources in our engine have a chance of getting things wrong. Profiling sources get things wrong because:

- The program may change behavior between when we collected the profiling and when we speculated on it.
- The profiling has some stochastic element and the program is getting unlucky, leading to wrong profiling.
- The profiling source has a logic bug that makes it not able to see that something happened.
- We neglected to implement a profiler for something and instead just speculated blind.

The first of these issues – behavior change over time – is inevitable and is sure to happen for some functions in any sufficiently large program. Big programs tend to experience phase changes, like some subroutine going from being called from one part of a larger library that uses one set of types, to being called from a different part with different types. Those things inevitably cause exits. The other three issues are all variants of the profiling being broken. We don't want our profiling to be broken, but we're only human. Recall that for speculation to have good EV, the probability of being right has to be about 1. So, it's not enough to rely on profiling that was written by imperfect lifeforms. Exit flags are a check on the rest of the profiling and are there to ensure that we get things right eventually for all programs.

In JavaScriptCore, every OSR exit is tagged with an exit kind. When a DFG or FTL function exits enough times to get jettisoned, we record all of the exit kinds that happened along with the bytecode locations that semantically caused the exits (for example if we do a type check for `add` at bytecode #63 but then hoist the check so that it ends up exiting to bytecode #45, then we will blame #63 not #45). Whenever the DFG or FTL decide whether to perform a kind of speculation, they are expected to check whether there is an exit flag for that speculation at the bytecode that we're compiling. Our exit flag checking discipline tends to be strictly better than our profiling discipline, and it's way easier to get right — every phase of the DFG has fast access to exit flags.

Here's an example of an actual OSR exit check in DFG:

```
speculationCheck(
    OutOfBounds, JSValueRegs(), 0,
    m_jit.branch32(
        MacroAssembler::AboveOrEqual,
        propertyReg,
        MacroAssembler::Address(storageReg, Butterfly::offsetOfPublicLength())));
```

Note that the first argument is `OutOfBounds`. That's an example exit kind. Here's another example, this time from the FTL:

```
speculate(NegativeZero, noValue(), nullptr, m_out.lessThan(left, m_out.int32Zero));
```

Again, the the first argument is the exit kind. This time it's `NegativeZero`. We have 26 exit kinds, most of which describe a type check condition (some are used for other uses of OSR, like exception handling).

We use the exit kinds by querying if an exit had happened at the bytecode location we are compiling when choosing whether to speculate. We typically use the presence of an exit flag as an excuse not to speculate at all for that bytecode. We effectively allow ourselves to overcompensate a bit. The exit flags are a check on the rest of the profiler. They are telling the compiler that the profiler had been wrong here before, and as such, shouldn't be trusted anymore for this code location.

Summary of Profiling

JavaScriptCore's profiling is designed to be cheap and useful. Our best profiling sources tend to either involve minimal instrumentation (like just setting a flag or storing a value to a known location) or be intertwined with optimizations (like inline caching). Our profilers gather lots of rich information and in some cases we even collect information redundantly. Our profiling is designed to help us avoid making speculative bets that turn out to be wrong even once.

Compilation and OSR

Now that we have covered bytecode, control, and profiling, we can get to the really fun part: how to build a great speculative optimizing compiler. We will discuss the OSR aspect of speculation in tandem with our descriptions of the two optimizing compilers.

This section is organized into three parts. First we give a quick and gentle introduction to DFG IR, the intermediate representation used by both the DFG and FTL tiers. Then we describe the DFG tier in detail, including how it handles OSR. Finally we describe how the FTL tier works.

DFG IR

The most important component of a powerful optimizing compiler is the IR. We want to have the best possible speculative optimizing compiler for JavaScript, so we have the following goals for our IR:

- The IR has to describe all of the parts of the program that are interesting to the optimizer. Like other high quality optimizing IRs, DFG IR has good support for talking about data flow, aliasing, effects, control flow, and debug information. Additionally, it's also good at talking about profiling data, speculation decisions, and OSR.
- The IR has to be mutable. Anything that is possible to express when first lowering a program to the IR should also be expressible during some later optimization. We prefer that decisions made during lowering to the IR can be be refined by optimizations later.

- The IR has to have some validation support. It's got to be possible to catch common mistakes in a validator instead of debugging generated code.
- The IR has to be purpose-built. If there exists an optimization whose most comprehensive implementation requires a change to the IR or one of its core data structures, then we need to be able to make that change without asking anyone for permission.

Note that IR mutability is closely tied to how much it describes and how easy it is to validate. Any optimization that tries to transform one piece of code into a different, better, piece of code needs to be able to determine if the new code is a valid replacement for the old code. Generally, the more information the IR carries and the easier it is to validate, the easier it is to write the analyses that guard optimizations.

Let's look at what the DFG IR looks like using a simple example:

```
function foo(a, b)
{
    return a + b;
}
```

This results in bytecode like:

```
[   0] enter
[   1] get_scope         loc3
[   3] mov               loc4, loc3
[   6] check_traps
[   7] add               loc6, arg1, arg2
[  12] ret               loc6
```

Note that only the last two lines (`add` and `ret`) are important. Let's look at the DFG IR that we get from lowering those two bytecode instructions:

```
  23:  GetLocal(Untyped:@1, arg1(B<Int32>/FlushedInt32), R:Stack(6), bc#7)
  24:  GetLocal(Untyped:@2, arg2(C<BoolInt32>/FlushedInt32), R:Stack(7), bc#7)
  25:  ArithAdd(Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7)
  26:  MovHint(Untyped:@25, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid)
  28:  Return(Untyped:@25, W:SideState, Exits, bc#12)
```

In this example, we've lowered the `add` opcode to four operations: two GetLocals to get the argument values from the stack (we load them lazily and this is the first operation that needs them), a speculative ArithAdd instruction, and a MovHint to tell the OSR part of the compiler about the ArithAdd. The `ret` opcode is just lowered to a Return.

In DFG jargon, the instructions are usually called *nodes*, but we use the terms *node*, *instruction*, and *operation* interchangeably. DFG nodes are simultaneously nodes in a data flow graph and instructions inside of a control flow graph, with semantics defined "as if" they executed in a particular order.

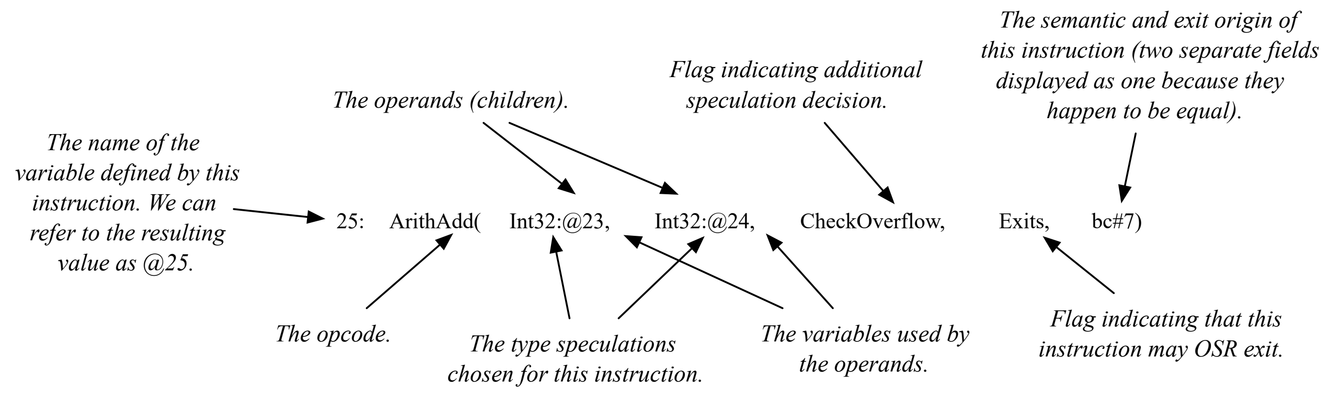

The semantic and exit origin of this instruction (two separate fields displayed as one because they happen to be equal).
Flag indicating additional speculation decision.
The operands (children).
The name of the variable defined by this instruction. We can refer to the resulting value as @25.
25: ArithAdd(Int32:@23, Int32:@24, CheckOverflow, Exits, bc#7)
The opcode.
The type speculations chosen for this instruction.
The variables used by the operands.
Flag indicating that this instruction may OSR exit.

Figure 24. Explanation of an example ArithAdd DFG instruction.

Let’s consider the ArithAdd in greater detail (Figure 24). This instruction is interesting because it’s exactly the sort of thing that the DFG is designed to optimize: it represents a JavaScript operation that is dynamic and impure (it may call functions) but here we have inferred it to be free of side effects using the `Int32:` type speculations. These indicate that that before doing anything else, this instruction will check that its inputs are Int32’s. Note that the type speculations of DFG instructions should be understood like function overloads. ArithAdd also allows for both operands to be double or other kinds of integer. It’s as if ArithAdd was a C++ function that had overloads that took a pair of integers, a pair of doubles, etc. It’s not possible to add any type speculation to any operand, since that may result in an instruction overload that isn’t supported.

Another interesting feature of this ArithAdd is that it knows exactly which bytecode instruction it originated from and where it will exit to. These are separate fields in the IR (the *semantic* and *forExit* origins) but when the are equal we dump them as one, `bc#7` in the case of this instruction.

Any DFG node that may exit will have the Exits flag. Note that we set this flag conservatively. For example, the Return in our example has it set not because Return exits but because we haven’t found a need to make the exit analysis any more precise for that instruction.

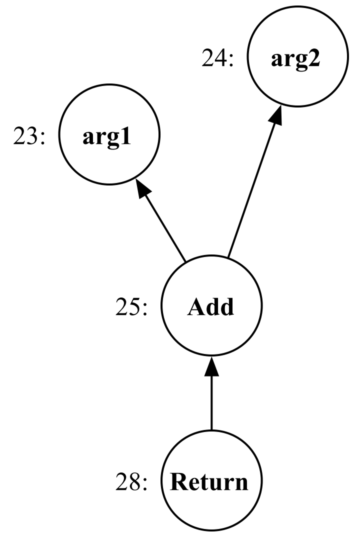


Figure 25. Example data flow graph.

DFG IR can be simultaneously understood as a sequence of operations that should be performed *as if* in the given order and as a data flow graph with backwards pointers. The data flow graph view of our running example is shown in Figure 25. This view is useful since lots of optimizations are concerned with asking questions like: *“what instructions produce the values consumed by this instruction?”* These data flow edges are the main way that values move around in DFG IR. Also, representing programs this way makes it natural to add SSA form, which we do in the FTL.

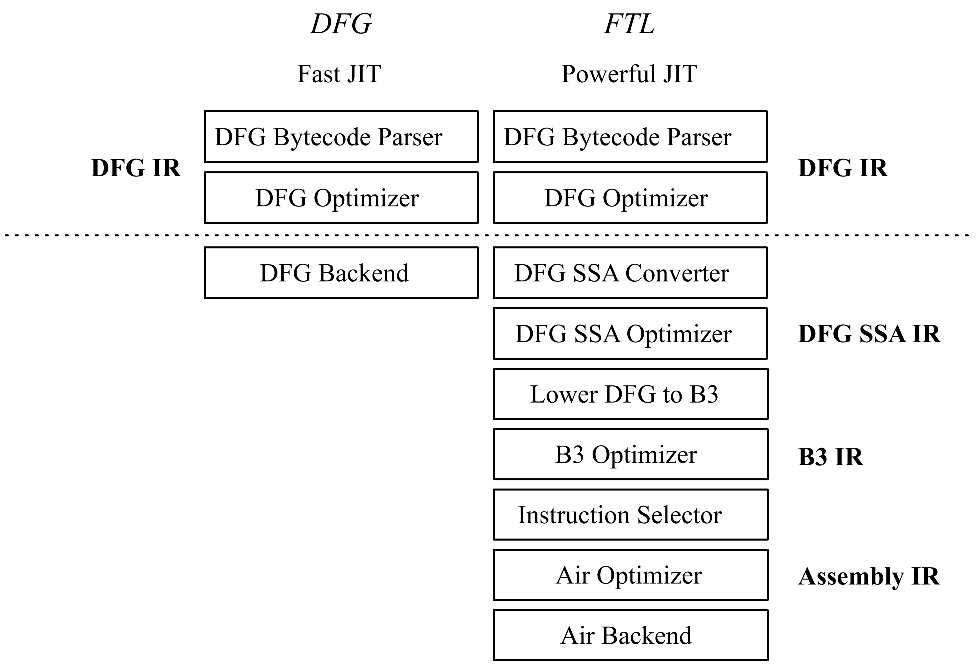


Figure 26. DFG and FTL compiler architecture. The pass pipeline depicted above the dotten line is shared between the DFG and FTL compilers. Everything below the dotted line is specialized for DFG or FTL.

DFG, in both non-SSA and SSA forms, forms the bulk of the DFG and FTL compilers. As shown in Figure 26, both JITs share the same frontend for parsing bytecode and doing some optimizations. The difference is what happens after the DFG optimizer. In the DFG tier, we emit machine code directly. In the FTL tier, we convert to DFG SSA IR (which is almost identical to DFG IR but uses SSA to represent data flow) and do more optimizations, and then lower through two additional optimizers (B3 and Assembly IR or Air). The remaining sections talk about the DFG and FTL compilers. The section on the DFG compiler covers the parts of DFG and FTL that are common.

DFG Compiler

The point of the DFG compiler is to remove lots of type checks quickly. *Fast compilation* is the DFG feature that differentiates it from the FTL. To get fast compilation, the DFG lacks SSA, can only do very limited code motion, and uses block-local versions of most optimizations (common subexpression elimination, register allocation, etc). The DFG has two focus areas where it does a great job despite compiling quickly: how it handles *OSR* and how it uses *static analysis*.

This section explains the DFG by going into these three concepts in greater detail:

- *OSR exit* as a first-class concept in the compiler.
- *Static analysis* as the main driver of optimization.
- *Fast compilation* so that we get the benefits of optimization as soon as possible.

OSR Exit

OSR is all about flattening control flow by making failing checks exit sideways. OSR is a difficult optimization to get right. It’s especially difficult to reason about at a conceptual level. This section tries to demystify OSR exit. We’re going to explain the DFG compiler’s approach to OSR, which includes both parts that are specific to the DFG tier and parts that are shared with the FTL. The FTL section explains extensions to this approach that we use to do more aggressive optimizations.

