形式验证辅助的超级优化系统的设计与实现

(申请清华大学工学硕士学位论文)

培 养 单 位：软件学院

学　　科：软件工程

研 究 生：谢 兴 宇

指 导 教 师：贺　飞　副教授

二〇二四年五月

形式验证辅助的超级优化系统的设计与实现

谢 兴 宇

Design and Implementation of a Superoptimization System Aided by Formal Verification

Thesis submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Master of Science

in

Software Engineering

by

Xie Xingyu

Thesis Supervisor: Associate Professor He Fei

May, 2024

学位论文公开评阅人和答辩委员会名单

公开评阅人名单

王朝坤	副教授	清华大学
周旻	副研究员	清华大学

答辩委员会名单

主席	刘世霞	教授	清华大学软件学院
委员	张慧	副研究员	清华大学软件学院
	王斌	副教授	清华大学软件学院
	周旻	副研究员	清华大学软件学院
	贺飞	副教授	清华大学软件学院
	万海	副研究员	清华大学软件学院
	赵曦滨	副教授	清华大学软件学院
秘书	梁杰	助理研究员	清华大学软件学院

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：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）按照上级教育主管部门督导、抽查等要求，报送相应的学位论文。

本人保证遵守上述规定。

作者签名：谢兴宇　　导师签名：[illegible]

日　　期：2024年5月27日　　日　　期：2024年5月27日

摘 要

虽然今天的工业级编译器中已经集成了各种代码优化的策略，但绝大多数为启发式策略。因此，即使经过优化，程序往往仍然存在性能提升的空间。在性能关键的代码段中，依然需要专家手动编写汇编代码，比如 OpenSSL。超级优化技术的目标是进一步优化代码，为一个程序找到其最优的版本。通常，超级优化的目标是汇编代码，以便考虑底层硬件对性能的影响。

超级优化是一项具有挑战性的任务，因为程序空间极其庞大，要在其中找到最优解非常困难。目前的典型超级优化方法主要基于随机化搜索。这些方法从初始的种子程序出发，不断对程序进行变异，例如删除或增加随机指令，以探索程序空间。然而，在变异程序时缺乏对程序语义的理解，很难探索庞大的程序空间。本研究引入了大语言模型来为随机化搜索生成种子程序，以利用大语言模型理解和生成代码的能力来生成高质量的种子程序。

形式化验证可用于严格检查所找到的优化程序是否正确。检查正确性的方法是，将优化后的程序与待优化程序的等价性编码为逻辑公式，然后将这些公式交由自动定理证明工具来证明或证否。然而，形式化验证的方法通常耗时较长。为了提高效率，本研究设计了一种两个程序之间的同步协调机制。

论文的主要内容如下：

- 引入大语言模型来为超级优化框架进行第一步优化，以生成随机化搜索的种子程序。设计了与大语言模型的对话，借助思维链技术来引导推理，并利用编译器进行初步检查，帮助大语言模型修复生成的代码中的编译错误。
- 为了提高在超级优化过程中自动验证程序等价性的效率，为等价性验证中的符号执行引入了同步协调机制。这一机制旨在减少所生成的逻辑公式的数量和长度，从而减轻自动定理证明工具的负担。
- 实现了一个基于即时编译和代码插桩技术的沙箱，以安全地运行程序，并通过程序的运行结果来评估程序的正确性和性能。
- 实现了一个面向 AArch64 架构的超级优化原型系统，收集了待优化程序集合，其中包括 22 个来自教科书的程序和 3 个来自 C 的标准库替代版 Newlib 的程序。在 16% 的待优化程序上，原型系统找到了超过工业级编译器最高选项优化的性能提升。这些性能提升均源自对 AArch64 架构特性的利用。

关键词：形式化验证；程序优化；大语言模型；符号执行；沙箱

Abstract

Although nowadays industrial compilers have integrated tremendous code optimization strategies, most strategies are heuristics. Thus, there is usually still room for performance improvement in the compiled programs. In performance-critical code, it is still necessary for experts to manually write assembly code, such as in OpenSSL. Superoptimization techniques aim to further optimize code, by *finding the most optimal version for a program*. The target programming language of superoptimization is usually assembly, taking into account the impact of the underlying hardware on performance.

The task of superoptimization is challenging due to the program space being a vast discrete space, where finding the optimal solution is not easy. Existing typical superoptimization methods are based on randomized search, mutating the program from an initial seed program, such as deleting and inserting an instruction, to explore the program space. This kind of methods lacks understanding of the program semantics during mutating, which obstacles the exploration of the vast program space. This study introduces the large language models to generate seed programs for randomized search, aiming to utilize the ability of large language models to understand and write code to generate high-quality seeds.

Formal verification can be used to rigorously check whether the found program is correct, by encoding the equivalence, between the found program and the program to be optimized, into logical formulas, which are then proved or disproved by automated theorem proving tools. However, formal verification is a heavy technique. To improve the efficiency, this study introduces a coordination mechanism between the two programs.

The main content of the thesis is as follows:

- Introduce large language models to perform the first step of optimization in the superoptimization framework: generating seed programs for randomized search. A dialogue with large language models was designed, utilizing the technique of chain-of-thought to guide reasoning, and the compilers to preliminarily check and help repair compilation errors in the code generated.
- To enhance the efficiency of automatically verifying program equivalence, introduce a synchronization coordination mechanism for symbolic execution in equivalence verification, reducing the number and length of generated logical formulas

and thus alleviating the burden on automated theorem proving tools.

- Implement a sandbox based on just-in-time compilation and code instrumentation technologies to efficiently evaluate the correctness and performance of programs during superoptimization by running test cases.
- Implement a superoptimization prototype tool targeting the AArch64 architecture, and collect a benchmark program dataset – including 22 textbook programs and 3 programs from the C standard library alternative Newlib. The experimental results show performance improvements over the highest optimization options of industrial-grade compilers in 16% of the programs to be optimized. The discovered performance improvements all stemmed from leveraging the characteristics of the AArch64 architecture.

Keywords: formal verification; program optimization; large language model; symbolic execution; sandbox

目 录

插图清单

附表清单

符号和缩略语说明

AArch64	ARM 架构的 64 位版本，全称为 ARM Architecture 64-bit
ARM	一个精简指令集架构系列，全称为 Advanced RISC Machine，其中 RISC 的全称为 Reduced Instruction Set Computer，意为精简指令集计算机
CPU	中央处理器（Central Processing Unit）
IR	中间表示（Intermediate Representation）
ISA	指令集架构（Instruction Set Architecture）
SAT	布尔可满足性问题（Boolean Satisfiability），也被称作命题可满足性问题（Propositional Satisfiability Problem）
SMT	可满足性模理论（Satisfiability Modulo Theories），是对 SAT（布尔可满足性问题）的扩展，将更复杂的公式囊括其中，比如包含整数、实数和数组等的公式
x86	一个复杂指令集架构系列，也被称为 80x86
x86-64	x86 指令集的 64 位版本，也被称为 x64、x86_64、AMD 64 和 Intel 64

第 1 章　引言

1.1　研究背景

超级优化指的是为一个程序找到其最优的版本，这一术语最早由 Massalin 于 1987 年提出[1]。这一术语的命名，是因为在英文中对程序优化（program optimization）[2-4]一词略微有误用，在英文中其字面含义原指程序的最优化（optimization）。不过，正如其中文翻译的字面含义一样，它实际被用来指代对程序的改进。这就给 Massalin 带来了困难，所以他只好再加上一个前缀 “超级”，将他所考虑的程序最优化的问题称作 “超级优化”（superoptimization）[5]。

超级优化可以在许多对代码性能有极高要求的底层软件或领域特定软件发挥作用，比如嵌入式系统和超文本传输安全协议（HTTPS，hypertext transfer protocol secure）。一般来讲，程序性能优化的任务主要是由编译器来承担，但编译器带来的提升往往有限，只有极巧合的情况下才会编译出最优的程序。专家人工优化的汇编代码有时会有比编译器生成的代码更高的性能[6-14]，这也从一个侧面说明了虽然现有的编译器中已经含有复杂的代码优化策略，但其所编译出的代码性能依然有提升的空间。超级优化旨在不依赖人工介入的情况下尽可能地进一步地优化程序性能。超级优化会尽可能全面地考虑程序空间中的各种程序，再借助形式化验证技术检验所找到的程序的正确性，也就是与待优化程序的等价性，以期在找到待优化程序的一个性能尽可能优的版本。

这里我们进一步明确超级优化任务的目标编程语言，经典的编译优化往往是在将高级语言翻译到低级语言的过程中或者是在高级语言和低级语言之间的中间表示（intermediate representation）上进行的，不过超级优化的目标编程语言一般是汇编语言，这有两点原因。一方面，汇编代码是编译器输出的最终结果，是编译器优化的最终目标，是真正要去执行的程序，其中囊括了诸多硬件细节，在性能攸关的场合，是需要将硬件——比如说 CPU 的流水线——对性能的影响考虑进来的；另一方面，汇编代码结构简单，最基本的语法单元只有两个：指令和标签。指令编码了计算机要去执行的操作，例如加减法或者读写内存，而标签用来标记出指令序列中的位置，借助标签和向标签去跳转的指令便可以实现高级语言中的分支、循环和函数调用等。这样形式简单的程序构成的程序空间更加规则，也就更加易于探索。

本研究着眼于如何提升现有的超级优化技术，会从两方面入手，一方面是如何更好地探索程序空间，找到更优的程序版本，另一方面是如何提升自动验证程

序等价性的能力。

1.2　研究现状

1.2.1　超级优化

1.2.1.1　基于程序综合的超级优化

程序综合（program synthesis）指的是自动化地构建满足给定形式化规约的程序。通过程序综合来解决超级优化问题的方法是这样的，计算满足规约“对于任意输入，待优化程序与程序 X 的输出相等，并且 X 的指令数目不超过限制 L”的程序 X。通过逐渐增加 L 的值，直到发现这样的 X 存在，就可以找到指令数最少——也就是最优——的版本。

Denali[5]是在超级优化问题中使用程序综合方法的先驱。它通过构建等价图（E 图），来表示所有可能的、与待优化程序等价的程序实现。然后将性质“E 图中存在指令数目不超过 L 的与待优化程序等价的程序”编码为一个 SAT（布尔可满足性问题）公式，再交由 SAT 求解器来求解。

Souper[15]设计了一种领域特定的中间表示（IR, intermediate representation），这种表示是 LLVM① IR 的一个函数式无控制流子集。Souper 从用这种表示撰写的待优化程序中提取出输出程序需要满足的规约，然后利用反例引导的归纳综合（CEGIS, counter-example guided inductive synthesis）技术，尝试生成满足规约且在指令数目限制之内的程序。Slumps[16]项目将 Souper 的方法移植到了 WebAssembly 上。

基于程序综合的超级优化方法难以处理含有循环或者递归的程序，因为这类程序的语义难以用逻辑公式来编码，这也就意味着“与某个包含循环的待优化程序等价”这一规约是难以被准确编码的。

1.2.1.2　基于回答集编程的超级优化

TOAST[17]通过回答集编程（ASP, answer set programming）[18]来解决超级优化问题，充分利用了回答集编程在处理复杂搜索问题时的高效性。回答集编程是一种基于逻辑的编程范式，它是声明式的，描述问题的约束而不是解决问题的具体步骤。回答集编程底层的计算求解器可以自动探索问题的解空间，寻找满足所有给定约束的解答集。类似于基于程序合成的方法，TOAST 也是按照程序长度逐步增加的方式进行搜索，以保证所找到的优化程序是指令条数最少的。在实验中，TOAST

① LLVM 是一套编译相关的开源工具链，在工业界有较广泛的使用。

能够完成包含最多 4 条 SPARC（scalable processor architecture）指令的程序的超级优化任务。

除了回答集编程求解器自身的求解能力所带来的规模上的限制之外，与基于程序综合的方法类似，TOAST 也难以处理含有循环或者递归的程序。

1.2.1.3　基于搜索的超级优化

基于穷举式搜索的超级优化　Massalin[1]最初用于解决超级优化问题的方法便是穷举式的暴力搜索。先搜索长度为 1 的指令序列，再搜索长度为 2 的指令序列，逐渐增加长度，直到找到一个与待优化的指令序列等价的序列。这一方法由 GNU Superoptimizer[19]工具化，它支持许多 1990 年代流行的指令集架构，并且针对不同指令集的特点做了剪枝。Bansal 和 Aiken[20]提出可以将程序规范化，比如将寄存器以在程序中出现的顺序来重命名，通过只搜索规范化的程序，来缩小需要搜索的程序空间——因为考虑一个程序，其实可以认为相当于是在考虑它的等价类中的程序。

GreenThumb[21]考虑了另一个视角的搜索问题，它将最优指令序列的生成问题转化为了一个图上的搜索问题：以多个测例下的程序状态为点，以指令为边的图，每条路径便对应一个程序。这种视角使得对一些高级算法技巧的引入成为可能：通过合并等价类来减少图中点的表示，通过双向搜索来加速搜索过程。

这些技术显著提高了穷举式搜索的实用性，但是由于程序空间关于指令数量是呈指数级增长的，基于穷举式搜索的方法难以解决空间爆炸的问题，所能处理的指令序列长度依然是十分有限的。

基于随机化搜索的超级优化　STOKE[22-23] 使用随机化搜索的方法，以扩大所能探索的程序空间的范围。随机化搜索通过一个启发式的估价函数（也被称作代价函数）来引导，其对程序的评估包括效率提升和正确性两方面，其被定义为：

$$c\,(\mathcal{R};\mathcal{T}) = \mathrm{eq}(\mathcal{R};\mathcal{T}) + \mathrm{perf}(\mathcal{R};\mathcal{T})$$

其中，$\mathcal{R}$ 为待评估程序，$\mathcal{T}$ 为待优化程序，$\mathrm{eq}(\mathcal{R},\mathcal{T})$ 为正确性项，表示两程序有多相似，可以是经过严格形式化验证的相等或者不等，也可以是通过比较在指定测例上的输出来估计出的一个相似度；$\mathrm{perf}(\mathcal{R},\mathcal{T})$ 为效率项，即候选程序相对于目标程序的性能提升，可以通过运行测例或者是数指令条数来估计。

STOKE 使用 Metropolis-Hastings 算法来随机化地变异程序以探索程序空间以尝试找到代价函数最小的解，其基本流程是这样的：从某个程序出发，不断对它作随机化地变异以探索程序空间。具体来讲，从 $\mathcal{R}$ 出发，STOKE 依据一个提议分布

来抽样出 $\mathcal{R}'$：

$$\mathcal{R}' \sim q(\cdot|\mathcal{R}),$$

这个提议分布指的便是从 $\mathcal{R}$ 出发作变异所能得到的程序的分布，比如假如我们只允许删除一条指令这种变异，再假设 $\mathcal{R}$ 中恰好有 5 条不同指令，那么在分布 $q(\cdot|\mathcal{R})$ 中就有 5 个程序有非零的概率，每个程序的概率是 1/5。变异得到的 $\mathcal{R}'$ 不一定会被接收，而是根据 $\mathcal{R}'$ 的代价函数，再计算一个接收标准：

$$\alpha(\mathcal{R} \to \mathcal{R}'; \mathcal{T}) = \min\left(1, exp\left(-\beta \cdot \frac{c\left(\mathcal{R}'; \mathcal{T}\right)}{c\left(\mathcal{R}; \mathcal{T}\right)}\right)\right).$$

然后，我们再在 [0, 1] 中均匀采样，如果采样值不超过 $\alpha(\mathcal{R} \to \mathcal{R}'; \mathcal{T})$，就接受 $\mathcal{R}$，否则拒绝。如果 $\mathcal{R}'$ 被接受了，那优化器就相当于是移动到了 $\mathcal{R}'$ 这个程序，它会被作为新的 $\mathcal{R}$ 来继续在程序空间中探索；否则的话，$\mathcal{R}$ 保持不变，再尝试生成新的 $\mathcal{R}'$。

在随机化搜索的过程中，为了高效地计算代价函数，需要提供一个测例集合，但测例集合所覆盖的路径可能是不足的，尤其是在程序中包含循环时。为了能够完善测例，可以使用一个有界验证器来生成反例[24]，这个有界验证器会将循环的迭代次数限定在一个阈值内，从使得在两个程序不等价时，可以发现一个反例，如果确实是未通过等价性验证得到反例，那么就将反例加入测例集中。

Bunel 等人[25]使用强化学习来改进 STOKE 的随机化搜索过程。他们注意到随机化搜索的过程可以被视作为一个马尔可夫决策过程，对当前程序的变异可以看作是智能体（agent）要采取的行动，变异前后的程序的代价函数的差可以看作是环境给智能体的反馈（reward），那么提议分布其实就可以看作是智能体的策略（policy）。所以可以用强化学习的技术来在随机化搜索的过程中学习得到更好的提议分布。

基于随机化搜索的超级优化方法可以优化数十行的汇编代码，是超级优化领域目前最领先的成果。但其在投入实用时依然面临困难，最大的困难在于，由于程序空间过于庞大，尽管随机化搜索可以避免去穷尽所有程序，但其探索程序空间的能力依然有限，尤其是涉及包含控制流的程序，仅仅通过变异难以找到合适的控制流变形。

1.2.1.4　基于神经网络的超级优化

Shypula 等人[26]使用一个序列到序列（sequence-to-sequence）的神经网络模型作为超级优化器，忽略程序的结构信息，仅仅将其视为词语（token）的序列。他们在 GitHub 上收集了约两万五千个函数，gcc -O0 编译出未优化版本的 x86-64 汇编

代码，再使用 gcc -Ofast 编译出一个优化版本的汇编代码，以此作为一个训练数据集。训练分两个阶段：使用常见的监督学习方法来做预训练，再做微调。微调可以使用同策略（on-policy）和异策略（off-policy）的方法，前者是指在探索程序空间的过程中即时学习调整，后者是指做完一些探索后，再统一学习。经典的同策略方法是 REINFORCE[27]，Shypula 等人[26]提出了一种新的异策略方法，面向优化的自模仿学习（SILO, self imitation learning for optimization）。它的每轮迭代会包含两步，一个探索步骤和一个学习步骤，在探索步骤中，从数据集中抽样出一批样本，在这些样本上尝试去发现比编译器生成的结果更高效的输出代码，在学习步骤中，从数据集中再独立地抽样出一批样本，其中可能会包含模型生成的优化代码，用这批样本来去更新模型参数。实验表明，对于 5.9% 的测试集，可以发现超过 gcc -Ofast 编译结果的优化。

AlphaDev[28] 在序列到序列的生成思路上更进了一步，借助深度强化学习和蒙特卡洛树搜索来寻找下一条指令。通过这种方式，AlphaDev 找到了更快的排序算法：对于不超过 5 个数的排序，发现了比现有的 LLVM C++ 标准库中更快的排序算法。

与第 1.2.1.3 节中随机化搜索的思路不同之处在于，神经网络可以直接去逐条地生成指令序列，而不是在一个指令序列上去做变形，这种方法已经展现了其潜力，但其对算力的要求很高，比如说，AlphaDev 使用了包含 16 个核心的第 4 代张量处理单元（TPU, tensor processing unit）。

1.2.2　基于大语言模型的程序优化

虽然大语言模型理解和生成代码的能力已经被广为认可[29-30]，不过尝试用大语言模型优化程序的工作还不多。Cummins 和 Grubisic 等人[31-32]借助大语言模型来生成编译器的优化选项。Shypula 等人[33]首次尝试用大语言模型来端对端地优化代码，也就是直接输入给大语言模型一段代码，让它输出优化后的代码。

大语言模型采用与第 1.2.1.4 节中类似的序列到序列的生成思路，逐词语（token）地生成代码。从使用角度上来看，大语言模型的一个优势在于，它可以在不调整网络权重的情况下，也可以较好地生成代码，这就降低了对算力的要求。不过，目前大语言模型的能力范围尚不清晰，还未有研究考虑用大语言模型来优化汇编代码的问题。

1.2.3　程序等价性的自动化验证

1.2.3.1　基于重写的等价性验证

基于重写的方法依赖于一系列人工提供的重写规则（rewriting rule），其中每一条规则描述了我们可以将满足某种模式的表达式重写成另一种形式，比如 $a \cdot (b + c) \rightarrow a \cdot b + b \cdot c$。重写规则描述了怎样的程序对是等价的，如果可以应用这些规则从一个程序重写到另一个程序，那么我们就可以知道这两个程序是等价的。

尽管重写规则在形式上有左右之分，但由于等价性本身的对称性，我们在考虑等价性问题时可以忽略重写规则的左右之分，而将一条重写规则视为在描述程序集合上的一个等价关系。借助这些重写规则，有两种算法可以判断两个程序是否等价：一种是将两个程序通过反复应用重写规则，将它们变换为标准形式，然后比较这两个标准形式是否相同[34]；另一种是利用特制的数据结构 E 图，通过将等价的节点组合成超级节点，从而紧凑地表示等价的程序集合，然后检查另一个程序是否在这个程序的等价类中[35-36]。

然而，基于重写的方法在实际应用中存在较大的局限性：重写规则的编写是一个非常困难的工作。编写规则需要对程序的语义有深刻的理解，并且很难保证所编写的重写规则集合是完备的。也就是说，可能会漏掉一些重写规则，导致无法证明两个实际上相等的程序的等价性。

1.2.3.2　基于可满足性模理论的等价性验证

对于无循环无递归的程序，可以将两个程序按照其语义编码为一阶逻辑的表达式，然后将描述表达式等价性的公式交由可满足性模理论求解器（SMT，satisfiability module theories）来检查。如果发现公式恒成立，就说明这两个程序等价[37]。这种将程序按语义编码为公式的方法也被称作符号执行，它通过模拟程序的执行来将程序的语义进行编码。

