介绍 B3 JIT 编译器

2016 年 2 月 15 日
作者：Filip Pizlo
@filpizlo

从r195562开始，WebKit 的FTL JIT（超光速即时编译器）现在在 OS X 上使用新的后端。Bare Bones Backend（简称 B3）取代了LLVM，成为 FTL JIT 中的低级优化器。虽然 LLVM 是一个出色的优化器，但它并不是专门为 JavaScript 等动态语言的优化挑战而设计的。我们认为，如果我们将 FTL 架构的最佳部分与新的编译器后端相结合，我们可以获得更大的速度提升，该编译器后端结合了我们从使用 LLVM 中学到的知识，但专门针对我们的需求进行了调整。这篇文章回顾了 FTL JIT 的工作原理，描述了 B3 的动机和设计，并展示了 B3 如何融入 WebKit 的编译器基础架构。

FTL JIT 背后的哲学

FTL JIT 的设计结合了高级优化编译器（用于推断类型并优化类型检查）和低级优化编译器（用于处理寄存器分配等传统优化）。我们使用现有的DFG（数据流图）编译器进行高级优化，并使用 LLVM 进行低级编译器优化。这种架构变化效果很好，它提供了额外的性能提升来优化热代码路径。

高级编译器和低级编译器使用称为 的降低阶段粘合在一起`FTL::LowerDFGToLLVM`。此阶段将低级编译器的中间表示抽象到称为 的接口后面`FTL::Output`。抽象低级编译器意味着我们在选择使用哪种后端时具有一定的灵活性。我们选择 LLVM 作为初始 FTL 实现，因为它是免费提供的，并且在生成高效代码方面表现出色。这意味着 WebKit 可以享受成熟的优化编译器的好处，而无需 WebKit 工程师来实现它。

FTL JIT 的最初作用是为代码提供额外的低级优化，这些代码已经由我们更注重延迟的低级 JIT（如 DFG JIT）进行了足够的高级优

化。但是，随着我们优化 FTL JIT，我们发现自己为其添加了强大的高级优化，而没有在低级 JIT 中实现等效的优化。这是有道理的，因为 FTL 使用低级优化后端意味着高级优化可以更草率，因此运行速度更快，编写速度更快。例如，专门为 FTL 编写的转换可能会留下死代码、无法访问的代码、未融合的加载/比较/分支序列和次优的 SSA 图，因为 LLVM 会清理这些东西。

FTL JIT 不仅仅是其各部分的总和，而且其吞吐量优势的很大一部分与 LLVM 无关。以下是 FTL JIT 执行的优化的不完整列表，但我们的其他 JIT 和 LLVM 都没有这些优化：

- 复杂的要点分析以消除或消除对象分配。
- 代数分析来证明整数范围的界限。

有时，即使 LLVM 也有优化，FTL 也会实现它。LLVM 有以下优化，但我们使用自己的 FTL 实现，因为它们需要了解 JavaScript 语义才能完全有效：

- 全局公共子表达式消除.
- 循环不变代码运动。

FTL 已经变得如此强大，以至于在任何测试中提高 WebKit JavaScript 性能的基本策略都是首先确保测试受益于 FTL。但在某些平台和某些工作负载上，这些工作负载要么总运行时间不够，要么有大量编译缓慢的代码，瓶颈就是 FTL 的编译时间。在某些程序中，我们付出了运行 FTL 编译器线程的代价，但当编译器完成时，程序已经运行完了这些函数。现在是解决编译时间问题的时候了，因为绝大多数（通常四分之三或更多）的 FTL 编译时间是 LLVM，这意味着重新考虑 LLVM 是否应该成为 FTL 后端。

设计新的编译器后端

我们知道，如果我们能够获得 FTL 的所有高级优化，而无需付出 LLVM 编译时间的全部代价，那么我们可以加速大量现实世界的 JavaScript 代码。但我们也知道，LLVM 的低级优化对于运行时间足够长以隐藏编译时间的基准测试至关重要。这是一个巨大的机

会，也是一个巨大的风险：击败 LLVM 的编译时间似乎很容易，但只有当新后端能够在吞吐量上竞争时，这才会是净胜利。

与 LLVM 竞争是一项艰巨的任务。LLVM 经过多年的努力，包括对类似 C 语言的全面优化。正如 FTL JIT 项目之前所展示的那样，LLVM 在优化 JavaScript 程序方面做得非常出色，而且它的编译时间也足够好，因此使用 LLVM 对 WebKit 来说是净胜球。

我们预计，如果我们可以构建一个编译速度比 LLVM 快 10 倍的后端，同时仍生成相当的代码，那么我们将能够加快当前在 LLVM 编译中受阻的代码的速度。机会大于风险。我们于2015 年 10 月下旬开始开发原型。我们的直觉是，我们需要一个在与 LLVM 相同的粒度级别上运行的编译器，以便允许相同类型的优化。为了让编译器运行得更快，我们使用了更紧凑的 IR（中间表示）。我们的计划是不仅通过实现与 LLVM 相同的优化，而且还通过进行 WebKit 特定的调整来匹配 LLVM 的吞吐量。

本节的其余部分描述了 Bare Bones Backend 的架构，特别关注我们计划如何使其实现高吞吐量，同时减少总体 FTL 编译时间。

B3 中级表示

我们知道，我们需要一种低级中间表示，它可以公开原始内存访问、原始函数调用、基本控制流和与目标处理器功能紧密匹配的数值运算。但低级表示意味着需要大量内存来表示每个函数。使用大量内存还意味着编译器在分析函数时必须扫描大量内存，这会导致编译器发生大量缓存未命中。B3 有两种中间表示 - 一种稍高级的表示称为B3 IR，另一种是机器级表示，称为 Assembly IR，简称 Air。这两种 IR 的目标都是表示低级操作，同时最大限度地减少分析代码所需的昂贵内存访问次数。这意味着四种策略：减少 IR 对象的整体大小、减少表示典型操作所需的 IR 对象数量、减少遍历 IR 时对指针追逐的需求以及减少中间表示的总数。

