清 华 大 学

综 合 论 文 训 练

题目：利用 LLVM 加速 x86 到 ARM 的动态二进制翻译机制

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：李祥凡

指导教师：任丰原教授

辅导教师：任丰原教授

2020 年 5 月 27 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：_____ 导师签名：_____ 日　期：_____

中文摘要

使用 Qemu 模拟器在 ARM 架构上运行 x86 程序时, 由于指令翻译造成的性能损失, 导致真实应用几乎不可用。本文阐述了使用 LLVM 的即时编译及代码优化技术, 提高生成机器码的执行效率, 改善 x86 程序在 ARM 架构上的执行性能. 由于时间有限, 目前仅实现了 Qemu 用户态下的优化工作。经过优化, Qemu 模拟器在 ARM 架构上执行 x86 程序的性能得到显著提升, nbench 测试下的 Integer Index 约提升至原来的 5 倍.

关键词：Qemu；LLVM；即时编译；代码优化；x86；ARM

ABSTRACT

When using the Qemu simulator to run x86 programs on the ARM architecture, the performance loss caused by instruction translation makes the real application almost unavailable. This article describes the use of LLVM's just-in-time compilation and code optimization technology to improve the execution efficiency of the generated machine code and improve the execution performance of x86 programs on the ARM architecture. Due to the limited time, only the optimization work in Qemu user mode is currently implemented. After optimization, the performance of the Qemu emulator to execute x86 programs on the ARM architecture has been significantly improved, and the Integer Index under the nbench test has been increased to about 5 times the original.

Keywords: Qemu; LLVM; Just-in-time compilation; Code optimization; x86; ARM

目　录

主要符号对照表

HPC	高性能计算 (High Performance Computing)
cluster	集群
Itanium	安腾
SMP	对称多处理
API	应用程序编程接口
PI	聚酰亚胺
MPI	聚酰亚胺模型化合物，N-苯基邻苯酰亚胺
PBI	聚苯并咪唑
MPBI	聚苯并咪唑模型化合物，N-苯基苯并咪唑
PY	聚吡咙
PMDA-BDA	均苯四酸二酐与联苯四胺合成的聚吡咙薄膜
ΔG	活化自由能 (Activation Free Energy)
χ	传输系数 (Transmission Coefficient)
E	能量
m	质量
c	光速
P	概率
T	时间
v	速度
劝学	君子曰：学不可以已。青，取之于蓝，而青于蓝；冰，水为之，而寒于水。木直中绳。輮以为轮，其曲中规。虽有槁暴，不复挺者，輮使之然也。故木受绳则直，金就砺则利，君子博学而日参省乎己，则知明而行无过矣。吾尝终日而思矣，不如须臾之所学也；吾尝跂而望矣，不如登高之博见也。登高而招，臂非加长也，而见者远；顺风而呼，声非加疾也，而闻者彰。假舆马者，非利足也，而致千里；假舟楫者，非能水也，而绝江河，君子生非异也，善假于物也。积土成山，风雨兴焉；积水成渊，蛟龙生焉；积善成德，而神明自得，圣心备焉。故不积跬步，无以至千里；不积小流，无以成江海。骐骥一跃，不能

十步；驽马十驾，功在不舍。锲而舍之，朽木不折；锲而不舍，金石可镂。蚓无爪牙之利，筋骨之强，上食埃土，下饮黄泉，用心一也。蟹六跪而二螯，非蛇鳝之穴无可寄托者，用心躁也。

——荀况

第 1 章　引言

1.1　QEMU 简介

QEMU 是一个主机上的 VMM（virtual machine monitor）, 通过动态二进制转换来模拟 CPU，并提供一系列的硬件模型，使 guest os 认为自己和硬件直接打交道，其实是同 QEMU 模拟出来的硬件打交道，QEMU 再将这些指令翻译给真正硬件进行操作。

1.1.1　QEMU 指令翻译过程

QEMU 模拟器的指令翻译过程采用 TCG(微型代码生成器), TCG 最初是 C 编译器的通用后端。它被简化为可以在 QEMU 中使用。指令翻译过程主要分为以下两步: 1) 将客户机程序的机器代码反汇编为独立于客户机架构的, 由一系列 TCG ops 组成的中间代码; 2) TCG 吸收中间代码中的 TCG ops, 并对它们进行一些优化, 包括活动性分析（liveness analysis）和琐碎的常量表达式评估（trivial constant expression evaluation). 这些优化用于为客户机虚拟寄存器的分配提供信息, 并尽可能消去冗余的指令, 进行常量传播以减少简化内存操作等, 此后 TCG ops 中的各临时变量被分配给宿主机上的虚拟寄存器结构. 得到最终的 TCG 代码并存储起来. 指令的翻译以 TB(Translation Block) 为单位, 单个 TB 的大小取决于两点: 1) 每个 TB 内最多能声明 512 个临时变量; 2) 每个 TB 最多能有两个跳转出口。最终生成的 TCG 代码描述了一系列客户机虚拟寄存器及内存之间的操作。