对于包含循环的程序，Badihi 等人[38]通过语法比较，抽取出其中相似的无控制流片段，然后使用上述处理无循环程序的方法来验证这些片段的等价性。这种方法对于仅在控制流块内部可能存在不同的程序对是有效的，但对于涉及跨控制流的程序优化，如循环展开等，则失效了。

Shemer 等人[39]采用了一种称为 “对齐” 的方法，其中 “对齐” 指的是在两个程序的切点（包括起点、终点和控制流分支出现的程序点）之间寻找一个映射关系。

Ramana 等人[40]证明了程序对齐在理论上对于等价性问题的完备性。换句话说，对于任意等价的程序对，总存在一种对齐方式，使得可以利用这种对齐方式来证明它们的等价性。

STOKE 系列工作[41-42]采用了测例驱动的方法来对齐程序，并在对齐点对上求出不变式，从而初步解决了如何验证带有循环程序的等价性问题。该方法首先通过模板生成在对齐点对上一定成立的公式，然后利用测例找到满足这些公式的点对。最终，通过将由不变式得到的验证条件（即一系列一阶逻辑公式）交由可满足性模理论求解器来求解。

Gupta 等人[43]进一步改进了该方法，用验证过程中所生成的反例来进一步引导寻找程序对齐点对及不变式生成。

由于程序等价性问题是一个不可判定问题，验证两个程序等价性的方法只能是启发式的，并且常常需要消耗大量的时间。其性能上所面临的制约是多方面的，包括底层可满足性模理论求解时的不可判定性、符号执行中可能遇到的路径爆炸问题和生成不变式时的不可判定性等。

1.3　研究内容

基于随机化搜索的超级优化方法的一个缺陷在于：在探索空间时考虑进来的程序的语义信息很少。例如，计算一个变量的二进制表示中 1 的数量的代码片段可以被 `popcnt` 指令替代，但是这种需要改变控制流的替代借助语法上的变异是很难发现的。然而，如果能掌握这段代码片段的含义，就有机会很自然地发现这种指令替代的优化。近年来，大型语言模型理解和生成编程语言的潜力逐渐被挖掘出来[29-30]，所以本研究将大语言模型引入超级优化任务中，以利用大语言模型理解语义和生成代码的能力来帮助优化代码。

等价性验证是超级优化中一个耗时巨大的任务，提高验证等价性的性能面临多方面困难，其中一个困难是符号执行中的路径爆炸问题。路径爆炸指的是程序的执行路径数目随着程序长度的增加在最坏情况下会呈指数级增长，比如在有若干分支语句顺序相连的程序中。而等价性验证又会放大路径爆炸给符号执行带来的问题，因为等价性验证需要考虑两个程序的每一对执行路径，所以路径数目相比于单个程序是平方级的。本研究在等价性验证中引入了符号执行的同步协调机制来缓解这一路径对数目的爆炸问题。该机制交替对两个程序进行符号执行，在执行的过程中，利用一个程序的已知路径来剪枝另一个程序的路径，以提前避免走入不可能的路径对。

图 1.1 展示了本研究完整的超级优化的框架图，从图中可以看出上述的两点改进在完整框架中的位置：种子生成器（大语言模型），验证器（等价性的自动化验证）。

图中的符号含义解释如下：{...} 表示一个程序，用烧杯表示输入给程序的测

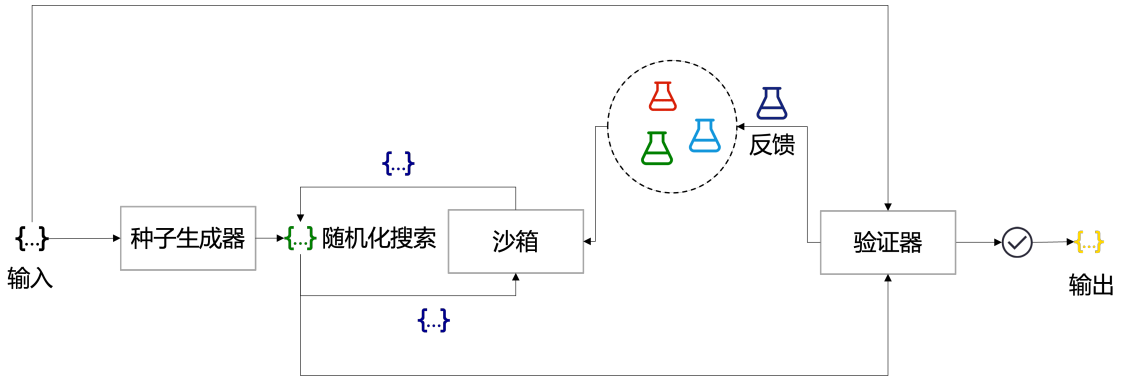

图 1.1　总体框架

试用例，不同颜色的 {...} 表示不同的程序，类似的，不同颜色的烧杯表示不同的测试用例。

框架的输入是一个汇编代码片段，在本文中被称为*待优化程序*。框架的目标输出是一个性能尽可能高的程序，它在语义上与待优化程序等价，换言之，对于任意的输入，它的输出与待优化程序的输出相同。

框架的工作流程如下：待优化程序首先会被交给种子生成器。种子生成器根据待优化程序生成种子程序，也就是随机化搜索的初始程序。系统的主体是二层循环。内层循环是随机化搜索，从种子程序出发变异得到新的程序。这些新程序会通过沙箱来评估其正确性和运行时间。沙箱会在若干个测试用例上执行程序，以评估其性能和正确性。外层循环负责更新测试用例集合。对于在内层循环中通过所有测试用例的程序，验证器需要进一步检验其与待优化程序的等价性。如果不等价，验证器需要生成一个能够暴露出两个程序之间区别的测试用例，并将其加入测试用例集合，然后重新启动内层的随机化搜索循环。通过了验证器检验的程序，即为该框架的一个合法输出程序。

图中用矩形框出的模块即为本文所重点研究的对象。下面会分别对其介绍。

种子生成器

种子生成器负责为随机化搜索生成初始程序。可以作为种子的程序包括：待优化程序、空程序、随机程序。为了获得更高质量的种子，本研究借助大型语言模型来在第一步优化代码。

在与大语言模型对话时，借助思维链（chain-of-thought）来帮助推理，同时，本研究还借助编译器来进行初步检查所生成代码的正确性。

为了剖析大语言模型在解决超级优化问题上的潜力，本研究进行了初步实验。针对来自《高效程序的奥秘（Hacker's Delight）》的 25 个程序，通过人工核查大型语言模型对程序所给出的自然语言总结，发现大型语言模型正确理解了 32% 的程

序，部分理解了 28% 的程序。在大型语言模型所给出的优化中，有 64% 发生了幻觉现象（即与待优化程序不等价），其中有 93% 的程序通过后续的随机化变异找到了正确的优化程序。

验证器

验证器负责自动化地形式化验证所找到的程序与待优化程序是否等价。

在本文中，所考虑的是汇编程序的等价性是一种较简化的定义。直观来讲，等价性指的是，对于任意的输入，如果两个程序的执行都会终止，那么这两个程序的输出是相同的。简化发生在对计算机系统的建模和对程序是否终止的考虑上。

为了缓解等价性验证中符号执行的路径爆炸问题，本文引入了一种符号执行的同步协调机制：先对一个程序进行一步符号执行，然后对另一个程序进行一步符号执行，这样交替进行。在等价性问题中，给定两个程序的输入是相同的，那么根据一个程序的已知路径条件，可以确定输入的一个范围，从而可以剪枝对另一个程序的路径搜索，避免走入该范围的输入下不可能进入的分支。

沙箱

沙箱负责评估程序的性能以及初步评估程序的正确性。在本研究中，沙箱借助即时编译器，通过直接运行待评估程序的方式来分析其性能，以及是否与待优化程序等价。由于待评估的程序可能会包含一些不安全的代码，例如访问非法地址，因此需要一个沙箱来确保汇编程序的安全运行。

分析性能的方式是统计待评估程序在测例上运行时，会执行多少条指令；分析正确性的方式是比较其输出与待优化程序的输出，如果在某个测例上的输出完全相同，可以认为通过了这个测例，如果待评估程序通过了所有测例，可以初步认为二者是等价的。

为了实现这些分析，并确保程序能够安全地执行和终止，沙箱借助代码插桩技术来监控程序的行为，并拦截可能的异常。具体而言，沙箱会在待评估程序中插入四种功能的汇编代码：载入测例和记录运行输出，对所执行指令计数，在程序超时时终止程序，以及拦截可能的异常。

系统实现与实验

另外，所设想的框架被实现为了一个 AArch64 架构上的原型系统，并进行了初步的评估实验。

用于实验评估的待优化程序集合共有 25 个程序，其中 22 个来自教科书《高效程序的奥秘（Hacker’s Delight）》，3 个来自 C 标准库的一个面向嵌入式设备的

开源替代 Newlib①。由于没有其他的超级优化工具支持 AArch64 架构的汇编代码，所以选取工业级编译器的最高优化选项（GCC -Ofast）的编译结果作为基准。

特别地，实验中发现了 STOKE[22] 所收集的《高效程序的奥秘（Hacker's Delight）》中的程序存在一个漏洞②。

1.4　主要贡献

本文的主要贡献总结如下：

1. 将大语言模型理解和生成代码的能力引入超级优化任务中，具体而言，利用大语言模型来为随机化搜索生成种子程序；
2. 在等价性验证时，对两个程序的符号执行作同步协调，以缓解等价性验证任务中的路径爆炸问题，具体而言，交替地对两个程序进行符号执行，利用一个程序的已知路径条件来排除另一个程序中不同时可达的路径；
3. 实现了面向 AArch64 架构的原型系统，并进行了实验评估。在 4/25 = 16% 的待优化程序上，找到了超过工业级编译器最高优化选项的编译结果的性能提升。这 4 个程序中，以执行的指令数量作为性能的衡量指标，其性能提升在 50% 到 300% 之间。

1.5　论文结构

本文的组织结构如下：第 2 章介绍了种子生成器，即利用大型语言模型来生成高质量的种子程序；第 3 章介绍了验证器——汇编程序等价性的形式化定义，以及在验证程序等价性时采用的符号执行的同步协调机制；第 4 章介绍了沙箱的实现，包括通过代码插桩来保障程序安全执行以及分析程序执行结果的方法；第 5 章介绍了原型系统的实现细节以及所进行的实验评估；最后，第 6 章对全文进行总结。

① https://sourceware.org/newlib/

② https://github.com/StanfordPL/stoke/pull/1021

第 2 章　基于大语言模型的程序优化

本章介绍了本研究对大型语言模型的应用，利用其来理解并帮助优化汇编代码。在第 1.3 节所介绍的超级优化整体设计中，大语言模型用于生成初始种子代码，旨在提高种子程序的质量。尽管大型语言模型给出的优化不一定是正确的，但作为初始种子，它并不要求与待优化程序在语义上完全一致。这意味着即使存在错误，后续的随机化变异方法仍有可能修复这些错误。

2.1　示例分析

本节将通过一个例子来引出大语言模型的潜在优势，会通过介绍现有的超级优化方法在这个例子上所面临的困难，并展示大语言模型如何优化这个汇编程序。这个例子揭示了大语言模型的潜力，它有可能辅助自动化地优化一部分过去只能由人类专家手工优化的程序。

```c
#include <stdint.h>

uint64_t roundup_power_2(uint64_t x) {  // x   = 1001
  uint64_t o1 = x - 1;                  // o1  = 1000
  uint64_t o2 = o1 >> 1;                // o2  = 0100
  uint64_t o3 = o1 | o2;                // o3  = 1100
  uint64_t o4 = o3 >> 2;                // o4  = 0011
  uint64_t o5 = o3 | o4;                // o5  = 1111
  uint64_t o6 = o5 >> 4;                // o6  = 0000
  uint64_t o7 = o5 | o6;                // o7  = 1111
  uint64_t o8 = o7 >> 8;                // o8  = 0000
  uint64_t o9 = o7 | o8;                // o9  = 1111
  uint64_t o10 = o9 >> 16;              // o10 = 0000
  uint64_t o11 = o9 | o10;              // o11 = 1111
  uint64_t o12 = o11 >> 32;             // o12 = 0000
  uint64_t o13 = o11 | o12;             // o13 = 1111
  return o13 + 1;                       //       10000
}
```

图 2.1　向上舍入到 2 的次幂（C 代码）

首先介绍一下这个待优化程序，图 2.1 是原始的 C 代码，图 2.2 是 AArch64 汇

```
1  sub     x0, x0, #1
2  orr     x0, x0, x0, lsr 1
3  orr     x0, x0, x0, lsr 2
4  orr     x0, x0, x0, lsr 4
5  orr     x0, x0, x0, lsr 8
6  orr     x0, x0, x0, lsr 16
7  orr     x0, x0, x0, lsr 32
8  add     x0, x0, 1
9  ret
```

图 2.2　向上舍入到 2 的次幂（AArch64 汇编代码，由 GCC 编译）

编代码①。它来自书《高效程序的奥秘（Hacker's Delight）》[44]，这本书被称作“位操作骇客圣经”，其中讲解了一系列位运算技术，用于将原本复杂的算法编码为小型的无循环的位运算指令序列。Gulwani[45] 指出其是程序合成和超级优化的很好的基准数据集来源，并从中收集了一个包含 25 个程序的基准数据集，其中的程序所做的事情包括“将字中最右边的 1 置 0”和“计算两个 32 位无符号数乘积的高 32 位”等。图 2.1 是其中的第 24 个程序的 C 代码，它实现的功能是“向上舍入到 2 的次幂“，也就是说，记函数的输入为 $x \in \mathbb{N} \cap [0, 2^{64})$，函数的输出是（当向上舍入的结果超过了 64 位无符号整数能表示的上限时，便会发生上溢出）

$$\text{roundup\_power\_2}(x) = \begin{cases} 0, & x = 0 \\ 2^{\lceil \log_2 x \rceil}, & x \in \mathbb{N} \cap (0, 2^{63}] \\ 0, & x \in \mathbb{N} \cap (2^{63}, 2^{64}) \end{cases}$$

程序右侧的注释展示了程序在给定输入 $x = 9$ 时是怎么运行的，为了方便展示，图 2.1仅展示出了每个局部变量的二进制表示的低 4 位，更高的位上均为 0。程序的工作原理是，先将 x 减去 1，然后将最高位的 1 右侧的位全部置为 1，得到一个为二的某次幂减 1 的数，然后再加上 1。用最高位的 1 来“填满”低位的方法是通过右移和或运算，将最高位右移 1 位并与自身取或之后，最高位及向右的连续 2 位会是 1（第 6 行），再右移 2 位并取或后，最高位及向右的连续 4 位会是 1（第 8 行），以此类推，直到最高位及向右的连续 32 位是 1（第 14 行），最后再右移 32 位并取或后，得到的数是最高位及向右的连续 64 位是 1（第 16 行）。如果最高位的 1 的右侧没有那么多位了，那么这种右移取或的操作便不会再产生影响（如第 9-16 行注释所示）。

下面来说明对“向上舍入到 2 的次幂”这个例子作优化的困难之处，以及为什

① 由编译器 ARM64 GCC 13.2.0 在 `-Ofast` 选项下编译出。

么传统超级优化方法都无法优化这个例子。下面会从两个角度来阐释。

第一，从可能的优化空间来看，如图 2.3 所示，优化空间出现在对复杂汇编指令的使用。

```
    subs    x1, x0, #1
    b.eq    L_RET
    clz     x1, x1
    mov     x0, #-1
    lsr     x1, x0, x1
    sub     x0, x1, x0
  L_RET:
    ret
```

图 2.3　向上舍入到 2 的次幂（AArch64 汇编代码，人工优化后）

如果 ISA 和 CPU 没有对高级位运算指令的支持，比如 `clz`（count leading zeros，统计有多少个前导零）或 `bsr`（bit-scan reverse，反向扫描找到首个出现的 `1`）这种指令，图 2.2 已经是目前所知的最优的汇编代码了。这种不支持复杂位运算指令的 CPU 在嵌入式市场尤其常见，例如 Sifive 的 E 系列（如 E31、E76）和 S 系列（如 S54）。

需要借助这种高级汇编指令就意味着，按目前的了解：

- 基于中间表示（intermediate representation）的程序优化或者超级优化技术[15]是无法优化这个例子的；
- 仅支持汇编语言中的基本指令的超级优化技术[1,5,19]也是无法优化这个例子的。

由于需要改变控制流结构并引入新的基本块，许多经典的超级优化技术无法适用。这些技术通常只能从无跳转的汇编代码片段优化至无跳转的汇编代码片段[1,5,20-21]，换句话说，它们只能应用于将单基本块的程序优化为单基本块的程序。这种限制使得这些传统技术难以处理需要引入新控制流结构的优化场景，因此对于复杂的优化需求，它们的适用性受到了限制。

第二，从程序的输入输出来看，这个例子的输入输出关系较难引导在程序空间中的探索优化过程。考虑一个总是返回零的函数，它与该程序返回的结果仅相差一个比特，而且对几乎一半的测例（输入在 $\{0\} \cup (\mathbb{N} \cap [2^{63}, 2^{64})$ 中），正确的返回结果恰好就是 0。所以，借助在不同输入测例上的输出来引导探索程序空间的超级优化技术[22,24-26,28]，也无法在这个示例程序上取得效果。

总的来说，传统的超级优化方法受制于程序空间爆炸的挑战，因此只能考虑

较小规模的指令组合，而实际上，汇编指令的数量可能多达几千条。而大语言模型能够生成类似于 `clz` 这样的高级指令，这些指令能够更有效地实现所需的计算目标。

```
1  sub   x0, x0, #1
2  clz   x1, x0
3  mov   x0, #-1
4  lsr   x1, x0, x1
5  sub   x0, x1, x0
```

图 2.4　向上舍入到 2 的次幂（AArch64 汇编代码，大语言模型优化后）

图 2.4 展示了大语言模型的能力，大语言模型则可以发现对 “向上舍入到 2 的次幂” 这一例子的进一步优化：借助 `clz`（count leading zeros，统计有多少个前导零）指令，来更高效地完成 “将 `x - 1` 的最高位的 `1` 的右侧位均置 `1`” 这一步（图 2.1 的 5-16 行，图 2.2 的 2-7 行）。方法是对 `-1`（在其补码表示中，64 位每位均为 `1`）作逻辑右移，使得其前导零数量与 `x - 1` 相同，这便是图 2.4 的 2-4 行所做的事情。图 2.4 的第 1 行与图 2.2 的第 1 行完全相同，并未作改变；图 2.4 的最后 1 行与图 2.2 的最后 1 行的行为从结果上看也是相同的，只是由把 `x0` 加 `1` 改为了 `x0` 减 `-1`。

值得注意的是，大语言模型所优化出的代码（图 2.4）虽然乍一看很有道理，但其实并不完全正确，这是因为第 4 行的 `lsr` 指令只会取出最后一个操作数寄存器的低 6 位来作为移位量，而不是整个操作数寄存器（在这里是 `x1`）的值。因为在第 4 行时，寄存器 `x1` 的值是第 2 行 `clz` 指令的计算结果，前导零的数量的范围是 $0-64$，而 $0-63$ 的二进制表示的位数都不超过 6 位，所以当且仅当 `clz` 的计算结果为 64 时，`lsr` 指令并非按其计算结果来移位。此时，`x - 1` 有 64 个前导零，也就意味着其恰好为 `0`，即，输入 `x0` 的值恰好为 1，此时正确的输出应也为 1，因为 $1 = 2^0$ 是不小于 1 的最小的 2 的次幂，然后，图 2.4 中由于 `lsr` 的右移结果依然是 -1（前导零计数结果 64 的低 6 位依然是 0，所以实际没有发生右移），最终程序会计算出 $-1-(-1)=0$，这并非正确答案。因此，在图 2.3 相较于图 2.4 添加了一条特判处理，即可修复错误，得到一个正确的优化程序：即对输入为 1——也就是输入减去 1 为 0——的情况，直接返回 1。这一瑕疵暴露了大语言模型的局限性，而且，其属于被人们广泛关注的大语言模型的幻觉现象（hallucination）[46-47]——大语言模型生成的文本会有错误，包括虚假事实、逻辑错误和过度推断等。对超级优化中所生成的程序的错误的修复，可以归类为对幻觉问题在程序优化问题

中的一种特殊场景下的修复方案；反之，对解决幻觉问题的诸多研究也有希望应用于基于大语言模型的超级优化中。

2.2　大语言模型汇编语言能力的初步分析

第 2.1 节通过一个例子展示了大语言模型在超级优化问题中的潜力。本节将对其能力做更进一步的剖析。大语言模型理解和生成自然语言、高级编程语言的能力已经得到广泛认可[29-30,48]。然而，超级优化的目标是汇编语言。目前学术界和工业界对于大语言模型在汇编语言方面的能力尚无深入的研究和讨论。相较于自然语言和接近自然语言的高级编程语言（如 Python），汇编语言更接近底层硬件。在代码中涉及到硬件的诸多细节，比如寄存器、内存访问、栈的压入和弹出等。同时，其代码也更为繁琐和冗长，对于人类来说，其理解和撰写的难度远超高级编程语言。为了初步剖析和论证大语言模型在超级优化问题中的潜力，本研究借鉴人类优化代码的过程，将大语言模型优化汇编代码的能力分为两方面来分析：理解汇编代码的能力和优化汇编代码的能力。