减小红外物体的尺寸

B3 和 Air 旨在使用小对象来描述程序中的操作。B3 通过删除可以延迟重新计算的冗余信息来减小 IR 对象的大小。Air 则更为激进，通过精心调整所有对象的布局来减小对象的大小，即使这意味着使 IR 更难操纵。

B3 IR 使用SSA（静态单次赋值），这意味着每个变量在程序文本中只被赋值一次。这在变量和赋值的语句之间创建了一对一映射。为了紧凑，SSA IR 通常使用一个对象来封装赋值给变量的语句和变量本身。这使得 SSA 看起来很像数据流，尽管 B3 并不是真正的数据流 IR，因为每个语句都锚定在控制流中的特定位置。

在 B3 中，类用于表示变量，而赋值的语句是`Value`。每个值都归某个基本块所有。每个基本块包含要执行的值序列。每个值都有一个对其他值的引用列表（在 B3 中我们称之为子代），这些值代表输入以及该操作码所需的任何元数据。

将子边设为双向边很诱人。这样，找到任何值的所有用途都很便宜。如果您想用其他值替换一个值，这种方法非常棒 — — 这种操作非常常见，以至于 LLVM 使用首字母缩略词“RAUW”（将所有用途替换为）并在其词典中对其进行定义。但双向边的开销很大。边本身必须表示为一个对象，该对象同时包含指向子级和父级（或子级的用户）的指针。子级值列表将成为 Use 对象的数组。这是 B3 IR 避免的内存使用示例之一。在 B3 中，您无法在一个独立操作中将一个值的所有用途替换为另一个值，因为值不知道其父级。这可以为每个值节省大量内存，并使读取整个 IR 便宜得多。B3 优化不是“将所有用途替换为”，而是使用`Value::replaceWithIdentity()`方法将值替换为身份。身份是具有一个子级并只返回该子级的值。这种方法最终会很便宜。我们可以`Identity`就地将一个值更改为另一个值。执行此类替换的大多数转换已经遍历了整个函数，或者`Procedure`用 B3 的话来说。因此，我们只需安排对过程的现有遍历也调用`Value::performSubstitution()`，这将`Identity`使用 的子代替换 的所有使用`Identity`。这让我们鱼与熊掌兼得：B3 阶段变得更快，因为处理一个值所需的内存访问更少，并且这些阶段不会因无法“用 替换所有使用”而获得任何渐近复杂性，因为工作只是纳入它们已经在进行的处理中。

Air 采取了更为极端的方法来减小尺寸。Air 中的基本操作单元是`Inst`。`Inst`是按值传递的。尽管`Inst`可以接受可变数量的参数，但它具有三个内联容量，这是常见情况。因此，基本块通常包含一个连续的内存块，可以完整描述其所有指令。用于表示参数的

Arg 对象完全是就地的。它只是一个紧密打包的位包，用于描述汇编指令将看到的所有类型的参数。

最小化红外物体的数量

最小化表示典型操作所需的对象数量也很重要。我们研究了 FTL 中使用的常见操作。分支到退出的基本块很常见。将常数添加到整数、将整数转换为指针并使用指针进行加载或存储很常见。在移位之前屏蔽移位量很常见。使用溢出检查进行整数数学运算很常见。B3 IR 和 Air 使常见操作仅使用一个 `Value` 或 `Inst` 在常见情况下。分支和退出操作不需要多个基本块，因为 B3 将其封装在 `Check` 操作码中，而 Air 将其封装在特殊 `Branch` 指令中。B3 IR 中的指针只是整数，所有加载/存储操作码都采用可选的偏移立即数，因此典型的 FTL 内存访问只需要一个对象。B3 IR 中的算术运算仔细平衡了现代硬件和现代语言的语义。掩码移位量是 C 语义的遗留，也是关于将 N 位整数移位超过 N 位的含义的古老分歧；现在这些都不再有意义了，因为 X86 和 ARM 都会为您掩码移位量，而所有现代语言都假定移位就是这样进行的。B3 IR 不需要您发出那些毫无意义的掩码，从而节省了内存。带有溢出检查的数学运算在 B3 中使用 `CheckAdd`、`CheckSub` 和操作码，在 Air 中使用、和操作码。这些操作码都理解慢速路径是优化代码的突然终止。这样就无需在每次 JavaScript 程序进行数学运算时创建控制流。`CheckMul` `BranchAdd` `BranchSub` `BranchMul`

减少指针追逐

大多数编译器阶段都必须遍历整个过程。很少会编写只针对单一类型操作码执行操作的编译器阶段。公共子表达式消除、常量传播和强度降低（最常见的优化类型）几乎都处理 IR 中的所有操作码，因此必须访问过程中的所有值或指令。B3 IR 和 Air 经过优化，通过尽可能多地使用数组并减少追逐指针的需要来减少与此类线性扫描相关的内存瓶颈。如果最常见的操作是读取整个内容，则数组往往比链接列表更快。访问数组中彼此相邻的两个值很便宜，因为它们很可能落在同一缓存行上。这意味着只有从数组加载 N 次才会遇到内存延迟，其中 N 是缓存行大小与值大小之间的比率。通常，缓存行很大：32 字节，或 64 位系统上 4 个指针的足够空间，如今往往是最小值。链表没有这个好处。因此，B3 和 Air 将过程表示为基本块数组，并将每个基本块表示为操作数组。在 B3 中，它是一个 `WTF::Vector` 指向值对象的指针数组（具体来说，是一个）。在

Air 中，它是一个 s 数组`Inst`。这意味着 Air 在遍历基本块的常见情况下不需要指针追逐。Air 在优化访问方面必然是最极端的，因为 Air 是较低级别的，因此本质上会看到更多代码。例如，您可以获取立即数的值、临时数或寄存器的类型以及每个参数的元属性（它读取值吗？写入值？它访问多少位？它是什么类型？）而无需读取除 内部包含的内容之外的任何内容`Inst`。

