计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220158
60(7): 1678−1691, 2023

针对 gem5 指令集实现及其功能测试的自动代码生成

赵紫微　涂　航　刘　芹　李　莉　余　涛
（武汉大学国家网络安全学院　武汉　430072）
（zhaoziwei0730@163.com）

An Automated Code Generation of Instruction Set Implementation and Functional Testing for gem5

Zhao Ziwei, Tu Hang, Liu Qin, Li Li, and Yu Tao
（*School of Cyber Science and Engineering*, *Wuhan University*, *Wuhan* 430072）

Abstract　Computer system simulators are important tools for research and prototype development of embedded systems. For interpretation-based simulators, the decoding process of CPU models has an important effect on their performance. Therefore, improving the performance of the decoding process is one of the key problems of simulation efficiency. Besides, for instruction sets without a standard test suite (such as custom instructions), writing functional tests manually leads to low development efficiency. The instruction information required by functional tests is practically the same as the implementation of the decoding process. To solve the above problems, we propose a code generation method, which takes an instruction set description as input and outputs its implementation codes optimized for gem5 and its functional tests. Firstly, we extend the instruction set description language of gem5 and divide it into code description, function description, and test description. Secondly, we optimize the construction algorithm of decoding decision trees for gem5 and generate decoding codes, instruction codes, and functional test cases. Lastly, we take the Cortex-M3 instruction set as an example and compare our method with the original method of gem5. The total generation time is reduced by about 64%, the compiled executable code size is reduced by about 407 KB, the performance is improved by about 13%, and our method can improve the development efficiency.

Key words　code generation; instruction set; decoding; simulator; functional testing

摘　要　在嵌入式领域，计算机系统模拟器是研究与原型开发的重要工具. 对于采用解释执行的模拟器，其 CPU 模型的译码过程会影响性能，如何提升译码过程的性能是提高仿真效率的关键问题之一. 此外，对于无标准测试集的指令集来说（例如自定义指令），手动编写指令功能测试的开发效率较低，并且其与实现译码过程所需的指令信息基本相同. 为解决上述问题，提出一个代码生成方案，输入一份指令集描述，输出针对 gem5 优化后的指令集实现代码和功能测试代码. 首先，扩展 gem5 的指令集描述语言，将其分为编码描述、功能描述和测试描述. 其次，针对 gem5 优化译码决策树构建算法，并为 gem5 生成译码模块代码、指令集实现代码和指令功能测试用例. 最后，以 Cortex-M3 指令集为例与原方案相比，总生成时间减少约 64%，编译后的可执行文件代码大小减少约 407 KB，性能提升约 13%，并且能够提高开发效率.

关键词　代码生成；指令集；译码；模拟器；功能测试

中图法分类号　TP319

收稿日期：2022-02-21；**修回日期**：2022-07-08

基金项目：国家重点研发计划项目（2018YFC1315404, 2018YFC1604004）
This work was supported by the National Key Research and Development Program of China (2018YFC1315404, 2018YFC1604004).

通信作者：涂航（tuhang@whu.edu.cn）

计算机系统模拟器[1]是运行于宿主机上的软件，用于模拟目标机器的硬件和软件环境，便于使用者研究目标机器的架构与执行过程，减少硬件成本. 嵌入式领域对于模拟器的需求量较大，并且在可扩展性和精确度方面有较多要求. 因此，如何基于现有的模拟器快速开发原型并且在保证正确性的同时具有较好的执行效率是一个值得研究的问题.