1.1.2　QEMU 指令执行过程

QEMU 模拟器的指令翻译过程采用 TCI(微型代码解释器), TCI 模拟了一个虚拟的客户机 CPU, 对指令翻译过程生成的 TCG 代码逐条解释执行, 执行过程中会模拟客户机 CPU, 进行一系列虚拟寄存器和内存之间的操作. 至于各种逻辑与算术运算, TCI 直接用 C 语言的相对应操作进行模拟。

1.1.3　TB chaining

QEMU 模拟器在指令翻译得到 TB 后, 记录下该 TB 的第一条指令的 guest pc 与该 TB 的对应关系, 并存储在多级缓存中。在执行一个 TB 结束后, QEMU 使用模拟的程序计数器 (PC) 和其他的虚拟 CPU 状态信息, 在哈希表中查找下一个 TB

的地址, 若哈希表中查找失败, 进而会在多级缓存中进行查找 **?**。这一查找过程带来的性能开销较大。而 TB chaining 机制有效降低了 TB 查找的开销。TB chaining 是指在某一次执行 TB 后, 记录下该次跳转出口所到达的目的 TB 的 TCG code 地址, 这样使得下一次执行至该 TB, 并从同一跳转出口离开该 TB 时, 无须再根据 pc 查询下一 TB 的地址, 而可以直接跳转至下一 TB 的 TCG code 处开始执行。TB chaining 使得在不考虑 TB 失效的前提下, 对于每个 TB 跳转, 只需要进行 + 一次 TB 查找, 大大减少了在缓存中查询 TB 的次数.

1.2 LLVM 简介

LLVM 是一个针对多种体系结构的编译器框架, 是一个用于构建编译器的工具包, 是将指令转换为计算机可以读取和执行的形式的程序。它會將中間語言（Intermediate Representation，IR）从编译器取出并最佳化，最佳化后的 IR 接着被转换及链接到目标平台的汇编语言。LLVM 可以接受來自 GCC 工具链所编译的 IR，包含它底下现存的编译器。

1.2.1 LLVM IR

LLVM 的一个优势就是提供了统一的内部表示 (LLVM IR), LLVM IR 旨在为编译器的优化器部分提供基础, LLVM IR 是一种抽象的低级指令, 是一个像 RISC 的指令集, 然而可以很好地表达更为高级的信息, 或者说高级语言可以很好地映射到 LLVM IR。LLVM IR 使得编译器的各个前端 (高级语言) 和多个后端 (体系结构) 能够共享中间表示, 共享代码优化器。

1.2.2 LLVM JIT

LLVM 与静态编译不同, 它可以在编译时期, 链接时期, 甚至是执行时期产生可重新定位的机器码。LLVM JIT(Just-in-time) 编译器的目的是在需要时” 即时” 编译代码。具体的说, 程序被人为划分为多个编译模块, 当这个模块中的指令第一次需要被执行时, 才编译整个模块, 即对该模块的 LLVM IR 执行一系列优化并产生机器码。采用 JIT 编译的系统通常会持续分析正在执行的代码, 并识别出从重新编译中获得的执行性能提升能超过编译开销的部分代码。JIT 编译将静态编译的执行效率与解释器的灵活性结合在一起，同时也结合了两者的额外开销。由于加载和编译字节码所花费的时间，JIT 在应用程序的初始执行中引起了轻微到明显的延迟。有时，这种延迟称为“启动时间延迟”或“预热时间”。通常，JIT 执行

的优化越多，它将生成的代码越好，但是初始延迟也会增加。因此，JIT 编译器必须在编译时间和希望生成的代码质量之间进行权衡。

1.3 使用 LLVM 改善 QEMU 指令翻译过程的好处

当前 QEMU 使用翻译得到的 TCG code 作为中间代码, 后端用软件模拟一个虚拟的客户机 CPU, 并由 TCI 解释执行 TCG code. 我的实现采用 LLVM IR 代替 TCG code 作为中间代码, 执行指令时有两种选择:1) 对 IR 进行一系列优化编译得到机器代码, 直接在宿主机上运行; 2) 直接用 LLVM IR 解释器解释执行; 用 LLVM 改善 QEMU 的指令翻译过程, 会带来以下几点好处:

1.3.1 更加丰富的 IR 优化类型

当前的 QEMU 模拟器在得到 TCG 中间代码后, 仅仅进行了以下几项优化: 活动性分析, 常量传播, 冗余指令/变量消除等, 然后将变量分配给虚拟 CPU 中的虚拟寄存器。而使用 LLVM IR 作中间代码, 可以执行种类更加繁多的优化, 能更大程度的减小最终生成代码的体积, 获取更佳的执行性能。