看起来，将基本块构造为数组会阻止将操作插入基本块中间。但是，我们又可以省去这一麻烦，因为大多数转换必须已经处理了整个过程，因为编写一个只会影响一小部分操作的转换的情况非常少见。当转换希望插入指令时，它会建立一个`InsertionSet`（这里是在 B3 和 Air 中）来列出应插入这些值的索引。一旦使用基本块完成转换，它就会执行该集合。这是一个线性时间操作，作为基本块大小的函数，它一次性执行所有插入和所有必要的内存重新分配。

减少中间表示的数量

B3 IR 涵盖了 LLVM IR 的所有用例，并且据我们所知，它不会妨碍我们进行 LLVM 会做的任何优化。它使用更少的内存，处理成本更低，从而缩短了编译时间。

Air 涵盖了 LLVM 低级 IR 的所有用例，例如选择 DAG、机器 SSA 和MC 层。它在一个紧凑的 IR 中实现这些功能。本节讨论了一些使 Air 如此简单的架构选择。

Air 旨在表示所有指令选择决策的最终结果。虽然 Air 确实提供了足够的信息来对其进行转换，但它并不是一个高效的 IR，只能执行特定类型的转换，例如寄存器分配和调用约定降低。因此，将 B3 转换为 Air 的阶段（称为`B3::LowerToAir`）必须巧妙地为每个 B3 操作选择正确的 Air 序列。它通过执行向后贪婪模式匹配来实现这一点。它反向执行每个基本块。当它看到一个值时，它会尝试遍历该值及其子项，甚至可能是其子项的子项（传递地，多层深度）以构造一个`Inst`。与结果匹配的 B3 模式`Inst`将涉及一个根值和一些仅由根使用并且可以作为结果的一部分进行计算的内部`Inst`值。指令选择器锁定这些内部值，从而阻止它们发出自己的指令——再次发出它们是不必要的，因为`Inst`我们选择的已经执行了这些内部值会做的事情。这就是反向选择的原因：我们希望在看到

值本身之前先看到值的用户，以防只有一个用户并且该用户可以将值的计算合并到单个指令中。

指令选择器需要知道哪些 Air 指令可用。这可能因平台而异。Air 提供了一个快速查询 API，用于确定特定指令是否可以在当前目标 CPU 上有效处理。因此，B3 指令选择器会级联所有可能的指令和与之匹配的 B3 值模式，直到找到 Air 认可的指令。这非常快，因为 Air 查询通常被折叠成常量。它也非常强大。我们可以融合涉及加载、存储、立即数、地址计算、比较、分支和基本数学运算的模式，以在 x86 和 ARM 上发出高效指令。这反过来又减少了指令总数，减少了临时变量所需的寄存器数量，从而减少了对堆栈的访问次数。

B3 优化

B3 IR 和 Air 的设计为我们构建高级优化提供了良好的基础。B3 的优化器已经包含了很多优化：

- 强度降低，包括：
  - 控制流图简化。
  - 常量折叠。
  - 积极消除死代码。
  - 整数溢出检查消除。
  - 许多杂项简化规则。
- 流敏感的常数折叠。
- 全局公共子表达式消除。
- 尾部重复。
- SSA 修复。
- 恒定物化的最优放置。

我们还对 Air 进行了优化，例如消除死代码、简化控制流图以及修复部分寄存器停顿和溢出代码问题。Air 中最重要的优化是寄存器分配；这将在后面的部分中详细讨论。

有了 B3，我们还可以灵活地调整任何专门针对 FTL 的优化。下一节将详细介绍我们的一些特定于 FTL 的功能。

调整 B3 以适应 FTL

B3 包含一些受 FTL 启发的功能。由于 FTL 大量使用自定义调用约定、自修改代码片段和堆栈替换，我们知道必须在 B3 中实现这些一流的概念。B3 使用 `Patchpoint` 和 `Check` 系列操作码来处理这些概念。我们知道 FTL 在某些情况下会在同一属性上生成大量重复分支，因此我们确保 B3 可以专门处理这些路径。最后，我们知道我们需要一个出色的寄存器分配器，它开箱即用，不需要进行大量调整。我们的寄存器分配策略与 LLVM 有很大不同。因此，我们有一个部分来描述我们的想法。

B3 补丁点

FTL 大量使用自定义调用约定、自修改代码片段和堆栈替换。虽然 LLVM 确实支持这些功能，但它将它们视为实验性内在函数，这限制了优化。我们从头开始设计 B3 以支持这些概念，因此它们会得到与任何其他类型的操作相同的处理。

LLVM 使用patchpoint 内在函数支持这些功能。它允许 LLVM 的客户端发出自定义机器代码来实现 LLVM 建模的操作（如调用）。B3 使用本机 `Patchpoint` 操作码支持这些功能。B3 对修补的本机支持提供了重要的优化：

- 可以将任意参数固定到任意寄存器或堆栈槽。还可以要求参数获取任意寄存器，或者使用对编译器最方便的任何方式（寄存器、堆栈槽或常量）来表示。
- 还可以限制返回结果的方式。可以将结果固定到任意寄存器或堆栈槽。
- 补丁点不需要具有预定的机器代码大小，这样就无需用无操作来填充补丁点内的代码。LLVM 的补丁点需要预定的机器代码大小。LLVM 的后端会发出那么多字节的无操作来代替补丁点，并向客户端报告这些无操作的位置。B3 的补丁点允许客户端与其自己的代码生成器一起发出代码，从而无需无操作和预定大小。
- B3 的补丁点可以提供有关效果的详细信息，例如补丁点是否可以读取或写入内存、是否会导致突然终止等。如果补丁点确实读取或写入内存，则可以使用与 B3 用于限制加载、存储和调用效果的相同别名语言来提供对访问的内存的界限。LLVM 有一种用于描述堆抽象区域的语言（称为TBAA），但您不能使用它来描述补丁点读取或写入的内存的界限。