2.2.1　理解汇编语言的能力

有一系列指标可以衡量对编程语言的理解能力，比如缺陷检测（Defect Detection）[49]、代码克隆检测（Clone Detecting）[50]、断言生成（Assertion Generation）[51]和代码总结（Code Summarization）[52]等，但它们的基准数据集都是由高级编程语言所撰写。

为了衡量大语言模型对汇编语言的理解能力，本研究采用一项不需要基准数据集的指标：代码总结。更具体而言，就是能否用自然语言正确描述出一个函数体的整体功能。

为了严格确认大语言模型仅仅是依靠所输入的汇编代码本身去理解汇编代码，本研究需要要求：

- 代码中没有用于解释代码的注释；
- 函数名是经过混淆的，因为一般来讲函数名可能会概括出函数的主要功能，所以需要对函数作重命名，使其不包含有关于函数功能的信息；
- 所输入的汇编代码不会在大语言模型的训练数据集中，即其是模型从未见过的。

为了便于实验，这里所考虑的汇编代码中没有函数调用，也没有系统调用。这是因为对于函数调用或者系统调用，要理解它们就需要补充关于系统的信息，或者被调用函数的信息（比如函数体代码）。

由于这个实验的评估无法自动化完成，对于所输出的自然语言描述是否正确的确认必须通过人工检查。因此，本研究并未进行太大规模的实验。

这里选用 Gulwani[45] 从《高效程序的奥秘（Hacker's Delight）》[44] 一书中收集的 25 个程序作为基准数据集，这些程序是一些很精巧的位运算序列，即便对于一般的程序员来说也不易理解，所以用它们来考察大语言模型的程序理解能力是合适的。

对每个程序，给两次机会让大语言模型去理解，使用的大语言模型为 Mixtral-7B①。实验结果如下：

- 有 8/25 = 32% 的程序完全被理解；
- 有 7/25 = 28% 的程序可以部分理解，比如说对于一个 "将最低位的 1 置 0" 的程序，大语言模型可以理解到这个程序是在 "找最低位的 1"，这种理解虽然不完全正确，但也与程序真正的完整目标是接近的；
- 有 10/25 = 40% 的程序并不能理解，然而，即便对于这部分程序，大语言模型也能理解汇编程序中每一条指令的作用，但无法对程序的整体功能有完整的理解。

实验结果表明，大语言模型在理解汇编代码上具有一定的能力，尽管并不总是能够完全理解。然而，优化代码并不一定需要对代码有完全正确的理解，这部分理解能力已经足够。这一点将在第 5.2 节的实验中进一步体现。

2.2.2　尝试优化汇编语言的能力

本小节会通过两个例子，展示大语言模型具有在以下两个方面优化汇编代码的潜力：

- 算法选择上的优化（第 2.2.2.1 节），比如要实现 "比较两个数组中的最大值" 的功能，可以将时间复杂度为 $O(n^2)$ 的二重枚举的算法优化为时间复杂度为 $O(n)$ 的线性算法；
- 发现和应用复杂指令（第 2.2.2.2 节），比如图 2.3 中的 `clz` 指令（count leading zeros，计算前导零数量）。

2.2.2.1　算法选择上的优化

这里通过一个经典的求幂算法——计算 x^y——来展示大语言模型选择更优算法的能力。图 2.5 展示了 C 代码，图 2.6 展示了 GCC 从 C 代码所编译出的 x86-64 汇编代码。这些代码通过 y 次乘法来计算幂，其时间复杂度为 $O(y)$。

大语言模型的优化结果如图 2.7 所示，它应用了更快的平方求幂算法（expo-

① https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2

```c
#include <stdint.h>

uint32_t exp(uint32_t x, uint32_t y) {
  uint32_t ans = 1;
  while (y--) ans *= x;
  return ans;
}
```

图 2.5　计算幂次 x^y（C 代码）

```asm
  lea     -0x1(%rsi), %edx    # 将 y - 1 加载到 edx 中
  test    %esi, %esi          # 测试 y 是否等于 0
  je      .L40069d            # 如果 y 等于 0，则跳转到 .L40069d
  mov     $0x1, %eax          # 将 eax 设置为 1
.L400690:
  imul    %esi, %eax          # 用 x 乘以 eax
  sub     $0x1, %edx          # edx 减 1
  cmpl    $0xffffffff, %edx   # 将 edx 与 -1 比较
  jne     .L400690            # 如果 edx 不等于 -1，跳回 .L400690
  nop                         # 空操作
  retq                        # 返回 eax
.L40069d:
  mov     $0x1, %eax          # 将 eax 设置为 1
  retq                        # 返回 eax
```

图 2.6　计算幂次 x^y（x86-64 汇编代码，由 GCC 编译）

nentiation by squaring），将时间复杂度由 $O(y)$ 降低到了 $O(\log_2 y)$。平方求幂法的原理是：

$$x^y = \begin{cases} x\,(x^2)^{(y-1)/2}, & 2 \nmid y \\ (x^2)^{y/2}, & 2 \mid y \end{cases}$$

故而，可以把一个计算 y 次幂的问题规约到计算 $\lfloor y/2 \rfloor$ 的问题，从而降低了时间复杂度。注意，与第 2.2.1 节中相同，在输入函数给大语言模型时，会对函数名作混淆，以使其不包括对函数体代码功能的概括信息，也就是说，在输入给大语言模型时，图 2.5 中的 `exp` 不会出现，而是会像 图 2.6 中的标签一样，被替换成一个没有含义的名字。

这个例子展示了大语言模型在优化算法选择上的潜力：识别出代码片段所完成的任务，并给出算法复杂度更低的代码。

```
1   test   %esi, %esi          # 检测 y 是否为零
2   je     .L40069d            # 如果 y 为零，跳转到.L40069d
3   movl   %edx, %ecx          # 初始化结果（ans）为 1
4   mov    $0x1, %eax          # 将 x 复制到%ecx
5 .L400690:
6   testb  $0x1, %sil          # 测试 y 的最低有效位
7   jz     .L400694            # 如果这个位是 0，跳转到.L400694
8   imull  %ecx, %eax          # 用 x 乘以 ans（ans *= x）
9 .L400694:
10  shrl   $0x1, %esi          # 将 y 向右移动 1 位（y >>= 1）
11  imull  %ecx, %ecx          # 平方 x（x *= x）
12  testl  %esi, %esi          # 检测 y 是否为零
13  jnz    .L400690            # 如果 y 不为零，跳回.L400690
14  retq                       # 以%eax 中的结果返回
15 .L40069d:
16  mov    $0x1, %eax          # 将结果（ans）设置为 1
17  retq                       # 以%eax 中的结果返回
```

图 2.7　计算幂次 x^y（x86-64 汇编代码，大语言模型优化后）

2.2.2.2　发现和应用复杂指令

第 2.1 节中的例子（图 2.2 至 2.4）已经展示出大语言模型发现和应用复杂指令的能力，不过由于其是一个非常经典而古老的案例（出处《高效程序的奥秘（Hacker's Delight）》[44]成书于 2002 年），大语言模型有在训练数据集中见过其优化出的代码之嫌，因此，本小节将会展示一个更新的例子，确保是不会出现在大语言模型训练数据集中的代码。

图 2.8 中所示的 C 代码函数是在线编程学习平台 LeetCode 上编号为 2859 的题目的一份正确提交代码 ①，它是 2023 年 9 月 17 日公布的，而本节所试验的大语言模型（ChatGPT 4）截止成文时所使用的训练数据集是 2023 年 4 月之前采集的，所以该代码不可能出现在其训练数据集中。这道题目要求解的问题是这样的：给一个数组 `nums` 和一个非负整数 `k`，计算 `nums` 中索引在其二进制表示中恰好有 k 个 `1` 的元素之和。在图 2.8 中，为了判断下标（`i`）的二进制表示中是否有 k 个 `1`，有一个内层循环来计算 `i` 的二进制表示中 `1` 的数量（第 9-13 行），每次右移一位，检查最低位是否为 `1`，如果是，则计数器 `cnt` 加一。图 2.9 展示了由 GCC 根据图 2.8 编译出的 x86-64 汇编代码，其用来计算二进制表示中 `1` 的数量的汇编指令序列同样用高亮标出（第 14-21 行）。

① 来自 https://leetcode.com/problems/sum-of-values-at-indices-with-k-set-bits/solutions/4054870/c-solution/，为了便于处理和展示，将原代码中的功能函数内联进了主函数，使其成为一个单函数、无函数调用的程序。

```c
int sumIndicesWithKSetBits(
    const int * const nums,
    const int numsLen,
    const int k
){
    int sum = 0;
    for (int i = 0; i < numsLen; i += 1){
        int cnt = 0;
        for (int copy = i; copy > 0; copy >>= 1){
            if ((copy & 1) == 1){
                cnt += 1;
            }
        }
        if (cnt == k){
            sum += nums[i];
        }
    }
    return sum;
}
```

图 2.8　数组中下标有 k 个 1 的元素之和（C 代码）

如图 2.10 所示，大语言模型可以成功地识别出这个内层循环的功能——计算二进制表示中 1 的数量，然后找到可以直接完成这个功能的指令 `popcnt` 来替代原来的内循环。虽然现代编译器（如 GCC 和 CLANG）也有对 `popcnt` 的支持选项 `-mpopcnt`，但经过实际试验，它们并不能像大语言模型那样识别出这里存在与 `popcnt` 相同功能的代码片段。

2.3　基于编译器反馈的对话设计

本节会介绍本研究是如何与大语言模型对话的，如何驱动它优化代码。本节会从设计目标讲起，进而介绍是如何实现设计目标的。最后，会额外提及在对话中用到的提示工程（prompt engineering）技巧。

本研究与大语言模型的交互并非一问一答式的，而是通过多轮对话进行的。大语言模型处理多轮对话的原理是，在每次生成新的答复时，都会完整回顾之前的全部对话，包括所接收的输入和自己生成的回答。利用这一特性，可以在对话中引导大语言模型思考，逐步地提供信息，并让它从自身的错误中吸收经验。

对话的设计目标包括：

1. 让大语言模型尽可能地去优化代码；

```
  testl %esi, %esi            # 检测 numsLen
  jle .L8                     # 如果 numsLen 小于等于 0，跳转到 .L8
  movq %rdi, %r9              # 将 nums 数组的地址复制到 %r9
  leal -1(%rsi), %eax         # 计算 numsLen-1，存入 %eax
  leaq 4(%rax,%rax,4), %rax   # 计算 nums 数组的长度
  leaq (%rdi,%rax,4), %r10    # 计算 nums 数组最后一个元素的地址
  movl $0, %eax               # 初始化 sum 为 0
.L7:
  movl (%r9), %r8d            # 加载 nums[i] 的值到 %r8d
  testl %r8d, %r8d            # 测试 nums[i]，用于判断正负
  jle .L9                     # 如果 nums[i] 小于等于 0，跳转到 .L9
  movl %r8d, %ecx             # 将 i 的值复制到 %ecx 用于后续位运算
  movl $0, %esi               # 初始化 cnt 为 0
.L5:
  movl %ecx, %edi             # 复制 i 到 %edi
  andl $1, %edi               # 计算 i & 1
  cmpl $1, %edi               # 比较结果是否为 1
  sbbl $-1, %esi              # 如果结果为 1，cnt 加 1
  sarl %ecx                   # i 右移一位，准备下一次循环
  testl %ecx, %ecx            # 测试右移后的 i
  jg .L5                      # 如果 i 大于 0，继续循环
.L9:
  movl $0, %esi               # 重置 cnt 为 0
.L3:
  addl %eax, %r8d             # 将 nums[i] 加到 sum 上
  cmpl %edx, %esi             # 比较 cnt 和 k
  cmove %r8d, %eax            # 如果 cnt 等于 k，更新 sum
  addq $4, %r9                # 将数组索引 i 增加 1，指向下一个元素
  cmpq %r10, %r9              # 比较当前元素地址和数组末元素地址
  jne .L7                     # 如果不相等，继续循环
  rep ret                     # 结束循环，返回 sum
.L8:
  movl $0, %eax               # 若 numsLen 非正，设置 sum 为 0
.L2:
  retq                        # 返回 sum
```

图 2.9　数组中下标有 k 个 1 的元素之和（x86-64 汇编代码）

2. 代码需要尽量保证正确。

为了达成目标 1，本研究主要应用了思维链（chain-of-thought）[53]的技术来引导大语言模型的思考。思维链指的是将复杂的问题分解成一系列更小的步骤，逐步地推理和解决问题。本研究将优化代码的问题拆解为理解代码和尝试优化代码

```
  testl %esi, %esi    # 检查 numsLen (存储在 %esi) 是否为 0 或负值
  jle .L8             # 如果是，跳转到 .L8 结束程序
  movl $0, %r10d      # 初始化循环变量 i = 0 (%r10d)
  movl $0, %eax       # 初始化 sum = 0 (%eax)
  movl $0, %r8d       # 初始化 cnt = 0 (%r8d)
  jmp .L3             # 跳转到标签 .L3
.L9:
  movl %r10d, %ecx    # 将循环变量 i 的值复制到 %ecx
  movl $0, %r8d       # 重置 cnt = 0 (%r8d)
  # 使用 popcnt 指令
  popcnt %ecx, %r8d   # 计算 i 中 1 的个数，结果存入 %r8d (即 cnt)
.L7:
  addq $4, %rdi       # 将指针 nums (%rdi) 移动到下一个整数
.L3:
  cmpl %edx, %r8d     # 比较 cnt (%r8d) 和 k (%edx)
  jne .L6             # 如果不等，跳转到 .L6
  addl (%rdi), %eax   # 如果相等，累加 nums[i] 到 sum (%eax)
.L6:
  addl $1, %r10d      # i 增加 1
  cmpl %esi, %r10d    # 比较 i (%r10d) 和 numsLen (%esi)
  je .L2              # 如果相等，即循环结束，跳转到 .L2 结束函数
  testl %r10d, %r10d  # 检查 i 的值
  jg .L9              # 如果 i > 0，跳转回 .L9 继续循环
  movl $0, %r8d       # 否则将 cnt 重置为 0
  jmp .L7             # 跳转到 .L7 继续执行
.L8:
  movl $0, %eax       # 如果 numsLen <= 0，设置 sum = 0
.L2:
  retq                # 返回函数结果，即 sum 的值
```

图 2.10　数组中下标有 k 个 1 的元素之和（x86-64 汇编代码，大语言模型优化后）

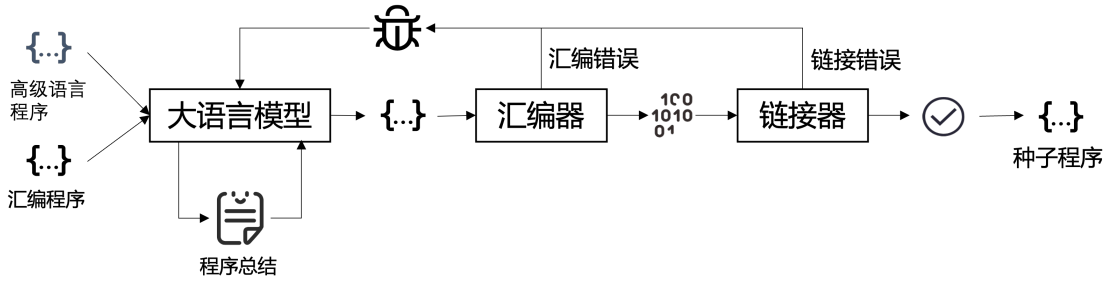

图 2.11　对话设计：借助编译器反馈来帮助大语言模型初步矫正生成程序

两个子问题。首先，让大语言模型先去理解代码，然后根据对代码的理解和对代码整体功能的总结，去生成功能相同但效率更优的代码实现。

对于理解代码的子问题，这里可以再将其分为两个步骤：理解高级语言代码和理解汇编代码。也就是说，首先，将高级语言代码（如 C 代码）输入给大语言模型，让它先理解高级语言的语句，并总结程序的整体行为；然后，借助大语言模型对高级语言代码的理解，来帮助其理解后续的汇编代码。

下面来论证这种步骤分解的合理性，包括两方面：一方面是大语言模型对高级编程语言的理解能力要强于底层的汇编语言；另一方面是对高级语言代码的理解能够帮助大语言模型更好地理解相应的汇编代码。首先，关于大语言模型对高级编程语言的较强理解能力。如第 2.2 节 中所述，大语言模型已经展现出了对高级编程语言的极高理解能力。而且，其训练数据集中的高级语言代码数量也远多于汇编代码。毕竟，汇编代码往往不会以文本形式直接存储。因此，有理由相信大语言模型对高级语言代码的理解能力强于汇编代码。其次，对高级代码的理解之所以能够帮助大语言模型更好地理解相应的汇编代码。可以将其类比于人的理解方式：如果能够从高级语言撰写的程序中总结出程序片段的整体行为，就无需再从汇编代码中去理解其整体功能。此外，根据汇编指令对应到的高级语言的语句，以及该汇编指令和高级语言语句在整体功能中的地位，也会让人更容易理解每条汇编语句的行为和执行目标。

为了达成目标 2，本工作将借助编译器——更具体地说，是汇编器（assembler）和链接器（linker）——来初步检查并帮助大语言模型矫正所生成的代码。矫正的方式是将汇编器或链接器的报错反馈给大语言模型，让其修复错误，直至通过编译。在这种方式中，大语言模型代替了此前开发者的位置：开发者在撰写完代码后，尝试将代码编译，根据编译器反馈的编译错误来修复代码。由于编译器报错信息的设计初衷是为了让开发者容易理解错误并修复错误，会结合自然语言的描述和具体的代码错误，因此编译器的报错信息适合反馈给具有很强自然语言理解能力的大语言模型。

图 2.11 总结了本研究如何与大语言模型对话的过程。在最开始，依次将高级语言代码和汇编语言代码输入给大语言模型。在大语言模型理解并尝试优化出汇编代码后，先借助汇编器来检查是否有汇编错误。如果可以将生成的代码汇编为二进制文件，并且将其与一个空的主程序二进制文件链接（因为可执行文件必须包含 `main` 函数作为入口函数），如果可以通过链接，就说明生成的代码是正确的。否则，如果在汇编阶段或者链接阶段出错，就将汇编器或链接器的报错反馈给大语言模型，让其修复生成的代码中的错误。

需要注意的是，每次与大语言模型交互时，输入给大语言模型的是一个完整的（可能有多轮的）会话。会话的最开始，有一些初始化提示作为热身，这些提示

包括“人格模式（persona pattern）”[54]的提示工程技巧，也就是让大语言模型来扮演汇编语言和程序优化的专家的角色，以让大语言模型快速地定位到它所需要启用的知识范围、快速地了解到它会接收的代码和需要输出的内容。在会话的最末尾，可能会有若干轮修复错误的尝试，这意味着每次大语言模型尝试修复错误时，它都能看到完整的历史，包括它对于程序的理解，和此前犯下的错误等，这或许会有助于其更好更快地修复错误——就像开发者在修复代码漏洞时，可能也要先经历多次试错，才能找到正确的修复，而这多次试错的经验对于发现正确的修复也是重要的。

2.4　幻觉现象及其修复可能性的初步分析

大语言模型面临的一个问题是，它可能会有幻觉现象（hallucination）[46-47]，即优化出的代码在表面上看起来是正确的，但实际上是错误的。本节试图通过初步实验来分析：

- 大语言模型在优化汇编代码时是否有可能面临严重的幻觉现象；
- 基于随机化搜索的超级优化方法是否有潜力修复大语言模型因幻觉现象而产生的微小错误。

初步实验所使用的待优化程序集来源于教科书《高效程序的奥秘（Hacker’s Delight）》，共包含 25 个程序。这些程序均为较难理解的位运算序列，指令数量一般在几条左右，最多不超过 20 条。本节所评估的大语言模型为 ChatGPT4，使用的基于随机化搜索的超级优化系统为 STOKE[22]。

针对这 25 个程序，一共进行了三轮实验。在每一轮中，首先将程序交由大语言模型来优化，然后将大语言模型生成的汇编代码交由 STOKE[22]进行测试。STOKE 会首先通过随机输入来测试大语言模型生成的汇编代码是否正确。如果发现汇编代码存在错误，STOKE 将从这个错误的程序出发，尝试通过随机化搜索的方式找到一个正确的程序。

在大语言模型所生成的 75 个优化程序中，有 5 个导致了 STOKE 的崩溃，在剩余的 70 个中：

- 有 45/70 = 64.3% 发生了幻觉现象，大语言模型给出的优化程序无法通过测试；
- 有 42/45 = 93.3% 的幻觉现象，通过 STOKE 的随机化变异，还能从中出发找到正确的优化程序。

图 2.12 展示了大语言模型在程序优化问题中的幻觉现象，以及基于随机化搜索的方法对这种幻觉现象的修复。高亮标出的第 1 行为被大语言模型所忽略的代

```
1  movl %edi, %eax      # 将输入变量 x 原本的值暂存于寄存器 %eax 中
2  negl %edi            # 计算 -x
3  andl %edi, %eax      # 从 %eax 中取出 x 的值，计算 -x & x
4  retq                 # 返回 x & - x 的计算结果
```

图 2.12　计算二进制表示中最低位的 1（x86-64 汇编代码，大语言模型优化后）

码，也就是说，大语言模型在理解了代码主体的计算逻辑（`x & -x`）之后，忘记了第 1 行的操作（将输入变量 `x` 的值暂存）。而随机化搜索，在测试结果的引导下，可以从大语言模型所输出的代码的基础上（第 3 - 4 行），再重新发现第 1 行，弥补大语言模型所犯下的微小错误。这个例子直观地展示了基于随机化搜索的超级优化方法的潜力，它有可能修复大语言模型在优化代码时因幻觉现象而犯下的微小错误。