1.3.2 针对不同的体系结构生成机器代码

由于每个 TB 实际上只会包含最多几百条指令, LLVM 提供的种类繁多的复杂的 IR 优化, 相较于 TCG 优化而言, 实际上能额外提供的优化限度相对有限。使用 LLVM 进行翻译过程优化的真正优势在于, LLVM 的代码生成器能根据宿主机的体系结构, 将 LLVM IR 编译生成最适合与在宿主机上运行的机器码。

当前 QEMU 模拟器使用一个虚拟宿主机 CPU 逐条解释执行 TCG code 中的指令, 用软件模拟真实 CPU 带来的性能开销是巨大的。例如, QEMU 在指令执行过程对虚拟寄存器的访问实际上是对栈上的内存空间的访问, 这与真实的硬件寄存器的访问速度有一定差距。另外, TCG code 的生成是独立于宿主机架构的, 即 TCG 无法得知宿主机的架构, 同一个客户机程序在不同的宿主机架构上都会得到相同的 TCG code，虚拟 CPU 相当于运行在宿主机上的一个用户程序, 这样无法充分利用不同体系结构的特点, 充分利用架构特点得到适合的代码。而使用 LLVM 的代码生成器可以根据特定宿主机的架构 (内存布局, 指令集, 寄存器等), 将 LLVM IR 编译为适合于在宿主机运行的机器码, 并直接由宿主机 CPU 执行，充分利用硬件优势。这与 QEMU 模拟器的后端逐条解释执行独立于宿主机体系结构的 TCG code 相比, 性能上有着巨大优势。

1.3.3 即时编译

LLVM 的即时编译机制与 QEMU 原本的动态翻译思想可以很好的契合, TCG 翻译的单位-——TB 可以作为 JIT 的编译模块, JIT 在需要时执行一个 TB 的编译。

综上所述, 我们有理由相信, 使用 LLVM 对 QEMU 模拟器进行优化, 即使用 LLVM IR 代替 TCG code 作为中间代码, 并用 LLVM JIT 代替虚拟 CPU 进行后端机器码的执行, 会使得 QEMU 模拟器的性能得到提升。

第 2 章　实验内容

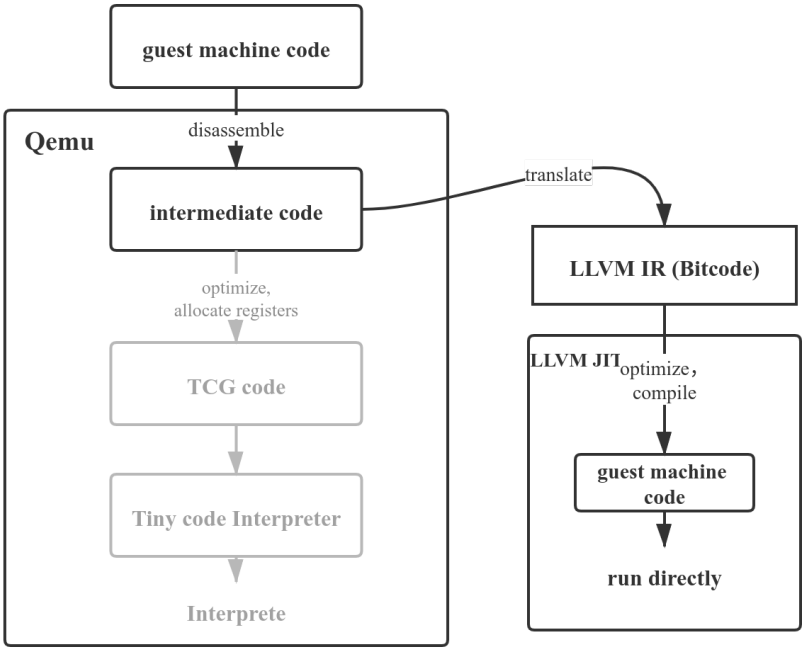

图 2.1　优化后的 QEMU 指令翻译, 执行过程粗略图

LLVM 优化后的 QEMU 模拟器的指令翻译，执行过程如图 2.1 所示

2.1　指令翻译得到 LLVM IR

如图 2.1 所示, 在我的项目实现中，LLVM IR 由客户程序的机器码反汇编得到的 TCG 中间代码翻译而得到。实际上 LLVM IR 也可以由机器码直接翻译得到, 但综合考虑代码实现的简便性, 代码的可读性, 以及后来的实验结果表明, 代码执行阶段带来的性能提升远远超过翻译时期的开销改变, 于是采用了最终的实现方案, 即直接沿用了 QEMU 模拟器原有的反汇编过程。TCG 中间代码包含了一系列 TCG ops, 我手写代码完成了各类 TCG ops 向 LLVM IR 指令的转化工作。具体来说, 由机器码反汇编得到的 TCG 中间代码描述了一系列临时变量 (最多达 512 个) 之间的操作以及一些内存操作。我使用了一个包含 512 个 LLVM 临时变量的数组来存储执行时临时变量的值。内存操作通过使用一些辅助函数, 并绑定到相对应

