A New Bytecode Format for JavaScriptCore

Jun 21, 2019
by Tadeu Zagallo
@tadeuzagallo

In revision r237547 we introduced a new bytecode format for JavaScriptCore (JSC). The goals of the new format were to improve memory usage and allow the bytecode to be cached on disk, while the previous format was optimized for interpreter throughput at the cost of memory usage.

In this post, we will start with a quick overview of JSC's bytecode, key aspects of the old bytecode format and the optimizations it enabled. Next, we will look into the new format and how it affects interpreter execution. Finally, we will look at the impact of the new format on memory usage and performance and how this rewrite improved type safety in JavaScriptCore.

Background

Before JSC executes any JavaScript code, it must lex, parse and generate bytecode for it. JSC has 4 tiers of execution:

- **Low Level Interpreter (LLInt)**: the start-up interpreter
- **Baseline JIT**: a template JIT
- **DFG JIT**: a low-latency optimizing compiler
- **FTL JIT**: a high-throughput optimizing compiler

Execution starts by interpreting the bytecode, at the lowest tier, and as the code gets executed more it gets promoted to a higher tier. This is described in a lot more details in this blog post about the FTL.

The bytecode is the source of truth throughout the whole engine. The LLInt executes the bytecode. The baseline is a template JIT, which emits snippets of machine code for each bytecode instruction. Finally, the DFG and FTL parse the bytecode and emit DFG IR, which is then run through an optimizing compiler.

Because the bytecode is the source of truth, it tends to stay alive in memory throughout the whole program execution. In JavaScript-heavy websites, such as Facebook or Reddit, the bytecode is responsible for **20%** of the overall memory usage.

The Bytecode

To make things more concrete, let’s look at a simple JavaScript program, learn how to inspect the bytecode generated by JSC and how to interpret the bytecode dump.

```
// double.js
function double(a) {
    return a + a;
}
double(2);
```

If you run the above program with `jsc -d double.js`, JSC will dump all the bytecode it generates to `stderr`. The bytecode dump will contain the bytecode generated for `double`:

```
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4
[   6] check_traps
[   7] add                loc7, arg1, arg1, OperandType
[  13] ret                loc7
```

Each line starts with the offset of the instruction in brackets, followed by the opcode name and its operands. Here we can see the operands `loc` for local variables, `arg` for function arguments and `OperandTypes`, which is metadata about the predicted types of the arguments.

Old Bytecode Format

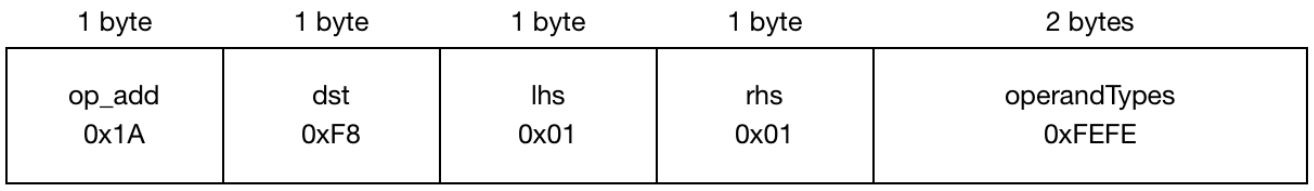
The old bytecode format had a few issues that we wanted to fix:

- It used too much memory.
- The instruction stream was writable, which prevented memory-mapping the bytecode stream.
- It had optimizations that we no longer benefited from, such as direct threading.

In order to better understand how we addressed these issues in the new format, we need a basic understanding of the old bytecode format. In the old format, instructions could be in one of two forms: *unlinked*, which is compact and optimized for storage and *linked*, which is inflated and optimized for execution, containing memory addresses of runtime objects directly in the instruction stream.