2.5　讨论

本节会讨论一些值得尝试的对大语言模型的使用上的改进方向。

第一点是，本研究目前所使用的大语言模型并未在汇编语言的数据集上做进一步的微调（fine-tune）。

虽然现有的实验结果和相关研究表明，未在特定编程语言的数据集上进行微调的大语言模型已经展现出了对编程语言的理解能力[29,55]，但也有研究表明，在特定编程语言的数据集上进行微调确实可以增强大语言模型理解和生成相应代码的能力[30,56]。

不过，如何构建用于理解和优化汇编代码的数据集用于微调，也并不容易，已知的困难包括：

1. 汇编代码一般不会以文本形式存储下来，不易于直接从互联网上收集；
2. 汇编程序相对于高级编程程序更加复杂，繁琐和冗长，也就更加难以理解；
3. 构建一个用于代码优化的数据集需要一对一对的代码——包括优化前的代码和优化后的代码。由于超级优化希望大语言模型能发现传统的（如编译器中的）代码优化方法无法自动化找到的优化，因此最好不要由现有的编译器或代码优化工具自动执行从优化前的代码生成优化后的代码这一步，而是需要专家来人工介入。

Shypula 等人的工作提供了一个值得参考的思路，可以将其用于构建汇编代码优化的数据集。他们构建了一个高级语言代码优化的数据集，用于微调大语言模型，使其在高级编程语言的优化上可以超越传统优化方法的效果。具体地，他们收

集了编程题库中同一个人相邻的两次正确的提交。如果这两次提交的性能相对提升超过 10%，则认为提交者经过思考，对同样的功能给出了更优的代码实现。然后将这两次提交的代码作为（优化前代码，优化后代码）的代码对加入数据集中。如果要构建汇编代码优化的数据集，可以参考这种思路。可以将有效率提升的相邻两次正确提交所编译出的汇编代码作为（优化前汇编代码，优化后汇编代码）的代码对加入数据集中。

第二点是，目前大语言模型理解代码的方式主要是静态的——即通过阅读代码来理解其行为。然而，人类在理解代码时通常会采用更加动态的方式，包括运行代码、观察代码的执行结果，甚至逐行调试观察每一行代码的执行对程序状态的变化。受此启发，可以尝试让大语言模型动态地运行代码，观察其执行结果，从而更好地理解代码的行为。这种方法可能需要结合模拟器或虚拟机等技术，使大语言模型能够模拟代码的执行过程，并观察每一步的执行结果和状态变化。通过动态地运行代码，大语言模型可以更深入地理解代码的行为，更准确地推断代码的功能和逻辑，从而提高其在代码理解和优化方面的能力。

具体而言，有三种方式可以让大语言模型动态运行或者说模拟动态运行代码：

- 为大语言模型提供输入和代码运行的输出；
- 为大语言模型提供每一行所改变的程序状态（变量、寄存器和内存）；
- 可以给大语言模型程序的输入，让其逐行模拟程序的运行，自行计算出每一行的运行结果。

这三种方式也对应了人类在除了阅读代码之外理解代码动态行为的三种方式：观察输入输出对，单步调试程序观察程序状态变化，和纸笔模拟程序运行。

2.6　本章小结

本章探讨了大语言模型在解决超级优化问题上的潜力，展示了它可能发现更优算法和应用复杂指令的能力。在初步实验中，对于来自《高效程序的奥秘（Hacker's Delight）》的 25 个程序，大语言模型正确理解了 32% 的程序，分理解了 28% 的程序，但其中 64% 的优化存在幻觉现象。不过，通过后续的随机化变异，93% 的程序找到了正确的优化程序。对于大语言模型在超级优化问题上所能实际发挥的作用，将于第 5.2 节通过实际实验来进行评估。

技术上，本章介绍了是如何借助编译器来构建与大语言模型的对话的，其中的一个关键是借助编译器来帮助初步检查和修复大语言模型所生成的程序。

第 3 章　基于符号执行的程序等价性验证

本章给出了汇编程序等价性的一个较为严格的定义，在此定义的基础上，介绍了在等价性验证中对符号执行作同步协调的一个机制。同时，有研究表明，相较于高级编程语言或者中间表示如 LLVM IR 来说，汇编层面的符号执行是最快的[57]。由于高级语言往往被编译成汇编语言来执行，因此尽管本章讨论的是汇编程序，但本章所讨论的技术也有望应用于高级语言如 C 语言程序。

3.1　等价性验证中的符号执行简介

自动化地分析两个程序是否等价是不可判定的（undecidable），即不存在一个通用的算法可以判定任意两个程序是否等价。这是因为，假如说存在这样一个通用的算法可以判断任意两个程序等价，那么我们可以用这个算法来判断任意一个程序和一个死循环程序（如 `while (1){}`）是否等价，从而得到一个判断任意一个程序是否会停止的算法，然而这种算法已知是不存在的[58]——对于用图灵完备（Turing-complete）的语言撰写的程序而言，图灵完备指的是这种语言拥有可以描述任何计算的能力，严格来说，就是其可以模拟通用图灵机的计算，所有现代常见的高级编程语言（如 C 语言）和汇编语言都是图灵完备的。

当然，虽然一般的等价性问题是不可判定的，对于一些特定的程序对，还是存在一些算法可以自动判断它们是否等价。

如第 1.2.3 节中所介绍的，自动化验证程序等价性的算法可以分为两类：

- 基于重写（rewriting）的方法，尝试将程序变形为一种 “标准形式”。这种方法会预先指定一系列不会改变程序含义而又性质良好的变形规则，仅考虑这些规则所能推导出的变形。通过比较两个程序的标准形式是否完全相同或者是否在由这些规则建立的等价类中，即可知道两个程序是否等价；
- 基于可满足性模理论（SMT, satisfiability modulo theories）的方法，将程序等价性编码为若干一阶逻辑公式，然后交由可满足性模理论求解器来求解。如果求解器判断这些公式恒成立，就说明两个程序是等价的。

当然，这两类方法之间也并非完全分离，而是会互相融合。

本研究采用了第二类方法，即基于可满足性模理论求解器的方法。这是因为基于重写的方法依赖于人工设计的规则，这些规则是一种等式，指示左右两边具有相同的语义可以进行变形，比如分配律 $x \cdot (y + z) = x \cdot y + x \cdot z$。然而，对于汇编程序来说，因为存在极多的操作类型，比如 AArch64 中有 763 种指令操作，设计出完

备的重写规则集合是极为困难的。相比之下，基于可满足性模理论求解器的方法不需要依赖于人工设计繁复的重写规则。其中用于编码汇编的位向量（bitvector）理论具有完备的判定过程（decision procedure）。换言之，对于仅由类型为 64 位的 01 串的变量构成的公式，总可以在有限时间内判断它是恒真的，或者找到一个让公式不成立的反例。尽管受到计算资源和时间的限制，在实践中可能会在短时间内求解不出来。

对于没有循环的程序，基于可满足性模理论来验证两个程序等价性的基本思路是：将两个程序根据其语义编码为一阶逻辑中的表达式，记作 $[\![P]\!]$ 和 $[\![Q]\!]$，这两个表达式描述了如何从输入状态计算出输出状态 ①，然后交由可满足性模理论求解器来检查 $[\![P]\!] = [\![Q]\!]$ 是否一定成立，如果不成立的话，求解器会抛出一组反例，该组反例即为一个能说明两个程序行为不同的输入状态，换言之，从该状态出发执行 P 和 Q，所得到的输出状态是不同的。

符号执行可以看作是一种编码无循环程序语义的方法。程序不再被考虑成实际状态之间的变迁，而是符号状态之间的变迁。符号状态可以被定义为一个从寄存器集到表达式集的函数

$$S : Reg \to Term,$$

其中 $Term$ 是由类型为 64 位的位向量的寄存器变量经过若干计算所能构成的可能的表达式的集合。从 S 出发，执行语句 $\hat{v} := \hat{e}$（$v_1 := e_1, v_2 := e_2, \cdots$）② 后，符号状态变迁为

$$S[\hat{v} \mapsto S(\hat{e})],$$

意味着将 $S(v_1)$ 更新，更新为将 e_1 中各寄存器变量替换为 S 中对应的表达式所新得到的表达式，也将 $S(v_2)$ 更新，更新为将 e_2 中各寄存器变量替换为 S 中对应的表达式所新得到的表达式，以此类推。比如说，对于 AArch64 中的汇编语句 `add x0, x1, x2`，从符号状态 S 出发执行后，得到的符号状态为 $S[\texttt{x0} \mapsto S(\texttt{x1}) + S(\texttt{x2})]$，若初始的符号状态为 $\{\texttt{x0} \mapsto x_0, \texttt{x1} \mapsto x_1, \texttt{x2} \mapsto x_2, \cdots\}$，那么执行后的符号状态为 $\{\texttt{x0} \mapsto x_1 + x_2, \texttt{x1} \mapsto x_1, \texttt{x2} \mapsto x_2, \cdots\}$。再举一个会改变多个寄存器的例子，比如说 AArch64 中的汇编语句 `subs x0, x1, x2`，在做减法的同时，也会根据 `x1` 减 `x2` 的结果去更新 CPU 内部的标志位，实际上，`n`，`z`，`c`，`v`

① 状态就是对 CPU 中各寄存器和存储空间的一组赋值，这里可以简化理解为对 CPU 中各寄存器的一组赋值，比如 $\{\texttt{x0} \mapsto 0, \texttt{x1} \mapsto 1, \cdots\}$。

② 所有的汇编语句都可以用 $\hat{v} := \hat{e}$ 这种形式来描述，比如 AArch64 中的汇编语句 `add x0, x1, x2`，可以描述为 $\texttt{x0} := \texttt{x1} + \texttt{x2}$；而会更新 CPU 内部标志位的 AArch64 语句 `subs x0, x1, x2`，可以描述为 $\texttt{x0} := \texttt{x1} - \texttt{x2}, \texttt{n} := \texttt{x1} - \texttt{x2} <_s 0, \texttt{z} := \texttt{x1} - \texttt{x2} = 0, \texttt{c} := \texttt{x1} >_u \texttt{x2}, \texttt{v} := \texttt{x1} <_s 0 \oplus \texttt{x2} <_s 0 \wedge \texttt{x1} <_s 0 \oplus \texttt{x2} <_s 0$，其中 $<_s$ 和 $>_u$ 分别表示有符号比较和无符号比较，$\oplus$ 表示异或。

这些标志位是放在同一个寄存器里的，这里为了便于展示，我们将它们区分开来，从符号状态 S 出发执行后，会得到 $S[\mathtt{x0} \mapsto S(\mathtt{x1}) - S(\mathtt{x2}), \mathtt{n} \mapsto S(\mathtt{x1}) - S(\mathtt{x2}) <_s 0, \mathtt{z} \mapsto S(\mathtt{x1}) - S(\mathtt{x2}) = 0, \mathtt{c} \mapsto S(\mathtt{x1}) >_u S(\mathtt{x2}), \mathtt{v} \mapsto (S(\mathtt{x1}) <_s 0 \oplus S(\mathtt{x2}) <_s 0) \wedge (S(\mathtt{x1}) <_s 0 \oplus S(\mathtt{x2}) <_s 0)]$，其中 $<_s$ 和 $>_u$ 分别表示有符号比较和无符号比较，$\oplus$ 表示异或。

在程序中有分支的情况下，符号执行技术会相应地分叉，以探索所有可能的路径。在遍历路径时，除了计算符号状态之外，还会额外计算一个路径条件（path condition），通常记为 pc。路径条件也是关于输入状态的一阶逻辑公式，它描述了当接收到何种性质的输入状态时，程序会沿着这条路径执行。从技术上讲，路径条件是该路径上所有分支条件的合取。

当程序中存在循环或递归调用时，符号执行技术就变得复杂。一种确保正确性的方法是利用循环不变式，将程序在循环头和递归调用的位置切开，分成多个不包含循环或递归调用的片段。然而，这种方法引入了新的困难——如何生成这样的不变式，这是一个不可判定的问题；另一种方法是限制循环的迭代次数，这样就可以将程序视为不含循环的。这种方法被称为限界模型检测（bounded model checking）。虽然这会引入一些不精确性，但却简化了问题。这也是本研究所采用的方法，具体来说，本研究会通过限制执行的指令数量来限制循环的迭代次数。

借助符号执行去做等价性验证，最直接的方法是：分别对两个程序进行符号执行，然后比较两个程序的每一对指令数量都不超过一定阈值的执行路径，检查当满足这一对路径上的路径条件时，两个程序在这一对路径上的运行结果是否一定是相同的。对于每一个路径对，设程序 1 的路径条件和符号状态为 (pc_1, S_1)、程序 2 的路径条件和符号状态为 (pc_2, S_2)，我们会生成形如

$$pc_1 \wedge pc_2 \rightarrow \bigwedge_{r \in Reg} S_1(r) = S_2(r)$$

的验证条件。

本研究对此的一个观察是：其实并不需要考虑那么多路径之间的对应关系，等价的程序之间的控制流结构往往是相似的，一个程序的一条路径很可能只对应于另一个程序中的一条或少数几条路径。

不过，找到这样的路径之间的对应关系也并非易事。如果我们可以 “同时” 符号执行两个程序，借助一个程序已知的路径条件，来避免另一个程序进入不对应的路径，这样就可以避免不必要的路径探索。当然，我们不可能真的做到同时执行两个程序，所以退而求其次，本研究交错地执行两个程序，以此来协调二者的执行，以模拟出类似同步执行的效果。

3.2　汇编程序的形式化

本节会定义出本研究所考虑的汇编程序的等价性的严格含义，会从汇编程序的状态（第 3.2.1 节）出发，定义出汇编程序的语义（第 3.2.2 节），根据语义再定义出汇编程序之间的等价性。需要注意的是，这里所考虑的汇编程序等价性是一个较为简化的模型，省略了对系统权限级别（用户态和系统态）以及内存的建模，同时也忽略了不终止的程序。

3.2.1　程序状态

严格来说，汇编程序的状态应该包括寄存器的值和内存中的值。特别地，内存中应该包括当前的栈帧，并且所执行的程序也应存放在内存中[59]。

在本文中，简单起见，仅考虑机器中所包含的通用寄存器和系统寄存器，通用寄存器是用户程序可以使用的寄存器，用于存储临时数据和计算中间结果，例如在 x86-64 架构中的 `rax` 和 AArch64 架构中的 `x0`。系统寄存器则是具有特殊用途的寄存器，比如程序计数器 `pc`（program counter），用于存储下一条要执行的指令的地址。记所考虑的寄存器集合为 Reg，下面给出了汇编程序状态的一个较为严格的定义。

定义 3.1 (汇编程序的状态)： 一个 w 位机器的状态为一个从寄存器集合到值域的映射 $s : Reg \to \mathbb{Z}_{2^w}$，其中 w 一般为 32 或 64。

例 3.1 (汇编程序的状态)： $\{\texttt{x0} \mapsto 3, \texttt{x1} \mapsto 2, \texttt{x2} \mapsto 1, \cdots\}$

3.2.2　程序语义

记所有指令的集合为 $Inst$，一个汇编程序是一个指令的有限序列，也就是 $P : [n] \to Inst$，其中 $[n] = \{0, \cdots, n-1\}$，$n \in \mathbb{Z}_{2^w} \setminus \{0\}$ 为程序 P 的长度，在后文中，程序的长度会被记作 $|P| = n$。

这里，本文会作以下简化：

- 对于程序指令，这里仅考虑常规计算指令和跳转指令，涉及抛出异常、系统调用或者是向量计算的指令不在考虑范围内；
- 状态中的程序计数器寄存器的值即为序列的下标；
- 汇编程序中所标注的标签会被忽略，在分支（跳转）语句中，目标标签会直接被替换为该标签在指令序列中所对应的下标，指示了下一条等待被执行的指令是序列中的第几条指令。

$Inst$ 中一条指令 i 的语义记作 $[\![i]\!]$，它是一个状态之间的映射，比如

`add x0, x0, x1` 的语义为：

$$s \mapsto s\left[s(\texttt{x0}) \mapsto s(\texttt{x0}) + s(\texttt{x1}), s(\texttt{pc}) \mapsto s(\texttt{pc}) + 1\right].$$

根据一条指令的语义，可以定义出汇编程序的语义。

定义 3.2 (汇编程序的小步操作语义)： 称在程序 P 中从状态 s 出发执行 k 条指令可以到达 s'（记作 $P \vdash s \rightsquigarrow^k s'$），如果

- $k = 0$ 而 $s = s'$，或者
- $k > 0$ 而 $P \vdash s \rightsquigarrow^{k-1} t$，$t(\texttt{pc}) < |P|$ 而且 $[\![P(t(\texttt{pc}))]\!](t) = s'$。

这里没有直接定义一个汇编程序的含义，而是给出了程序执行 k “步”（k 条指令）的含义：根据 `pc` 来取出指令，然后执行指令，如此 k 次。

例 3.2 (汇编程序的小步操作语义)： 考虑一个简单的汇编程序 P，它的指令序列如下所示。

```
add x0, x0, x1
add x0, x0, x2
```

那么，有 $P \vdash s_0 \rightsquigarrow^1 s_1$，$P \vdash s_1 \rightsquigarrow^1 s_2$，$P \vdash s_0 \rightsquigarrow^2 s_2$，其中

$$\begin{aligned} s_0 = &\ \{\texttt{pc} \mapsto 0, \texttt{x0} \mapsto 3, \texttt{x1} \mapsto 2, \texttt{x2} \mapsto 1, \cdots\} \\ s_1 = &\ \{\texttt{pc} \mapsto 1, \texttt{x0} \mapsto 5, \texttt{x1} \mapsto 2, \texttt{x2} \mapsto 1, \cdots\} \\ s_2 = &\ \{\texttt{pc} \mapsto 2, \texttt{x0} \mapsto 6, \texttt{x1} \mapsto 2, \texttt{x2} \mapsto 1, \cdots\} \end{aligned}$$

3.2.3　程序等价性

程序等价性指的是对于任意输入，两个程序都会返回相同的输出，形式化的定义如下。

定义 3.3 (汇编程序的等价性)： 称汇编程序 P_1 和 P_2 是等价的，如果任意程序状态 s, s_1, s_2 使得

- $s(\texttt{pc}) = 0$（s 为初始状态），
- $s_1(\texttt{pc}) = |P_1|$（s_1 为程序 P_1 的终止状态），
- $s_2(\texttt{pc}) = |P_2|$（s_2 为程序 P_2 的终止状态），
- 存在自然数 $k_1 \geqslant 0$，使得 $P_1 \vdash s \rightsquigarrow^{k_1} s_1$（在程序 P_1 中从初始状态 s 出发执行若干条指令后可以到达 s_1），
- 存在自然数 $k_2 \geqslant 0$，使得 $P_2 \vdash s \rightsquigarrow^{k_2} s_2$（在程序 P_2 中从初始状态 s 出发执行若干条指令后可以到达 s_2），

有

$$\forall r \in Reg \setminus \{\texttt{pc}\}, s_1(r) = s_2(r).$$

```
; P1                    ; P2                    ; P3
add x0, x0, #2          add x0, x0, #1          add x0, x0, #3
                        add x0, x0, #1
```

图 3.1　将寄存器 x0 自增（AArch64 汇编代码）

例 3.3 (汇编程序的等价性)： 考虑图 3.1中的三个 AArch64 汇编程序 P_1、P_2 和 P_3，P_1 和 P_2 是等价的，P_1 和 P_3 不等价，P_2 和 P_3 不等价。

注意，在讨论等价性时，有一个特殊情况：执行过程不终止。定义 3.3 采取了一种较为简单的处理方法，仅考虑两个程序的执行都终止的情况，而忽略其中一个程序会不终止的情况。

然而，即便程序的执行会终止，其运行所需的时间可能取决于输入，有可能会超出实际执行所能承受的范围，同时也可能给等价性的验证带来极大困难。因此，吸纳限界模型检测的思想，本文引入了一个时间限制（或者是模拟时间限制）的机制。汇编程序可以通过执行的指令条数来粗略估计运行时间。超过时间限制的情况同样被排除在等价性的讨论范围之外。本研究将这一改进过的等价性称作限界等价性，其形式化的定义如下。

定义 3.4 (汇编程序的限界等价性)： 称汇编程序 P_1 和 P_2 是 b-限界等价的（$b \in \mathbb{Z}_+$），如果任意程序状态 s, s_1, s_2 使得

- $s(\mathsf{pc}) = 0$（s 为初始状态），
- $s_1(\mathsf{pc}) = |P_1|$（s_1 为程序 P_1 的终止状态），
- $s_2(\mathsf{pc}) = |P_2|$（s_2 为程序 P_2 的终止状态），
- 存在自然数 $0 \leqslant k \leqslant b$，$P_1 \vDash s \rightsquigarrow^k s_1$（在程序 P_1 中从初始状态 s 出发执行不超过 b 条指令后可以到达 s_1），
- 存在自然数 $0 \leqslant k \leqslant b$，$P_2 \vDash s \rightsquigarrow^k s_2$（在程序 P_2 中从初始状态 s 出发执行不超过 b 条指令后可以到达 s_2），

有

$$\forall r \in Reg \setminus \{\mathsf{pc}\}, s_1(r) = s_2(r).$$

例 3.4 (汇编程序的限界等价性)： 考虑图 3.1中的前两个 AArch64 汇编程序 P_1 和 P_2，P_1 和 P_2 是 2-等价的，但它们不是 1-等价的。

3.3　符号执行的同步协调机制

本节将介绍一种在等价性验证过程中符号执行的同步协调机制：两个程序在符号执行的过程中，一个程序走一步，另一个程序跟着走一步，彼此用对方的路径条件来约束自己所能走的路径。