的 LLVM 函数来实现。这样, 每个 TB 的 TCG 中间代码, 都会逐条指令被翻译为 LLVM 指令, 并得到一个 LLVM 函数。当需要执行某个 TB 时, 就通过 LLVM 的执行引擎执行相对应的 LLVM 函数即可, TB 对应的 LLVM 函数的返回值包含有最后执行到的 TB 所在地址以及跳转出口 (0 或 1)。

2.2 指令的执行

在翻译得到 LLVM IR 后, 就到了指令的执行阶段。LLVM 对指令的执行提供了两种类型的执行引擎: JIT Compiler(即时编译器) 和 Interpreter(解释器)。JIT Compiler 采用即时编译, 即在某个编译模块中的任意函数第一次被执行时, 才编译整个模块, 进行 IR 优化并生成机器代码, 此后该编译模块中的函数被执行时, 只需要执行相应的机器码即可; Interpreter 则不编译生成机器码, 而是逐条指令解释执行 LLVM IR, 也不负责进行任何 IR 优化。

表 2.1　使用不同的执行引擎, 某一 TB 先后被执行多次的时间开销 (us)

执行轮次 / 执行引擎	1st	2st	3rd	4th	5th	6th	7th
JIT Compiler	7452.01	0.285	0.294	0.257	0.227	0.283	0.207
Interpreter	70.748	66.999	66.959	67.504	66.727	66.399	66.828
regular qemu	0.779	0.539	0.491	0.553	0.430	0.562	0.490

表 2.1 是分别使用 LLVM 提供的 JIT Compiler, Interpreter 以及常规 QEMU 模拟器的 TCI 解释器作执行引擎, 执行某一测试程序时, 某一会被反复执行的特定 TB 在前 7 次被执行时的时间开销。通过表中数据可以得知, 在代码执行性能方面, LLVM 提供的 JIT Compiler 在第一次执行时的编译过程会造成较大的延时, 但由此带来的好处是后续执行用时的大幅度减少, 编译得到的机器码运行耗时约为常规 QEMU 使用 TCI 解释执行的 50%。而 LLVM 提供的 Interpreter 对 IR 解释执行的效率与常规 QEMU 模拟器相比, 仍有较大差距。至此, 我们可以确定, 使用 LLVM 提供的 JIT 编译器发射得到宿主机上的机器码, 相较于常规的 QEMU 模拟器, 可以通过牺牲第一次编译的时间成本, 换取执行时显著的性能提升。对于频繁执行的 TB, 这样的牺牲是值得的, 这可以使 QEMU 模拟器的整体性能得到明显改善。

2.3 性能评估

在项目实现过程中, 我尝试了多种不同的代码执行策略, 目的是尽可能提高 QEMU 模拟器的执行性能。为了定量地评估并比较各种实现策略下的 QEMU 模拟器的性能和相较常规 QEMU 模拟器的性能变化, 我使用 NBench 作为基准程序对优化后的 QEMU 进行测试。NBench(Native mode Benchmark) 是一种合成计算基准程序, 开发于 1990 年代中期, 旨在测量计算机的 CPU，FPU 和内存系统速度。我使用改进后的 qemu 模拟器在用户态下执行另一平台架构的 NBench 程序, NBench 程序会执行一系列计算任务, 并给出 CPU(qemu 模拟器模拟的) 的性能评分, 我将主要关注评测结果中的 Integer Index 指标。Integer Index 是一系列仅涉及整数处理的的测试结果的几何平均值, 包括数字排序, 字符串排序, 位域操作, 模拟浮点, 任务分配, 霍夫曼编码, IDEA (国际数据加密算法)。该指数指出了被评测系统相较于 90MHz 奔腾 Intel CPU 的基准系统的相对性能分数，从而可以总体了解被测机器的性能。

2.3.1 仅使用 JIT Compiler 作为执行引擎

起初, 我仅使用 JIT Compiler 作为 QEMU 的执行引擎, 即对于翻译得到的每一个 TB, 都在第一次执行时将 LLVM IR 编译为机器代码, 然后执行这一段机器代码。这样的执行策略的缺点在于, 对于那些执行频次较低的 TB 而言, 耗时的 IR 优化以及编译过程带来的执行期性能提升将是得不偿失的。LLVM 提供的 JIT Compiler 可以被设置为四种不同的优化级别, 用数字 0-3 表示, 对应与即时编译时对 IR 的不同优化力度, 优化级别为 0, 表示不对 IR 执行任何优化, 3 表示执行最高级别的优化。

如表 2.2 所示, 我统计了仅使用 LLVM JIT Compiler 作为执行引擎, 并设置不同的优化级别时, 使用 NBench 测试所得到的 Integer Index 数值, 并与常规的 QEMU 模拟器进行了比较。由于此时我并未实现 TB chaining 机制, 出于性能比较的公平性, 在对比时我禁用了常规 QEMU 模拟器的 TB chaining 机制。