补丁点可用于生成任意复杂的可能自修改的代码片段。其他 WebKit JIT 可以访问大量实用程序，用于生成稍后将修补的代码。它通过我们的内部 `MacroAssembler` API 公开。创建补丁点时，您可以为其提供代码生成回调，该回调以 C++ lamda 的形式提供，补丁点随附该回调。一旦 Air 发出补丁点，就会调用 lambda。它被赋予一个 MacroAssembler 引用（具体来说，是一个 CCallHelpers 引用，它使用一些更高级的实用程序增强了汇编程序）和一个对象，该对象描述了 B3 中补丁点的高级操作数与机器位置（如寄存器和堆栈槽）之间的映射。这个对象称为 `StackmapGenerationParams`，还提供有关代码生成状态的其他数据，如使用的寄存器集。B3 和补丁点的生成器之间的契约是：

- 生成器必须生成突然终止或失败的代码。它不能跳转到 B3 发出的其他代码。例如，让一个补丁点跳转到另一个补丁点是错误的。
- 补丁点必须考虑生成器使用对象可以产生哪些类型的效果 `Effects`。例如，补丁点不能访问内存或突然终止，除非效果中注明。任何补丁点的默认效果都声称它可以访问任何内存并可能突然终止。
- 补丁点不得破坏 StackmapGenerationParams 声明正在使用的任何寄存器。也就是说，补丁点可以使用它们，只要它先保存它们并在失败之前恢复它们即可。补丁点可以请求任意数量的临时 GPR 和 FPR，因此无需清除寄存器。

由于 B3 补丁点从来不需要预先调整机器代码片段的大小，因此它们比 LLVM 的补丁点更便宜，并且使用范围更广。由于 B3 可以请求临时寄存器，因此 FTL 不太可能在它发送到补丁点的代码中溢出寄存器。

以下是 B3 客户端如何发出补丁点的示例。这是动态语言的一种经典补丁点：它从内存中加载一个偏移量值，该值可以在编译器完成后进行修补。此示例说明了将可修补代码引入 B3 过程是多么自然。

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

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

    // The generator is where the magic happens. It's
    // during final code generation from Air to machin
    patchpoint->setGenerator(
        [=] (CCallHelpers& jit, const StackmapGenerati
            // The 'jit' will let us emit code inline
            // is generating. The 'params' give us the
            // result (params[0]) and the register use
            // (params[1]). This emits a load instruct
            // 32-bit offset. The 'offsetLabel' is the
            // immediate in the current assembly buffe
            CCallHelpers::DataLabel32 offsetLabel =
                jit.load32WithAddressOffsetPatch(
                    CCallHelpers::Address(params[1].gp

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

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

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

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

补丁点只是 B3 动态语言支持的一半。另一半是Check系列操作码，用于实现栈上替换（简称 OSR）。

B3中的OSR

JavaScript 是一种动态语言。它不具备本机执行速度快的类型和操作。FTL JIT 通过推测这些操作的行为将这些操作转换为快速代码。推测意味着如果遇到某些不良情况，则放弃 FTL JIT 代码。这可确保后续代码不必再次检查此条件。推测使用 OSR 来处理从 FTL JIT 中放弃。OSR 包含两种策略：

- 安排运行时将函数的所有未来执行切换到由非优化（即基线）JIT 编译的版本。当运行时检测到出现了 FTL JIT 未预料到的某些情况时，就会发生这种情况，因此优化的代码不再有效。WebKit 通过重写所有调用链接数据结构以指向基线入口点来实现这一点。如果该函数当前位于堆栈中，WebKit 会在优化函数内的所有无效点上涂写。无效点在生成的代码中是不可见的 - 它们只是可以安全地通过跳转到 OSR 处理程序覆盖机器代码的位置，OSR 处理程序会找出如何重塑堆栈框架以使其看起来像基线堆栈框架，然后跳转到基线代码中的适当位置。

- 发出执行某些检查的代码，如果检查失败，则跳转到 OSR 处理程序。这允许后续代码假设不需要重复检查。例如，FTL 推测整数数学不会溢出，这使得它能够使用整数来表示许多 JavaScript 数字，这些数字在语义上表现得像双精度数。FTL JIT 还会发出许多推测性类型检查，例如快速检查某个值是否真的是具有某些字段集的对象，或者检查某个值是否真的是整数。

可以使用 实现失效点`Patchpoint`。补丁点的生成器不会发出任何代码，只是警告我们的 MacroAssembler 我们想要一个标签，稍后可能会在该标签上写一个跳转。我们的 MacroAssembler 早就支持这一点了。只有当标签太靠近其他控制流时，它才会在此标签上发出跳转大小的无操作。只要没有其他控制流，就不会发出任何内容。补丁点获取 OSR 处理程序所需的所有实时状态。当 OSR 处理程序运行时，它将查询 StackmapGenerationParams 留下的数据，以确定哪些寄存器或堆栈位置包含状态的哪一部分。请注意，这可能在 B3 完成编译后很长时间才会发生，这就是为什么 B3 在名为 的按值对象中表示这些数据的原因`B3::ValueRep`。这与我们在 LLVM 中所做的没什么不同，尽管由于 LLVM 没有使用我们的 MacroAssembler 发出指令，因此我们不得不使用单独的机制来警告我们用跳转重新修补。

对于涉及显式检查的 OSR，`Patchpoint`操作码就足够了，但不一定有效。在 LLVM 中，我们将每个 OSR 出口建模为一个基本块，该块调用补丁点并将所有活动状态作为参数传递。这需要将每个推测检查表示为：

- 正在检查谓词的一个分支。
- 一个基本块，表示检查成功后执行的代码。
- 表示 OSR 退出的基本块。此基本块包含一个补丁点，该补丁点将重建基线堆栈框架所需的所有状态作为其参数。

这准确地代表了推测的含义，但它是通过打破基本块来实现的。每次检查都会将其基本块一分为二，然后引入另一个基本块用于退出。这不仅会增加编译时间。它还会削弱任何围绕控制流保守的优化。我们发现这影响了 LLVM 中的许多方面，包括指令选择和死存储消除。

B3 旨在处理 OSR 而不会破坏基本块。我们使用`Check`操作码来实现这一点。A就像是 a和 a`Check`之间的结合。与 a 一样，它以谓词作为输入。就像它对 所做的那样，指令选择器支持 的比较/分支融合。与 a 一样， a也需要额外的状态和生成器。不是基本块终端。它封装了谓词上的分支，在谓词为真时发出 OSR 退出处理程序，并在谓词为假时转到下一条指令。传递给 的生成器在链接到的分支的线外路径上发出代码。不允许转到或跳回 B3 代码。B3 可以假设转到 a证明谓词为假。