执行时的步长可以以指令（语句）为单位，或以基本块为单位，本研究以基本块为单位。

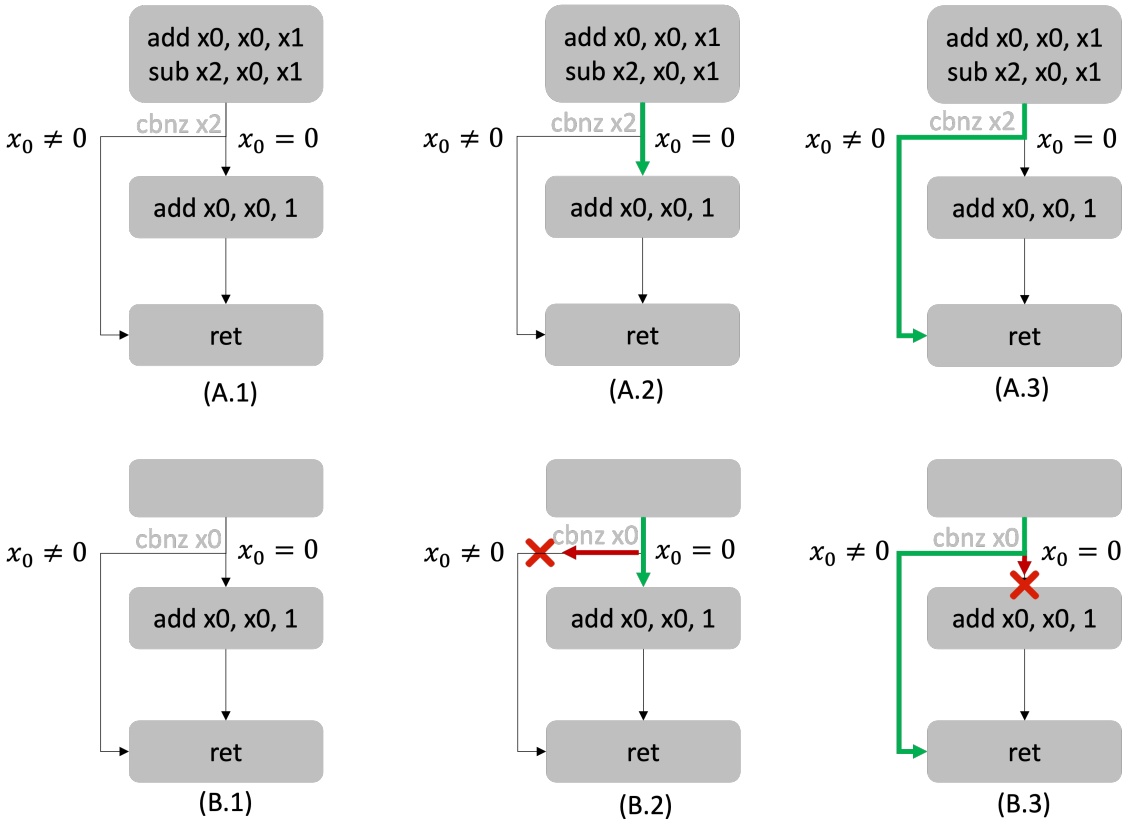

图 3.2　符号执行的同步协调机制

下面通过图 3.2 中的示例来进一步介绍直观。图中以控制流图（control-flow graph）的形式展示了两个程序 A 和 B。第一行展示了对程序 A 的符号执行，而第二行展示了对程序 B 的符号执行。在执行过程中，程序 A 先进行了一步操作，而程序 B 则能够利用程序 A 已知的路径条件，从而及时地避免探索不必要的路径。

控制流图中的每个节点代表一个基本块（basic block），基本块中是无跳转的汇编语句序列，比如 `add x0, x0, 1`，有向边指示在执行完一个基本块后可能到达的下一个基本块。基本块可能具有一条或两条出边。当存在一条出边时，边上不会标注条件，表示在执行完该基本块后将无条件执行指向的下一个基本块；当存在两条出边时，边上会标注两个互补的条件，如图中的 $x_0 = 0$ 和 $x_0 \neq 0$，它们被称为守卫条件（guard condition）。这意味着在执行完该基本块后，根据 $x_0 = 0$ 是否成立，来决定选择哪条出边。

本研究中的控制流图与标准控制流图的区别是：

- 本研究会额外添加一个退出块（exit block），用来标记程序的返回节点，所有的 `ret` 语句都会被连接到那个退出块上。

算法 3.1 同步协调的符号执行

输入： 当前正要执行程序的基本块 B_{cur}、符号状态 S_{cur}、当前闲置程序的基本块 B_{idle}、当前符号状态 S_{idle}、已有路径条件 pc

输出： 是否等价，如果不等价，返回一个可以区分两个程序的输入。

```
1:  function InterleavedSymExec(B_cur, S_cur, B_idle, S_idle, pc)
2:      if B_cur is the exit block ∧ B_idle is the exit block then
3:          if S_cur = S_idle is valid then
4:              return true
5:          else
6:              return false, counter example as input
7:      if B_cur is the exit block then
8:          return InterleavedSymExec(B_idle, S_idle, B_cur, S_cur, pc, c)        ▷ 切换执行程序
9:      S'_cur ← BlockSymExec(B_cur, S_cur, pc)
10:     for each successor B'_cur of B_cur do
11:         gc ← the guard condition from B_cur to B'_cur
12:         if pc ∧ gc is satisfiable then                    ▷ 根据现有路径条件剪枝
13:             res ← InterleavedSymExec(B_idle, S_idle, B'_cur, S'_cur, pc ∧ gc, c) ▷ 切换执行程序
14:             if res is not true then
15:                 return res
16:     return true
```

- 守卫条件会以输入寄存器的值为表达式来标记出，比如在程序 A 中，在分支时，虽然是根据 `x2` 当时的值来判断是否跳转，但 `x2` 当时的值是 `x0` 最初的值，所以这里的守卫条件是 $x_0 = 0$ 和 $x_0 \neq 0$。

在图 3.2 中，首先由程序 A 执行一步，然后由程序 B 执行一步，如此交替执行，所探索的路径用绿色标注。红色标出了被剪枝的路径探索。在 (B.2) 中，由于程序 A 已经收集了路径条件 $x_0 = 0$，因此程序 B 可以利用这个条件作为已知信息，知道此时无需探索条件为 $x_0 \neq 0$ 的路径。类似地，在 (B.3) 中，由于程序 A 已经收集到了路径条件 $x_0 \neq 0$，程序 B 可以知道此时无需探索条件为 $x_0 = 0$ 的路径。这种方式就不再需要枚举 $2 \times 2 = 4$ 条路径，而只需考虑 2 条路径。

主体算法如算法 3.1 所示。InterleavedSymExec 是一个递归函数，按轮次来深度优先遍历两个程序。每一轮中，先执行一步当前要执行的程序，然后将控制权转交给当前闲置的程序，这样交替执行，直到两个程序都到达退出块。算法的输入包括当前要执行的程序的基本块 B_{cur} 和当前符号状态 S_{cur}，以及闲置程序的基本块 B_{idle} 和符号状态 S_{idle}，还有已知的路径条件 pc，它是两个程序的控制流图路径上所有条件的合取。第 4-8 行是处理一个完整路径对的情况，这时两个程序都已经到达退出基本块，便来借助可满足性模理论求解器来判断两个程序的符号状态是否一定相同，如果相同，说明这一对路径是等价的，否则，求解器会返回一个反例，它是一个可以区分这两个程序的输入。第 9-10 行处理特殊情况。如果当前程序已

算法 3.2 借助双方路径条件剪枝的单基本块符号执行

输入： 基本块 B，符号状态 S，路径条件 pc
输出： 从 S 出发执行完该基本块后的符号状态

```
1:  function BlockSymExec(B, S, pc)
2:      for instruction i in B do
3:          switch i do
4:              case v̂ := if c then ê1 else ê2
5:                  if pc ∧ S(c) is unsatisfiable then
6:                      S ← S[v̂ ↦ S(ê2)]          ▷ 根据现有双方路径条件删减符号状态
7:                  else if pc ∧ ¬S(c) is unsatisfiable then
8:                      S ← S[v̂ ↦ S(ê1)]          ▷ 根据现有双方路径条件删减符号状态
9:                  else
10:                     S ← S[v̂ ↦ ite(S(ĉ), S(ê1), S(ê2))]
11:             case v̂ := ê
12:                 S ← S[v̂ ↦ S(ê)]
13:     return S
```

经到达退出块，就切换执行程序，将当前闲置的程序变为要去执行的程序，然后继续执行。第 11 行调用辅助函数 BlockSymExec 来符号执行一个基本块，根据输入的符号状态 S 计算出执行完该基本块后的符号状态，这里也会借助已知的路径条件 pc 来帮助简化符号状态，具体的工作原理我们会在之后详细介绍。在第 12-17 行，算法会遍历当前要执行的程序的基本块的所有后继基本块，对于每一个后继基本块，记从当前基本块到这个后继基本块的守卫条件为 gc，然后判断已知的路径条件 pc 和守卫条件 gc 是否是可满足的，如果不可满足，在这里剪枝，否则就继续递归调用 InterleavedSymExec，切换执行程序，直到两个程序都到达退出块。

算法 3.2 介绍了辅助函数 BlockSymExec，它是用来符号执行一个基本块的，除了第 3.1 节中所介绍的之外，对于条件执行指令（第 4 行），比如 AArch64 中的 `cinc`（conditional increment），一般来说需要用到 ite（if-then-else）来编码（第 10 行）。然而，由于本研究额外引入了另一个程序的路径条件，因此可以根据额外引入的路径条件来简化符号状态，从而省去 ite 中不必要的分支（第 5-8 行）。

一个未在算法（算法 3.2 和算法 3.1）体现出来的细节是限界模型检测，如定义 3.4所述，本研究会将指令的执行数量限制在一个阈值内，例如，只考虑指令数量不超过 10000 的路径，以应对程序可能无限循环的情况。

3.4　本章小结

本章介绍了符号执行，并给出了汇编程序等价性的一个严格但简化的定义，即限界等价性（定义 3.4）。该定义规定，对于任意相同的输入，在两个汇编程序都

在执行不超过一定数量的指令后终止的情况下，它们的输出应该相同。

基于这个定义，本章进而介绍了一种在等价性验证过程中符号执行的同步协调机制，其可以更高效地验证两个程序的限界等价性。

第 4 章　基于代码插桩的汇编程序沙箱

本章会介绍超级优化中的一个重要模块的实现：执行汇编程序的沙箱环境。

沙箱的总体设计如图 4.1 所示。对程序行为的监控和保障是通过应用代码插桩技术来实现的。沙箱接受一段汇编程序和一个输入测例，然后在这个测例上执行程序，并测量程序的运行时间。在可能出现非法行为（如异常和死循环）时，沙箱会终止程序。如果程序正常执行，沙箱最终会返回程序的输出状态；否则返回一个错误码。在这里，程序的输入测例和输出状态指的是计算机系统的一个快照，包括寄存器和内存中存储的值。

4.1　设计目标

本节会介绍为什么本系统需要这样一个会执行并监控程序的沙箱环境，先从用于引导程序空间探索的评估函数开始讲起，说明为什么沙箱对于超级优化评估函数的计算是必要的；在厘清沙箱在执行程序时所需要满足的保障之后，会分析为何需要应用代码插桩技术，以完成沙箱的功能保障。

在超级优化问题中，对程序的评估包括两方面：正确性和效率。正确性指的是当前程序的含义是否与待优化程序一致，而效率则指程序的期望执行时间。

对程序的评估值有以下两点要求：

- **高效性**：由于超级优化系统需要在探索程序空间时即时评估程序，因此能够更快地评估程序将意味着能够更快地探索程序空间，从而更有可能以及更快地找到一个优化版本的程序；
- **平滑性**：有连续的评估值，以用于引导对程序空间的探索。比如说对于正确性，用一个正实数来度量两个程序之间的 “距离”，而不是用 0/1（不相等/相等）这种二元值。

注意，对于正确性项和效率项，这两点要求都需要被满足。

按是否执行程序，评估的方法可以被分为静态（不执行程序）和动态（执行程序）两类。接下来将介绍静态评估方法以及其缺陷，以阐明为何本研究需要采用动态方法来评估程序。

评估程序正确性——也就是两个程序的语义是否等价——的静态方法是形式化验证，这种方法可以严格判断程序正确与否。但判断两个程序是否等价是一个不可判定问题，具体来讲，对于一个图灵完备的程序语言，不存在一个算法总可以在有限时间内判定任意两个用该种语言撰写的程序是否是等价的，尽管随着形式

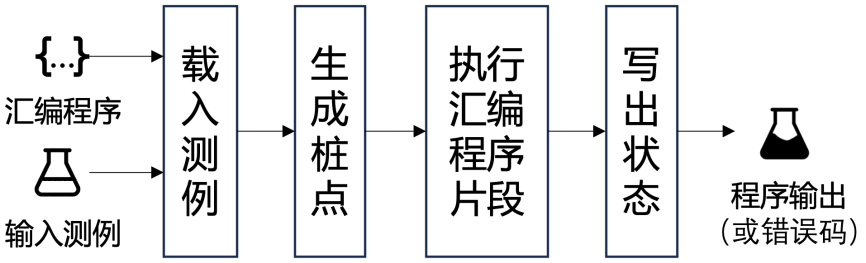

图 4.1　沙箱总体设计

验证技术的发展，一些启发式的方法[37-38,41-42,60-61]有机会自动判定一部分程序之间的等价性，但它们往往耗时极大，即便对于两个十余行无循环的汇编代码片段，也会花费数十秒的时间[22]，因此不满足高效性的要求。而且，形式化验证的方法只能输出一个二元结果（正确/不正确），也不满足平滑性的要求。

评价程序效率的基本静态方法是数指令条数。当然，不同指令的执行时间也是不同的，例如，移动指令 `mov` 的延迟通常不足乘法指令 `mul` 的延迟的一半[62]。因此，根据指令类型与执行延迟的映射表[62-63]，可以计算程序所有指令的预估延迟的累和，作为程序运行时间的估计，这比简单地数指令条数更精细一些。更精细的静态方法会考虑 CPU 的流水线，将指令之间的资源竞争导致的流水线阻塞考虑在内，以 CPU 执行的预期周期数作为程序运行时间的估计。例如，静态分析汇编代码效率的工具 LLVM MCA（machine code analyzer）就是这样的工具。但这些静态方法都面临一个问题，即无法估计一条指令被执行的次数。例如，当程序中存在循环时，循环具体执行多少次通常由输入决定，因此静态方法难以准确地估计包含循环的程序的运行时间。由于汇编程序不直接包含循环结构，控制流由跳转指令决定，也就是说，当程序中存在跳转指令时，静态方法就难以准确估计程序的运行时间。

以上两段的分析已经建立了对动态评估方法的需求，由于超级优化系统中要评估的程序可以是程序空间中的任意一个程序，为了安全地执行程序，并分析程序实际执行的时间开销，在超级优化系统中是需要一个沙箱环境的。

沙箱需要保障：

1. 不会影响主程序（超级优化器）的执行，比如沙箱内部的异常不会导致主程序崩溃；
2. 能够发现程序可能进入死循环的情况，并在此时及时终止程序的执行；
3. 可以准确地测出代码的执行时间，需要达到指令级别的准确度，这意味着需要能够区分出一条汇编指令是否执行所带来的时间差异。

这 3 条所需要满足的保障中，前 2 条有较为直接的解决方案，比如可以通过启动子进程让待评估代码在子进程中执行，借助操作系统的进程间隔离机制保护

主程序，设定时限，如果子进程没有在一定时间内完成任务，就通过系统调用杀死子进程；最困难的是第 3 条，因为代码的执行时间可能非常短。比如用于个人电脑的最新苹果 M3 芯片，其高性能 CPU 的最高时钟频率为 4.05 GHz，这意味着执行一条延迟为一个时钟周期的加法指令仅需 $1/(4.05 \times 10^9) \approx 0.25$ ns。

计算机上对程序的实际执行时间的测量需要用到硬件上的 “计时器”，一般计算机上有两个承担 “计时器” 的硬件模块：实时时钟（RTC, real-time clock）和 CPU 时钟。下面会分别介绍这二者，并论述对于本工作而言，二者的精度均是有限的。

- 实时时钟一般作为一个独立模块位于主板上，对于计算机系统来说是一个外设。频率一般为 32.768 kHz，与石英钟相同，远小于 CPU 频率（个人电脑的 CPU 频率一般在 GHz 级别），故其精度不足以准确衡量耗时仅有几纳秒的汇编代码。
- CPU 时钟是 CPU 内部的一个计时单元，来记录 CPU 执行的周期数，比如 x86-64 架构中的 TSC（time stamp counter）和 AArch64 架构中的通用计时器（generic timer），它们都是一个 64 位的寄存器，记录了 CPU 自上次重置以来的时钟周期数。不过，这种方法会受到不同 CPU 架构的影响，对相同的汇编代码片段，不同的 CPU 架构会在流水线上作不同的处理，例如，虽然现在个人电脑所使用的 CPU 架构往往是可以在运行时动态调整指令的执行顺序的，但也有一些用于低功耗高性能场景的 CPU 如 ARM Cortec-A53 总是会顺序执行指令。由于在本工作中，在超级优化时并不清楚程序具体会在什么型号的 CPU 上执行，所以尽管借助 CPU 时钟可以很精准地获取程序的真实执行时间，但本工作也并未采用此种方法。

因此，本工作使用基于代码插桩的方法来作动态指令计数。简单来说，就是在待评估程序开始执行前额外准备一个计数器并清零，在待评估程序的每条指令前插桩，让计数器自增。这样，在程序执行结束后，就可以知道待评估程序执行了多少条指令。依次运行所有测试用例，然后取平均值，以此作为对程序执行时间的估计。

4.2　桩点设计

本节会介绍本工作的桩点设计，即在哪些程序位置插桩以及插入什么样的代码。

图 4.2 展示了本工作的桩点设计，白色部分是待评估程序，灰色部分是插桩代码。桩点依据位置可分为三类：前处理桩点、逐指令桩点和后处理桩点。前处理桩点在整个待评估程序之前，逐指令桩点是在每条指令之前，后处理桩点是在整个

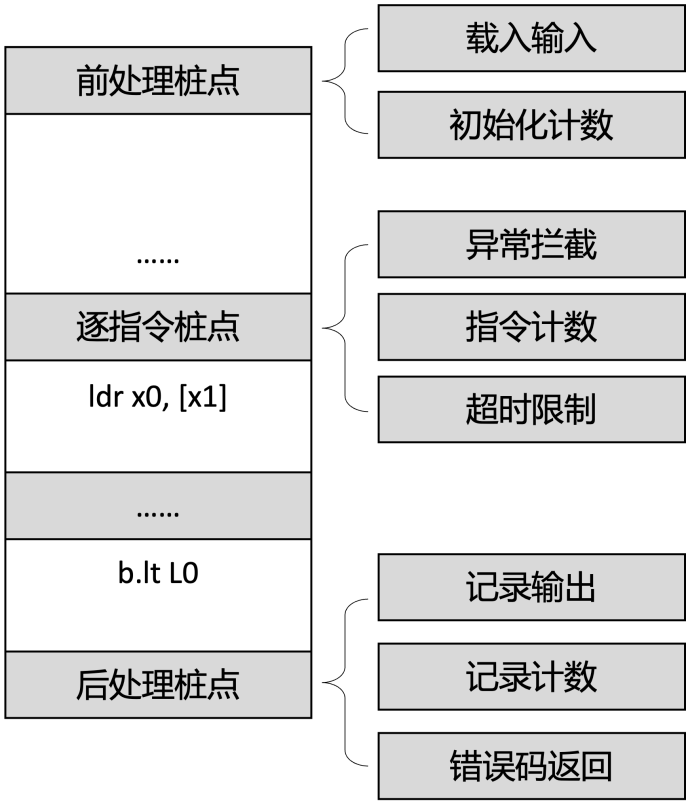

图 4.2　桩点设计

待评估程序之后。前处理桩点负责载入测例，以及初始化沙箱会额外占用的功能寄存器；逐指令桩点在待评估程序的每条指令之前，包括异常拦截（当将要发生异常时，终止程序）、指令计数（计数器自增）和超时限制（在程序运行过久之后掐断）；后处理桩点负责将沙箱功能寄存器的值记录，以及模拟异常抛出返回错误码。

本节接下来会按功能依次介绍插桩代码。

4.2.1　输入状态的载入和输出状态的记录

在前处理桩点中，需要将测例中的寄存器值载入到相应的寄存器中，包括通用寄存器和会影响指令执行的标志寄存器，还需要将测例中内存的值载入到相应内存。在后处理桩点中，需要将当前的程序状态写出，也就是待评估程序执行完毕后的状态，包括内存和寄存器。需要注意的是，沙箱额外使用的寄存器不算在状态内。

4.2.2　异常拦截

由于沙箱是在同一个线程中探索程序空间并执行在程序空间中发现的程序，如果在执行待评估程序时遇到异常（比如访问非法内存），可能会导致整个程序崩

代码 1 异常拦截及模拟抛出

```
1   ; 逐指令桩点
2   if E1 will occur in this instruction,
3     jump to LABEL_E1
4   if E2 will occur in this instruction,
5     jump to LABEL_E1
6   if E3 will occur in this instruction,
7     jump to LABEL_E3
8   ; ...
9
10  ; 后处理桩点
11  set the return value as 0
12  jump to the function epilog
13LABEL_E1:
14  set the return value as 1
15  jump to the function epilog
16LABEL_E2:
17  set the return value as 2
18  jump to the function epilog
19LABEL_E3:
20  set the return value as 3
21  jump to the functino epilog
22; ...
```