Unlinked Instructions

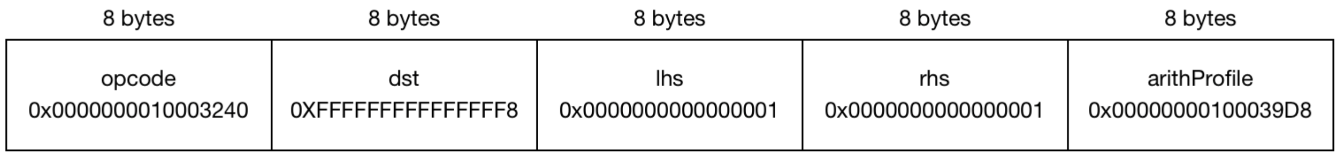
The instructions were encoded using variable-width encoding. The opcode and each operand took as little space as possible, ranging from 1 to 5 bytes. Take the add instruction from the program above as an example, it would take 6 bytes: one for the opcode (add), one for each of the registers (`loc7`, `arg1` and `arg1` again) and two bytes for the operand types.



1 byte	1 byte	1 byte	1 byte	2 bytes
op_add 0x1A	dst 0xF8	lhs 0x01	rhs 0x01	operandTypes 0xFEFE

Linking / Linked Instructions

Before executing the bytecode it needed to be linked. Linking inflated all the instructions, making the opcode and each of the operands pointer-sized. The opcode was replaced by the actual pointer to the implementation of the instruction and the profiling-related metadata replaced with the memory address of the newly allocated profiling data structures. The `add` from above took 40 bytes to represent:



8 bytes	8 bytes	8 bytes	8 bytes	8 bytes
opcode 0x0000000010003240	dst 0XFFFFFFFFFFFFFFF8	lhs 0x0000000000000001	rhs 0x0000000000000001	arithProfile 0x00000000100039D8

Execution

The bytecode is executed by the LLInt, which is written in a portable assembly called *offlineasm*. Here are a few notes about offlineasm that may help understand the code snippets that will follow:

- Temporary registers are `t0–t5`, argument registers are `a0–a3` and return registers are `r0` and `r1`. For floating point, the equivalent registers are `ft0–ft5`, `fa0–fa3` and `fr`. `cfp` and `sp` are special registers that hold the call frame and stack pointer respectively.
- Instructions have one of the following suffixes: `b` for byte, `h` for 16-bit word, `i` for 32-bit word, `q` for 64-bit word and `p` for pointer (either 32- or 64-bit depending on the architecture).
- Macros are lambda expressions that take zero or more arguments and return code. Macros may be named or anonymous and can be passed as arguments to other macros.

If you want to learn more about offlineasm, LowLevelInterpreter.asm is the main offlineasm file in JSC and it contains a more in depth explanation of the language at the top.

The old bytecode format had two big advantages related to interpreter throughput: direct threading and inline caching.

Direct Threading

The linked bytecode had the actual pointer to the offlineasm implementation of the instruction in place of the opcode, which made executing the next instruction as simple as advancing the program counter (PC) by the size of the current instruction and making an indirect jump. That is illustrated in the following offlineasm code:

```
macro dispatch(instructionSize)
    addp instructionSize * PtrSize, PC
    jmp [PC]
end
```

Notice that `dispatch` is a macro, this way the code is duplicated at the end of every instruction. This is very efficient, since it's just an addition plus an indirect branch. The duplication reduces branch prediction pollution since we don't have all instructions jumping to a common label and sharing the same indirect jump to the next instruction.