`Branch` `Patchpoint` `Branch` `Branch` `Check` `Patchpoint` `Check` `Check` `C`

我们发现这`Check`是一项必不可少的优化。它大大减少了表示推测操作所需的控制流数量。与 LLVM 一样，B3 也有一些优化，这些优化在实际控制流方面变得保守，但它们处理得`Check`很好。

尾部重复

FTL 生成的代码通常在同一属性上有许多重复的分支。可以使用跳转线程和尾部重复等优化来消除这些分支。尾部重复的功能更强大一些，因为它允许在编译器意识到该专业化将揭示哪些类型的优化之前进行有限量的代码专业化。LLVM 直到后端才进行尾部重复，因此错过了许多它会揭示的高级优化机会。B3 具有尾部重复，添加此优化可使 Octane 几何平均分数提高 2% 以上。LLVM 曾经具有尾部重复，但它在 2011 年被删除。删除时有关的讨论似乎表明它对编译时间产生了不利影响并且对 clang 来说是无利可图的。我们还发现它会稍微缩短编译时间 - 但与 clang 不同，我们看到整体速度有所提升。如果它在我们跟踪的基准测试中能带来净收益，我们可以接受损失一些编译时间。这是我们可以在 B3 中自由进行的特定于 Web 的调整的一个示例。

寄存器分配

寄存器分配仍然是计算机科学中一个蓬勃发展的探索领域。LLVM 在过去五年中对其寄存器分配器进行了彻底改造。GCC八年前对其分配器进行了彻底改造，目前仍在进行另一次彻底改造。

B3 通过 Air 进行寄存器分配。Air 类似于汇编代码，但它允许在目标处理器允许寄存器的任何地方使用临时变量。然后，寄存器分配器的工作就是转换程序，使其引用寄存器和溢出槽而不是临时变量。后面的阶段将溢出槽转换为具体的堆栈地址。

选择寄存器分配器很难。LLVM 和 GCC 的历史似乎表明，编译器最初选择的寄存器分配器很少是最终选择。我们决定使用经典的图形着色寄存器分配器，称为迭代寄存器合并（简称 IRC）。该算法非常简洁，我们过去有实现和调整它的经验。其他分配器（如 LLVM 和 GCC 中使用的分配器）没有如此详细的文档，因此需要更长的时间才能集成到 Air 中。我们的初始实现直接将 IRC 论文中的伪代码转换为真实代码。我们目前的性能结果并未显示 LLVM 和 B3 之间生成的代码质量存在显著差异，这似乎意味着这个寄存器分配器的工作效果与 LLVM 的贪婪寄存器分配器一样好。除了添加一个非常基本的修复阶段并调整实现以减少对哈希表的依赖外，我们没有改变算法。我们喜欢 IRC 能够轻松地模拟我们在补丁点和疯狂的 X86 指令中遇到的各种寄存器约束，而且 IRC 有一个智能解决方案，可以通过合并变量来消除无意义的变量分配，这很棒。IRC 也足够快，不会妨碍我们实现将编译时间相对于 LLVM 缩短一个数量级的目标。本文末尾的性能评估部分更详细地介绍了我们的 IRC 实现和 LLVM 的贪婪寄存器分配器之间的权衡。请参阅此处了解我们的 IRC 实现。

我们计划重新考虑寄存器分配器的选择。我们应该选择在实际 Web 工作负载上性能最佳的寄存器分配器。根据我们迄今为止的性能结果，我们有一个关于在 Air 中尝试 LLVM 的 Greedy 寄存器分配器的未解决错误。

B3 设计概要

B3 旨在实现与我们之前从 LLVM 获得的相同类型的优化，同时在更短的时间内生成代码。它还支持特定于 FTL JIT 的优化，例如精确建模自修改代码的影响。下一节将介绍 B3 如何适应 FTL JIT。

将 B3 JIT 融入 FTL JIT

FTL JIT 的设计初衷是抽象其后端的中间表示。我们这样做是因为 LLVM 的 C API 非常冗长，而且通常需要很多指令才能完成 FTL 认为应该是一条指令的工作，例如从指针和偏移量加载值。使用我们的抽象意味着我们可以让它更简洁。同样的抽象让我们可以“换入”B3 JIT，而无需重写整个 FTL JIT。

FTL JIT 的大部分更改都是以有益的重构形式进行的，这些重构利用了 B3 的补丁点。LLVM 补丁点要求 FTL 拦截 LLVM 的内存分配，以便获取指向称为“堆栈映射”的内存块的指针。获取内存块的方式因平台而异，因为 LLVM 将其部分命名策略与平台使用的动态链接器相匹配。然后，FTL 必须解析二进制堆栈映射块，以生成与 B3 所称的 StackmapGenerationParams 道德等效的面向对象描述：每个补丁点处正在使用的寄存器列表、从高级操作数到寄存器的映射以及发出每个补丁点的机器代码所需的信息。我们本来可以围绕 B3 编写一个层，使用 B3 生成类似 LLVM 的堆栈映射 `Patchpoint`，但我们选择重写使用补丁点的 FTL 部分。使用 B3 的补丁点要容易得多。例如，以下是使用补丁点发出 double-to-int 指令所需的全部代码。B3 本身不支持该指令，因为此类指令的具体行为在不同目标之间有很大差异。

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

请注意，查询用于输入的寄存器 (params[1].fpr()) 和用于输出的寄存器 (params[0].gpr()) 是多么容易。大多数修补点都比这个复杂得多，因此拥有清晰的 API 会使这些修补点更易于管理。