从表中数据分析可知, 使用 LLVM JIT Compiler 作执行引擎, 即使不对 IR 执行任何优化 (opt-level=0), 模拟 CPU 的性能也显著优于常规的 QEMU 模拟器, 在对 LLVM IR 进行一些优化 (opt-level>0) 后, 模拟 CPU 的性能进一步得到显著提升, 在优化级别为 2 时, 评测得到的 Integer Index 数值最高。当优化级别由 2 升高至 3 后, 性能评分反而略微降低。这一现象可以理解为过高的优化级别使得 IR 优化力度过大, 这带来的代码执行性能上的提升不足以弥补优化, 编译期的额外开

表 2.2　使用 JIT 编译器作执行引擎, 并设置不同的代码优化级别得到的 NBench 评测结果

实现	Integer Index
regular qemu, no tb chaining	0.451
llvm-qemu, opt-level = 0	0.971
llvm-qemu, opt-level = 1	1.744
llvm-qemu, opt-level = 2	1.756
llvm-qemu, opt-level = 3	1.696

销。实际上, JIT Compiler 的默认优化级别为 2, 将优化级别设置为 3 是更激进的做法, 并不是普适的选择。在实验的余下部分, 我都将 JIT Compiler 的优化级别设置为 2(默认的优化级别)。

2.3.2　将 Interpreter 与 JIT Compiler 结合使用

为了克服仅使用 JIT Compiler 作为执行引擎的缺点, 我实现了一种将 Interpreter 和 JIT Compiler 结合起来使用的简单的执行策略: 设置一个执行次数的阈值, 并为每一个 TB 设置一个被执行次数的计数器, 当需要执行一个 TB 时, 若该 TB 的被执行总次数未超过设定的阈值, 则使用 LLVM 提供的 Interpreter 直接逐指令地对该 TB 的 LLVM IR 解释执行，直到执行次数达到阈值时, 才使用 JIT Compiler 进行 IR 优化并生成机器码, 此后对该 TB 的执行都会直接运行机器码。我将执行频次达到阈值的 TB 称为热点 (hotspot)TB, 这样的策略能优先对那些被认为会被多次执行的 TB 执行优化, 而对于那些执行频次低的 TB, 执行期的性能提升不再那么重要, 使用 Interpreter 执行 IR 能避免编译带来的巨大开销。而根据著名的二八定律, 计算机在 80% 的时间里执行 20% 的常用代码, 这样的只对热点 TB 进行优化并编译的策略是具有较高的实际意义的。

表 2.2 中展示了将执行频次阈值设置为 5, 并对热点 TB 执行 JIT 编译, 非热点 TB 使用解释器解释执行, 带来的执行性能变化, 可见 Integer Index 相较于仅使用 JIT 编译时有略微提高。我使用的 hotspot 判定规则较为简单, 若结合实际执行情况使用更加复杂且合理的策略, 相信能得到更好的结果。

表 2.3　对热点 TB 使用 JIT 编译, 非热点 TB 使用解释器解释执行, 得到的 NBench 评测结果

实现	Integer Index
regular qemu, no tb chaining	0.451
llvm-qemu, opt-level = 2	1.756
llvm-qemu, opt-level = 2, hotspot(threshold=5)	1.804

2.3.3　TB chaining 的实现

由于在执行 TB 之前, 根据 guest PC 在多级缓存中查找对应 TB 地址的这一过程的性能开销可能较大, 所以我在我的项目实现中也使用了 TB chaining 机制, 实现的原理与常规 QEMU 中的 TB chaining 相似, 区别之处在于, 常规 QEMU 在执行过程中需要记录下跳转的目的 TB 的 TCG code 的地址相对于上一 TB 的 TCG code 的相应跳转出口地址的偏移, 在此后进行同一 TB 跳转时, 只需要在当前 TCG code 指针上加上这一偏移量, 就能接着执行下一个 TB 的 TCG code; 而由于 LLVM IR 在优化并编译为机器码后, 将难以定位 LLVM IR 中的特定指令, 这使得在 TB 对应的机器码之间执行跳转是不现实的, 而且在机器码层次执行跳转, 将会违背将 TB 作为执行单元的原则, 难以将每一 TB 作为独立个体分别执行优化力度的控制等, 于是我的方案是对于每一个 TB, 在执行过程中记录下每个跳转出口跳转至的 TB 的地址 (即将 TB 链接起来), 在执行 TB 时, 将顺着这一链接关系一路执行下去, 直到无法通过链接关系定位下一 TB 为止。

表 2.3 比较了使用 LLVM 优化的 qemu 模拟器在实现 TB chaining 前后的 Nbench 评测结果。可以看到性能显著提升, Integer Index 相较未实现 TB chaining 时提升了 68.7%。