Our discussion proceeds as follows. First we use a high-level example to illustrate what OSR exit is all about. Then we describe what OSR exit means at the machine level, which will take us into the details of how optimizing compilers handle OSR. We will show a simple OSR exit IR idea based on *stackmaps* to give a sense of what we're trying to achieve and then we describe how DFG IR compresses stackmaps. Finally we talk about how OSR exit is integrated with watchpoints and invalidation.

High-level OSR example. To start to demystify DFG exit, let's think of it as if it was an optimization we were doing to a C program. Say we had written code like:

```
int foo(int* ptr)
{
    int w, x, y, z;
    w = ... // lots of stuff
    x = is_ok(ptr) ? *ptr : slow_path(ptr);
    y = ... // lots of stuff
    z = is_ok(ptr) ? *ptr : slow_path(ptr);
    return w + x + y + z;
}
```

Let's say we wanted to optimize out the second `is_ok` check. We could do that by duplicating all of the code after the first `is_ok` check, and having one copy statically assume that `is_ok` is true while another copy either assumes it's false or makes no assumptions. This might make the fast path look like:

```
int foo(int* ptr)
{
    int w, x, y, z;
    w = .. // lots of stuff
    if (!is_ok(ptr))
        return foo_base1(ptr, w);
    x = *ptr;
    y = ... // lots of stuff
    z = *ptr;
    return w + x + y + z;
}
```

Where `foo_base1` is the original `foo` function after the first `is_ok` check. It takes the live state at that point as an argument and looks like this:

```
int foo_base1(int* ptr, int w)
{
    int x, y, z;
    x = is_ok(ptr) ? *ptr : slow_path(ptr);
    y = ... // lots of stuff
    z = is_ok(ptr) ? *ptr : slow_path(ptr);
    return w + x + y + z;
}
```

What we've done here is OSR exit. We're optimizing control flow on the fast path (removing one `is_ok` check) by exiting (tail-calling `foo_base1`) if `!is_ok`. OSR exit requires:

- *Somewhere to exit*, like `foo_base1` in this case. It should be a thing that can complete execution of the current function without getting stuck on the same speculation.
- The *live state at exit*, like `ptr` and `w` in this case. Without that, the exit target can't pick up where we left off.

That's OSR exit at a high level. We're trying to allow an optimizing compiler to emit checks that exit out of the function on failure so that the compiler can assume that the same check won't be needed later.

OSR at the machine level. Now let's look at what OSR exit looks like at a lower level. Figure 27 shows an example of OSR at a particular bytecode index.

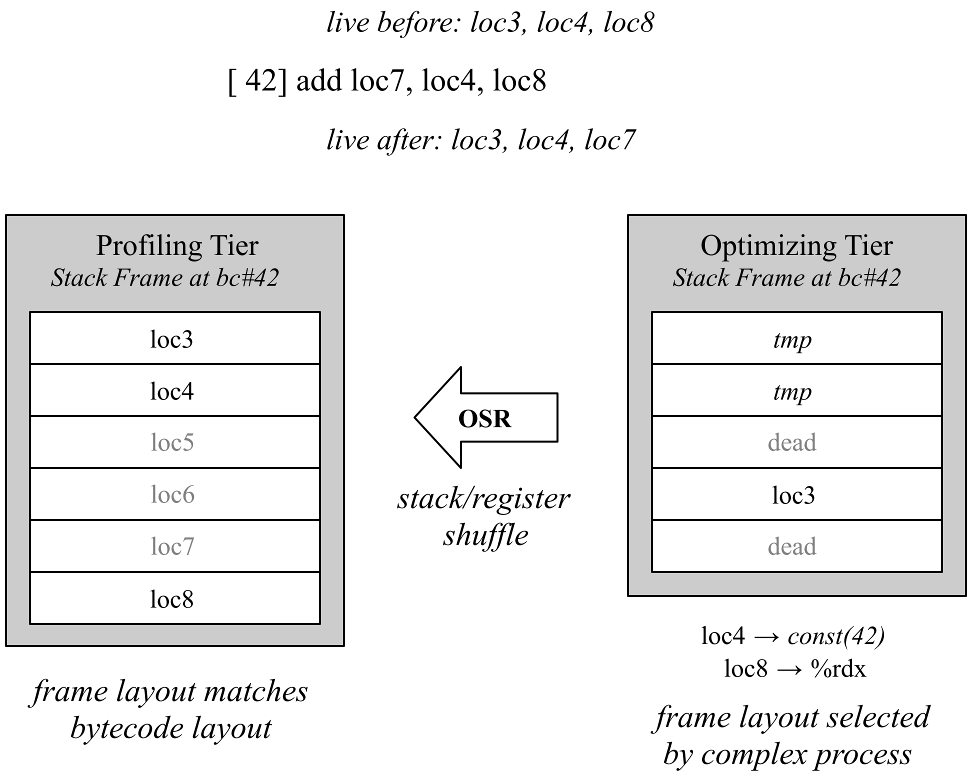


Figure 27. OSR exit at the machine level for an example bytecode instruction.

OSR is all about replacing the current stack frame and register state, which correspond to some bytecode index in the optimizing tier, with a different frame and register state, which correspond to the same point in the profiling tier. This is all about shuffling live data from one format to another and jumping to the right place.

Knowing where to jump to is easy: each DFG node (aka instruction or operation) has *forExit*, or just *exit*, origin that tells us which bytecode location to exit to. This may even be a bytecode stack in case of inlining.

The live data takes a bit more effort. We have to know what the set of live data is and what its format is in both the profiling and optimizing tiers. It turns out that knowing what the set of live data is and how to represent it for the profiling tiers is easy, but extracting that data from the optimizing tier is hard.

First let's consider what's live. The example in Figure 27 says that we're exiting at an `add` and it has `loc3`, `loc4`, and `loc8` live before. We can solve for what's live at any bytecode instruction by doing a liveness analysis. JavaScriptCore has an optimized bytecode liveness analysis for this purpose.

Note that the frame layout in the profiling tier is an orderly representation of the bytecode state. In particular, `locN` just means `framePointer - 8 * N` and `argN` just means `framePointer + FRAME_HEADER_SIZE + 8 * N`, where `FRAME_HEADER_SIZE` is usually 40. The only difference between frame layouts between functions in the profiling tier is the frame size, which is determined by a constant in each bytecode function. Given the frame pointer and the bytecode virtual register name, it's always possible to find out where on the stack the profiling tiers would store that variable. This makes it easy to figure out how to convert any bytecode live state to what the Baseline JIT or LLInt would expect.

The hard part is the optimizing tier's state. The optimizing compiler might:

- Allocate the stack in any order. Even if a variable is on the stack, it may be anywhere.

- Register-allocate a variable. In that case there may not be any location on the stack that contains the value of that variable.
- Constant-fold a variable. In that case there may not be any location on the stack or in the register file that contains the value of that variable.
- Represent a variable's value in some creative way. For example, your program might have had a statement like `x = y + z` but the compiler chose to never actually emit the add except lazily at points of use. This can easily happen because of pattern-matching instruction selection on x86 or ARM, where some instructions (like memory accesses) can do some adds for free as part of address computation. We do an even more aggressive version of this for object allocations: some program variable semantically points to an object, but because our compiler is smart, we never actually allocated any object and the object's fields may be register-allocated, constant-folded, or represented creatively.

We want to allow the optimizing compiler to do things like this, since we want OSR exit to be an enabler of optimization rather than an inhibitor. This turns out to be tricky: how do we let the optimizing compiler do all of the optimizations that it likes to do while still being able to tell us how to recover the bytecode state?

The trick to extracting the optimized-state-to-bytecode-state shuffle from the optimizing compiler is to leverage the original bytecode→IR conversion. The main difference between an SSA-like IR (like DFG IR) and bytecode is that it represents data flow relationships instead of variables. While bytecode says `add x, y, z`, DFG IR would have an Add node that points to the nodes that produced `y` and `z` (like in Figure 25). The conversion from bytecode to DFG IR looks like this pseudocode:

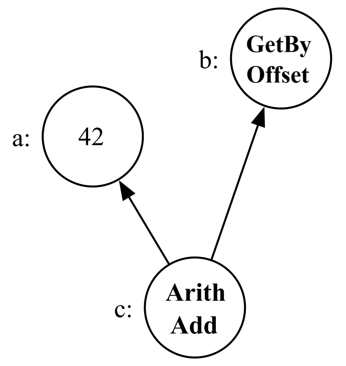
```
case op_add: {
    VirtualRegister result = instruction->result();
    VirtualRegister left   = instruction->left();
    VirtualRegister right  = instruction->right();

    stackMap[result] = createAdd(
        stackMap[left], stackMap[right]);
    break;
}
```

This uses a standard technique for converting variable-based IRs to data-flow-based IRs: the converter maintains a mapping from variables in the source IR to data flow nodes in the target IR. We're going to call this the `stackMap` for now. Each bytecode instruction is handled by modeling the bytecode's data flow: we load the left and right operands from the `stackMap`, which gives us the DFG nodes for those locals' values. Then we create an ArithAdd node and store it into the result local in the `stackMap` to model the fact that the bytecode wanted to store the result to that local. Figure 28 shows the before-and-after of running this on the `add` bytecode in our running example.

live before: loc3, loc4, loc8

[42] add loc7, loc4, loc8

live after: loc3, loc4, loc7



stackMap *before bc#42*

Virtual Register	Value (name: opcode)
loc3	s: GetScope
loc4	a: JSConstant(42)
loc5	*dead*
loc6	*dead*
loc7	*dead*
loc8	b: GetByOffset

stackMap *after bc#42*

Virtual Register	Value (name: opcode)
loc3	s: GetScope
loc4	a: JSConstant(42)
loc5	*dead*
loc6	*dead*
loc7	c: ArithAdd
loc8	*dead*

Figure 28. Example of `stackMap` before and after running the SSA conversion on `add` at bc#42 along with an illustration of the data flow graph around the resulting ArithAdd.

The `stackMap`, pruned to bytecode liveness as we are doing in these examples, represents the set of live state that would be needed to be recovered at any point in bytecode execution. It tells us, for each live bytecode local, what DFG node to use to recover the value of that local. A simple way to support OSR would be to give each DFG node that could possibly exit a data flow edge to each node in the liveness-pruned `stackMap`.

This isn't what the DFG actually does; DFG nodes do not have data flow edges for the stackmap. Doing literally that would be too costly in terms of memory usage since basically every DFG node may exit and stackmaps have O(live state) entries. The DFG's actual approach is based on delta-compression of stackmaps. But it's worth considering exactly how this uncompressed stackmap approach would work because it forms part of the FTL's strategy and it gives a good mental model for understanding the DFG's more sophisticated approach. So, we will spend some time describing the DFG IR as if it really did have stackmaps. Then we will show how the stackmap is expressed using delta compression.

OSR exit with uncompressed stackmaps. Imagine that DFG nodes really had extra operands for the stackmap. Then we would have an ArithAdd like the following, assuming that bc#42 is the exit origin and that `loc3`, `loc4`, and `loc8` are live, as they are in Figures 27 and 28:

```
c: ArithAdd(@a, @b, loc3->@s, loc4->@a, loc8->@b, bc#42)
```

In this kind of IR, we'd let the first two operands of ArithAdd behave the expected way (they are the actual operands to the add), and we'd treat all of the other operands as the stackmap. The exit origin, bc#42, is a control flow label. Together, this tells the ArithAdd where to exit (bc#42) and the stackmap (@s, @a, and @b). The compiler treats the ArithAdd, and the stackmap operands, as if the ArithAdd had a *side exit* from the function the compiler was compiling.

One way to think about it is in terms of C pseudocode. We are saying that the semantics of ArithAdd and any other instruction that may exit are as if they did the following before any of their effects:

```
if (some conditions)
    return OSRExit(bc#42, {loc3: @s, loc4: @a, loc8: @b});
```

Where the `return` statement is an early return from the compiled function. So, this terminates the execution of the compiled function by tail-calling (jumping to) the `OSRExit`. That operation will transfer control to bc#42 and pass it the given stackmap.

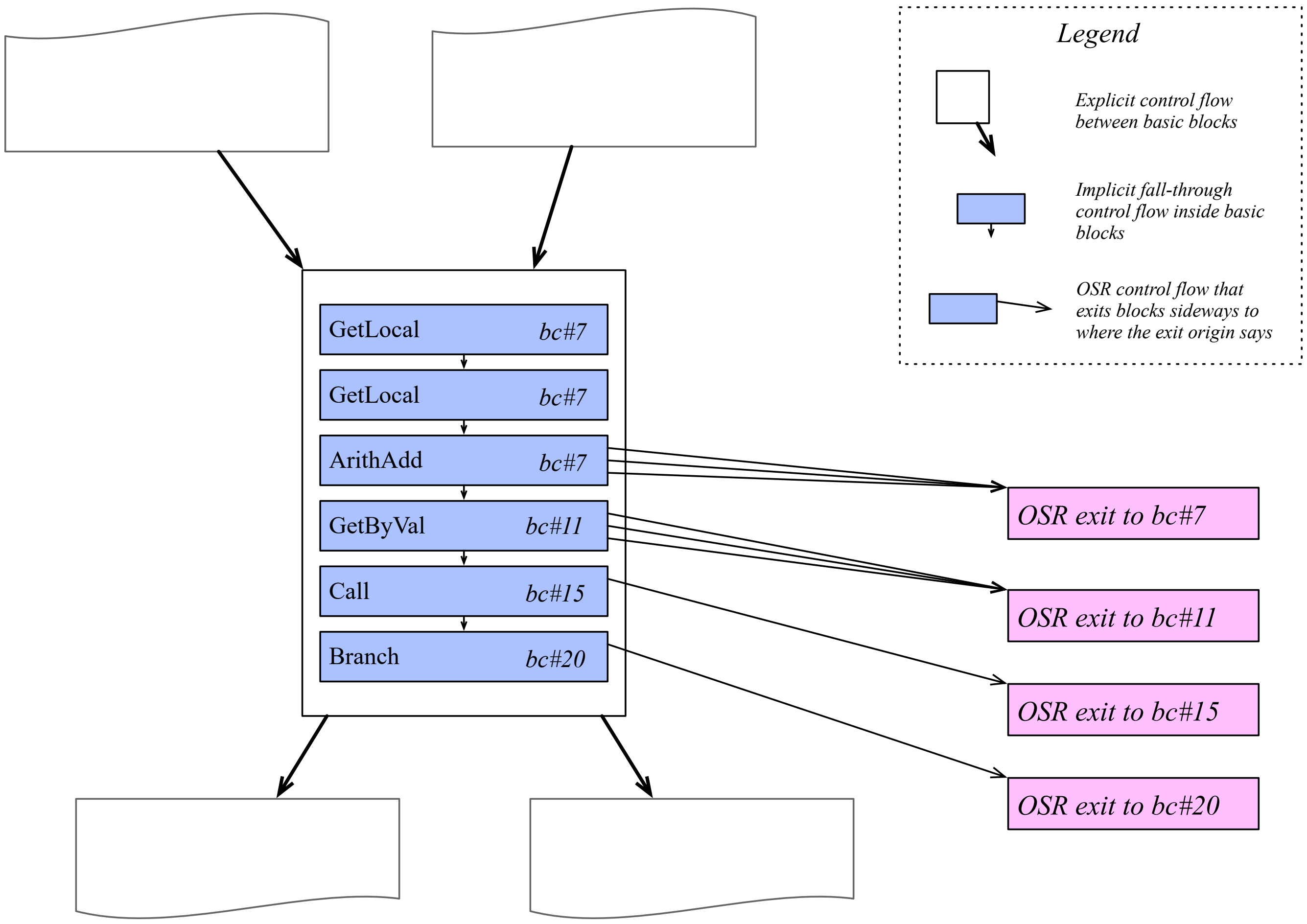

Legend
Explicit control flow between basic blocks
Implicit fall-through control flow inside basic blocks
OSR control flow that exits blocks sideways to where the exit origin says
GetLocal bc#7
GetLocal bc#7
ArithAdd bc#7
GetByVal bc#11
Call bc#15
Branch bc#20
OSR exit to bc#7
OSR exit to bc#11
OSR exit to bc#15
OSR exit to bc#20

Figure 29. Example of control flow in a compiler with OSR exit. OSR exit means having an additional implicit set of control flow edges that come out of almost every instruction and represent a side exit from the control flow graph.