溃，从而终止超级优化的过程。然而，在整个程序空间中，这种会有异常的畸形程序是大量存在的。因此，沙箱需要一种异常拦截机制，以确保这种异常不会影响主程序的进行。此外，由于不同的异常意味着不同的行为，知道待评估的程序会抛出哪种异常也是评估的一部分。因此，本工作通过代码插桩设计了一种拦截并模拟异常抛出的机制。

代码 1 展示了异常拦截及模拟抛出机制的插桩代码的基本设计，逐指令桩点会在每条指令之前检查这条指令是否会抛出异常，如果会，就跳转到后处理桩点中对应的异常处理代码。后处理桩点中对应的异常处理代码会将用于存放返回值的寄存器（比如 x86-64 中的 `rax` 和 `x0`）设置为相应的异常码，然后跳转至函数尾言；如果没有异常，返回值为 0，表示待评估的程序正常执行至终止（第 11-12 行）。

4.2.3　指令计数

如第 4.1 节所述，本工作以待评估程序执行的指令数量作为其运行时间的估计。为了动态地统计执行的指令数量，本工作额外使用一个不常用的通用寄存器作为计数器，在待评估程序的每条指令执行之前将计数器自增。在前处理桩点，需

要将计数器清零；在后处理桩点，需要将计数器的值记录到超级优化器主程序中的一个变量中。

4.2.4　超时限制

为了避免程序陷入死循环，需要设置一个“时限”，当程序运行超过这个时限时，就中断程序的执行。由于本工作以指令执行条数作为时间的度量，所以可以设定一个指令执行条数的阈值，当执行的指令条数超出阈值时，便认为程序已超时。因为已经有了异常处理机制，程序超时可以视为一种异常（超时异常）来处理。在后处理桩点中为加入超时异常的处理代码片段，在逐指令桩点中，如果判断超时，则跳转到后处理桩点中的超时异常处理代码片段。该处理代码片段会设置返回值为超时异常码，然后跳转至函数尾言。

下面额外指出几个实现中需要特别注意的细节：

- 逐指令桩点不能影响 CPU 的标志寄存器，比如比较指令 `cmp` 和条件跳转指令 `b.eq` 之间的插桩代码如果改变了标志状态，就会影响程序的行为，`b.eq` 是否跳转不再由 `cmp` 指令的结果决定。比如说，对比插桩前的汇编代码

  ```
  cmp x0, x1
  b.eq L0
  ```

  和插桩后的汇编代码

  ```
  cmp x0, x1
  ; --- 逐指令桩点
  cmp x0, #2
  ; ---
  b.eq L0
  ```

  ，二者中 `b.eq` 指令的行为可能不同：前者是当 `x0` 与 `x1` 相等时跳转，而后者是当 `x0` 等于 2 时跳转。为了避免逐指令桩点影响 CPU 的标志寄存器，可以在逐指令桩点的最开头将标志寄存器的值保存在栈中，然后在桩点的最后再弹栈，恢复标志寄存器的值。
- 为了防止待评估程序篡改主程序的内存，沙箱需要在主程序的内存中分配和划分出可供待评估程序读写的区域。当待评估程序读写内存时，利用异常拦截机制来检查访问的内存是否在主程序允许的区域范围内。如果不在范围内，则视为非法内存访问，并模拟抛出异常。
- 由于沙箱会额外占用寄存器，在待评估程序要用到被沙箱占用的寄存器的时候，类似于编译器在分配寄存器时的寄存器溢出（register spilling），可以将需要用到的被占用的寄存器的值临时存储到栈中，使用完后再弹栈。

4.3　讨论

本节会讨论目前的沙箱实现方法的局限性，以及为了克服这些局限性值得在未来尝试的解决方法。

首先，目前通过动态指令计数来评估时间的方法不够精准，现代 CPU 有复杂的机制，比如流水线机制，在流水线中指令的类型和指令之间的依赖关系都会影响到程序运行的时间。

一个更精确一些的评估方法是不仅仅统计指令条数，还统计出每种类型的指令的条数，然后根据某种 CPU 上每种类型的指令的延迟表，将所有指令的总延迟，作为对程序运行时间的估计。比如说在 Cortex-X1 上，`mul` 的延迟为 2 个时钟周期，`mov` 的延迟为 1 个时钟周期，那么，AArch64 汇编程序

```
mul x2, x1, x0
mov x0, #1
```

的运行时间的估计就是 3 个时钟周期。

但这种评估方法依然无法考虑到 CPU 完整的流水线机制，比如说 CPU 在执行 `mul` 的后半阶段同时也可以执行 `mov`，因为当 `mul` 获知了 `x0` 的值后，其值就不再被需要，所以可以尽早将 `x0` 修改为 1。如果要将完整的 CPU 的机制纳入考虑，包括流水线机制和分支预测等复杂设计，一种可行的方案是借助 CPU 模拟器，比如 Gem 5[64]，在各种不同的 CPU 架构上、若干输入测例上，借助模拟器来作性能统计，不过这样会大大增加评估的时间开销。

当然，即便是完善实现的 CPU 模拟器，与实际的物理执行也会有出入，如果可以确定唯一的目标 CPU 架构，并且有条件在目标 CPU 架构上执行代码，那么通过 CPU 时钟来测量是最准确的方法。更具体地讲，需要撤销逐指令插桩，在执行待评估程序前后读取 CPU 时钟，再将前后的时间戳相减，以计算出运行时间区间长度。CPU 时钟一般可以通过读取一个系统寄存器得到，比如 AArch64 架构中的 `cntpct_el0`（counter-timer physical count register, exception level 0）① 和 x86-64 架构中的 `tsc`（time-stamp counter），其中记录了 CPU 从上电开始经过的时钟周期数。

其次，目前本工作在汇编语言特性上的支持尚不完善，包括只支持跳转到静态标签的跳转指令，不支持系统调用等。静态标签指的是在汇编代码中通过字符串和冒号标记出的标签，例如代码 1中第 13 行的 `LABEL_E1`，这种标签在编译时会被解析为一个代码行所在的地址，因此可以直接跳转到这个地址。而动态标签

① EL0（exception level 0）是 ARM 架构中定义的四个异常级别之一，用于指定运行级别或权限级别，EL0 是用户模式，用于运行应用程序，是最低级别。`cntpct_el0` 这个系统寄存器位于 EL0 级别，就意味着它可以在任何级别被读取到。

指的是在代码执行时才会确定的标签，例如在运行时根据某个寄存器运行时的值来决定跳转到哪里；系统调用是操作系统提供的一些特定服务，例如读写文件等，这会导致 CPU 从用户态切换到内核态，并且可能带来难以预期的副作用。

4.4　本章小结

本章介绍了一个用于评估程序的沙箱环境，该沙箱环境通过应用代码插桩技术来监控程序的行为，包括估计待评估程序的执行时间、拦截即将出现的异常以及中断可能死循环的超时程序等。这个沙箱环境可以安全地在测例上执行程序，以评估程序的正确性和效率。

第 5 章　系统实现与实验

为了检验本研究所提出的方法的有效性，本工作实现了一个基于 AArch64 架构的超级优化原型系统，收集了一个基准数据集并在其上进行了实验评估。

5.1　系统实现

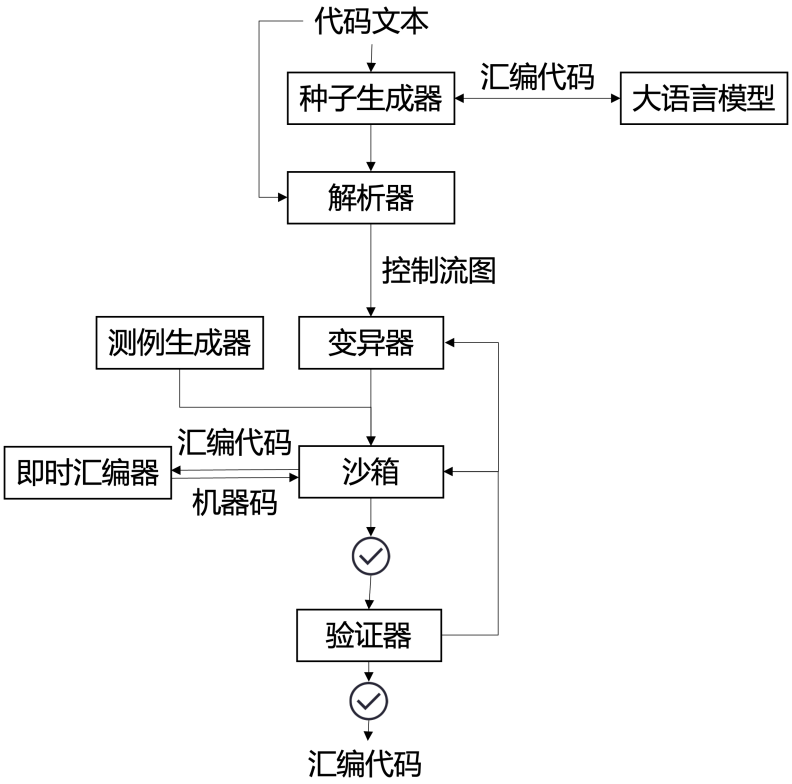

图 5.1　系统架构

本节介绍了所实现的原型系统，其整体架构如图 5.1 所示。输入为汇编代码文件。首先，借助大语言模型来初步优化程序，生成种子。待优化代码和大语言模型生成的代码均为文本格式，读入内存后即为字符串，将会被解析器解析为控制流图，这是系统内部程序的表示结构。接下来进入随机化搜索的主循环。这里有两层循环：内层循环中，变异出的程序会交由一个沙箱来评估。沙箱借助一个即时汇编器将控制流图编译为可以直接执行的机器码，然后在测例生成器生成的测例集合上执行程序。如果程序通过了所有的测例，就会进入外层循环。对于通过所有测例进入外层循环的程序，会被交由一个等价性验证器。经过严格形式化验证的汇编代码会被输出，否则，验证器会生成一个反例，以补充测例集合。

本系统主要使用 C++ 开发，在代码行数上共计有七千余行，所依赖的第三方库包括：

- antlr[①]：为了实现前端解析器，本系统采用了 antlr 来自动生成解析器代码，它能够根据给定的语法规则自动生成解析器代码。
- asmjit[②]：为了生成和操作低级机器码，本系统依赖于 asmjit 库，这是一个运行时代码生成库。asmjit 使得直接在内存中生成可执行代码变得简单而高效，为本系统提供了强大的底层支持。
- argparse[③]：命令行参数的解析由 argparse 库处理，它可以方便地定义和解析命令行参数。
- tomlplusplus[④]：配置文件的解析则采用 tomlplusplus，这是一个专门用于处理 TOML（Tom's obvious minimal language）语言撰写的配置文件的库，使得用户能够以简洁、易于理解的格式定义配置。
- spdlog[⑤]：日志记录功能通过 spdlog 实现，spdlog 是一个快速的日志记录库，支持多种输出目标和格式，会方便开发和调试。
- nlohmann[⑥]：处理 JSON（JavaScript object notation）的任务交给了 nlohmann JSON 库，它可以将 C++ 中的数据结构序列化成 JSON 数据，或者是将 JSON 数据反序列化成 C++ 中的数据结构，这样就可以通过 REST API（representational state transfer application programming interface）与大语言模型交互。
- z3[⑦]：z3 是一个主流的自动定理证明器，也就是 SMT solver（可满足性模理论求解器），本系统的验证器组件基于 z3 开发，由 z3 来自动判断逻辑公式是否成立。

在实现中，为了提高效率并便于调试，将需要考虑的程序空间适当缩小是必要的。本系统采用了两种启发式方法来缩减程序空间：

- 缩小选项范围：圈定操作码池、寄存器池和立即数池。合法的程序只能使用操作码池中的操作码、寄存器池中的寄存器和立即数池中的立即数。
- 依据数据流来检查程序的规范性：只有输入寄存器在程序入口处活跃，也就是说，程序不会使用到未被定义的寄存器。

这两种方法都会被应用于大语言模型所生成的种子以及变异器进行的程序变异中。

① https://www.antlr.org/
② https://asmjit.com/
③ https://github.com/p-ranav/argparse
④ https://marzer.github.io/tomlplusplus/
⑤ https://github.com/gabime/spdlog
⑥ https://github.com/nlohmann/json
⑦ https://github.com/Z3Prover/z3

5.1.1　中间表示

本小节会介绍汇编程序在系统内部是如何表示的。

5.1.1.1　控制流图

```
1  is_leap_year:
2    tst     x0, #3
3    b.eq    .LBB0_2
4    mov     x0, xzr
5    ret
6  .LBB0_2:
7    mov     x8, 0x8f5c28f5c28f5c29
8    mov     x10, 0x28f5c28f5c28f5c
9    mul     x8, x0, x8
10   ror     x9, x8, #2
11   cmp     x9, x10
12   b.ls    .LBB0_4
13   mov     w0, #1
14   ret
15 .LBB0_4:
16   mov     x9, 0xa3d70a3d70a3d8
17   ror     x8, x8, #4
18   cmp     x8, x9
19   cset    w0, lo
20   ret
```

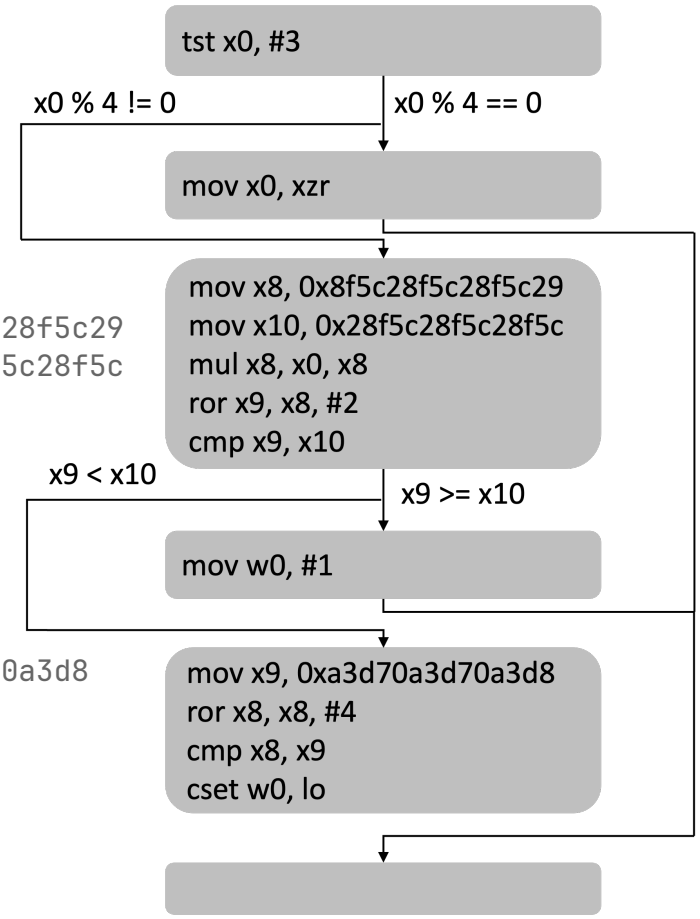

图 5.2　原始汇编代码及其控制流图示例

汇编程序在本系统中以控制流图的结构来存储，这种结构便于第 3 章中对符号执行的同步协调机制，以及第 5.1.4 节中对程序的变异。图 5.2 展示了一个例子，这个程序的功能是判断一个年份是否是闰年，原始的汇编代码如左图所示，其对应的控制流图如右图所示，每一个灰底圆角矩形表示一个基本块（basic block），也就是不含有分支跳转的指令序列——既不会有跳转从基本块内部发生，也不会跳转到基本块里 ① 。基本块之间的有向边被称作控制流边，表示在执行完前一个基本块后有可能会接着执行下一个基本块，边上标出的是走这条控制流边的条件，被称作守卫条件（guard condition），在图 5.2 中以 C 风格的布尔表达式来表示。一个

① 注意，如有根据寄存器决定跳转目标的指令，如 AArch64 架构中的 `blr` 或 x86-64 架构中的长跳转，哪一个程序位置有可能会被跳转到是难以静态分析出的。所以在本工作中，并未将此类在运行时确定跳转目标的跳转纳入考虑，仅根据标签和以标签为目的地跳转的指令来划分基本块。

基本块最多可能有两条出边，如果有一条出边，表示在执行完这个基本块后一定会沿着这条边走；如果有两条出边，两条出边上会标着互补的守卫条件，表示在执行完这个基本块后，会根据守卫条件的执行情况选一条出边来走，而且一定存在一条可以走的出边。

相较于图 5.2 ，实际实现还有两点额外的细节：

- 图 5.2 直接通过 `mov` 来表示给一个寄存器赋值。但由于 AArch64 每条指令的编码长度固定为 32 位，实际上这个赋值并不能通过一条指令来完成，比如 `mov x8, 0x8f5c28f5c28f5c29` 实际上应为

  ```
  mov     x8, #23593
  movk    x8, #49807, lsl #16
  movk    x8, #10485, lsl #32
  movk    x8, #36700, lsl #48
  ```

  这四条指令，通过给每 16 位依次赋值来达成给 64 位寄存器赋值的效果。
- 跳转指令在控制流图中并未展示出来，在实际实现中，跳转指令会且仅有可能会出现在基本块的末尾，也就是说，每个基本块的最后一条指令有可能会是跳转指令，且仅有最后一条指令可能是跳转指令。

相较于经典的控制流图，本系统中的控制流图会额外添加一个特殊的基本块，被称为“退出块（exit block）”。这是由于本系统不仅仅会考虑一个完整的函数体作为程序，也会考虑其中的一个程序片段：这二者的区别在于是否有 `ret` 指令。为了统一处理，即便在函数体（有 `ret` 指令的程序片段）中，其中的 `ret` 指令也会被忽略，并添加一个特殊的基本块作为退出块。如图 5.2 右所示，退出块不含任何指令，程序执行进入此基本块表示程序终止，函数体中原本的 `ret` 指令会被替换成一条指向退出块的控制流边。

5.1.1.2　类型系统

为了方便检查指令的合法性，本系统对于指令的操作数设计了一个简易的类型系统。

$$
\begin{aligned}
\mathit{operandType} \quad ::= \quad & \mathit{num} \\
| \quad & \mathit{reg} \\
| \quad & \mathit{shift} \\
| \quad & \mathit{cond} \\
| \quad & \mathit{label} \\
| \quad & \mathit{mem}
\end{aligned}
$$

操作数被分为六个类型：

- 立即数（*num*），如 `2123123`，本系统会将这个类型根据值域进一步划分子类型；
- 寄存器（*reg*），如 64 位通用寄存器 `x0`；
- 偏移量（*shift*），如 `1`，取值范围取决于作二进制移位的寄存器的位数，对于 64 位寄存器来说，只能是在 0 到 63 之间的整数 ①；
- 条件码（*cond*），如 `eq`，用于判断是否执行某些特定操作；
- 标签（*label*），在文本中是一个字符串，在解析过程中会被解析为一个独一无二的数字；
- 内存地址（*mem*），如 `[x0]`，表示寄存器 `x0` 中的值指向的内存地址。

借助这个简易类型系统，以及不同指令操作码所需的操作数的类型，还可以方便地直接生成一条合法指令。

5.1.1.3　规范性检查

为了缩小需要探索的程序空间，可以约定哪些程序是符合规范的。本系统会对大语言模型生成的种子程序以及在随机化搜索过程中变异出的程序进行规范性检查。如果程序不符合规范，系统会拒绝该程序，并要求生成方（大语言模型或者变异器）重新生成。

具体的规范性检查的方式参考了 STOKE[22]。在优化程序时，本系统会配置出哪些寄存器是程序的输入，哪些寄存器被视作程序的输出，在已知输入的情况下，可以要求程序满足规范仅当：*在程序入口处，只有输入寄存器是活跃的*，也就是说，只有输入寄存器的值可能在之后被使用到，其他寄存器在程序入口的值一定不会再被使用到②。

计算在程序入口处活跃的寄存器是一个经典的数据流分析问题，也就是活跃变量分析。其基本思想是逆序计算，利用靠后的程序位置的活跃寄存器集合，根据指令语义和数据流关系，来计算靠前的程序位置的活跃寄存器集合。然而，在存在循环时，不同程序位置之间会相互依赖，因此这种计算难以通过一遍遍历来实现。某些程序位置可能活跃的寄存器集合需要反复更新。

本系统采用了一种基于先进先出队列的实现方法，如算法 5.1 所示，借助一个队列来存放刚刚被更新、可用于更新其他基本块的基本块。算法的输入，寄存器集合 R，即为存放程序输出的寄存器集合。算法的主要计算目标是 $live_{in}$ 和 $live_{out}$，这两个从基本块到寄存器集合的映射，其中 $live_{in}$ 是基本块入口的活跃寄存器集

① 取值范围是否是 0 和 63 之间的全部整数，也与具体的指令有关，比如 `movk` 中移位立即数的取值范围是 $\{0, 16, 32, 48\}$。

② 简单起见，目前只实现了对寄存器的活跃性分析，并未将内存纳入考虑。

算法 5.1 计算程序入口处的活跃寄存器

输入： 已知在程序出口处活跃的寄存器集合 R
输出： 在程序入口处活跃的寄存器集合

```
1:  live_out[block_exit] ← R
2:  q ← [block_exit]
3:  while q is not empty do
4:      b ← pop the head of q
5:      L ← live_out[b]
6:      for all inst i in reversed(b) do
7:          L ← (L − kill[i]) ∪ gen[i]
8:      live_in[b] ← L
9:      for all p in preds[b] do
10:         if L ⊈ live_out[p] ∧ p ∉ q then
11:             push p into the tail of q
12:         live_out[p] ← live_out[p] ∪ L
13: end while
14: return live_in[block_entry]
```