在进行了上述的优化和改进后, 最终得到的项目, 相较于常规的 QEMU 模拟器, 在 64 位 ARM 架构上运行 x86 程序的性能得到了大幅度提升, NBench 下的评测结果显示, Integer Index 增长为原来的 5.62 倍。

表 2.4　实现 TB chaining 机制后, 得到的 NBench 评测结果

实现	Integer Index
regular qemu, tb chaining implemented	0.542
llvm-qemu, opt-level = 2, hotspot(threshold=5), no TB chaining implemented	1.804
llvm-qemu, opt-level = 2, hotspot(threshold=5), TB chaining implemented	3.044

2.4　实验总结

我认为该项目的实现有着重要的实际意义, 由于 LLVM 在代码优化上的强大能力以及它能根据特定的机器架构执行优化这一特点, 使用 LLVM 改善 QEMU 模拟器的指令翻译机制, 对于模拟器的性能提升是有很大潜力。我的实验证实了使用 LLVM 提供的工具链改善 QEMU 性能的可行性, 并为优化后的性能提升提供了数据支持, 对今后可能开展的使用 LLVM 对 QEMU 模拟器进行优化的工作有一定的借鉴意义。通过牺牲一定的编译时性能开销, 换取执行时性能的大幅度提升, 这使得 x86 真实应用在 ARM 架构机器上的可用性得到提升。目前还存在一些值得展开的后续工作: 首先, 由于本人的能力和时间有限, 目前仅实现了 QEMU 用户态下的优化工作, 全系统模拟尚未实现成功, 将优化工作扩展到全系统态, 将使得在 ARM 架构上完全模拟 x86 时的性能得到明显提升; 我在对 LLVM IR 进行优化并编译生成机器码时, 采用了统一的优化级别, 更加合理的做法应该是根据 TB 的具体执行情况 (被执行次数等), 动态地调节对 IR 的优化力度, 这样能更充分合理地挖掘优化带来的性能提升潜力; 同时, 对热点 TB 的判定可以采用更加复杂, 合理的策略, 而不是简单地依据 TB 的总被执行次数。这些都有待于在后续工作中加以改进完善。

插图索引

表格索引

公式索引

致　谢

衷心感谢导师 xxx 教授和物理系 xxx 副教授对本人的精心指导。他们的言传身教将使我终生受益。

在美国麻省理工学院化学系进行九个月的合作研究期间，承蒙 xxx 教授热心指导与帮助，不胜感激。感谢 xx 实验室主任 xx 教授，以及实验室全体老师和同学们的热情帮助和支持！本课题承蒙国家自然科学基金资助，特此致谢。

感谢 LaTeX 和 ThuThesis?，帮我节省了不少时间。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

附录 A　单目标规划

As one of the most widely used techniques in operations research, *mathematical programming* is defined as a means of maximizing a quantity known as *bjective function*, subject to a set of constraints represented by equations and inequalities. Some known subtopics of mathematical programming are linear programming, nonlinear programming, multiobjective programming, goal programming, dynamic programming, and multilevel programming.

It is impossible to cover in a single chapter every concept of mathematical programming. This chapter introduces only the basic concepts and techniques of mathematical programming such that readers gain an understanding of them throughout the book.

A.1　Single-Objective Programming

The general form of single-objective programming (SOP) is written as follows,

$$\begin{cases} \max \ \ f(x) \\ \text{subject to:} \\ \qquad g_j(x) \leqslant 0, \quad j = 1, 2, \cdots, p \end{cases}$$

which maximizes a real-valued function f of $x = (x_1, x_2, \cdots, x_n)$ subject to a set of constraints.

Definition A.1： In SOP, we call x a decision vector, and $x_1, x_2, \cdots, x_n$ decision variables. The function f is called the objective function. The set

$$S = \left\{ x \in \mathbf{R}^n \;\middle|\; g_j(x) \leqslant 0,\ j = 1, 2, \cdots, p \right\}$$

is called the feasible set. An element x in S is called a feasible solution.

Definition A.2： A feasible solution x^* is called the optimal solution of SOP if and only if

$$f(x^*) \geqslant f(x) \tag{A-1}$$

for any feasible solution x.

One of the outstanding contributions to mathematical programming was known as the Kuhn-Tucker conditions**??**. In order to introduce them, let us give some definitions. An inequality constraint $g_j(x) \leqslant 0$ is said to be active at a point x^* if $g_j(x^*) = 0$. A point x^* satisfying $g_j(x^*) \leqslant 0$ is said to be regular if the gradient vectors $\nabla g_j(x)$ of all active constraints are linearly independent.