This is easy to model in a compiler. We don't allocate any kind of control flow constructs to represent the condition check and side exit but we assume it to exist implicitly when analyzing ArithAdd or any other node that may exit. Note that in JavaScript, basically every instruction is going to possibly exit, and JavaScriptCore's may exit analysis defaults to `true` for most operations. Figure 29 illustrates what this looks like. We are going to have three kinds of control flow edges instead of the usual two:

1. The normal control flow edges between basic blocks. This is what you normally think of as "control flow". These edges are explicitly represented in the IR, as in, there is an actual data structure (usually vector of successors and vector of predecessors) that each block uses to tell what control flow edges it participates in.
2. The implicit fall-through control flow for instructions within blocks. This is standard for compilers with basic blocks.
3. A new kind of control flow edge due to OSR, which goes from instructions in blocks to OSR exit landing sites. This means changing the definition of basic blocks slightly. Normally the only successors of basic blocks are the ones in the control flow graph. Our basic blocks have a bunch of OSR exit successors as well. Those successors don't exist in the control flow graph, but we have names for them thanks to the exit origins found in the

exiting instructions. The edges to those exit origins exit out of the middle of blocks, so they may terminate the execution of blocks before the block terminal.

The OSR landing site is understood by the compiler as having the following behaviors:

- It ends execution of this function in DFG IR. This is key, since it means that there is no merge point in our control flow graph that has to consider the consequences of exit.
- It possibly reads and writes the whole world. The DFG has to care about the reads (since they may observe whatever happened just before the exit) but not the writes (since they affect execution after execution exited DFG).
- It reads some set of values, namely those passed as the stackmap.

This understanding is abstract, so the compiler will just assume the worst case (after exit every location in memory is read and written and all of the bits in all of the values in the stackmap are etched into stone).

This approach is great because it allows precise reconstruction of baseline state when compiling OSR exit and it mostly doesn't inhibit optimization because it "only" involves adding a new kind of implicit control flow edge to the control flow graph.

This approach allows for simple reconstruction of state at exit because the backend that compiles the DFG nodes would have treated the stackmap data flow edges (things like `loc3->@s` in our example) the same way it would have treated all other edges. So, at the ArithAdd, the backend would know which registers, stack slots, or constant values to use to materialize the stackmap values. It would know how to do this for the same reason that it would know how to materialize the two actual add operands.

If we survey the most common optimizations that we want the compiler to do, we find that only one major optimization is severely inhibited by this approach to OSR exit. Let's first review the optimizations this doesn't break. It's still possible to perform common subexpression elimination (CSE) on ArithAdd. It's still possible to hoist it out of loops, though if we do that then we have to edit the exit metadata (the exit destination and stackmap will have to be overwritten to be whatever they are at the loop pre-header). It's still possible to model the ArithAdd to be pure in lots of the ways that matter, like that if there are two loads, one before and one after the ArithAdd, then we can assume them to be redundant. The ArithAdd could only cause effects on the exit path, in which case the second load doesn't matter. It's still possible to eliminate the ArithAdd if it's unreachable.

The only thing we cannot easily do is what compilers call dead code elimination, i.e. the elimination of instructions if their results are not used. Note that the compiler terminology is confusing here. Outside the compiler field we use the term *dead code* to mean something that compilers call *unreachable code*. Code is unreachable if control flow doesn't reach it and so it doesn't execute. Outside the compiler field, we would say that such code is *dead*. It's important that compilers be able to eliminate unreachable code. Happily, our approach to OSR has no impact on unreachable code elimination. What compilers call *dead code* is code that is reached by control flow (so *live* in the not-compiler sense) but that produces a result that no subsequent code uses. Here's an example of dead code in the compiler sense:

```
int tmp = a + b;
// nobody uses tmp.
```

Dead code elimination (DCE) is the part of a compiler that removes this kind of code. Dead code elimination doesn't quite work for the ArithAdd because:

- ArithAdd's speculation checks must be assumed live even if the result of the add is unused. We may do some optimization to a later check because we find that it is subsumed by checks done by this ArithAdd. That's a pretty fundamental optimization that we do for OSR checks and it's the reason why OSR ultimately flattens control flow. But we don't bother recording whenever this ArithAdd's check is used to unlock a later optimization, so we have to assume that some later operation is already depending on the ArithAdd doing all of its checks. This means that: say that the result of some operation A is used by a dead operation B. B will still have to do whatever checks it was doing on its inputs, which will keep A alive even though B is dead. This is particularly devastating for ArithAdd, since ArithAdd usually does an overflow check. You have to do the add to check overflow. So, ArithAdd is never really dead. Consider the alternative: if we did not considider the ArithAdd's overflow check's effect on abstract state, then we wouldn't be able to do our range analysis, which uses the information inferred from overflow checks to remove array bounds checks and vice versa.

- The ArithAdd is almost sure to end up in the stackmap of some later operation, as is basically every node in the DFG program, unless the node represents something that was dead in bytecode. Being dead in bytecode is particularly unlikely because in bytecode we must assume that everything is polymorphic and possibly effectful. Then the `add` is really not dead: it might be a loop with function calls, after all.

The DFG and FTL still do DCE, but it's hard and usually only worth the effort for the most expensive constructs. We support decaying an operation just to its checks, for those rare cases where we can prove that the result is not used. We also support sinking to OSR, where an operation is replaced by a *phantom* version of itself that exists only to tell OSR how to perform the operation for us. We mainly use this complex feature for eliminating object allocations.

To summarize the effect on optimizations: we can still do most of the optimizations. The optimization most severely impacted is DCE, but even there, we have found ways to make it work for the most important cases.

The only real downside of this simple approach is repetition: almost every DFG operation may exit and the state at exit may easily have tens or hundreds of variables, especially if we have done significant inlining. Storing the stackmap in each DFG node would create a case of O(n^2) explosion in memory usage and processing time within the compiler. Note that the fact that this explosion happens is somewhat of a JavaScript-specific problem, since JavaScript is unusual in the sheer number of speculations we have to make per operation (even simple ones like `add` or `get_by_id`). If the speculations were something we did seldom, like in Java where they are mostly used for virtual calls, then the simple approach would be fine.

Stackmap compression in DFG IR. Our solution to the size explosion of repeated stackmaps is to use a delta encoding. The stackmaps don't change much. In our running example, the `add` just kills `loc8` and defines `loc7`. The kill can be discovered by analyzing bytecode, so there's no need to record it. All we have to record about this operation is that it defines `loc7` to be the ArithAdd node.

We use an operation called MovHint as our delta encoding. It tells which bytecode variable is defined by which DFG node. For example, let's look at the MovHint we would emit for the `add` in Figure 28:

```
c: ArithAdd(@a, @b, bc#42)
   MovHint(@c, loc7, bc#42)
```

We need to put some care into how we represent MovHints so that they are easy to preserve and modify. Our approach is two-fold:

- We treat MovHint as a store effect.
- We explicitly label the points in the IR where we expect it to be valid to exit based on the state constructed out of the MovHint deltas.

Let's first look at how we use the idea of store effects to teach the compiler about MovHint. Imagine a hypothetical DFG IR interpreter and how it would do OSR exit. They key idea is that in that interpreter, the state of the DFG program comprises not just the mapping from DFG nodes to their values, but also an OSR exit state buffer containing values indexed by bytecode variable name. That OSR exit state buffer contains exactly the stack frame that the profiling tiers would use. MovHint's interpreter semantics are to store the value of its operand into some slot in the OSR exit state buffer. This way, the DFG interpreter is able to always maintain an up-to-date bytecode stack frame in tandem with the optimized representation of program state. Although no such interpreter exists, we make sure that the way we compile MovHint produces something with semantics consistent with what this interpreter would have done.

MovHint is not compiled to a store. But any phase operating on MovHints or encountering MovHints just needs to understand it as a store to some abstract location. The fact that it's a store means that it's not dead code. The fact that it's a store means that it may need to be ordered with other stores or loads. Lots of desirable properties we need for soundly preserving MovHints across compiler optimizations fall out naturally from the fact that we tell all the phases that it's just a store.

The compiler emits zero code for MovHint. Instead, we use a *reaching defs* analysis of MovHints combined with a bytecode liveness analysis to rebuild the stackmaps that we would have had if each node carried a stackmap. We perform this analysis in the backend and as part of any optimization that needs to know what OSR is doing. In the DFG tier, the reaching defs analysis happens lazily (when the OSR exit actually occurs — so could be long after the DFG compiled the code), which ensures that the DFG never experiences the $O(n^2)$ blow-up of stackmaps. OSR exit analysis is not magical: in the "it's just a store" model of MovHint, this analysis reduces to load elimination.

DFG IR's approach to OSR means that OSR exit is possible at some points in DFG IR and not at others. Consider some examples:

- A bytecode instruction may define multiple bytecode variables. When lowered to DFG IR, we would have two or more MovHints. It's not possible to have an exit between those MovHints, since the OSR exit state is only partly updated at that point.
- It's not possible to exit after a DFG operation that does an observable effect (like storing to a JS object property) but before its corresponding MovHint. If we exit to the current exit origin, we'll execute the effect again (which is wrong), but if we exit to the next exit origin, we'll neglect to store the result into the right bytecode variable.

We need to make it easy for DFG transformations to know if it's legal to insert operations that may exit at any point in the code. For example, we may want to write instrumentation that adds a check before every use of @x. If that use is a MovHint, then we need to know that it may not be OK to add that check right before that MovHint. Our approach to this is based on the observation that the lowering of a bytecode instruction produces two phases of execution in DFG IR of that instruction:

- The speculation phase: at the start of execution of a bytecode, it's both necessary and possible to speculate. It's necessary to speculate since those speculations guard the optimizations that we do in the subsequent DFG nodes for that bytecode instruction. It's possible to speculate because we haven't done any of the instruction's effects, so we can safely exit to the start of that bytecode instruction.
- The effects phase: as soon as we perform any effect, we are no longer able to do any more speculations. That effect could be an actual effect (like storing to a property or making a call) or an OSR effect (like MovHint).

To help validate this, all nodes in DFG IR have an exitOK flag that they use to record whether they think that they are in the speculative phase (exitOK is true) or if they think that they might be in the effects phase (exitOK is false). It's fine to say that exitOK is false if we're not sure, but to say exitOK is true, we have to be completely sure. The IR validation checks that exitOK must become false after operations that do effects, that it becomes true again only at prescribed points (like a change in exit origin suggesting that we've ended the effects phase of one instruction and begun the speculation phase of the next one), and that no node that may exit has exitOK set to false. This validator helps prevent errors, like when dealing with bytecode operations that can be lowered to multiple effectful DFG nodes. One example is when `put_by_id` (i.e. something like `o.f = v`) is inferred to be a transition (the property `f` doesn't exist on `o` so we need to add it), which results in two effects:

- Storing a value `v` into the memory location for property `o.f`.
- Changing `o`'s structure to indicate that it now has an `f`.

The DFG IR for this will look something like:

```
CheckStructure(@o, S1)
PutByOffset(@o, @v, f)
PutStructure(@o, S2, ExitInvalid)
```

Note that PutStructure will be flagged with ExitInvalid, which is the way we say that exitOK is false in IR dumps. Failing to set exitOK to false for PutStructure would cause a validation error since PutByOffset (right before it) is an effect. This prevents us from making mistakes like replacing all uses of @o with some operation that could speculate, like:

```
a: FooBar(@o, Exits)
   CheckStructure(@a, S1)
b: FooBar(@o, Exits)
   PutByOffset(@b, @v, f)
c: FooBar(@o, Exits)
   PutStructure(@c, S2, ExitInvalid)
```

In this example, we've used some new FooBar operation, which may exit, as a filter on @o. It may seem absurd to instrument code this way, but it is a goal of DFG IR to:

- Allow replacing uses of nodes with uses of other nodes that produce an equivalent value.
  Let's assume that FooBar is an identity that also does some checks that may exit.
- Allow inserting new nodes anywhere.

Therefore, the only bug here is that @c is right after the PutByOffset. The validator will complain that it is not marked ExitInvalid. It should be marked ExitInvalid because the previous node (PutByOffset) has an effect. But if you add ExitInvalid to @c, then the validator will complain that a node may exit with ExitInvalid. Any phase that tries to insert such a FooBar would have all the API it needs to realize that it will run into these failures. For example, it could ask the node that it's inserting itself in front of (the PutStructure) whether it has ExitInvalid. Since it is ExitInvalid, we could do either of these things instead of inserting @c just before the PutStructure:

1. We could use some other node that does almost what FooBar does but without the exit.
2. We could insert @c earlier, so it can still exit.