合，$live_{out}$ 是基本块出口的活跃寄存器集合。在第 1 行，退出块的出口活跃寄存器集合被初始化为 R，其他的基本块的 $live_{in}$ 和 $live_{out}$ 会被初始化为 $\varnothing$（未在算法中显式标注出）。在第 2 行，工作队列 q 被初始化为仅包含退出块的单元素队列。第 3-13 行是算法主体循环：第 4 行取出位于当前队列头的基本块 b；第 5-8 行逆序枚举 b 中的指令从 $live_{out}[b]$ 计算出 $live_{in}[b]$，其中，$gen[i]$ 是指令 i 所读取的寄存器集合，$kill[i]$ 是指令 i 只写的寄存器集合；第 9-12 行枚举 b 的前驱基本块，如果该前驱基本块的出口活跃寄存器集合会被更新的话，就将其加入工作队列——当然，如果其已经在工作队列中，就不必加入了。

5.1.2　解析器

解析器负责读入文本格式的源代码，将其转为系统内的中间表示，也就是控制流图。

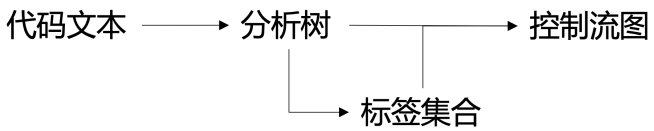

图 5.3　解析流程

解析器可以分为两个阶段，如图 5.3 所示，第一阶段将文本格式的代码转换为分析树（parse tree）[①]，第二阶段将分析树转换为控制流图。第一阶段的主体代码

① 分析树（parse tree）也被称作具体语法树（concrete syntax tree）。

由 antlr 根据语法文件自动生成。在第二阶段，为了将 antlr 生成的分析树转为控制流图，需要预先遍历一遍分析树，以获取汇编代码中所有的标签集合。这种预处理是必要的，因为在汇编代码中，可能会出现跳转到后面的标签的情况。例如，在图 5.2 左侧的第 3 行的跳转指令，会跳转到第 5 行的标签。通过这种预处理，可以确定指令中跳转到的标签是否在当前代码中存在，以及存在的话具体是哪一个标签。

本系统会这样将分析树（具体语法树）转为控制流图：

- 基本块：解析器顺序扫描指令序列及其中的标签。它不断将指令添加到当前基本块中。当遇到标签、跳转指令或返回指令时，当前基本块结束，然后重新启动一个新的基本块。
- 控制流边：解析时会构建三种控制流边，第一种是由 `ret` 指令解析成的指向退出块的控制流边，第二种是由跳转（分支）指令解析成的控制流边，第三种是指令顺次执行所带来的控制流边，比如在图 5.2 右侧，连接上下相邻两个基本块的控制流边，因为条件跳转指令的存在导致指令片段被分隔为不同的基本块，当跳转需要满足的条件不成立时，就会走这种控制流边。

5.1.3　测例生成器

测例生成器允许用户针对待优化程序来提供以下配置：

- 输入测例包含哪些寄存器和内存位置；
- 以何种策略来生成测例。

比如，测例生成器可以是为输入寄存器 `x0` 来在 $\mathbb{N} \cap [0, 2^{64})$ 均匀随机生成。

除了随机生成之外，有的程序还会对输入之间关系的有假设，比如函数的第二个整型参数是第一个数组参数的数组长度，所以这时就需要为这种假设撰写专门的生成测例的策略。

5.1.4　变异器

参考 STOKE[22-24,41]，本系统支持以下几种变异：

- **指令替换**：随机挑选一条指令，将其替换为另一条随机生成的新指令；
- **操作数替换**：随机挑选一条指令，再随机挑选其中的操作数，替换成同类型的其他值，比如说将立即数从 2 换成 5，或者是将寄存器从 `x0` 换成 `x1`；
- **操作码替换**：随机挑选一条指令，将其中的操作码随机替换为另外一个操作码，比如说将 `add` 换为 `sub`，这时需要保证当前的操作数对于新的操作码而言也是合法的；
- **全局交换**：随机交换程序中的两条指令；

- **局部交换**：随机交换一个基本块中的两条指令；
- **指令删除**：随机删除程序中的一条指令；
- **指令移动**：随机挑选一个基本块，将其中的一条指令移动到另一个位置（某两条相邻指令中间）。

在实际变异时，由于可以产生的新指令的范围十分庞大，本系统会从变异的指令码范围和操作数范围两个角度进一步缩减探索的程序空间：划定出变异所允许的操作码和操作数的范围，包括三个集合，操作码池，寄存器池和立即数池。在现有的实现中，指令池和寄存器池是可配置的，指令池默认包含所有指令，寄存器池包含所有通用寄存器，立即数池比较特殊，它包括待优化程序中的所有寄存器，和人为圈定的一些常见数字——$0, 1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6, 7, -7, 8, -8, 16, -16, 32, -32, 64, -64, 128, -128$——在变异时，将立即数操作数的合法范围与立即数池取交集，在这个交集中再随机选取一个立即数即可。

5.1.5　随机化搜索

本系统在 AArch64 架构上复现了 STOKE[22] 中的随机化搜索算法，它是 Metropolis-Hastings 算法的变体。简单来说，就是根据对变异结果的评估值来决定是否接受新的程序，以新的程序为当前程序去做变异：如果对新的程序的评估值更优了，那么就必然接受；如果更劣了，那么就根据变劣的程度来有概率地接受，越劣概率越低。

本系统会定义一个参数 β，在评估一个新程序时，将新程序的评估值减去 $[0, \beta]$ 中的一个随机数，再将其与当前程序的评估值作比较，根据比较结果（孰优孰劣）来决定是否接受变异。形式化地来讲，记当前程序评估出的代价函数为 c，新程序所评估出的代价函数为 c'，变异的接受条件为

$$c' - p < c, p \sim U[0, \beta] \tag{5.1}$$

其中 $p \sim U[0, \beta]$ 表示 p 为在 $[0, \beta]$ 这个区间中均匀随机采样所得。

程序的评估值包含两项，通过在沙箱中运行测例分析得出，并对所有测试用例的运行结果取平均。

- 正确性项：输出和待优化程序的输出在多少个比特上不同；
- 性能项：执行了多少条指令。

参考 STOKE[22]，随机化搜索被进一步划分为两个阶段：

1. 在这个阶段，对程序的评估只考虑正确性项，当在第一阶段中发现了一个正确的程序，也就是与待优化程序在所有测例上运行结果相同的程序，第一阶

段结束，进入第二阶段。

2. 在这个阶段，对程序的评估不仅考虑正确性项，还要考虑性能项。

在实现中，接收变异的随机阈值 β 在第一个阶段中会被设为 1，在第二个阶段会被设为 10。这是因为在第二阶段，程序是否正确不再是唯一的考量，此时就期望——或者说允许——对程序进行更激烈的变异。

5.1.6　讨论

本节会讨论实现中值得改进的方向，主要是用于帮助更快探索程序空间或者是减小所需探索的程序空间的启发式策略。

首先，来讨论第 5.1.1.3 节中所作的规范性检查。虽然借助这种规范性检查可以缩减所需探索的程序空间，但也可能导致错过优化程序的机会。这是因为虽然待优化程序和优化后程序都是符合优化的，但是可能存在一条变异路径，路径中某个中间程序是不符合规范的。比如说待优化程序为（输出为寄存器 x0）

```
add x2, x0, x1
sub x0, x2, x1
```

经由删除第 1 条指令的变异可以得到

```
sub x0, x2, x1
```

虽然这个程序不符合规范（因为用到了未定义的 x2 中的值），但再做一次删除指令的变异便可得到一个正确的优化——一个空程序。目前规范性检查的限制导致这种变异路径无法被发现。

第二，现有的测例生成器（见第 5.1.3 节）主要采用随机生成方法。然而，当输入代码涉及控制流分支时，随机生成的测试用例很难覆盖所有可能被执行的指令。在这种情况下，可以考虑再次使用第 3 章中所介绍的符号执行配合限界模型检测的方法。先收集所有可能的代码执行路径，如果存在循环，则仅考虑其执行不超过一定次数（例如 5 次）的情况。然后，将这些路径交由 SMT 求解器求解，以找到一条能够经过特定路径的输入测试用例。这个求解公式中的变量是输入的寄存器、内存和 CPU 的标志位。公式的形式是所有控制流守卫条件的合取，即路径上所有条件跳转语句的跳转条件或跳转条件的否定的合取。使用符号执行的方法来生成测例会需要更长的时间，但通常覆盖率更高。主要瓶颈在于 SMT 求解器，只要公式不过于复杂以至于超出求解器的能力，就可以得到沿着特定路径执行的输入，或者发现不存在这样的输入。

另外，在变异时，本系统将生成的立即数范围限制在一个较小的立即数池中，让本系统无法找到非平凡的立即数。一种或许值得尝试的方法是，可以再次利用 SMT 求解器来额外调整程序中的立即数。比如对于程序

```
sub     x1, x0, #48
and     x2, x0, #-33
sub     x2, x2, #65
```

可以将程序中希望调整的立即数设为变量 a, b, c，

```
sub     x1, x0, a
and     x2, x0, b
sub     x2, x2, c
```

这样，将程序与待优化程序等价视为一个约束，通过 SMT 求解器来找到一组可以满足等价性约束的 a, b, c。

5.2　实验

5.2.1　实验设置

5.2.1.1　基准数据集

基准数据集有两个来源：Hacker's Delight 和 Newlib。前者是在超级优化领域中先前工作所对比的一个数据集[21-22]，而后者是 C 语言标准库的一个面向嵌入式系统的开源替代实现。

Hacker's Delight　教科书《高效程序的奥秘（Hacker's Delight）》[44]中包含了 25 个程序，最初由 Gulwani[45] 从书中收集，用作程序合成和超级优化的基准数据集。书中讲解了一系列位运算技术，用于将原本复杂的算法实现为小型的无循环的位运算指令序列，例如“将最右边的 `1` 置 `0`”（p01_turnoff）和“计算两个 32 位无符号数乘积的高 32 位”（p25_higher_mul）等。这 25 个程序中不含有分支、循环和递归调用，仅由加减乘除和位运算所构成，其形式简单，不过也不易人类理解，即使是饱受训练的程序员，也很难直接看出这些程序的功能。

本工作沿用 STOKE 做实验所用的基准数据集中的 C 代码 ①，使用 GCC ② 来将其编译成 AArch64 汇编代码。特别的，在实验过程中，STOKE 原始的 p24（第 24 个程序）被发现有 bug，在返回语句时，将 `o11` 误用为了 `o10` ③。

由于一些程序的优化涉及向量寄存器，目前的实现尚不支持，对这部分程序（p18, p22, p23），本实验暂不考虑。具体而言，对这部分程序的优化需要使用指令 `popcnt`（统计二进制表示中 `1` 的数量），而在 AArch64 中，`popcnt` 只支持向量

① https://github.com/StanfordPL/stoke/tree/develop/examples/hacker
② 使用的是 ARM64 GCC 13.2.0，为目前最新版本的 ARM64 GCC。
③ https://github.com/StanfordPL/stoke/pull/1021

寄存器。

Newlib　Newlib[①] 是 C 语言标准库的一个开源替代实现，专门面向嵌入式设备所设计，嵌入式设备对于性能会有更高的要求，而且需要兼容各种不同的硬件型号，而且这也意味着其已经做了很多性能上的优化。

在其中收集待优化程序的方式是：在所有 C 代码文件（以.c 为后缀名）中筛选出包含函数体定义，但函数定义中不包含浮点数变量、内存读写和非内置函数[②]的函数调用 C 代码文件。共计收集到了 4 个这样的 C 代码文件，再使用 GCC[③]将其编译为 AArch64 代码文件。

这 4 个源代码文件分别为 divll.c（计算两个 64 位有符号整数的除法），ffsll.c（查找一个 64 位有符号整数最低边的 `1` 在第几位，计数从 1 开始，如果输入的整数为 `0`，则直接返回 0），iswxdigit.c（判断字符是否是十六进制数字）和 mulsi3.c（计算两个 32 位有符号整数的乘法）。其中，mulsi3.c 由于其汇编代码中包含高级条件跳转指令 `cbz` 和 `cbnz`，现有实现因为技术原因尚未支持，所以本实验也并未将其纳入考虑。

为了让这些代码片段能够被先有实现直接支持，还需要对这 3 个源代码文件稍作改动：divll.c 中的结果默认会存放在一个结构体 `lldiv_t` 中，将其改为用两个临时变量来存储商和余数，在汇编代码中就是两个寄存器；ffsll.c 中的结果默认会存放在一个 32 位整数中作为返回值，方便起见也将其改为 64 位整数，这样输入与输出相统一；iswxdigit.c 中会使用 `wint_t` 来存储一个字符，也统一改为用 64 位整数来存储。

5.2.1.2　实验环境

本实验的实验环境分为两部分：

- 用于运行本系统的环境，由于需要直接执行 AArch64 汇编代码，所以需要一个 AArch64 架构的设备；
- 用于运行大语言模型的环境，由于需要大量的计算资源，所以需要一个 GPU。

两者之间通过 REST API（representational state transfer application programming interface）来通信。

运行本系统的环境如下：

- 操作系统：Ubuntu 22.04

① https://sourceware.org/newlib/

② 内置函数指在命名上以 `__builtin_` 开头的函数，内置函数通常有可内联的汇编代码实现，比如 `__builtin_ffsll` 和 `__builtin_popcount`。

③ 使用的是 ARM64 GCC 13.2.0，为目前的最新版本的 ARM64 GCC。

- CPU 型号：ARM Cortex-A76（时钟频率为 2GHz）
- 内存大小：1.7GiB

本实验所使用的大语言模型为 Mixtral-7B[①]，运行大语言模型的环境如下：

- 操作系统：Ubuntu 20.04
- GPU 型号：NVIDIA GeForce RTX 4090
- 显存大小：24 GiB

5.2.1.3　时间限制

对于基准数据集中的每一个待优化程序，本实验将会设定一个小时的时间限制来进行超级优化。这一小时是指时钟时间，包括了 CPU 的计算时间、GPU 的计算时间，以及两台机器之间的通信时间。

5.2.2　实验结果

实验结果如表 5.1 所示。共计 25 个待优化程序，对每个程序，都会运行两轮超级优化，以观察算法中的随机性对优化效果的影响。在评估中，使用程序执行的指令条数来估计程序的运行时间（之所以以此作为衡量，详细讨论可见于第 4.1 节）。GCC 编译的结果是在最高优化选项 -Ofast 下编译的，代表了工业级编译器的最高优化结果，由于在由于在 AArch64 上未发现其他超级优化系统的开源实现，因此这里以 GCC 最高优化选项下的编译结果作为本实验所对比的基准。超级优化的结果在表 5.1 的后两列中被标出，如果并未发现与待优化程序等价的程序，那么就用 “-” 标出。如果发现了通过验证器检查的正确优化，其所发现的最优的程序会以一个数字标出，这个数字代表这个程序在所有测例上平均执行的指令条数。当然，本系统所发现的程序也有可能与 GCC 所编译出的程序相同。如果发现了优于 GCC 编译结果的优化，那么就会在表中以灰底标出。

总得来说，在 4/25 = 16% 的待优化程序上，本系统找到了超过 GCC -Ofast 编译结果的优化，在这部分程序中，性能提升了 50% – 300%。从基准数据集中程序的优化空间来讲，这些程序的优化空间并不多，仅有 1 个程序是已知有更优的版本，但本系统并未发现的（p24_roundup）。之所以优化空间不多，是因为 Hacker’s Delight 和 Newlib 中的程序已经是经过高度优化的了，加之本实验所搜集的程序大多极为短小只有几条指令，所以对于大多数程序，想要人工找到更优的程序版本也是极为困难的。

特别地，为了分析大语言模型在其中发挥的作用，加入了一组消融实验，也

① https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2

表 5.1　超级优化实验结果（运行时间估量单位：指令条数）

来源	待优化程序	GCC 编译结果	第零轮（空种子）	第一轮	第二轮
Hacker's Delight	p1_turnoff	2	2	2	2
	p2_test_power	2	2	2	3
	p3_lowbit	2	2	3	-
	p4_mask_lowbit	2	2	2	2
	p5_right_propagate	2	2	2	2
	p6_turnon_lowzero	2	4	*2*	*2*
	p7_lowzero	2	2	4	-
	p8_mask_trailing_zeros	2	-	*2*	-
	p9_abs	2	-	-	-
	p10_lz_eq	4	-	-	-
	p11_lz_lt	3	-	-	-
	p12_lz_le	3	-	-	-
	p13_sign	3	-	*3*	*3*
	p14_floor_avg	3	2	2	2
	p15_ceil_avg	3	-	-	-
	p16_max	2	2	-	2
	p17_turnoff_lowbits	4	-	-	*4*
	p19_whether_power	6	-	-	-
	p20_bits_higher	7	-	-	-
	p21_cycle	8	-	*5*	-
	p24_roundup	9	-	-	-
	p25_higher_mul	12	2	8	3
Newlib	divll	6	-	*3*	*4*
	ffsll	4	-	-	-
	iswxdigit	6	-	-	-

就是表 5.1 中的第零轮。在第零轮中，使用一个空程序作为初始的种子程序，这个空程序全由空指令构成（nop），其空指令数量与待优化程序的指令数量相同。从表 5.1 中可以看到，在 6/25 = 24% 的程序中，大语言模型辅助的超级优化方法显示了相较于传统超级优化方法的优越性，这部分对比通过斜体来标出，包括 p6_turnon_lowzero、p8_mask_trailing_zeros、p13_sign、p17_turnoff_lowbits、p21_cycle 和 divll。

下面，我们来分析一下本系统找到的超过 GCC -Ofast 编译结果的优化，总得来讲，所找到的优化均来自于 AArch64 的架构特性。

```c
#include "stdint.h"

int32_t p14(int32_t x, int32_t y) {
  int32_t o1 = x & y;
  int32_t o2 = x ^ y;
  int32_t o3 = o2 >> 1;
  return o1 + o3;
}
```

图 5.4　p14_floor_avg（C 代码）

```
  eor    w2, w0, w1
  and    w0, w0, w1
  add    w0, w0, w2, lsr 1
```

图 5.5　p14_floor_avg（AArch64 汇编代码，由 GCC 编译）

```
  add    x2, x0, x1
  asr    x0, x2, 1
```

图 5.6　p14_floor_avg（AArch64 汇编代码，超级优化后）

p14_floor_avg 是一个计算两个 32 位有符号整数的平均值的程序，其 C 代码如图 5.4 所示，由 GCC 从这段 C 代码编译出的汇编代码如图 5.5 所示。这段代码最初是为 32 位架构来编写的，利用位运算避免了先直接计算 (x + y) / 2 会带来的整数溢出问题。不过，由于本实验是在 AArch64 架构上实施的，这就为其引入了新的优化空间。本系统发现的优化代码如图 5.6 所示，在 64 位机器上，32 位整数的加减法不再有溢出问题，所以可以直接依表达式计算 (x + y) / 2，这段代码比 GCC 编译出的 3 条指令的代码（图 5.5）更短，只用了 2 条指令。

程序 p21_cycle 的功能是：接收 4 个 64 位无符号整数 x, a, b, c，其中 $x \in \{a, b, c\}$，

```c
#include "stdint.h"

uint64_t p21(uint64_t x, uint64_t a, uint64_t b, uint64_t c) {
  return ((-(x == c)) & (a ^ c)) ^ ((-(x == a)) & (b ^ c)) ^ c;
}
```

图 5.7　p21_cycle（C 代码）

```
  eor     x4, x3, x1
  cmp     x0, x3
  csel    x4, x4, xzr, eq
  eor     x2, x3, x2
  cmp     x0, x1
  csel    x2, x2, xzr, eq
  eor     x0, x4, x2
  eor     x0, x0, x3
```

图 5.8　p21_cycle（AArch64 汇编代码，由 GCC 编译）

```
  cmp     x0, x1
  csel    x2, x2, x3, eq
  csel    x3, x2, x3, pl
  cmp     x0, x3
  csel    x0, x1, x3, eq
```

图 5.9　p21_cycle（AArch64 汇编代码，超级优化后）

返回 x 在 $\langle a, b, c\rangle$ 这个序列中的下一个元素，也就是说，如果 $x = a$，返回 b，如果 $x = b$，返回 c，如果 $x = c$，那么返回 a。一个直接的实现需要分支跳转，不过一般认为分支对 CPU 的流水线机制不太友好，在 Hacker's Delight 中，通过应用复杂的位运算技巧来避免了分支，其 C 代码如图 5.7 所示，由 GCC 从这段 C 代码编译出的汇编代码如图 5.8 所示。在 AArch64 架构中，条件执行语句让我们可以无分支的情况下直接地实现这个功能，这就为超级优化带来了优化空间。本系统所发现的优化程序如图 5.9 所示，这段代码比 GCC 编译出的 8 条指令的代码（图 5.8）更短，只有 5 条指令。不过，可以看到，图 5.9 中本系统所发现的代码实际依然是有冗余的，第 2 行和第 3 行的 `csel` 指令可以合并为一条。

p25_higher_mul 是一个计算两个 32 位无符号整数的乘积的高 32 位的程序，其 C 代码如图 5.10 所示，由 GCC 从这段 C 代码编译出的汇编代码如图 5.11 所示。这段代码最初也是为 32 位架构来编写的，在没有 64 位寄存器的情况下，直接将两