Let x^* be a regular point of the constraints of SOP and assume that all the functions $f(x)$ and $g_j(x), j = 1, 2, \cdots, p$ are differentiable. If x^* is a local optimal solution, then there exist Lagrange multipliers $\lambda_j, j = 1, 2, \cdots, p$ such that the following Kuhn-Tucker conditions hold,

$$\begin{cases} \nabla f(x^*) - \sum_{j=1}^{p} \lambda_j \nabla g_j(x^*) = 0 \\ \lambda_j g_j(x^*) = 0, \quad j = 1, 2, \cdots, p \\ \lambda_j \geqslant 0, \quad j = 1, 2, \cdots, p. \end{cases} \tag{A-2}$$

If all the functions $f(x)$ and $g_j(x), j = 1, 2, \cdots, p$ are convex and differentiable, and the point x^* satisfies the Kuhn-Tucker conditions (**??**), then it has been proved that the point x^* is a global optimal solution of SOP.

A.1.1 Linear Programming

If the functions $f(x), g_j(x), j = 1, 2, \cdots, p$ are all linear, then SOP is called a *linear programming*.

The feasible set of linear is always convex. A point x is called an extreme point of convex set S if $x \in S$ and x cannot be expressed as a convex combination of two points in S. It has been shown that the optimal solution to linear programming corresponds to an extreme point of its feasible set provided that the feasible set S is bounded. This fact is the basis of the *simplex algorithm* which was developed by Dantzig as a very efficient method for solving linear programming.

Roughly speaking, the simplex algorithm examines only the extreme points of the feasible set, rather than all feasible points. At first, the simplex algorithm selects an extreme point as the initial point. The successive extreme point is selected so as to improve the objective function value. The procedure is repeated until no improvement in objective function value can be made. The last extreme point is the optimal solution.

Table 1　This is an example for manually numbered table, which would not appear in the list of tables

Network Topology		# of nodes	# of clients			Server
GT-ITM	Waxman Transit-Stub	600	2%	10%	50%	Max. Connectivity
Inet-2.1		6000				
Xue	Rui	Ni	ThuThesis			
	ABCDEF					

A.1.2 Nonlinear Programming

If at least one of the functions $f(x), g_j(x), j = 1, 2, \cdots, p$ is nonlinear, then SOP is called a *nonlinear programming*.

A large number of classical optimization methods have been developed to treat special-structural nonlinear programming based on the mathematical theory concerned with analyzing the structure of problems.

Figure 1　This is an example for manually numbered figure, which would not appear in the list of figures

Now we consider a nonlinear programming which is confronted solely with maximizing a real-valued function with domain $\mathbf{R}^n$. Whether derivatives are available or not, the usual strategy is first to select a point in $\mathbf{R}^n$ which is thought to be the most likely place where the maximum exists. If there is no information available on which to base such a selection, a point is chosen at random. From this first point an attempt is made to construct a sequence of points, each of which yields an improved objective function value over its predecessor. The next point to be added to the sequence is chosen by analyzing the behavior of the function at the previous points. This construction continues until some termination criterion is met. Methods based upon this strategy are called *ascent methods*, which can be classified as *direct methods*, *gradient methods*, and *Hessian methods* according to the information about the behavior of objective function f. Direct methods require only that the function can be evaluated at each point. Gradient methods require the evaluation of first derivatives of f. Hessian methods require the evaluation of

second derivatives. In fact, there is no superior method for all problems. The efficiency of a method is very much dependent upon the objective function.

A.1.3 Integer Programming

Integer programming is a special mathematical programming in which all of the variables are assumed to be only integer values. When there are not only integer variables but also conventional continuous variables, we call it *mixed integer programming*. If all the variables are assumed either 0 or 1, then the problem is termed a *zero-one programming*. Although integer programming can be solved by an *exhaustive enumeration* theoretically, it is impractical to solve realistically sized integer programming problems. The most successful algorithm so far found to solve integer programming is called the *branch-and-bound enumeration* developed by Balas (1965) and Dakin (1965). The other technique to integer programming is the *cutting plane method* developed by Gomory (1959).

Uncertain Programming (*BaoDing Liu, 2006.2*)

A.2 单目标规划

北冥有鱼，其名为鲲。鲲之大，不知其几千里也。化而为鸟，其名为鹏。鹏之背，不知其几千里也。怒而飞，其翼若垂天之云。是鸟也，海运则将徙于南冥。南冥者，天池也。

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}, y)}{p(\mathbf{x})} = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} \tag{123}$$

吾生也有涯，而知也无涯。以有涯随无涯，殆已！已而为知者，殆而已矣！为善无近名，为恶无近刑，缘督以为经，可以保身，可以全生，可以养亲，可以尽年。

A.2.1 线性规划

庖丁为文惠君解牛，手之所触，肩之所倚，足之所履，膝之所倚，砉然响然，奏刀騞然，莫不中音，合于桑林之舞，乃中经首之会。

文惠君曰：“嘻，善哉！技盖至此乎？”庖丁释刀对曰：“臣之所好者道也，进乎技矣。始臣之解牛之时，所见无非全牛者；三年之后，未尝见全牛也；方今之时，臣以神遇而不以目视，官知止而神欲行。依乎天理，批大郤，导大窾，因其