Let's look at what the second option would look like:

```
a: FooBar(@o, Exits)
   CheckStructure(@a, S1)
b: FooBar(@o, Exits)
c: FooBar(@o, Exits)
   PutByOffset(@b, @v, f)
   PutStructure(@c, S2, ExitInvalid)
```

Usually this is all it takes to deal with regions of code with !exitOK.

Note that in cases where something like FooBar absolutely needs to do a check after an effect, DFG IR does support exiting into the middle of a bytecode instruction. In some cases, we have no choice but to use that feature. This involves introducing extra non-bytecode state that can be passed down OSR exit, issuing OSR exit state updates before/after effects, using an exit origin that indicates that we're exiting to some checkpoint in the middle of a bytecode instruction's execution, and implementing a way to execute a bytecode starting at a checkpoint during OSR exit. It's definitely possible, but not the sort of thing we want to have to do every time that some DFG node needs to do an effect. For this reason, canonical DFG IR use implies having !exitOK phases (aka effect phases) during some bytecode instructions' execution.

Watchpoints and Invalidation. So far we have considered OSR exit for checks that the compiler emits. But the DFG compiler is also allowed to speculate by setting watchpoints in the JavaScript heap. If it finds something desirable — like that `Math.sqrt` points to the sqrt intrinsic function — it can often incorporate it into optimization without emitting checks. All that is needed is for the compiler to set a watchpoint on what it wants to prove (that the `Math` and `sqrt` won't change). When the watchpoint fires, we want to invalidate the compiled code. That means making it so that the code never runs again:

- no new calls to that function go to the optimized version and
- all returns into that optimized function are redirected to go to baseline code instead.

Ensuring that new calls avoid optimized code is easy: we just patch all calls to the function to call the profiled code (Baseline, if available, or LLInt) instead. Handling returns is the interesting part.

One approach to handling invalidation is to walk the stack to find all returns to the invalidated code, and repoint those returns to an OSR exit. This would be troublesome for us due to our use of effects phases: it's possible for multiple effects to happen in a row in a phase of DFG IR execution where it is not possible to exit. So, the DFG approach to invalidation involves letting the remaining effects of the current bytecode instruction finish executing in optimized code and then triggering an OSR exit right before the start of the next bytecode instruction.

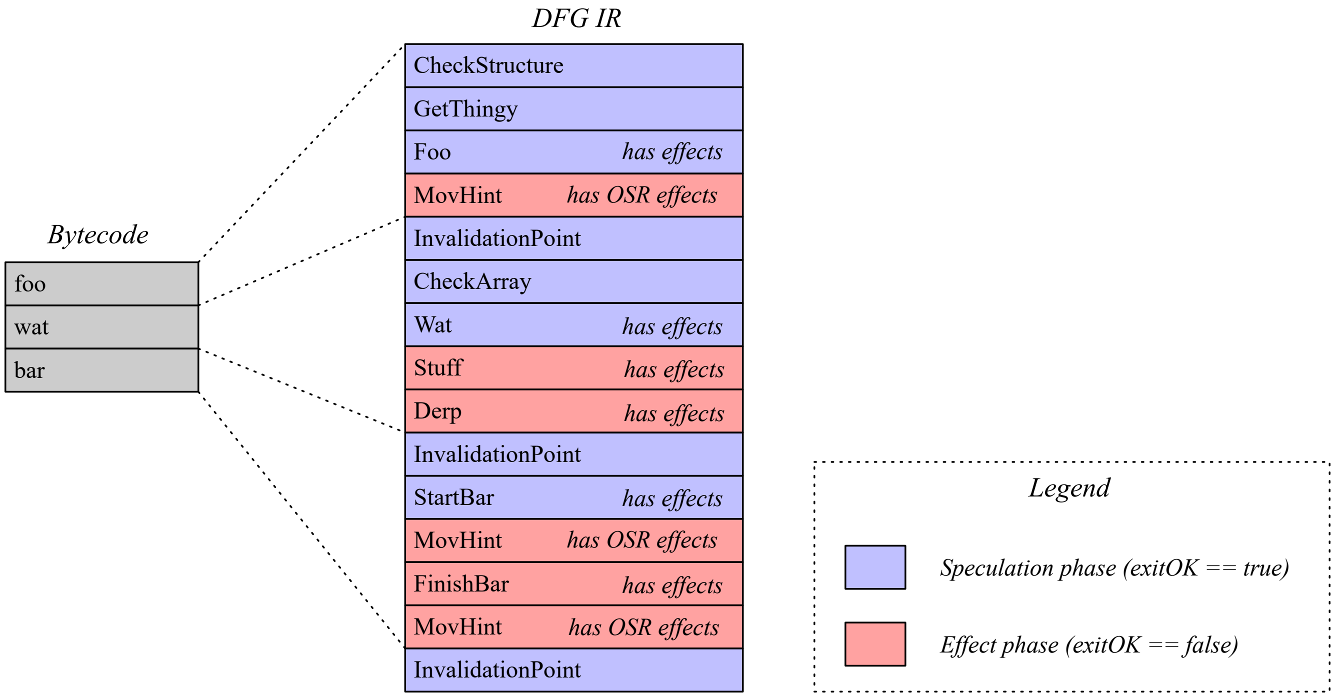


Figure 30. How OSR exit and invalidation might work for hypothetical bytecodes.

Invalidation in DFG IR is enabled by the InvalidationPoint instruction, which is automatically inserted by the DFG frontend at the start of every exit origin that is preceded by effects that could cause a watchpoint to fire. InvalidationPoint is modeled as if it was a conditional OSR exit, and is given an OSR exit jump label as if there was a branch to link it to. But, InvalidationPoint emits no code. Instead, it records the location in the machine code where the InvalidationPoint would have been emitted. When a function is invalidated, all of those labels are overwritten with unconditional jumps to the OSR exit.

Figure 30 shows how OSR exit concepts like speculation and effect phases combine with InvalidationPoint for three hypothetical bytecode instructions. We make up intentionally absurd instructions because we want to show the range of possibilities. Let's consider `wat` in detail. The first DFG IR node for `wat` is an InvalidationPoint, automatically inserted because the previous bytecode (`foo`) had an effect. Then `wat` does a CheckArray, which may exit but has no effects. So, the next DFG node, Wat, is still in the speculation phase. Wat is in a sort of perfect position in DFG IR: it is allowed to perform speculations and effects. It can perform speculations because no previous node for `wat`'s exit origin has performed effects. It can also perform effects, but then the nodes after it (Stuff and Derp) cannot speculate anymore. But, they can perform more effects. Since `wat` has effects, an InvalidationPoint is immediately inserted at the start of the next bytecode (`bar`). Note that in this example, Foo, Wat, and StartBar are all in the perfect position (they can exit and have effects). Since Stuff, Derp, and FinishBar are in the effects region, the compiler will assert if they try to speculate.

Note that InvalidationPoint makes code layout tricky. On x86, the unconditional jump used by invalidation is five bytes. So, we must ensure that there are no other jump labels in the five bytes after an invalidation label. Otherwise, it would be possible for invalidation to cause one of those labels to point into the middle of a 5-byte invalidation jump. We solve this by adding `nop` padding to create at least a 5-byte gap between a label used for invalidation and any other kind of label.

To summarize, DFG IR has extensive support for OSR exit. We have a compact delta encoding of changes to OSR exit state. Exit destinations are encoded as an exit origin field in every DFG node. OSR exit due to invalidation is handled by automatic InvalidationPoint insertion.

Static Analysis

The DFG uses lots of static analysis to complement how it does speculation. This section covers three static analyses in the DFG that have particularly high impact:

- We use *prediction propagation* to fill in predicted types for all values based on value profiling of some values. This helps us figure out where to speculate on type.
- We use the *abstract interpreter* (or just *AI* for short in JavaScriptCore jargon) to find redundant OSR speculations. This helps us emit fewer OSR checks. Both the DFG and FTL include multiple optimization passes in their pipelines that can find and remove redundant checks but the abstract interpreter is the most powerful one. The abstract interpreter is the DFG tier's primary optimization and it is reused with small enhancements in the FTL.
- We use *clobberize* to get aliasing information about DFG operations. Given a DFG instruction, clobberize can describe the aliasing properties. In almost all cases that description is O(1) in time and space. That description implicitly describes a rich dependency graph.

Both the prediction propagator and the abstract interpreter work by forward-propagating type infromation. They're both built on the principles of abstract interpretation. It's useful to understand at least some of that theory, so let's do a tiny review. Abstract interpreters are like normal interpreters, except that they describe program state abstractly rather than considering exact values. A classic example due to Kildall involves just remembering which variables have known constant values and forgetting any variable that may have more than one value. Abstract interpreters are run to *fixpoint*: we keep executing every instruction until we no longer observe any changes. We can execute forward (like

Kildall) or backward (like liveness analysis). We can either have sets that shrink as we learn new things (like Kildall, where variables get removed if we learn that they may have more than one value) or we can have sets that grow (like liveness analysis, where we keep adding variables to the live set).

Now let's go into more details about the two abstract interpreters and the alias analysis.

Prediction propagation. The prediction propagator's abstract state comprises variable to speculated type (Figure 13) mappings. The speculated type is a set of fundamental types. The sets tell which types a value is predicted to have. The prediction propagator is not flow sensitive; it has one copy of the abstract state for all program statements. So, each execution of a statement considers the whole set of input types (even from program statements that can't reach us) and joins the result with the speculated type of the result variable. Note that the input to the prediction propagator is a data flow IR, so multiple assignments to the same variable aren't necessarily joined.

The prediction propagator doesn't have to be sound. The worst case outcome of the prediction propagator being wrong is that we either:

- do speculations that are too strong, and so we exit too much and then recompile.
- do speculations that are too weak, so we run slower than we could forever.

Note that the second of those outcomes is generally worse. Recompiling and then speculating less at least means that the program eventually runs with the optimal set of speculations. Speculating too weakly and never recompiling means that we never get to optimal. Therefore, the prediction propagator is engineered to sometimes be unsound instead of conservative, since unsoundness can be less harmful.

The abstract interpreter. The DFG AI is the DFG tier's most significant optimization. While there are many abstract interpreters throughout JavaScriptCore, this one is the biggest in terms of total code and the number of clients — hence to us it is *the* abstract interpreter.

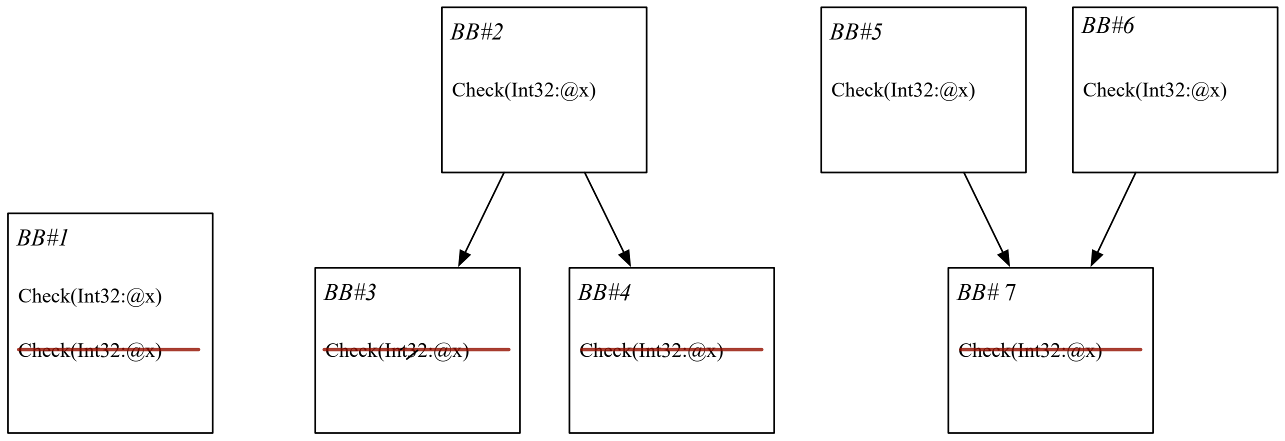
The DFG AI's abstract state comprises variable to abstract value mappings where each abstract value represents a set of possible JSValues that the variable could have. Those sets describe what type information we have proved from past checks. We join abstract states at control flow merge points. The solution after the fixpoint is a minimal solution (smallest possible sets that have a fixpoint). The DFG AI is flow-sensitive: it maintains a separate abstract state per instruction boundary. AI looks at the whole control flow graph at once but does not look outside the currently compiled function and whatever we inlined into it. AI is also sparse conditional.

The DFG abstract value representation has four sub-values:

- Whether the value is known to be a constant, and if so, what that constant is.
- The set of possible types (i.e. a SpeculatedType bitmap, shown in Figure 13).
- The set of possible indexing types (also known as array modes) that the object pointed to by this value can have.
- The set of possible structures that the object pointed to by this value can have. This set has special infinite set powers.

The last two sub-values can be mutated by effects. DFG AI assumes that all objects have escaped, so if an effect happens that can change indexing types and structures, then we have to *clobber* those parts of all live abstract values.

We interpret the four sub-values as follows: the abstract value represents the set of JSValues that reside in the intersection of the four sub-value sets. This means that when interpreting abstract values, we have the option of just looking at whichever sub-value is interesting to us. For example, an optimization that removes structure checks only needs to look at the structure set field.



(a) Eliminating checks in the same block

(b) Eliminating checks in different blocks

(c) Eliminating checks by merging information from multiple predecessors

Figure 31. Examples of check elimination with abstract interpretation.

The DFG AI gives us constant and type propagation simultaneously. The type propagation is used to remove checks, simplify checks, and replace dynamic operations with faster versions.

Figure 31 shows examples of checks that the DFG AI lets us remove. Note that in addition to eliminating obvious same-basic-block check redundancies (Figure 31(a)), AI lets us remove redundancies that span multiple blocks (like Figure 31(b) and (c)). For example, in Figure 31(c), the AI is able to prove that @x is an Int32 at the top of basic block #7 because it merges the Int32 states of @x from BB#5 and #6. Check elimination is usually performed by mutating the IR so that later phases know which checks are really necessary without having to ask the AI.