Inline Caching

Since the instruction stream is writable and all the arguments are pointer-sized, we can store metadata in the instruction stream itself. The best example of this is for the `get_by_id` instruction, which is emitted when we load from an object in JavaScript.

```
object.field
```

This emits a `get_by_id` for loading the property `field` from `object`. Since this is one of the most common operations in JavaScript, it's crucial that it be fast. JSC uses inline caching as a way of speeding this up. The way this was done in the interpreter was by reserving space in the instruction stream to cache metadata about the load. More specifically, we recorded the `StructureID` of the object we are loading from and the memory offset from which we have to load the value. The LLInt implementation of `get_by_id` looked like the following:

```
_llint_op_get_by_id:
    // Read operand 2 from the instruction stream as a
    // i.e. the virtual register of `object`
    loadisFromInstruction(2, t0)

    // Load `object` from the stack and its structureID
    loadConstantOrVariableCell(t0, t3, .opGetByIdSlow)
    loadi JSCell::m_structureID[t3], t1

    // Read the cached StructureID and verify that it m
    loadisFromInstruction(4, t2)
    bineq t2, t1, .opGetByIdSlow

    // Read the PropertyOffset of `field` and load the
    loadisFromInstruction(5, t1)
    loadPropertyAtVariableOffset(t1, t3, t0)

    // Read the virtual register where the result shoul
    // the value to it
    loadisFromInstruction(1, t2)
    storeq t0, [cfr, t2, 8]
    dispatch(constexpr op_get_by_id_length)

.opGetByIdSlow
    // Jump to C++ slow path
    callSlowPath(_llint_slow_path_get_by_id)
    dispatch(constexpr op_get_by_id_length)
```

New Bytecode Format

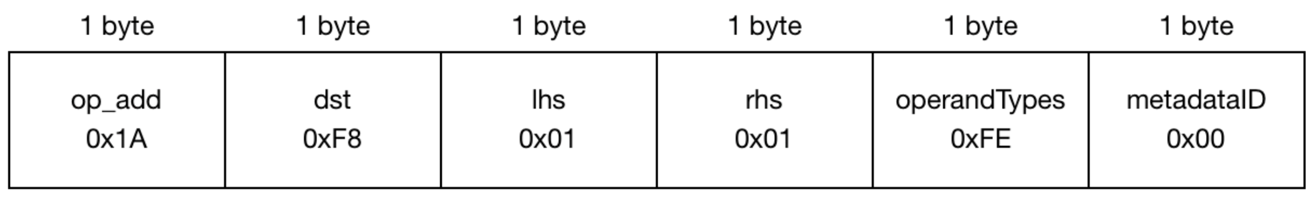
When designing the new bytecode, we had two major goals: it should be more compact and easily cacheable on disk. With these two goals, we expected significant improvements on memory usage and set ourselves to improve runtime performance through caching. This shaped how we encoded instructions as well as runtime metadata.

The first and biggest change is that we no longer have a separate linked encoding for execution. This immediately means that the bytecode can no longer be direct threaded, since the address of the instruction could not be stored to disk, as it changes with every program invocation. Removing this optimization was a deliberate choice of the design.

In order to make the single format suitable for both storage and execution, each instruction can be encoded as either *narrow* or *wide*.

Narrow Instructions

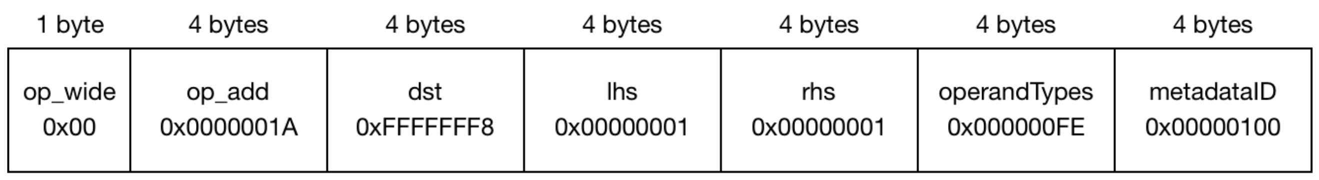
In a narrow instruction, the opcode and its operands each take 1-byte. Here's the `add` instruction again, this time as a narrow instruction in the new format:



Ideally, all instructions would be encoded as narrow, but not all operands fit into a single byte. If any of the operands (or the opcode for that matter) requires more than 1 byte, the whole instruction is promoted to wide.