性能结果

在撰写本文时，仍然可以使用编译时标志在 LLVM 和 B3 之间切换。这样就可以对 LLVM 和 B3 进行直接比较。本节总结了使用三

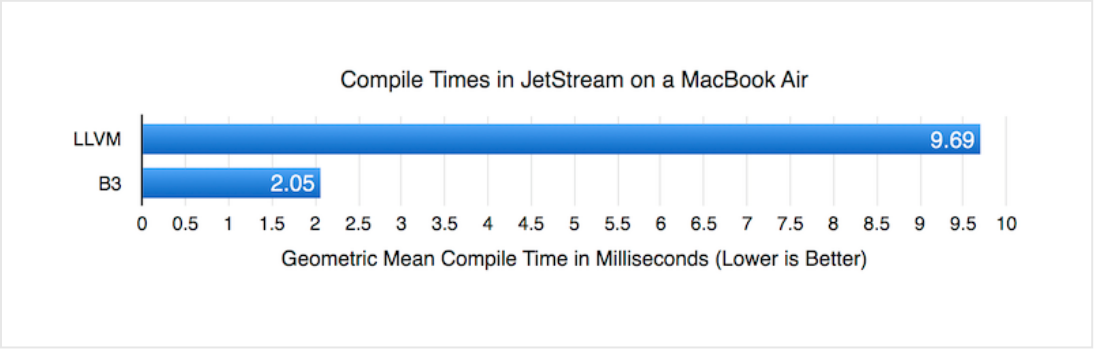
个基准测试套件在两台机器上进行的性能测试结果。

我们在两台不同的 Mac 上进行了测试：一台旧款 Mac Pro，配备 12 个超线程核心（两个 6 核 Xeon）和 32 GB 内存；以及稍新的 MacBook Air，配备 1.3 GHz Core i5 和 8 GB 内存。Mac Pro 有 24 个逻辑 CPU，而 MacBook Air 只有 4 个。测试不同类型的机器很重要，因为可用 CPU 的数量、内存的相对速度以及特定 CPU 型号擅长的方面都会影响最终结果。我们已经看到 WebKit 的更改在一种机器上提高了性能，而在另一种机器上却降低了性能。

我们使用了三种不同的基准测试套件：JetStream、Octane 和 Kraken。JetStream 和 Kraken 之间没有重叠，并且测试不同类型的代码。Kraken 拥有大量运行时间相对较短的数值代码 — 这是 JetStream 所缺少的。Octane 测试的内容与 JetStream 类似，但重点不同。我们认为 WebKit 应该在所有基准测试中都表现良好，因此我们始终将各种基准测试作为实验方法的一部分。

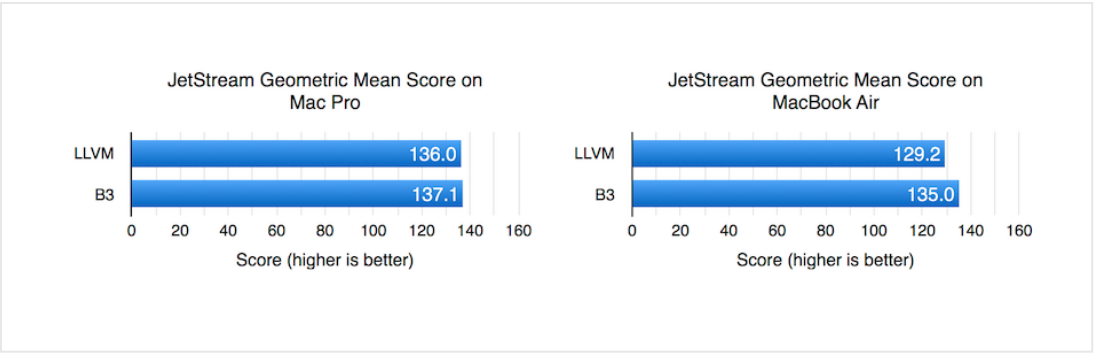
所有测试均将 LLVM 与同一 WebKit 版本 r195946 中的 B3 进行比较。测试在浏览器中运行，以使其尽可能真实。

JetStream 中的编译时间



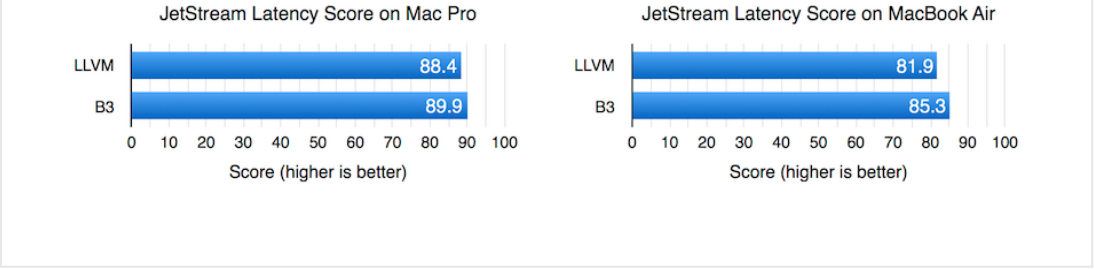
我们对 B3 的假设是，我们可以构建一个减少编译时间的编译器，这将使 FTL JIT 受益于更广泛的应用程序。上图显示了 JetStream 中热门函数的几何平均编译时间。LLVM 的编译时间比 B3 长 4.7 倍。虽然这偏离了我们最初假设的将编译时间缩短 10 倍的目标，但这是一个很大的进步。接下来的部分展示了这种编译时间的减少与 B3 整体良好的代码生成质量相结合如何提高 JetStream 和其他基准测试的速度。

JetStream 性能



在 Mac Pro 上，LLVM 和 B3 之间的性能几乎没有任何差异。但在 MacBook Air 上，B3 总体提速了 4.5%。我们可以通过 JetStream 的延迟和吞吐量组件进一步细分。本节还探讨了寄存器分配性能，并考虑了 JetStream 在各种 LLVM 配置下的性能。