个 32 位整数乘起来会发生溢出，所以先将其每个 32 位整数拆分为两个 16 位整数，再来作计算，以避免溢出。在 AArch64 架构中，64 位寄存器的引入带来了优化空间，我们可以直接计算两个 32 位整数的乘积，然后再从 64 位的乘法结果中取出高 32 位。本系统所发现的优化程序如图 5.12 所示，这段代码比 GCC 编译出的代码（图 5.11）更短，只有 3 条指令，而 GCC 编译出的代码有 12 条指令。不过，可以看到，图 5.12 中本系统所发现的代码实际是有冗余的，第 2 行的 `lsl` 和 `ubfx` 指令可以合并为一条，因为第 2 行所做的事情其实仅仅是将 `x0` 的值赋给 `x1`。

divll 是一个计算两个 64 位有符号整数的除法的程序，其 C 代码如图 5.13 所示，由 GCC 从这段 C 代码编译出的汇编代码如图 5.14 所示。当除法除不尽时，整数除法的含义可能有多种不同的理解，商可以是 “向上取整”、“向下取整” 或者是 “向零取整”，不同的编译器和 CPU 架构对于这时的除法结果可能会有不同的实现。所以，为了在底层有不同实现时统一程序的功能，divll 中会根据不同的除法结果来作调整，将其统一为 “向零取整”，也就是说，如果被除数为非负，余数也应非负，这部分调整在 图 5.13 的 9-12 行。在 AArch64 中，有符号除法指令（`sdiv`）一定是遵循 “向零取整” 的，所以是无需调整的，这就为超级优化带来了优化空间。本系统所发现的优化程序如图 5.15 所示，这段代码比 GCC 编译出的代码（图 5.14）更短，只有 3 条指令，而 GCC 编译出的代码有 8 条指令 ① 。

5.3　本章小结

本章介绍了本文所提出方法的具体实现，以及在基准数据集上的实验结果。

由于在 AArch64 上没有相关的超级优化系统，本实验以 GCC 的最高优化选项（-Ofast）的编译结果作为基准进行比较。在 4/25 = 16% 的待优化程序上，发现了超过 GCC -Ofast 编译结果的优化。在这 4 个程序中，以执行的指令数量作为性能的衡量指标，观测到性能有 50% – 300% 的提升。

这 4 个程序的性能提升来来源于 AArch64 的架构特性——原始的 C 程序由硬件无关的高级语言撰写，编译器在翻译到汇编代码时能做的优化有限，而超级优化则可以进一步利用这一优化空间。具体来讲，p14_floor_avg 和 p24_higher_mul 原本是为 32 位机器撰写的，在 64 位机器上可以不必考虑两个 32 位整数计算带来的溢出问题；p21_cycle 可以利用 AArch64 中的条件执行指令，来避免原本所需要的分支；在 divll 中通过一个分支来统一不同平台上除法的取整方式，而由于 AArch64 标准规定了除法指令本身是向零取整，所以可以省去这一分支语句（更详细的分

① 由于实际执行时，第 4 行的跳转必然发生，第 5-6 行并不会被执行到，所以根据运行时所执行的指令条数，表 5.1 中将 divll 的运行时间记为 6。

析可见于第 5.2.2 节）。

特别的，实验中发现了 STOKE[22] 使用的基准数据集中待优化程序的一个漏洞①。

① https://github.com/StanfordPL/stoke/pull/1021

```c
#include "stdint.h"

uint64_t p25(uint64_t x, uint64_t y) {
  uint64_t o1 = x & 0x0000ffff;
  uint64_t o2 = x >> 16;
  uint64_t o3 = y & 0x0000ffff;
  uint64_t o4 = y >> 16;
  uint64_t o5 = o1 * o3;
  uint64_t o6 = o2 * o3;
  uint64_t o7 = o1 * o4;
  uint64_t o8 = o2 * o4;
  uint64_t o9 = o5 >> 16;
  uint64_t o10 = o6 + o9;
  uint64_t o11 = o10 & 0x0000ffff;
  uint64_t o12 = o10 >> 16;
  uint64_t o13 = o7 + o11;
  uint64_t o14 = o13 >> 16;
  uint64_t o15 = o14 + o12;
  return o15 + o8;
}
```

图 5.10　p25_higher_mul（C 代码）

```asm
  and     x2, x0, #65535
  asr     x0, x0, #16
  and     x4, x1, #65535
  asr     x1, x1, #16
  mul     x3, x2, x4
  mul     x4, x0, x4
  add     x3, x4, x3, lsr #16
  and     x4, x3, #65535
  madd    x2, x1, x2, x4
  asr     x3, x3, #16
  add     x2, x3, x2, asr #16
  madd    x0, x0, x1, x2
```

图 5.11　p25_higher_mul（AArch64 汇编代码，由 GCC 编译）

```asm
  mul     x0, x1, x0
  lsl     x1, x0, 0
  ubfx    x0, x0, 32, 32
```

图 5.12　p25_higher_mul（AArch64 汇编代码，超级优化后）

```c
#include <stdint.h>

int64_t fxn(int64_t numer, int64_t denom) {
  int64_t quot;  /* quotient */
  int64_t rem;   /* remainder */

  quot = numer / denom;
  rem = numer % denom;
  if (numer >= 0 && rem < 0) {
    quot++;
    rem -= denom;
  }

  return quot ^ rem; // 不出现在汇编里，只是为了让函数有返回值
}
```

图 5.13　divll（C 代码）

```
  sdiv    x3, x0, x1
  msub    x2, x3, x1, x0
  bics    xzr, x2, x0
  bpl     .L2
  add     x3, x3, 1
  sub     x2, x2, x1
.L2:
  mov     x0, x3
  mov     x1, x2
```

图 5.14　divll（AArch64 汇编代码，由 GCC 编译）

```
  and     x3, x0, x0
  sdiv    x0, x0, x1
  msub    x1, x0, x1, x3
```

图 5.15　divll（AArch64 汇编代码，超级优化后）

第 6 章　总结

本文研究了一个有三十余年历史的经典问题：超级优化，即为给定程序寻找最优版本。程序空间是一个极其庞大的空间，这给传统的基于搜索的方法带来了很大的挑战。然而，大语言模型的出现为这一经典问题带来了新的机遇。本文探讨了将大语言模型引入超级优化的一种直接方法，即使用大语言模型来生成程序，并将其作为随机化搜索的种子程序，然后在这个种子程序的基础上去探索程序空间。由于形式化验证耗时较长，本文也引入了一种符号执行的同步协调机制，以缓解等价性验证中符号执行的路径爆炸问题。

本文实现了一个基于 AArch64 架构的超级优化原型系统，并对其做了初步实验评估。用于实验评估的待优化程序集合共有 25 个程序，其中 22 个来自教程《高效程序的奥秘（Hacker’s Delight）》，这是一个由前人收集的数据集；另有 3 个来自 Newlib，这是 C 标准库的一个面向嵌入式设备的开源替代实现。实验表明，在 1 小时的时间内，在 4/25 = 16% 的待优化程序上，发现了超过工业级编译器的最高优化选项（GCC -Ofast）的编译结果的性能提升。分析表明，这 4 个程序的性能提升是由于 AArch64 的架构特性——原始的 C 程序由硬件无关的高级语言撰写，编译器在翻译到汇编代码时能做的优化有限，而超级优化进一步利用了这一优化空间。

这一结果表明，本文提出的方法能够发现传统优化方法所错失的优化机会，特别是针对指令集架构特性的优化机会。

参考文献

[1] Massalin H. Superoptimizer: a look at the smallest program[J]. ACM SIGARCH Computer Architecture News, 1987, 15(5): 122-126.

[2] 毕钧. 深度学习编译优化研究[D]. 中国科学技术大学, 2023.

[3] 沈莉, 周文浩, 王飞, 等. swLLVM: 面向神威新一代超级计算机的优化编译器[J]. 软件学报, 2024, 35(5): 2359-2378.

[4] 李维, 王瑞. 嵌入式通信软件编译优化方法研究[J]. 单片机与嵌入式系统应用, 2023, 23(08): 83-86+91.

[5] Joshi R, Nelson G, Randall K. Denali: a goal-directed superoptimizer[J]. ACM SIGPLAN Notices, 2002, 37(5): 304-314.

[6] Bernstein D J, Chou T, Schwabe P. Mcbits: Fast constant-time code-based cryptography[C]// Cryptographic Hardware and Embedded Systems, Proceedings. Berlin, Heidelberg, 2013: 250-272.

[7] Bernstein D J, Chuengsatiansup C, Lange T. Curve41417: Karatsuba revisited[C]// Cryptographic Hardware and Embedded Systems, Proceedings. 2014: 316-334.

[8] Bernstein D J, Chuengsatiansup C, Lange T, et al. Kummer strikes back: New dh speed records [C]//Sarkar P, Iwata T. Advances in Cryptology, Proceedings. 2014: 317-337.

[9] Bernstein D J, Chuengsatiansup C, Lange T, et al. Ntru prime: Reducing attack surface at low cost[C]//Selected Areas in Cryptography, Proceedings. 2018: 235-260.

[10] Chou T. Sandy2x: New curve25519 speed records[C]//Selected Areas in Cryptography, Proceedings. 2016: 145-160.

[11] Chou T. Qcbits: Constant-time small-key code-based cryptography[C]//Cryptographic Hardware and Embedded Systems, Proceedings. 2016: 280-300.

[12] Chuengsatiansup C, Naehrig M, Ribarski P, et al. Panda: Pairings and arithmetic[C]//Pairing-Based Cryptography, Proceedings. Cham, 2014: 229-250.

[13] Chuengsatiansup C, Stehlé D. Towards practical ggm-based prf from (module-)learning-with-rounding[C]//Selected Areas in Cryptography, Proceedings. 2019: 693–713.

[14] Kannwischer M J, Rijneveld J, Schwabe P. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on cortex-m4 to speed up nist pqc candidates[C]//Applied Cryptography and Network Security, Proceedings. Bogota, Colombia, 2019: 281–301.

[15] Sasnauskas R, Chen Y, Collingbourne P, et al. Souper: A synthesizing superoptimizer[A]. 2018. arXiv: 1711.04422.

[16] Cabrera Arteaga J, Donde S, Gu J, et al. Superoptimization of webassembly bytecode[C]// Programming, Proceedings. 2020: 36–40.

[17] Brain M, Crick T, De Vos M, et al. Toast: applying answer set programming to superoptimisation [C]//Logic Programming, Proceedings. 2006: 270–284.

[18] 霍子伟. 一种改进的回答集逻辑程序分割方法及程序化简研究[D]. 中山大学, 2015.

[19] Granlund T, Kenner R. Eliminating branches using a superoptimizer and the GNU C compiler [C]//Programming language design and implementation, Proceedings. 1992: 341-352.

[20] Bansal S, Aiken A. Automatic generation of peephole superoptimizers[C]//Architectural support for programming languages and operating systems, Proceedings. 2006: 394-403.

[21] Phothilimthana P M, Thakur A, Bodik R, et al. Scaling up Superoptimization[C]//Architectural Support for Programming Languages and Operating Systems, Proceedings. 2016: 297-310.

[22] Schkufza E, Sharma R, Aiken A. Stochastic superoptimization[J]. ACM SIGPLAN Notices, 2013: 305-316.

[23] Schkufza E, Sharma R, Aiken A. Stochastic optimization of floating-point programs with tunable precision[C]//Programming Language Design and Implementation, Proceedings. 2014: 53-64.

[24] Churchill B, Sharma R, Bastien J, et al. Sound loop superoptimization for google native client [J]. ACM SIGPLAN Notices, 2017, 52(4): 313–326.

[25] Bunel R, Desmaison A, Kumar M P, et al. Learning to superoptimize programs[A]. 2017. arXiv: 1611.01787.

[26] Shypula A, Yin P, Lacomis J, et al. Learning to superoptimize real-world programs[A]. 2022. arXiv: 2109.13498.

[27] Williams R J. Simple statistical gradient-following algorithms for connectionist reinforcement learning[J]. Machine Learning, 1992, 8(3): 229-256.

[28] Mankowitz D J, Michi A, Zhernov A, et al. Faster sorting algorithms discovered using deep reinforcement learning[J]. Nature, 2023, 618(7964): 257-263.

[29] OpenAI, Achiam J, Adler S, et al. Gpt-4 technical report[A]. 2024. arXiv: 2303.08774.

[30] Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code[A]. 2021. arXiv: 2107.03374.

[31] Cummins C, Seeker V, Grubisic D, et al. Large language models for compiler optimization[A]. 2023. arXiv: 2309.07062.

[32] Grubisic D, Cummins C, Seeker V, et al. Compiler generated feedback for large language models [A]. 2024.

[33] Shypula A, Madaan A, Zeng Y, et al. Learning performance-improving code edits[A]. 2024. arXiv: 2302.07867.

[34] Tristan J B, Govereau P, Morrisett G. Evaluating value-graph translation validation for llvm[J]. ACM SIGPLAN Notices, 2011, 46(6): 295–305.

[35] Stepp M, Tate R, Lerner S. Equality-based translation validator for llvm[C]//Computer Aided Verification, Proceedings. 2011: 737–742.

[36] Tate R, Stepp M, Tatlock Z, et al. Equality saturation: a new approach to optimization[C]// Principles of Programming Languages, Proceedings. 2009: 264–276.

[37] Ramos D A, Engler D R. Practical, low-effort equivalence verification of real code[C]// Computer Aided Verification, Proceedings: Vol. 6806. 2011: 669-685.

[38] Badihi S, Akinotcho F, Li Y, et al. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code[C]//Foundations of Software Engineering, Proceedings. 2020: 13-24.

[39] Shemer R, Gurfinkel A, Shoham S, et al. Property Directed Self Composition[C]//Computer Aided Verification, Proceedings: Vol. 11561. 2019: 161-179.

[40] Nagasamudram R, Naumann D A. Alignment completeness for relational hoare logics[C]// Logic in Computer Science, Proceedings. 2021.

[41] Sharma R, Schkufza E, Churchill B, et al. Data-driven equivalence checking[C]//Object Oriented Programming Systems Languages & Applications, Proceedings. 2013: 391-406.

[42] Churchill B, Padon O, Sharma R, et al. Semantic program alignment for equivalence checking [C]//Programming Language Design and Implementation, Proceedings. 2019: 1027-1040.

[43] Gupta S, Rose A, Bansal S. Counterexample-guided correlation algorithm for translation validation[J]. Proceedings of the ACM on Programming Languages, 2020, 4(OOPSLA): 1-29.

[44] Warren H S. Hacker's delight[M]. 2nd ed ed. Upper Saddle River, NJ: Addison-Wesley, 2013.

[45] Gulwani S, Jha S, Tiwari A, et al. Synthesis of loop-free programs[J]. ACM SIGPLAN Notices, 2011, 46(6): 62–73.

[46] Zhang Y, Li Y, Cui L, et al. Siren's song in the ai ocean: A survey on hallucination in large language models[A]. 2023. arXiv: 2309.01219.

[47] Zhuo T Y, Huang Y, Chen C, et al. Red teaming chatgpt via jailbreaking: Bias, robustness, reliability and toxicity[A]. 2023. arXiv: 2301.12867.

[48] Li Y, Choi D, Chung J, et al. Competition-level code generation with alphacode[J]. Science, 2022, 378(6624): 1092-1097.

[49] Zhou Y, Liu S, Siow J, et al. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks[C]//Advances in Neural Information Processing Systems: Vol. 32. Curran Associates, Inc., 2019.

[50] Svajlenko J, Islam J F, Keivanloo I, et al. Towards a big data curated benchmark of inter-project code clones[C]//International Conference on Software Maintenance and Evolution, Proceedings. 2014: 476-480.

[51] Watson C, Tufano M, Moran K, et al. On learning meaningful assert statements for unit test cases[C]//International Conference on Software Engineering, Proceedings. 2020: 1398–1409.

[52] Husain H, Wu H H, Gazit T, et al. Codesearchnet challenge: Evaluating the state of semantic code search[A]. 2020. arXiv: 1909.09436.

[53] Wei J, Wang X, Schuurmans D, et al. Chain-of-thought prompting elicits reasoning in large language models[C]//Advances in Neural Information Processing Systems. 2022: 24824-24837.

[54] White J, Fu Q, Hays S, et al. A prompt pattern catalog to enhance prompt engineering with chatgpt[A]. 2023. arXiv: 2302.11382.

[55] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners[C]//Advances in Neural Information Processing Systems: Vol. 33. 2020: 1877-1901.

[56] Chang Y, Wang X, Wang J, et al. A survey on evaluation of large language models[J]. ACM Transactions on Intelligent Systems and Technology, 2024, 15(3).

[57] Poeplau S, Francillon A. Systematic comparison of symbolic execution systems: intermediate representation and its generation[C]//Annual Computer Security Applications Conference, Proceedings. 2019: 163–176.

[58] Turing A M. On Computable Numbers, with an Application to the Entscheidungsproblem[J]. Proceedings of the London Mathematical Society, 1937, s2-42(1): 230-265.

[59] Ahmed A, Appel A W, Richards C D, et al. Semantic foundations for typed assembly languages [J]. ACM Transactions on Programming Languages and Systems, 2010, 32(3): 1-67.

[60] Govereau P. Denotational translation validation[D]. USA: Harvard University, 2012.

[61] Ding H, Wang Z, Yang Y, et al. Proving Query Equivalence Using Linear Integer Arithmetic [J]. Proceedings of the ACM on Management of Data, 2023, 1(4): 1-26.

[62] ocxtal/insn_bench_aarch64: Instruction latency & throughput profiler for aarch64[EB/OL]. [05/27/2024]. https://github.com/ocxtal/insn_bench_aarch64/.

[63] Intel® 64 and ia-32 architectures software developer manuals[EB/OL]. [05/27/2024]. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[64] Binkert N, Beckmann B, Black G, et al. The gem5 simulator[J]. ACM SIGARCH Computer Architecture News, 2011, 39(2): 1–7.

致　谢

衷心感谢导师贺飞副教授对本人的指导和帮助。贺老师在学术上严谨治学，工作上严谨负责，他的言传身教将使我终生受益。

感谢实验室的同学给我的支持，包括范洪宇、刘诗月、徐荣琛、徐志杰、张洋、朱俸民等。范洪宇教了我很多事理，生活上也照顾了我很多，还请我去家里吃饭。刘诗月为本文提供了很大的帮助，尤其是第五章的实验章节，和我讨论了本课题的方方面面。韩志磊为本文第三章等价性验证的章节提供了宝贵的建议，比如如何刻画汇编程序的语义，也与我在学术上有很多探讨。徐荣琛教了我许多工作上的技能，比如怎么在 Excel 中导入 csv，也在我调不出代码的时候帮我调了很多 bug。徐志杰与我在学术上和生活上有很多探讨，尤其是在程序逻辑的基础知识方面，为本文也提供了宝贵的建议。张洋为本文的方方面面都提供了许多宝贵的建议，也与我在学术上和人生上有诸多探讨。朱俸民带我初窥了科研的门径，也教了我很多基础的技能。

此外，感谢实验室房间里其他组的同学，包括方寸谛和符景洲等。方寸谛为本文第二章大语言模型的章节提供了许多宝贵的建议。符景洲在本研究的开发与本文的写作过程中都给了我很多帮助，写作上，包括行文的思路和架构图的绘制，开发上，尤其是第四章的沙盒部分，他在二进制分析上的知识帮助了我许多，另外，也与我探讨了很多事理。

最后，感谢罗逸菲、邹振华和王文惠，我不太会生活，他们给了我很多生活上的支持。

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：_____

个人简历、在学期间完成的相关学术成果

个人简历

1999 年 6 月 3 日出生于山东省东营市。

2017 年 9 月免试进入清华大学计算机科学与技术系，2021 年 6 月本科毕业并获得工学学士学位。

2021 年 9 月免试进入清华大学软件学院攻读软件工学硕士至今。

在学期间完成的相关学术成果

学术论文：

[1] Zhu F*, **Xie X***, Feng D, Meng N, He F. On the methodology of three-way structured merge in version control systems: Top-down, bottom-up, or both[J]. Journal of Systems Architecture, 2023, 145: 103011.49. (CCF-B)

[2] Zhu F*, **Xie X***, Feng D, Meng N, He F. Mastery: Shifted-code-aware structured merging[C]//Dependable Software Engineering. Theories, Tools, and Applications. Cham: Springer Nature Switzerland, 2022: 70-87. (CCF-C, **Best Paper Award**)

指导教师评语

本论文研究超级优化系统的设计与实现，主要完成的工作包括：

1. 设计了一套基于大语言模型的优化程序搜索方法；
2. 设计了一种针对程序等价性验证的符号执行同步协调机制；
3. 基于上述方法，开发了一个基于 AArch64 架构的超级优化原型工具，并进行了实验验证。

超级优化是一种用于寻找性能最优（或近似最优）的程序版本的技术。超级优化必须辅以形式验证技术，才能确保优化前后代码的等价性。作者沿着这一方向开展了相关研究，并实现了一个原型系统。论文结构合理，符合硕士毕业论文要求。

答辩委员会决议书

论文面向超级优化问题开展研究工作，选题具有实践价值。论文的主要研究成果包括：

1. 提出了一种基于语言模型的端对端的汇编种子程序的生成方法；
2. 设计了一种汇编程序符号执行的同步协调机制，提升了等价性验证的性能；
3. 开发了一个面向 AArch64 架构的超级优化原型工具。

论文工作表明，作者掌握了软件工程领域坚实的基础理论和系统的专门知识，具有较丰富的专业实践经验和独立开展科研工作的能力。论文答辩叙述清楚，回答问题正确，系统演示正常。答辩委员会经无记名投票一致同意通过硕士学位论文答辩，并建议授予谢兴宇同学工学硕士学位。