Wide Instructions

A wide instruction consists of a special 1-byte opcode, `op_wide`, followed by a series of 4-byte slots for the original opcode and each of its arguments.

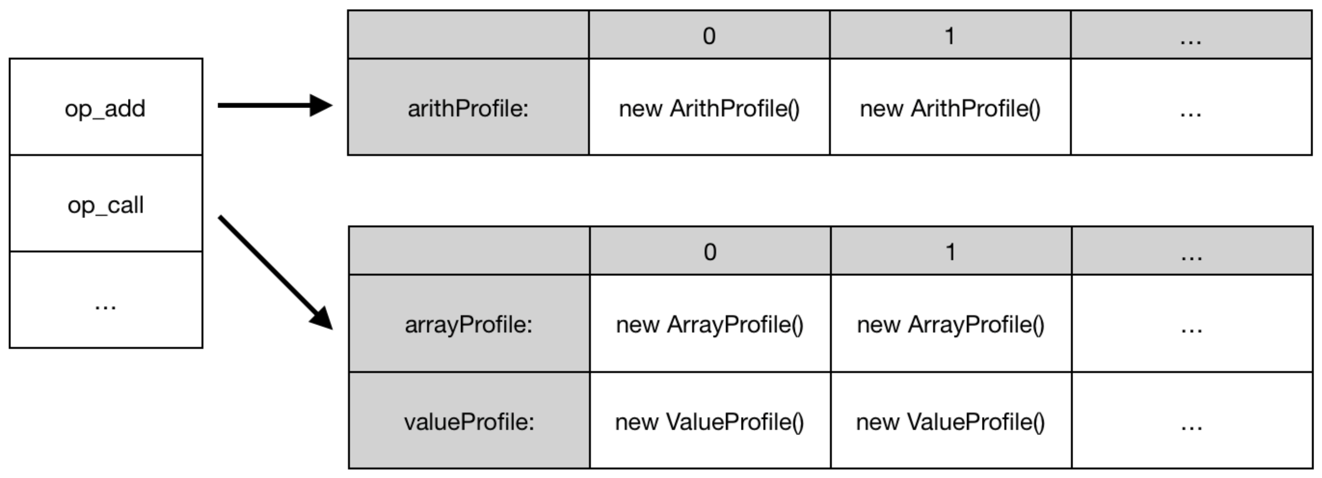


1 byte	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes
op_wide 0x00	op_add 0x0000001A	dst 0xFFFFFFF8	lhs 0x00000001	rhs 0x00000001	operandTypes 0x000000FE	metadataID 0x00000100

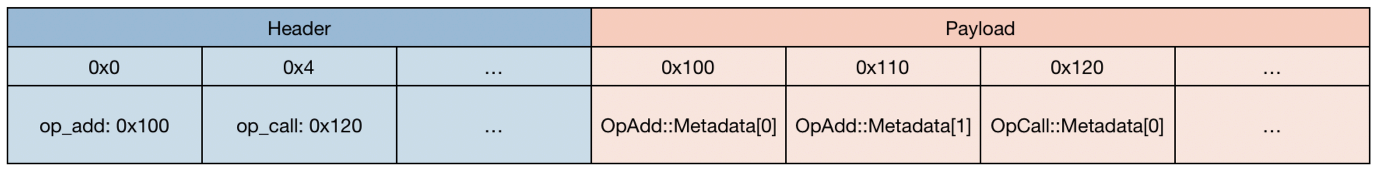
An important thing to notice here is that it is not possible to have a single wide operand – if one operand overflows, the whole instruction becomes wide. This is an important compromise, because even though it might seem wasteful at first, it makes the implementation much simpler: the offset of any given operand is the same regardless of the instruction being narrow or wide. The only difference is whether the instruction stream is treated as an array of 4-byte values or 1-byte values.