JetStream 延迟

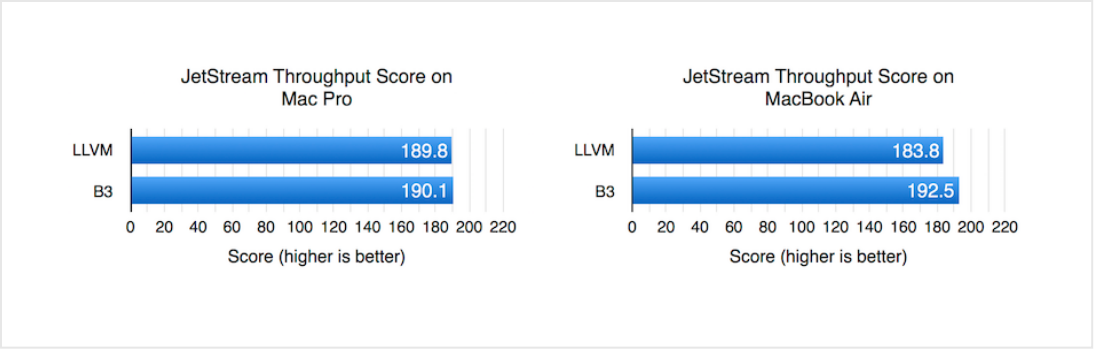


JetStream 的延迟组件有许多运行时间较短的测试。运行时间过长的 JIT 编译器会以多种方式影响延迟分数：

- 它延迟了优化代码的可用性，因此程序会花费额外的时间运行缓慢的代码。
- 它会对内存总线造成压力，并减慢运行基准测试的线程上的内存访问速度。
- 编译器线程并非完全并发。它们有时可能与主线程调度在同一个 CPU 上。它们有时可能会占用导致主线程阻塞的锁。

B3 似乎可以改善 Mac Pro 和 MacBook Air 上的 JetStream 延迟得分，但对 MacBook Air 的影响要大得多。

喷射流吞吐量



JetStream 的吞吐量测试运行时间足够长，因此编译它们所需的时间应该不会有太大影响。但只要测试提供预热周期，持续时间就是固定的，与硬件无关。因此，在较慢的硬件上，我们预计并非所有热代码都会在预热周期完成时被编译。

在 Mac Pro 上，无论我们选择哪个后端，吞吐量得分似乎都相同。但在 MacBook Air 上，B3 显示出 4.7% 的胜利。

Mac Pro 的性能结果似乎表明 LLVM 和 B3 生成的代码质量大致相当，但我们不能确定。虽然可能性不大，但有可能 B3 导致稳定状态性能下降 X%，而编译器延迟减少导致性能加速 X%，两者完全相互抵消。我们认为 LLVM 和 B3 生成的代码质量大致相当的可能性更大。

我们怀疑在 MacBook Air 上，编译时间非常重要。因为许多 JetStream 测试都有 1 秒的预热时间，在此期间不会记录性能，所以速度足够快的计算机（如 Mac Pro）将能够完全隐藏编译时间的影响。只要所有相关的编译都在一秒钟内完成，就没有人会知道它花了整整一秒钟还是只花了一小段时间。但是在速度足够慢的计算机上，或者像 MacBook Air 或 iPhone 这样没有 24 个逻辑 CPU 的计算机上，热函数的编译可能在 1 秒的预热后仍在进行。此时，由于在主线程上运行较慢的代码以及必须与编译器线程争夺内存带宽的复合影响，编译所花费的每一分钟都会从分数中扣除。编译时间减少了 4.7 倍，再加上 MacBook Air 的低并行性，这可能解释了为什么它在 B3 上获得了 JetStream 吞吐量加速，尽管在 Mac Pro 上的分数相同。

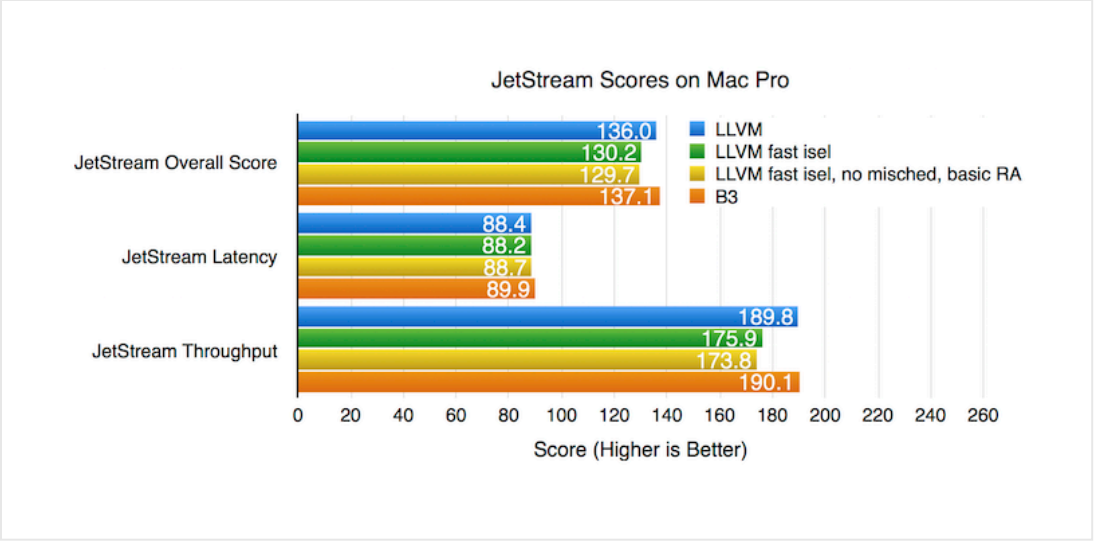
JetStream 中的寄存器分配性能

我们的吞吐量结果似乎表明，B3 和 LLVM 的寄存器分配器生成的代码质量没有显著差异。这引出了一个问题：哪个寄存器分配器最快？我们测量了 JetStream 中寄存器分配所花费的时间。就像我们对编译时间所做的那样，我们关联了 JetStream 所有热函数中的寄存器分配时间，并计算了 LLVM 的 Greedy 寄存器分配器与 B3 的 IRC 所花费时间比率的几何平均值。