The DFG AI has many clients, including the DFG backend and the FTL-to-B3 lowering. Being an AI client means having access to its estimate of the set of JSValues that any variable or DFG node can have at any program point. The backends use this to simplify checks that were not removed. For example, the backend may see an Object-or-Undefined, ask AI about it, and find that AI already proved that that we must have either an object or a string. The backend will be able to combine those two pieces of information to only emit an is-object check and ignore the possibility of the value being undefined.

Type propagation also allows us to replace dynamic heap accesses with inlined ones. Most fast property accesses in DFG IR arise from inline cache feedback telling us that we should speculate, but sometimes the AI is able to prove something stronger than the profiler told us. This is especially likely in inlined code.

Clobberize. Clobberize is the alias analysis that the DFG uses to describe what parts of the program's state an instruction could read and write. This allows us to see additional dependency edges between instructions beyond just the ones expressed as data flow. Dependency information tells the compiler what kinds of instruction reorderings are legal. Clobberize has many clients in both the DFG and FTL. In the DFG, it's used for common subexpression elimination, for example.

To understand clobberize, it's worth considering what it is about a program's control flow that a compiler needs to remember. The control flow graph shows us one possible ordering of the program and we know that this ordering is legal. But both the DFG and FTL tiers want to move code around. The DFG tier mostly only moves code around within basic blocks rather than between them while the FTL tier can also move code between basic blocks. Even with the DFG's block-local code motion, it's necessary to know more than just the current ordering of the program. It's also necessary to know how that ordering can be changed.

Some of this is already solved by the data flow graph. DFG IR provides a data flow graph that shows *some* of the dependencies between instructions. It's obvious that if one instruction has a data flow edge to another, then only one possible ordering (source executes before sink) is valid. But what about:

- Stores to memory.
- Loads from memory.
- Calls that can cause any effects.
- OSR effects (like MovHint).

Data flow edges don't talk about those dependencies. Data flow also cannot tell which instructions have effects at all. So, the data flow graph cannot tell us anything about the valid ordering of instructions if those instructions have effects.

The issue of how to handle dependencies that arise from effects is particularly relevant to JavaScript compilation — and speculative compilation in general — because of the precision about aliasing that speculation gives us. For example, although the JavaScript `o.f` operation could have any effect, after speculation we often know that it can only affect properties named `f`. Additionally, JavaScript causes us to have to emit lots of loads to fields that are internal to our object model and it's good to know exactly when those loads are redundant so that we can remove as many of them as possible. So, we need to have the power to ask, for any operation that may access internal VM state, whether that state could be modified by any other operation, and we want that answer to be as precise as it can while being O(1)-ish.

Clobberize is a static analysis that augments the data flow and control flow graphs by telling us constraints on how instructions can be reordered. The neat thing about clobberize is that it avoids storing dependency information in the instructions themselves. So, while the compiler is free to query dependency information anytime it likes by running the analysis, it doesn't have to do anything to maintain it.

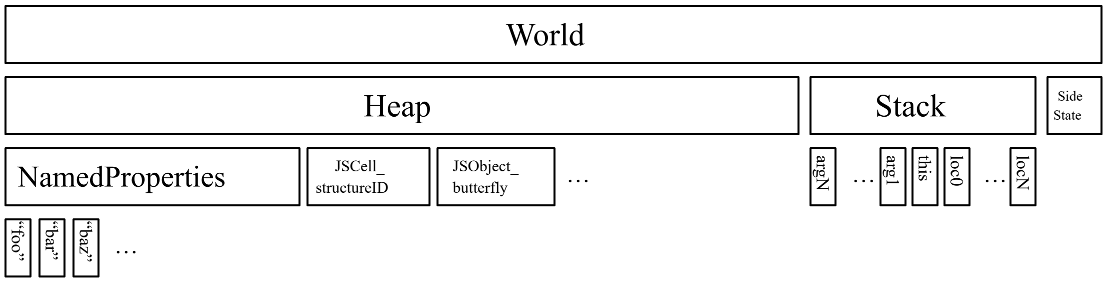


Figure 32. Some of the abstract heap hierarchy. All heaps are subsets of `World`, which is subdivided into `Heap`, `Stack` and `SideState`. For example, JS function calls say that they `write(Heap)` and `read(World)`. Subheaps of `Heap` include things like `JSObject_butterfly`, which refer to fields that are internal to the JSC object model and are not directly user-visible, and things like `NamedProperties`, a heap that contains subheaps for every named property the function accesses.

For each DFG instruction, clobberize reports zero or more reads or writes. Each read or write says which abstract heaps it is accessing. Abstract heaps are sets of memory locations. A read (or write) of an abstract heap means that the program will read (or write) from zero or more actual locations in that abstract heap. Abstract heaps form a hierarchy with `World` at the top (Figure 32). A write to `World` means that the effect could write to anything, so any read might see that write. The hierarchy can get very specific. For example, fully inferred, direct forms of property access like GetByOffset and PutByOffset report that they read and write (respectively) an abstract heap that names the property. So, accesses to properties of different names are known not to alias. The heaps are known to alias if either one is a descendant of the other.

It's worth appreciating how clobberize combined with control flow is just a way of encoding a dependence graph. To build a dependence graph from clobberize information, we apply the following rule. If instruction B appears after instruction A in control flow, then we treat B as having a dependence edge to A (B depends on A) if:

- any heap read by B overlaps any heap written by A, or
- any heap written by B overlaps any heap read or written by A.

Conversely, any dependence graph can be expressed using clobberize. An absurd but correct representation would involve giving each edge in the dependence graph its own abstract heap and having the source of the edge write the heap while the sink reads it. But what makes clobberize such an efficient representation of dependence graphs is that every dependence that we've tried to represent can be intuitively described by reads and writes to a small collection of abstract heaps.

Those abstract heaps are either collections of concrete memory locations (for example the `"foo"` abstract heap is the set of memory locations used to represent the values of properties named "foo") or they are metaphorical. Let's explore some metaphorical uses of abstract heaps:

- MovHint wants to say that it is not dead code, that it must be ordered with other MovHints, and that it must be ordered with any real program effects. We say this in clobberize by having MovHint write `SideState`. `SideState` is a subheap of `World` but disjoint from other things, and we have any operation that wants to be ordered with OSR exit state either read or write something that overlaps `SideState`. Note that DFG assumes that operations that may exit implicitly `read(World)` even if clobberize doesn't say this, so MovHint's write of `SideState` ensures ordering with exits.
- NewObject wants to say that it's not valid to hoist it out of loops because two successive executions of NewObject may produce different results. But it's not like NewObject clobbers the world; for example if we had two accesses to the same property on either sides of a NewObject then we'd want the second one to be eliminated. DFG IR has many NewObject-like operations that also have this behavior. So, we introduce a new abstract heap called `HeapObjectCount` and we say that NewObject is metaphorically incrementing (reading and writing) the `HeapObjectCount`. `HeapObjectCount` is treated as a subheap of `Heap` but it's disjoint from the subheaps that describe any state visible from JS. This is sufficient to block hoisting of NewObject while still allowing interesting optimizations to happen around it.

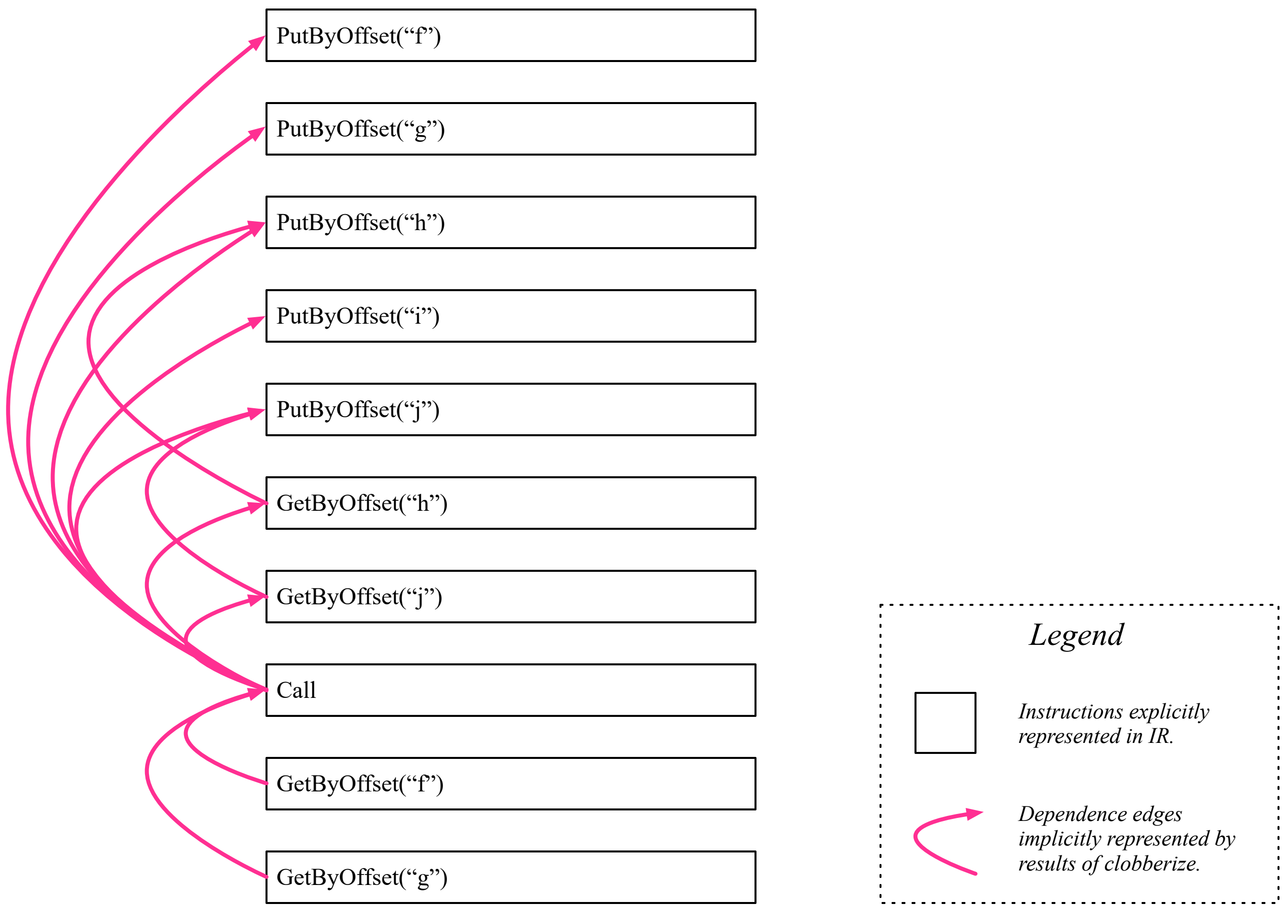


Figure 33. Sample sequence of DFG IR instructions and their dependence graph. DFG IR never stores the dependence graph in memory because we get the information implicitly by running clobberize.

The combination of clobberize and the control flow graph gives a scalable and intuitive way of expressing the dependence graph. It's scalable because we don't actually have to express any of the edges. Consider for example a dynamic access instruction that could read any named JavaScript property, like the Call instruction in Figure 33. Clobberize can say this in O(1) space and time. But a dependence graph would have to create an edge from that instruction to any instruction that accesses any named property before or after it. In short, clobberize gives us the benefit of a dependence graph without the cost of allocating memory to represent the edges.

The abstract heaps can also be efficiently collected into a set, which we use to summarize the aliasing effects of basic blocks and loops.

To summarize, the DFG puts a big emphasis on static analysis. Speculation decisions are made using a combination of profiling and an abstract interpreter called prediction propagation. Additionally, we have an abstract interpreter for optimization, simply called the DFG abstract interpreter, which serves as the main engine for redundant check removal. Abstract interpreters are a natural fit for the DFG because they give us a way to forward-propagate information about types. Finally, the DFG uses the clobberize analysis to describe dependencies and aliasing.

Fast Compilation

The DFG is engineered to compile quickly so that the benefits of OSR speculations can be realized quickly. To help reduce compile times, the DFG is focused about what optimizations it does and how it does them. The static analysis and OSR exit optimizations discussed so far represent the most powerful things that the DFG is capable of. The DFG does a quick and dirty job with everything else, like instruction selection, register allocation, and removal of redundant code that isn't checks. Functions that benefit from the compiler doing a good job on those optimizations will get them if they run long enough to tier up into the FTL.

The DFG's focus on fast compilation happened organically, as a result of many separate throughput-latency trade-offs. Initially, JavaScriptCore just had the Baseline JIT and then later Baseline as the profiling tier and DFG as the optimizing tier. The DFG experienced significant evolution during this time, and then experienced additional evolution after the FTL was introduced. While no single decision led to the DFG's current design, we believe that it was most significantly shaped by tuning for short-running benchmarks and the introduction of the FTL.

The DFG was tuned for a diverse set of workloads. On the one hand, it was tuned for long-running tests in which one full second of warm-up was given to the speculative compiler for free (like the old V8 benchmarks, which live on in the form of Octane and JetStream, albeit without the freebie warmup), but on the other hand, it was also tuned for shorter-running benchmarks like SunSpider and page load tests. SunSpider focused on smallish programs running for very short bursts of time with little opportunity for warm-up. Compilers that do more optimizations than the DFG tend to lose to it on SunSpider because they fail to complete their optimizations before SunSpider finishes running. We continue to use tests that are in the spirit of SunSpider, like Speedometer and JetStream. Speedometer has a similar code-size-to-running-time ratio, so like SunSpider, it benefits a lot from DFG. JetStream includes a subset of SunSpider and puts a big emphasis on short-running code in all of its other tests. That's not to say that we don't also care about long-running code. It's just that our methodology for improving the DFG was to try to get speed-ups on both short-running things and long-running things with the same engine. Since any long-running optimization would regress the short-running tests, we often avoided adding any long-running optimizations to the DFG. But we did add cheap versions of many sophisticated optimizations, giving respectable speed-ups on both short-running and long-running workloads.