Linking / Metadata Table

The other fundamental part of the new bytecode is the metadata table. During linking, instead of constructing a new instruction stream, we initialize a side table with all the writable data associated with any given instruction. Conceptually, the table is 2-dimensional, its first index is the opcode for the instruction, e.g. `add`, which points to a monomorphic array of metadata entries for that specific instruction. For example, we have a struct called `OpAdd::Metadata` to hold the metadata for the `add` instruction, so accessing `metadataTable[op_add]` will result in a pointer of type `OpAdd::Metadata*`. Additionally, each instruction has a special operand, `metadataID`, which is used as the second index in the metadata table.



For compactness, the metadata table is laid out in memory as a single block of memory. Here's a representation of what the table above would actually look like in memory.



Header			Payload			
0x0	0x4	…	0x100	0x110	0x120	…
op_add: 0x100	op_call: 0x120	…	OpAdd::Metadata[0]	OpAdd::Metadata[1]	OpCall::Metadata[0]	…

At the start of the table is the header, an array containing an integer for each opcode representing the offset from the start of the table where the metadata for that opcode starts. The next region is the payload, which contains the actual metadata for each opcode.

Execution

The biggest changes to execution are that the interpreter is now *indirect threaded*, and for any writable or runtime metadata, we need to read from the metadata table. Another interesting aspect is how wide instructions are executed.

Indirect Threading

The process of mapping from opcode to instruction address, which used to be done as part of linking in the old format, is now part of the interpreter dispatch. This means that extra loads are required, and the dispatch macro now looks like the following:

```
macro dispatch(instructionSize)
    addp instructionSize, PC
    loadb [PC], t0
    leap _g_opcodeMap, t1
    jmp [t1, t0, PtrSize]
end
```

Metadata Table

The same kind of "inline caching"† still applies to `get_by_id`, but given that we need to write the metadata at runtime and the instruction stream is now read-only, this data needs to live in the metadata table.

Loads from the metadata table are a bit costly, since the `CodeBlock` needs to be loaded from the call frame, the metadata table from the `CodeBlock`, the array for the current opcode from the table and then finally the metadata entry from the array based on the current instruction's `metadataID`.

† I write "inline caching" in quotes here since it's not technically inline anymore.

Wide Instruction Execution

The execution of wide instructions is surprisingly simple: there are two functions for each instruction, one for the narrow version and another for the wide. We take advantage of *offlineasm* to generate the two functions from a single implementation. (e.g. the implementation of `op_add` will automatically generate its wide version, `op_add_wide`.)

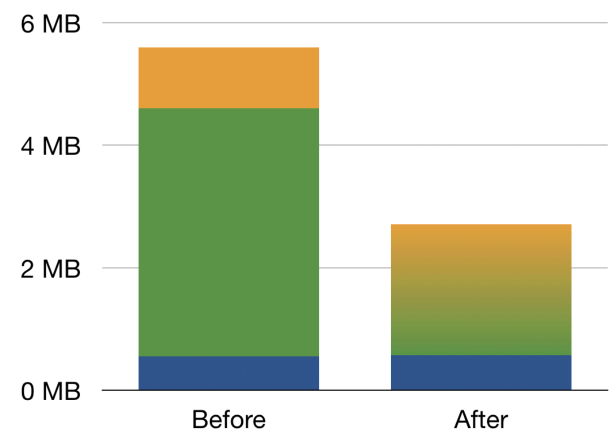
By default, the narrow version of the instruction is executed, until the special instruction `op_wide` is executed. All it does is read the next opcode, and dispatch to its wide version, e.g. `op_add_wide`. Once the wide opcode is done, it goes back to dispatching a narrow instruction, since every wide instruction needs the 1-byte `op_wide` prefix.

Memory Usage

The new bytecode format uses approximately **50%** less memory, which means a reduction of **10%** in overall memory usage for JavaScript-heavy websites, such as Facebook or Reddit.