LLVM 和 B3 都有昂贵的寄存器分配器，但 LLVM 的贪婪寄存器分配器平均运行速度更快——在 MacBook Air 上，LLVM 时间与 B3 时间的比率约为 0.6。这不足以克服 LLVM 在其他方面的性能缺陷，但它确实表明我们应该在 WebKit 中实现 LLVM 的贪婪分配器，看看这是否会加快速度。我们还可能继续调整我们的 IRC 分配器，因为还有许多改进可以做，比如删除哈希表的剩余用途并切换到更好的数据结构来表示干扰。

这些结果表明，IRC仍然是新编译器的绝佳选择，因为它易于实现，不需要大量调整，并且性能结果可观。值得注意的是，虽然 IRC 大约有 1300 行代码，但如果考虑分配器本身（超过 2000 行）及其支持数据结构的代码，LLVM 的 Greedy 接近 5000 行。

与 LLVM 的快速指令选择器的比较

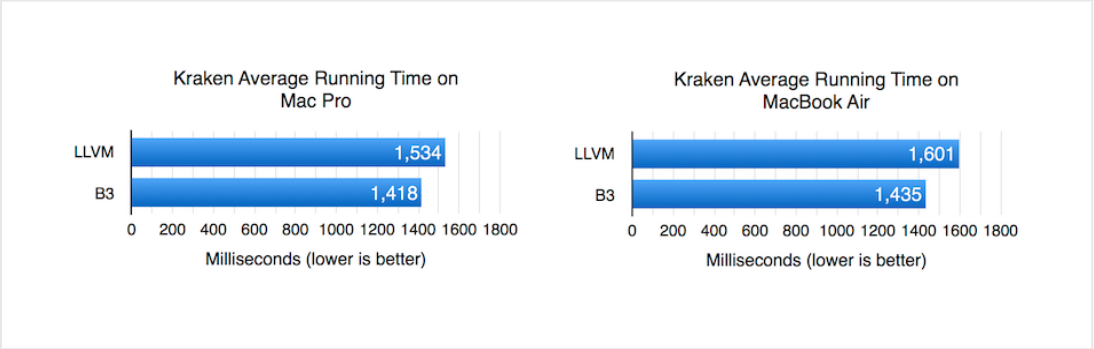


我们最后的 JetStream 比较考虑了 FTL JIT 在“fast isel”配置中运行 LLVM 的能力，我们在 ARM64 上使用了该配置。此配置通过禁用一些 LLVM 优化来减少编译时间：

- 使用快速指令选择器代替选择 DAG 指令选择器。当我们切换到快速 isel 时，我们也始终启用 LowerSwitch 通道，因为快速 isel 无法处理 switch 语句。
- 错误的通行证已被禁用。
- 使用基本寄存器分配器代替贪婪寄存器分配器。

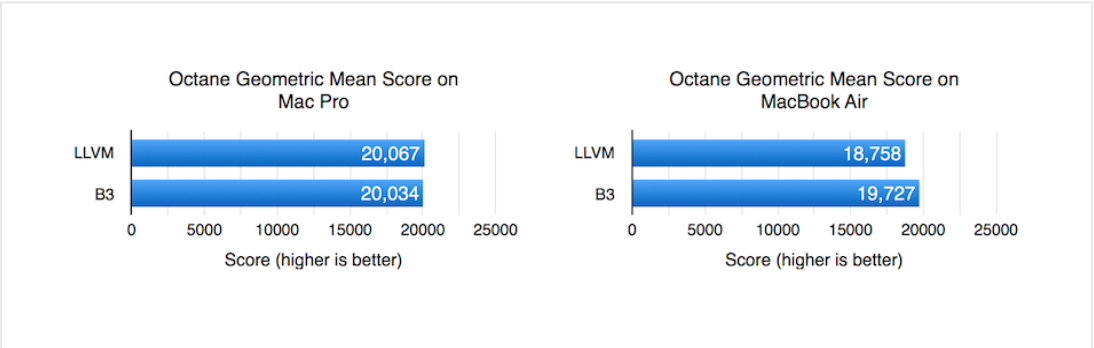
我们测试了仅切换到快速 isel，以及切换到快速 isel 并进行其他两项更改。如上图所示，使用快速 isel 会损害吞吐量：JetStream 吞吐量得分下降 7.9%，导致总体得分下降 4.5%。此外，禁用其他优化似乎会进一步损害吞吐量：吞吐量得分下降 9.1%，总体得分下降 4.9%，尽管这些差异接近误差幅度。B3 能够减少编译时间而不会损害性能：在这台机器上，B3 和 LLVM 的吞吐量得分相差无几。

Kraken 性能



Kraken 基准测试对 FTL JIT 来说是一个挑战，因为它的测试运行时间相对较短，但其中包含大量需要 WebKit 仅在 FTL JIT 中进行的优化的代码。事实上，当我们第一次想到做 B3 时，我们就已经想到了 Kraken。因此，B3 提供加速也就不足为奇了：在 Mac Pro 上加速 8.2%，在 MacBook Air 上加速 11.6%。

辛烷值性能



Octane 与 JetStream 有很多重叠之处：JetStream 大约一半的吞吐量组件来自 Octane。Octane 结果显示，Mac Pro 上没有差异，MacBook Air 上的 B3 速度提高了 5.2%，这与我们在 JetStream 吞吐量中看到的结果一致。

结论

B3 生成的代码在我们尝试的基准测试中与 LLVM 一样好，并且通过减少编译时间使 WebKit 整体运行速度更快。我们很高兴在 FTL JIT 中启用了新的 B3 编译器。拥有自己的低级编译器后端让我们在某些基准测试中获得了立竿见影的提升。我们希望将来它能让我们更好地调整我们的 Web 编译器基础架构。

B3 尚未完成。我们仍需要完成将 B3 移植到 ARM64 的工作。B3 通过了所有测试，但我们尚未完成对 ARM 的性能优化。一旦所有使用 FTL 的平台都切换到 B3，我们计划从 FTL JIT 中删除 LLVM 支持。

如果您对 B3 的更多信息感兴趣，我们计划维护最新的文档— 每次更改 B3 工作方式的提交都必须相应地更改文档。该文档重点介绍 B3 IR，我们希望很快添加Air 文档。 ■

下一个

使用视觉样式侧栏编辑 CSS

了解更多