The introduction of the FTL solidified the DFG's position as the compiler that optimizes less. So long as the DFG generates reasonably good code quickly, we can get away with putting lots of expensive optimizations into the FTL. The FTL's long compile times mean that many programs do not run long enough to benefit from the FTL. So, the DFG is there to give those programs a speculative optimization boost in way less time than an FTL-like compiler could do. Imagine a VM that only had one optimizing compiler. Unless that one compiler compiled as fast as the DFG and generated code that was as good as the FTL, it would end up being reliably slower than JavaScriptCore on some workloads. If that compiler compiled as fast as the DFG but didn't have the FTL's throughput then any program that ran long enough would run faster in JavaScriptCore. If that compiler generated code that was as good as the FTL but compiled slower than the DFG then any program that ran short enough to tier up into the DFG but not that compiler would run faster in JavaScriptCore. JavaScriptCore has multiple compiler tiers because we believe that it is not possible to build a compiler that compiles as fast as the DFG while generating code that is as good as the FTL.

To summarize, the DFG focuses on fast compilation because of the combination of the history of how it was tuned and the fact that it sits as the tier below the FTL JIT.

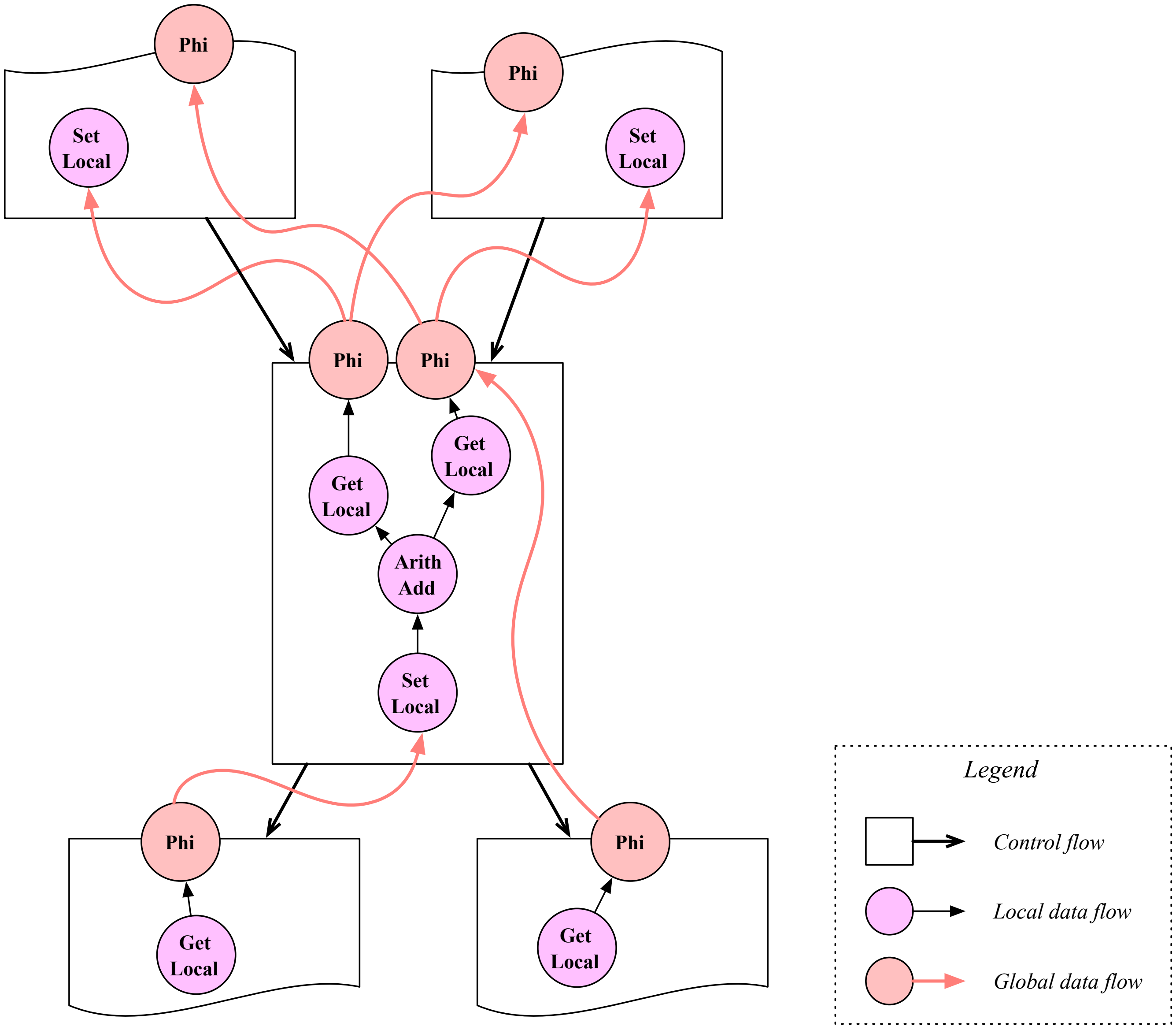


Figure 34. Illustration of a sample DFG IR program with all three graphs: local data flow, global data flow, and control flow.

The DFG compiler's speed comes down to an emphasis on block-locality in the IR. The DFG IR used by the DFG tier has a two-level data flow graph:

- Local data flow graph. The local data flow graph is used within basic blocks. This graph is a first-class citizen in the IR, when working with data flow in the DFG's C++ code, it sometimes seems like this is the only data flow graph. DFG IR inside a basic block resembles SSA form in the sense that there's a 1:1 mapping between instructions and the variables they assign and data flow is represented by having users of values point at the instructions (*nodes*) that produce those values. This representation does not allow you to use a value produced by an instruction in a different block except through tedious escape hatches.
- Global data flow graph. We say *global* to mean the entire compilation unit, so some JS function and whatever the DFG inlined into it. So, *global* just means spanning basic blocks. DFG IR maintains a secondary data flow graph that spans basic blocks. DFG IR's approach to global data flow is based on spilling: to pass a value to a successor block, you store it to a spill slot on the stack, and then that block loads it. But in DFG IR, we also thread data flow relationships through those loads and stores. This means that if you are willing to perform the tedious task of traversing this secondary data flow graph, you can get a global view of data flow.

Figure 34 shows an example of how this works. The compilation unit is represented as three graphs: a control flow graph, local data flow, and global data flow. Data flow graphs are represented with edges going from the user to the value being used. The local data flow graphs work like they do in SSA, so any SSA optimization can be run in a block-local manner on this IR. The global data flow graph is made of SetLocal/GetLocal nodes that store/load values into the stack. The data flow between SetLocal and GetLocal is represented completely in DFG IR, by threading data flow edges through special *Phi* nodes in each basic block where a local is live.

From the standpoint of writing outstanding high-throughput optimizations, this approach to IR design is like kneecapping the compiler. Compilers thrive on having actual SSA form, where there is a single data flow graph, and you don't have to think about an instruction's position in control flow when traversing data flow. The emphasis on locality is all about excellent compile times. We believe that locality gives us compile time improvements that we can't get any other way:

- Instruction selection and register allocation for a basic block can be implemented as a single pass over that basic block. The instruction selector can make impromptu register allocation decisions during that pass, like deciding that it needs any number of scratch registers to emit code for some DFG node. The combined instruction selector and register allocator (aka the DFG backend) compiles basic blocks independently of one another. This kind of code generation is good at register allocating large basic blocks but bad for small ones. For functions that only have a single basic block, the DFG often generates code that is as good as the FTL.
- We never have to decompress the delta encoding of OSR exit. We just have the backend record a log of its register allocation decisions (the variable event stream). While the DFG IR for a function is thrown out after compilation, this log along with a minified version of the DFG IR (that only includes MovHints and the things they reference) is saved so that we can replay what the backend did whenever an OSR exit happens. This makes OSR exit handling super cheap in the DFG – we totally avoid the O(n^2) complexity explosion of OSR stackmaps despite the fact that we speculate like crazy.
- There is no need to enter or exit SSA. On the one hand, SSA conversion performance is a solved problem: it's a nearly-linear-time operation. Even so, the constant factors are high enough that avoiding it entirely is profitable. Converting out of SSA is worse. If we wanted to combine SSA with our block-local backend, we'd have to add some sort of transformation that discovers how to load/store live state across basic blocks. DFG IR plays tricks where the same store that passes data flow to another block doubles as the OSR exit state update. It's not obvious that exiting out of SSA would discover all of the cases where the same store can be reused for both OSR exit state update and the data flow edge. This suggests that any version of exiting out of SSA would make the DFG compiler either generate worse code or run slower. So, not having SSA makes the compiler run faster because entering SSA is not free and exiting SSA is awful.
- Every optimization is faster if it is block-local. Of course, you could write block-local optimizations in an SSA IR. But having an IR that emphasizes locality is like a way to statically guarantee that we won't accidentally introduce expensive compiler passes to the DFG.

The one case where global data flow is essential to the DFG's mission is static analysis. This comes up in the prediction propagator and the abstract interpreter. Both of them use the global data flow graph in addition to the local data flow graphs, so that they can see how type information flows through the whole compilation unit. Fortunately, as shown in Figure 34, the global data flow graph is available. It's in a format that makes it hard to edit but relatively easy to analyze. For example, it implicitly reports the set of live variables at each basic block boundary, which makes merging state in the abstract interpreter relatively cheap.

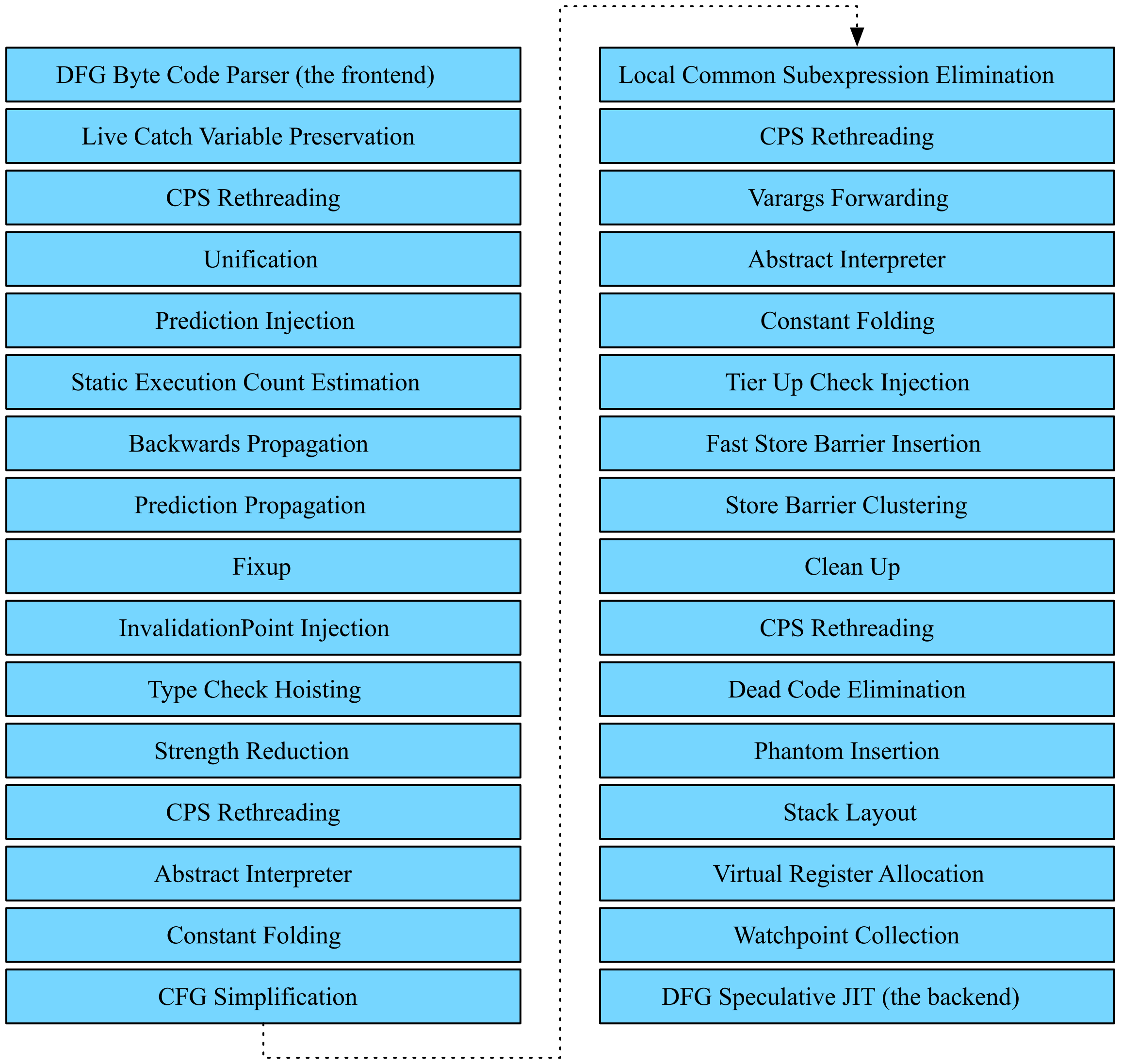


Figure 35. The DFG pipeline.

Figure 35 shows the complete DFG optimization pipeline. This is a fairly complete pipeline: it has classics like constant folding, control flow simplification, CSE, and DCE. It also has lots of JavaScript-specifics like deciding where to put checks (unification, prediction injection and propagation, prediction propagation, and fixup), a pass just to optimize common patterns of varargs, some passes for GC barriers, and passes that help OSR (CPS rethreading and phantom insertion). We can afford to do a lot of optimizations in the DFG so long as those optimizations are block-local and don't try too hard. Still, this pipeline is way smaller than the FTL's and runs much faster.

To summarize, the DFG compiler uses OSR exit and static analysis to emit an optimal set of type checks. This greatly reduces the number of type checks compared to running JavaScript in either of the profiled tiers. Because the benefit of type check removal is so big, the DFG compiler tries to limit how much time it spends doing other optimizations by restricting itself to a mostly block-local view of the program. This is a trade off that the DFG makes to get fast compile times. Functions that run long enough that they'd rather pay the compile time to get those optimizations end up tiering up to the FTL, which just goes all out for throughput.

FTL Compiler

We've previously documented some aspects of the FTL's architecture in the original blog post and when we introduced B3. This section provides an updated description of this JIT's capabilities as well as a deep dive into how FTL does OSR. We will structure our discussion of the FTL as follows. First we will enumerate what optimizations it is capable of. Then we will describe how it does OSR exit in detail. Finally we will talk about patchpoints — an IR operation based on a lambda.