表 1　这是手动编号但不出现在索引中的一个表格例子

Network Topology		# of nodes	# of clients			Server
GT-ITM	Waxman Transit-Stub	600	2%	10%	50%	Max. Connectivity
Inet-2.1		6000				
Xue	Rui	Ni	ThuThesis			
	ABCDEF					

固然。技经肯綮之未尝，而况大軱乎！良庖岁更刀，割也；族庖月更刀，折也；今臣之刀十九年矣，所解数千牛矣，而刀刃若新发于硎。彼节者有间而刀刃者无厚，以无厚入有间，恢恢乎其于游刃必有余地矣。是以十九年而刀刃若新发于硎。虽然，每至于族，吾见其难为，怵然为戒，视为止，行为迟，动刀甚微，謋然已解，如土委地。提刀而立，为之而四顾，为之踌躇满志，善刀而藏之。”

文惠君曰：“善哉！吾闻庖丁之言，得养生焉。”

A.2.2　非线性规划

孔子与柳下季为友，柳下季之弟名曰盗跖。盗跖从卒九千人，横行天下，侵暴诸侯。穴室枢户，驱人牛马，取人妇女。贪得忘亲，不顾父母兄弟，不祭先祖。所过之邑，大国守城，小国入保，万民苦之。孔子谓柳下季曰：“夫为人父者，必能诏其子；为人兄者，必能教其弟。若父不能诏其子，兄不能教其弟，则无贵父子兄弟之亲矣。今先生，世之才士也，弟为盗跖，为天下害，而弗能教也，丘窃为先生羞之。丘请为先生往说之。”

图 1　这是手动编号但不出现索引中的图片的例子

柳下季曰：“先生言为人父者必能诏其子，为人兄者必能教其弟，若子不听父之诏，弟不受兄之教，虽今先生之辩，将奈之何哉？且跖之为人也，心如涌泉，意如飘风，强足以距敌，辩足以饰非。顺其心则喜，逆其心则怒，易辱人以言。先生必无往。”

孔子不听，颜回为驭，子贡为右，往见盗跖。

A.2.3 整数规划

盗跖乃方休卒徒大山之阳，脍人肝而餔之。孔子下车而前，见谒者曰：“鲁人孔丘，闻将军高义，敬再拜谒者。”谒者入通。盗跖闻之大怒，目如明星，发上指冠，曰：“此夫鲁国之巧伪人孔丘非邪？为我告之：尔作言造语，妄称文、武，冠枝木之冠，带死牛之胁，多辞缪说，不耕而食，不织而衣，摇唇鼓舌，擅生是非，以迷天下之主，使天下学士不反其本，妄作孝弟，而侥幸于封侯富贵者也。子之罪大极重，疾走归！不然，我将以子肝益昼餔之膳。”

在学期间参加课题的研究成果

个人简历

xxxx 年 xx 月 xx 日出生于 xx 省 xx 县。

xxxx 年 9 月考入 xx 大学 xx 系 xx 专业，xxxx 年 7 月本科毕业并获得 xx 学士学位。

xxxx 年 9 月免试进入 xx 大学 xx 系攻读 xx 学位至今。

发表的学术论文

[1] Yang Y, Ren T L, Zhang L T, et al. Miniature microphone with silicon- based ferroelectric thin films. Integrated Ferroelectrics, 2003, 52:229-235. (SCI 收录, 检索号:758FZ.)

[2] 杨轶, 张宁欣, 任天令, 等. 硅基铁电微声学器件中薄膜残余应力的研究. 中国机械工程, 2005, 16(14):1289-1291. (EI 收录, 检索号:0534931 2907.)

[3] 杨轶, 张宁欣, 任天令, 等. 集成铁电器件中的关键工艺研究. 仪器仪表学报, 2003, 24(S4):192-193. (EI 源刊.)

[4] Yang Y, Ren T L, Zhu Y P, et al. PMUTs for handwriting recognition. In press. (已被 Integrated Ferroelectrics 录用. SCI 源刊.)

[5] Wu X M, Yang Y, Cai J, et al. Measurements of ferroelectric MEMS microphones. Integrated Ferroelectrics, 2005, 69:417-429. (SCI 收录, 检索号:896KM)

[6] 贾泽, 杨轶, 陈兢, 等. 用于压电和电容微麦克风的体硅腐蚀相关研究. 压电与声光, 2006, 28(1):117-119. (EI 收录, 检索号:06129773469)

[7] 伍晓明, 杨轶, 张宁欣, 等. 基于 MEMS 技术的集成铁电硅微麦克风. 中国集成电路, 2003, 53:59-61.

研究成果

[1] 任天令, 杨轶, 朱一平, 等. 硅基铁电微声学传感器畴极化区域控制和电极连接的方法: 中国, CN1602118A. (中国专利公开号)

[2] Ren T L, Yang Y, Zhu Y P, et al. Piezoelectric micro acoustic sensor based on ferroelectric materials: USA, No.11/215, 102. (美国发明专利申请号)