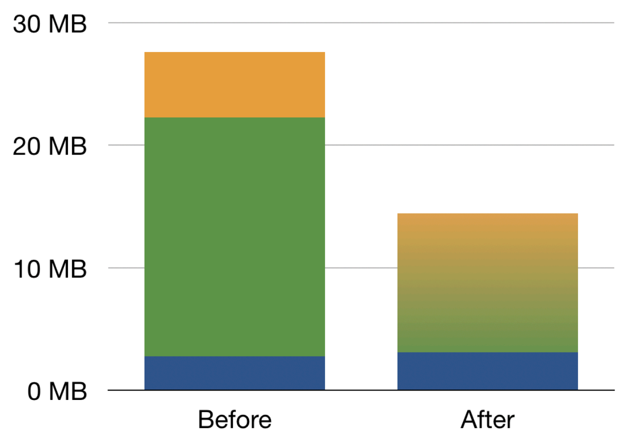
Here’s a chart comparing the **bytecode size** for reddit.com, apple.com, facebook.com and gmail.com.

apple.com



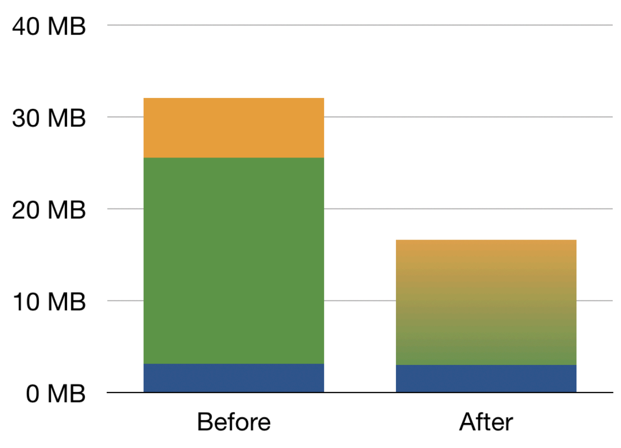
Description	Before	After	%
Unlinked	0.55 MB	0.57 MB	3.6%
Linked	4.05 MB	2.14 MB	-57%
Metadata	0.99 MB		
Total	**5.6 MB**	**2.71 MB**	**-52%**

reddit.com



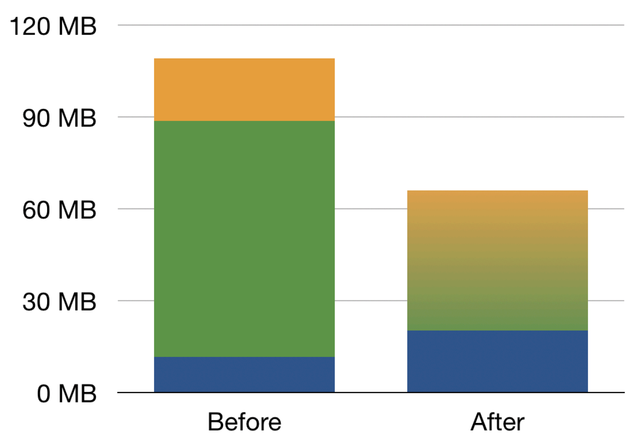
Description	Before	After	%
Unlinked	2.76 MB	3.08 MB	11.8%
Linked	19.51 MB	11.37 MB	-54%
Metadata	5.34 MB		
Total	**27.61 MB**	**14.45 MB**	**-48%**

facebook.com



Description	Before	After	%
Unlinked	3.11 MB	2.99 MB	-3.8%
Linked	22.43 MB	13.66 MB	-52%
Metadata	6.51 MB		
Total	**32.04 MB**	**16.65 MB**	**-48%**

gmail.com



Description	Before	After	%
Unlinked	11.65 MB	20.28 MB	74%
Linked	77.13 MB	45.69 MB	-53%
Metadata	20.33 MB		
Total	**109.12 MB**	**65.96 MB**	**-40%**

Notice that in the *After* case the *Metadata* has been merged with *Linked* to represent the memory used by the metadata table and *Unlinked* represents the actual bytecode instructions. The reason is that a significant part of what lives in the new metadata used to live in the old linked bytecode. Otherwise, the comparison would look skewed, since the whole *Linked* bar would be gone and the *Metadata* would seem to have doubled in size, when in reality it was part of the data from *Linked* that moved into *Metadata*.