All The Optimizations

The point of the FTL compiler is to run all the optimizations. This is a compiler where we never compromise on peak throughput. All of the DFG's decisions that were known trade-offs in favor of compile time at the expense of throughput are reversed in the FTL. There is no upper limit on the amount of cycles that a function compiled with the FTL will run for, so it's the kind of compiler where even esoteric optimizations have a chance to pay off eventually. The FTL combines multiple optimization strategies:

- We reuse the DFG pipeline, including the weird IR. This ensures that any good thing that the DFG tier ever does is also available in the FTL.
- We add a new DFG SSA IR and DFG SSA pipeline. We adapt lots of DFG phases to DFG SSA (which usually makes them become global rather than local). We add lots of new phases that are only possible in SSA (like loop invariant code motion).
- We lower DFG SSA IR to B3 IR. B3 is an SSA-based optimizing JIT compiler that operates at the abstraction level of C. B3 has lots of optimizations, including global instructcion selection and graph coloring register allocation. The FTL was B3's first customer, so B3 is tailored for optimizing at the abstraction level where DFG SSA IR can't.

Having multiple ways of looking at the program gives the FTL maximum opportunities to optimize. Some of the compiler's functionality, particularly in the part that decides where to put checks, thrives on the DFG's weird IR. Other parts of the compiler work best in DFG SSA, like the DFG's loop-invariant code motion. Lots of things work best in B3, like most reasoning about how to simplify arithmetic. B3 is the first IR that doesn't know anything about JavaScript, so it's a natural place to implement textbook optimization that would have difficulties with JavaScript's semantics. Some optimizations, like CSE, work best when executed in every IR because they find unique opportunities in each IR. In fact, all of the IRs have the same fundamental optimization capabilities in addition to their specialized optimizations: CSE, DCE, constant folding, CFG simplification, and strength reductions (sometimes called peephole optimizations or instruction combining).

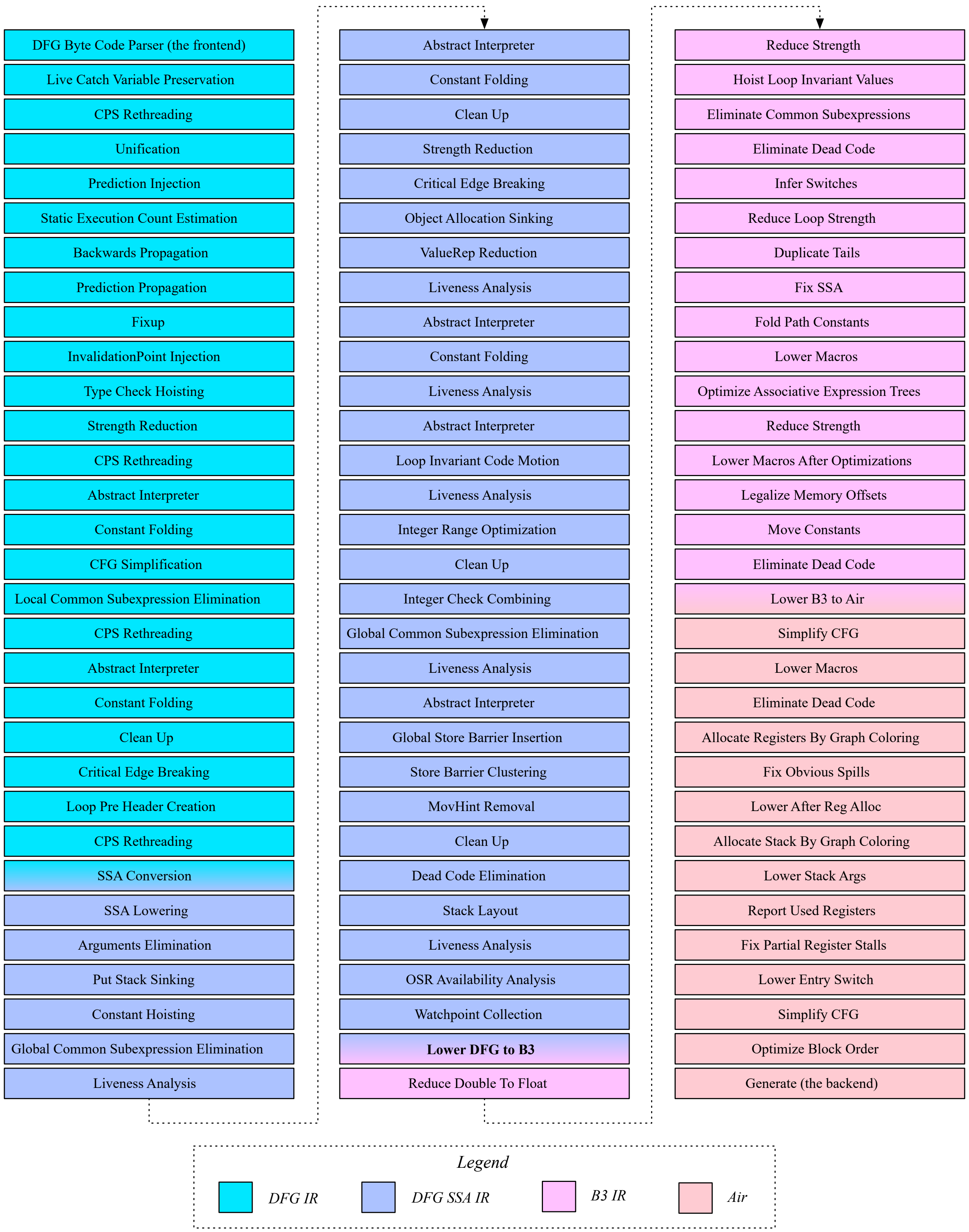


Figure 36. The FTL pipeline. Note that Lower DFG to B3 is in bold because it's FTL's biggest phase; sometimes when we say "FTL" we are just referring to this phase.

The no-compromise approach is probably best appreciated by looking at the FTL optimization pipeline in Figure 36. The FTL runs 93 phases on the code in encounters. This includes all phases from Figure 35 (the DFG pipeline), except Varargs Forwarding, only because it's completely subsumed by the FTL's Arguments Elimination. Let's review some of the FTL's most important optimizations:

- DFG AI. This is one of the most important optimizations in the FTL. It's mostly identical to the AI we run in the DFG tier. Making it work with SSA makes it slightly more precise and slightly more expensive. We run the AI a total of six times.
- CSE (common subexpression elimination). We run this in DFG IR (Local Common Subexpression Elimination), DFG SSA IR (Global Common Subexpression Elimination), B3 IR (Reduce Strength and the dedicated Eliminate Common Subexpressions), and even in Air (Fix Obvious Spills, a CSE focused on spill code). Our CSEs can do value numbering and load/store elimination.
- Object Allocation Sinking is a must-points-to analysis that we use to eliminate object allocations or sink them to slow paths. It can eliminate graphs of object allocations, including cyclic graphs.
- Integer Range Optimization is a forward flow-sensitive abtract interpreter in which the state is a system of equations and inequalities that describe known relationships between variables. It can eliminate integer overflow checks and array bounds checks.
- The B3 Reduce Strength phase runs a fixpoint that includes CFG simplification, constant folding, reassociation, SSA clean-up, dead code elimination, a light CSE, and lots of miscellaneous strength reductions.
- Duplicate Tails, aka tail duplication, flattens some control flow diamonds, unswitches small loops, and undoes some cases of relooping. We duplicate small tails blindly over a CFG with critical edges broken. This allows us to achieve some of what splitting achieved for the original speculative compilers.
- Lower B3 to Air is a global pattern matching instruction selector.
- Allocate Registers By Graph Coloring implements the IRC and Briggs register allocators. We use IRC on x86 and Briggs on arm64. The difference is that IRC can find more opportunities for coalescing assignments into a single register in cases where there is high register pressure. Our register allocators have special optimizations for OSR exit, especially the OSR exits we emit for integer overflow checks.

OSR Exit in the FTL

Now that we have enumerated some of the optimizations that the FTL is capable of, let's take a deep dive into how the FTL works by looking at how it compiles and does OSR. Let's start with this example:

```
function foo(a, b, c)
{
    return a + b + c;
}
```

The relevant part of the bytecode sequence is:

```
[   7] add loc6, arg1, arg2
[  12] add loc6, loc6, arg3
[  17] ret loc6
```

Which results in the following DFG IR:

```
  24:  GetLocal(Untyped:@1, arg1(B<Int32>/FlushedInt32), R:Stack(6), bc#7)
  25:  GetLocal(Untyped:@2, arg2(C<BoolInt32>/FlushedInt32), R:Stack(7), bc#7)
  26:  ArithAdd(Int32:@24, Int32:@25, CheckOverflow, Exits, bc#7)
  27:  MovHint(Untyped:@26, loc6, W:SideState, ClobbersExit, bc#7, ExitInvalid)
  29:  GetLocal(Untyped:@3, arg3(D<Int32>/FlushedInt32), R:Stack(8), bc#12)
  30:  ArithAdd(Int32:@26, Int32:@29, CheckOverflow, Exits, bc#12)
  31:  MovHint(Untyped:@30, loc6, W:SideState, ClobbersExit, bc#12, ExitInvalid)
  33:  Return(Untyped:@3, W:SideState, Exits, bc#17)
```

The DFG data flow from the snippet above is illustrated in Figure 37 and the OSR exit sites are illustrated in Figure 38.

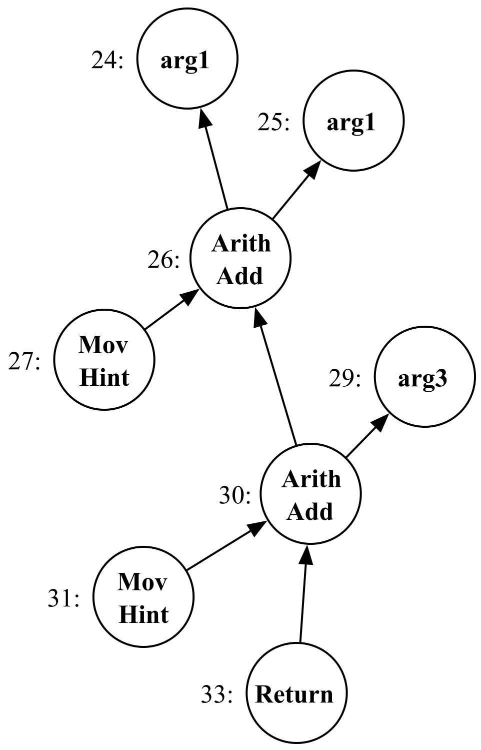
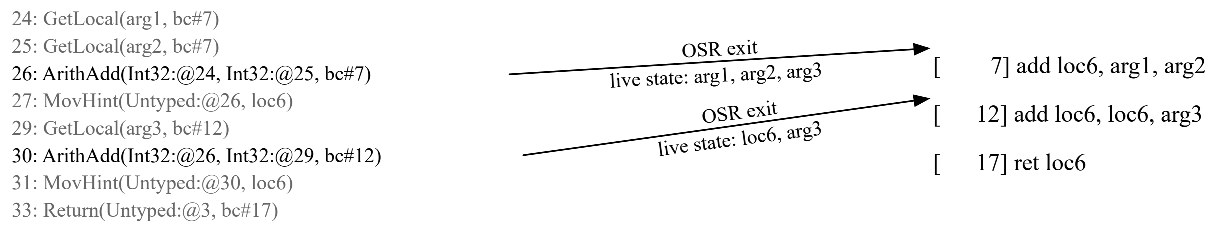


Figure 37. Data flow graph for FTL code generation example.



```
24: GetLocal(arg1, bc#7)
25: GetLocal(arg2, bc#7)
26: ArithAdd(Int32:@24, Int32:@25, bc#7)
27: MovHint(Untyped:@26, loc6)
29: GetLocal(arg3, bc#12)
30: ArithAdd(Int32:@26, Int32:@29, bc#12)
31: MovHint(Untyped:@30, loc6)
33: Return(Untyped:@3, bc#17)
```

```
[     7] add loc6, arg1, arg2
[    12] add loc6, loc6, arg3
[    17] ret loc6
```

Figure 38. DFG IR example with the two exiting nodes highlighted along with where they exit and what state is live when they exit.

We want to focus our discussion on the MovHint @27 and how it impacts the code generation for the ArithAdd @30. That ArithAdd is going to exit to the second `add` in the bytecode, which requires restoring `loc6` (i.e. the result of the first `add`), since it is live at that point in bytecode (it also happens to be directly used by that `add`).

This DFG IR is lowered to the following in B3:

```
Int32 @42 = Trunc(@32, DFG:@26)
Int32 @43 = Trunc(@27, DFG:@26)
Int32 @44 = CheckAdd(@42:WarmAny, @43:WarmAny, generator = 0x1052c5cd0,
                     earlyClobbered = [], lateClobbered = [], usedRegisters = [],
                     ExitsSideways|Reads:Top, DFG:@26)
Int32 @45 = Trunc(@22, DFG:@30)
Int32 @46 = CheckAdd(@44:WarmAny, @45:WarmAny, @44:ColdAny, generator = 0x1052c5d70,
                     earlyClobbered = [], lateClobbered = [], usedRegisters = [],
                     ExitsSideways|Reads:Top, DFG:@30)
Int64 @47 = ZExt32(@46, DFG:@32)
Int64 @48 = Add(@47, $-281474976710656(@13), DFG:@32)
Void @49 = Return(@48, Terminal, DFG:@32)
```

CheckAdd is the B3 way of saying: do an integer addition, check for overflow, and if it overflows, execute an OSR exit governed by a `generator`. The generator is a lambda that is given a JIT generator object (that it can use to emit code at the jump destination of the OSR exit) and a *stackmap generation parameters* that tells the *B3 value representation* for each stackmap argument. The B3 value reps tell you which register, stack slot, or constant to use to get the value. B3 doesn't know anything about how exit works except that it involves having a stackmap and a `generator` lambda. So, CheckAdd can take more than 2 arguments; the first two arguments are the actual add operands and the rest are the stackmap. It's up to the client to decide how many arguments to pass to the stackmap and only the generator will ever get to see their values. In this example, only the second CheckAdd (@46) is using the stackmap. It passes one extra argument, @44, which is the result of the first add — just as we would expect based on MovHint @27 and the fact that loc6 is live at bc#12. This is the result of the FTL decompressing the delta encoding given by MovHints into full stackmaps for B3.