Performance

As discussed earlier, indirect threading increases the overhead of the interpreter dispatch. However, given the average complexity of the bytecode instructions in JSC we did not expect the dispatch overhead to play a meaningful role in the overall performance of the interpreter. This is further amortized by the multiple JIT tiers. We profiled purely switching from direct to indirect threading in CLoop, the C++ backend for LLInt, and the results were neutral even with the JITs disabled.

Loads from the metadata table are costly, involving a large chain of loads. In order to reduce this chain, we pin a callee save register in the interpreter to hold the pointer to the metadata table at all times. Even with this optimization, it takes three loads to access the metadata entry. This was a 10% interpreter slowdown, but was neutral when running with all JIT tiers enabled.

Bonus: Type Safety

Changing the bytecode format required changes across the whole engine, so we decided to take this as an opportunity to improve the bytecode-related infrastructure, increasing the type safety, readability and maintainability of the code.

To support all this changes, first we needed to change how instructions were specified. Originally, instructions were declared in a JSON file, with their respective name and length (the number of operands plus one, for the opcode). Here is how the `add` instruction was declared:

```
{ "name": "op_add", "length": 5 }
```

When accessing the instruction and its operands from C++ the code would look roughly like this:

```
SLOW_PATH_DECL(slow_path_add)
{
    JSValue lhs = OP_C(2).jsValue();
    JSValue rhs = OP_C(3).jsValue();
    ...
}
```

Notice that operands were accessed by their offset: `OP_C` was a macro that would access the `Instruction* pc` argument (hidden here by the `SLOW_PATH_DECL` macro) at the given offset. However, the type of the data at the specified offset is unknown, so the result would have to be cast to the desired type. This is error-prone and makes the code harder to understand. The only way to learn which operands an instruction had (and their types) was to look for usages of it and look at variable names and type casts.

With the new infrastructure, when declaring an instruction, a name and type must be given for each of the operands, as well as declaring all the data that will be stored in the metadata table.

```
op :add,
    args: {
        dst: VirtualRegister,
        lhs: VirtualRegister,
        rhs: VirtualRegister,
        operandTypes: OperandTypes,
    },
    metadata: {
        arithProfile: ArithProfile,
    }
```

With this extra information, it is now possible to generate a fully typed `struct` for each instruction, which results in safer C++ code:

```
SLOW_PATH_DECL(slow_path_add)
{
   OpAdd bytecode = pc->as<OpAdd>();
   JSValue lhs = GET_C(bytecode.m_lhs);
   JSValue rhs = GET_C(bytecode.m_rhs);
   ...
}
```

In the example above, we first need to convert the generic `Instruction* pc` to the instruction we want to access, which will perform a runtime check. If the opcode matches, it returns an instance of the generated struct, `OpAdd`, which includes the fields `m_lhs` and `m_rhs`, both of type `VirtualRegister` as specified in our instruction declaration.

The extra information also allowed us to replace a lot of mechanical code with safer, generated code. For example, we auto generate all the type safe code for writing instructions to the instruction stream, which automatically does the narrow/wide fitting. We also generate all the code to dump the instructions for debugging.

Conclusion

JavaScriptCore has a new bytecode format that uses 50% less memory on average and is available on Safari 12.1 and Safari Technology Preview. The work on the caching API for the new bytecode is already underway in the WebKit repository, and anyone interested is welcome to follow along and contribute on bugs.webkit.org! You can also get in touch with me on Twitter with any questions or comments. ■

Next

Release Notes for Safari Technology Preview 86

Learn more

Previously

Release Notes for Safari Technology Preview 85