Stackmap	Mapping Type	How Computed
DFG IR stackmap	bytecode local ⇒ { DFG Node stack slot	DFG OSR exit analysis
B3 value reps	argument index ⇒ { register stack slot immediate	B3 register allocation *(works for any B3 data flow edge)*
FTL OSR exit descriptor	bytecode local ⇒ { argument index stack slot immediate materialization	DFG IR stackmap and FTL lowering
FTL lowering value mapping	DFG node ⇒ B3 value	Implicitly during FTL lowering
FTL OSR exit	bytecode local ⇒ { register stack slot immediate materialization	Lazily computed during FTL OSR exit handling

Figure 39. The stackmaps and stackmap-like mappings maintained by the FTL to enable OSR.

FTL OSR exit means tracking what happens with the values of bytecode locals through multiple stages of lowering. The various stages don't know a whole lot about each other. For example, the final IRs, B3 and Air, know nothing about bytecode, bytecode locals, or any JavaScript concepts. We implement OSR exit by tracking multiple stackmap-like mappings per exit site that give us the complete picture when we glue them together (Figure 39):

- The *DFG IR stackmaps* that we get be decompressing MovHint deltas. This gives a **mapping** from **bytecode local** to **either a DFG node or a stack location**. In some cases, DFG IR has to store some values to the stack to support dynamic variable introspection like via `function.arguments`. DFG OSR exit analysis is smart enough recognize those cases, since it's more optimal to handle those cases by having OSR exit extract the value from the stack. Hence, OSR exit analysis may report that a bytecode local is available through a DFG node or a stack location.
- The *B3 value reps* array inside the stackmap generation parameters that B3 gives to the generator lambdas of Check instructions like CheckAdd. This is a **mapping** from **B3 argument index** to a **B3 value representation**, which is either a register, a constant, or a stack location. By *argument index* we mean index in the stackmap arguments to a Check. This is three pieces of information: some user value (like @46 = CheckAdd(@44, @45, @44)), some index within its argument list (like 2), and the value that index references (@44). Note that since this CheckAdd has two argument indices for @44, that means that they may end up having different value representations. It's not impossible for one to be a constant and another to be a register or spill slot, for example (though this would be highly unlikely; if it happened then it would probably be the result of some sound-but-inefficient

antipattern in the compiler). B3's client gets to decide how many stackmap arguments it will pass and B3 guarantees that it will give the generator a value representation for each argument index in the stackmap (so starting with argument index 2 for CheckAdd).
- The *FTL OSR exit descriptor* objects, which the FTL's DFG→B3 lowering creates at each exit site and holds onto inside the generator lambda it passes to the B3 check. Exit descriptors are based on DFG IR stackmaps and provide a **mapping** from **bytecode local** to **B3 argument index, constant, stack slot, or materialization**. If the DFG IR stackmap said that a bytecode local is a Node that has a constant value, then the OSR exit descriptor will just tell us that value. If the DFG stackmap said that a local is already on the stack, then the OSR exit descriptor will just tell that stack slot. It could be that the DFG stackmap tells us that the node is a *phantom object allocation* — an object allocation we optimized out but that needs to be rematerialized on OSR exit. If it is none of those things, the OSR exit descriptor will tell us which B3 argument index has the value of that bytecode local.
- The FTL's DFG→B3 lowering already maintains a **mapping** from **DFG node** to **B3 value**.
- The *FTL OSR Exit* object, which is a **mapping** from **bytecode local** to **register, constant, stack slot, or materialization**. This is the final product of the FTL's OSR exit handling and is computed lazily from the B3 value reps and FTL OSR exit descriptor.

These pieces fit together as follows. First we compute the DFG IR stackmap and the FTL's DFG node to B3 value mapping. We get the DFG IR stackmap from the DFG OSR exit analysis, which the FTL runs in tandem with lowering. We get the DFG to B3 mapping implicitly from lowering. Then we use that to compute the FTL OSR exit descriptor along with the set of B3 values to pass to the stackmap. The DFG IR stackmap tells us which DFG nodes are live, so we turn that into B3 values using the DFG to B3 mapping. Some nodes will be excluded from the B3 stackmap, like object materializations and constants. Then the FTL creates the Check value in B3, passes it the stackmap arguments, and gives it a generator lambda that closes over the OSR exit descriptor. B3's Check implementation figures out which value representations to use for each stackmap argument index (as a result of B3's register allocator doing this for every data flow edge), and reports this to the generator as an array of B3 value reps. The generator then creates a `FTL::OSRExit` object that refers to the FTL OSR exit descriptor and value reps. Users of the FTL OSR exit object can figure out which register, stack slot, constant value, or materialization to use for any bytecode local by asking the OSR exit descriptor. That can tell the constant, spill slot, or materialization script to use. It can also give a stackmap argument index, in which case we load the value rep at that index, and that tells us the register, spill slot, or constant.

This approach to OSR exit gives us two useful properties. First, it empowers OSR-specific optimization. Second, it empowers optimizations that don't care about OSR. Let's go into these in more detail.

FTL OSR empowers OSR-specific optimizations. This happens in DFG IR and B3 IR. In DFG IR, OSR exit is a mutable part of the IR. Any operation can be optimized by adding more OSR exits and we even have the ability to move checks around. The FTL does sophisticated OSR-aware optimizations using DFG IR, like object allocation sinking. In B3 IR, OSR exit gets special register allocation treatment. The stackmap arguments of Check are understood by B3 to be *cold uses*, which means that it's not expensive if those uses are spilled. This is powerful information for a register allocator. Additionally, B3 does special register allocation tricks for addition and subtraction with overflow checks (for example we can precisely identify when the result register can reuse a stackmap register and when we can coalesce the result register with one of the input registers to produce optimal two-operand form on x86).

FTL OSR also empowers optimizations that don't care about OSR exit. In B3 IR, OSR exit decisions get frozen into stackmaps. This is the easiest representation of OSR exit because it requires no knowledge of OSR exit semantics to get right. It's natural for compiler phases to treat extra arguments to an instruction opaquely. Explicit stackmaps are particularly affordable in B3 because of a combination of factors:

1. the FTL is a more expensive compiler anyway so the DFG OSR delta encoding optimizations matter less,
2. we only create stackmaps in B3 for exits that DFG didn't optimize out, and
3. B3 stackmaps only include a subset of live state (the rest may be completely described in the FTL OSR exit descriptor).

We have found that some optimizations are annoying, sometimes to the point of being impractical, to write in DFG IR because of explicit OSR exit (like MovHint deltas and exit origins). It's not necessary to worry about those issues in B3. So far we have found that every textbook optimization for SSA is practical to do in B3. This means that we only end up having a bad time with OSR exit in our compiler when we are writing phases that benefit from DFG's high-level knowledge; otherwise we write the phases in B3 and have a great time.

This has some surprising outcomes. Anytime FTL emits a Check value in B3, B3 may duplicate the Check. B3 IR semantics allow any code to be duplicated during optimization and this usually happens due to tail duplication. Not allowing code duplication would restrict B3 more than we're comfortable doing. So, when the duplication happens, we handle it by having multiple FTL OSR exits share the same OSR exit descriptor but get separate value reps. It's also possible for B3 to prove that some Check is either unnecessary (always succeeds) or is never reached. In that case, we will have one FTL OSR exit descriptor but zero FTL OSR exits. This works in such a way that DFG IR never knows that the code was duplicated and B3's tail duplication and unreachable code elimination know nothing about OSR exit.

Patchpoints: Lambdas in the IR

This brings us to the final point about the FTL. We think that what is most novel about this compiler is its use of lambdas in its IRs. Check is one example of this. The DFG has some knowledge about what a Check would do at the machine code level, but that knowledge is incomplete until we fill in some blanks about how B3 register-allocated some arguments to the Check. The FTL handles this by having one of the operands to a B3 Check be a lambda that takes a JIT code generator object and value representations for all of the arguments. We like this approach so much that we also have B3 support Patchpoint. A Patchpoint is like an inline assembly snippet in a C compiler, except that instead of a string containing assembly, we pass a lambda that will generate that assembly if told how to get its arguments and produce its result. The FTL uses this for a bunch of cases:

- Anytime the B3 IR generated by the FTL interacts with JavaScriptCore's internal ABI. This includes all calls and call-like instructions.
- Inline caches. If the FTL wants to emit an inline cache, it uses the same inline cache code generation logic that the DFG and baseline use. Instead of teaching B3 how to do this, we just tell B3 that it's a patchpoint.
- Lazy slow paths. The FTL has the ability to only emit code for a slow path if that slow path executes. We implement that using patchpoints.
- Instructions we haven't added to B3 yet. If we find some JavaScript-specific CPU instruction, we don't have to thread it through B3 as a new opcode. We can just emit it directly using a Patchpoint. (Of course, threading it through B3 is a bit better, but it's great that it's not strictly necessary.)

Here's an example of the FTL using a patchpoint to emit a fast double-to-int conversion:
```
if (MacroAssemblerARM64::
    supportsDoubleToInt32ConversionUsingJavaScriptSemantics()) {
    PatchpointValue* patchpoint = m_out.patchpoint(Int32);
    patchpoint->appendSomeRegister(doubleValue);
    patchpoint->setGenerator(
        [=] (CCallHelpers& jit,
             const StackmapGenerationParams& params) {
            jit.convertDoubleToInt32UsingJavaScriptSemantics(
                params[1].fpr(), params[0].gpr());
        });
    patchpoint->effects = Effects::none();
    return patchpoint;
}
```

This tells B3 that it's a Patchpoint that returns Int32 and takes a Double. Both are assumed to go in any register of B3's choice. Then the generator uses a C++ lambda to emit the actual instruction using our JIT API. Finally, the patchpoint tells B3 that the operation has no effects (so it can be hoisted, killed, etc).

This concludes our discussion of the FTL. The FTL is our high throughput compiler that does every optimization we can think of. Because it is a speculative compiler, a lot of its design is centered around having a balanced handling of OSR exit, which involves a separation of concerns between IRs that know different amounts of things about OSR. A key to the FTL's power is the use of lambdas in B3 IR, which allows B3 clients to configure how B3 emits machine code for some operations.

Summary of Compilation and OSR

To summarize, JavaScriptCore has two optimizing compilers, the DFG and FTL. They are based on the same IR (DFG IR), but the FTL extends this with lots of additional compiler technology (SSA and multiple IRs). The DFG is a fast compiler: it's meant to compile faster than typical optimizing compilers. But, it generates code that is usually not quite optimal. If that code runs long enough, then it will also get compiled with the FTL, which tries to emit the best code possible.

Related Work

The idea of using feedback from cheap profiling to speculate was pioneered by the Hölzle, Chambers, and Ungar paper on polymorphic inline caches, which calls this *adaptive compilation*. That work used a speculation strategy based on *splitting*, which means having the compiler emit many copies of code, one for each possible type. The same three authors later invented OSR exit, though they called it *dynamic deoptimization* and only used it to enhance debugging. Our approach to speculative compilation means using OSR exit as our primary speculation strategy. We do use splitting in a very limited sense: we emit diamond speculations in those cases where we are not sure enough to use OSR and then we allow tail duplication to split the in-between code paths if they are small enough.

This speculative compilation technique, with OSR or diamond speculations but not so much splitting, first received extraordinary attention during the Java performance wars. Many wonderful Java VMs used combinations of interpreters and JITs with varied optimization strategies to profile virtual calls and speculatively devirtualize them, with the best implementations using inline caches, OSR exit, and watchpoints. Java implementations that used variants of this technique include (but are not limited to):

- the IBM JIT, which combined an interpreter and an optimizing JIT and did diamond speculations for devirtualization.
- HotSpot and HotSpot server, which combined an interpreter and an optimizing JIT and used diamond speculations, OSR exit, and lots of other techniques that JavaScriptCore uses. JavaScriptCore's FTL JIT is similar to HotSpot server in the sense that both compilers put a

big emphasis on great OSR support, comprehensive low-level optimizations, and graph coloring register allocation.
- Eclipse J9, a major competitor to HotSpot that also uses speculative compilation.
- Jikes RVM, a research VM that used OSR exit but combined a baseline JIT and an optimizing JIT. I learned most of what I know about this technique from working on Jikes RVM.

Like Java, JavaScript has turned out to be a great use case for speculative compilation. Early instigators in the JavaScript performance war included the Squirrelfish interpreter (predecessor to LLInt), the Squirrelfish Extreme JIT (what we now call the Baseline JIT), the early V8 engine that combined a baseline JIT with inline caches, and TraceMonkey. TraceMonkey used a cheap optimizing JIT strategy called tracing, which compiles lots of speculative paths. This JIT sometimes outperformed the baseline JITs, but often lost to them due to overspeculation. V8 upped the ante by introducing the speculative compilation approach to JavaScript, using the template that had worked so well in Java: a lower tier that does inline caches, then an optimizing JIT (called Crankshaft) that speculates based on the inline caches and exits to the lower tier. This version of V8 used a pair of JITs (baseline JIT and optimizing JIT), much like Jikes RVM did for Java. JavaScriptCore soon followed by hooking up the DFG JIT as an optimizing tier for the baseline JIT, then adding the LLInt and FTL JIT. During about the same time, TraceMonkey got replaced with IonMonkey, which uses similar techniques to Crankshaft and DFG. The ChakraCore JavaScript implementation also used speculative compilation. JavaScriptCore and V8 have continued to expand their optimizations with innovative compiler technology like B3 (a CFG SSA compiler) and TurboFan (a sea-of-nodes SSA compiler). Much like for Java, the top implementations have at least two tiers, with the lower one used to collect profiling that the upper one uses to speculate. And, like for Java, the fastest implementations are built around OSR speculation.

Conclusion

JavaScriptCore includes some exciting speculative compiler technology. Speculative compilation is all about speeding up dynamically typed programs by placing bets on what types the program would have had if it could have types. Speculation uses OSR exit, which is expensive, so we engineer JavaScriptCore to make speculative bets only if they are a sure thing. Speculation involves using multiple execution tiers, some for profiling, and some to optimize based on that profiling. JavaScriptCore includes four tiers to also get an ideal latency/throughput trade-off on a per-function basis. A control system chooses when to optimize code based on whether it’s hot enough and how many times we’ve tried to optimize it in the past. All of the tiers use a common IR (bytecode in JavaScriptCore’s case) as input and provide independent implementation strategies with different throughput/latency and speculation trade-offs.

This post is an attempt to demystify our take on speculative compilation. We hope that it’s a useful resource for those interested in JavaScriptCore and for those interested in building their own fast language implementations (especially the ones with really weird and funny features).