目前的模拟器从精确度方面可以分为功能级、指令级、周期级. 功能级保证其执行结果与目标机器相同，不考虑执行过程的精确性，在性能上表现最好，接近宿主机的执行速度，大多使用二进制翻译等技术提升性能，例如 QEMU[2]. 指令级保证每条指令按顺序全部执行，不考虑指令周期和流水线的问题，大多使用即时编译(just-in-time compilation, JIT)等技术提升性能. 周期级保证指令周期精确，可以仿真指令流水线，执行速度最慢，但提供更多执行细节信息，例如 gem5[3]. 考虑到嵌入式开发中对周期精确的需求，本文选择基于 gem5 进行研究. gem5 是一个开源的计算机架构模拟器，由密歇根大学的 m5 项目和威斯康星大学的 GEMS 项目合并而来，在学术界和工业界应用广泛. gem5 是模块化并以事件驱动的周期精确模拟器，可扩展性强. gem5 的 CPU 模块采用解释执行技术，其译码模块和指令集实现独立于 CPU 模块，可以与不同精确度的 CPU 模型相结合. 以不考虑访存延迟和流水线的 AtomicSimpleCPU 模型为例，主要有 3 个步骤：取指、译码和执行. 其中，译码过程是由各个指令集架构中的译码模块负责.

gem5 中的指令集描述语言可以半自动地生成指令集功能代码，但需要开发者手动处理指令编码的判断并为指令编写模板替换函数. 对于复杂的指令编码，手动处理指令编码的判断过于繁琐并且难以得到性能最优的实现. 没有统一的指令模板替换函数导致逻辑复杂且存在冗余，总体开发效率较低. gem5 在取指过程后会由译码模块对指令编码进行预解析，之后再调用函数解析完整的指令编码，这 2 个解析过程存在部分重复. 此外，在实现指令集和译码模块后还需要对模拟器进行功能测试，但一些指令集没有提供公开的标准测试集，这种情况下开发者需要根据指令集文档自行编写测试程序. 其中，测试用例的编写依赖于指令操作数的取值范围描述和指令的功能描述. 因此，可以依据 gem5 中所使用的指令集描述语言来生成功能测试中的部分数据，提高开发效率.

前人[4-5]提出了一些指令集描述方法和译码过程的优化算法，但不易与现有模拟器相结合，也没有针对 gem5 进行优化. 目前还没有一套为现有模拟器提供的从指令集描述到生成具体实现和测试用例的完整方案. 这对于有自定义的指令需求的用户来说，在模拟器性能和项目开发效率方面有所欠缺.

本文的工作难点在于根据统一的指令集描述语言提供一个完整的开发与测试方案，并针对 gem5 优化译码过程. 用户可以根据指令集文档直观地描述出所有指令的信息，得到自动生成的指令集实现代码、译码模块代码和功能测试用例.

gem5 原本未支持 Cortex-M3 指令集架构，本文实现了该指令集架构并引入了优化. 主要工作包括指令集功能实现和内存管理单元的修改，难点在于优化译码决策树和译码模块，意义在于提供一种更高效的指令集实现方案以及增加 gem5 对嵌入式芯片的仿真支持.

本文方案与 gem5 原方案的流程比较如图 1 所示. 本文方案包括 3 个阶段，首先是用词法和语法分析指令集描述文件，之后是根据指令的编码描述生成译码决策树，最后是使用统一的模板替换规则生成指令功能代码和译码模块代码. 与原方案相比，本文方案重新设计了指令集描述文件格式，修改了模板替换的逻辑，增加了译码决策树生成功能并优化译码模块. 此改进的作用在于简化指令定义，自动化生成译码决策树和译码模块代码，优化译码执行时间，提升开发效率.

本文的主要贡献包括 3 个方面:

1) 设计了一种指令集描述语言和一个基于 gem5 的指令集代码生成方案. 根据指令集的编码描述和功能描述，自动为模拟器生成指令集和译码模块代码，提升模拟器性能和开发效率.

2) 提出了一个针对 gem5 优化的译码决策树构建算法. 该算法基于前人的算法进行扩展，并减少了 gem5 中指令编码预解析阶段的重复判断，提升模拟器运行性能.

3) 实现了一个指令集功能测试框架. 根据指令的测试描述，使用框架中的模板为每条指令生成功能测试用例，在 gem5 上运行测试程序并输出测试报告.

1 相关工作

前人的研究[4-5]提到很多针对处理器或系统仿真的描述语言的研究可以用于生成指令集功能的描述，例如 Pydgin[6], LISA[7], nML[8]，但这些不是针对译码过

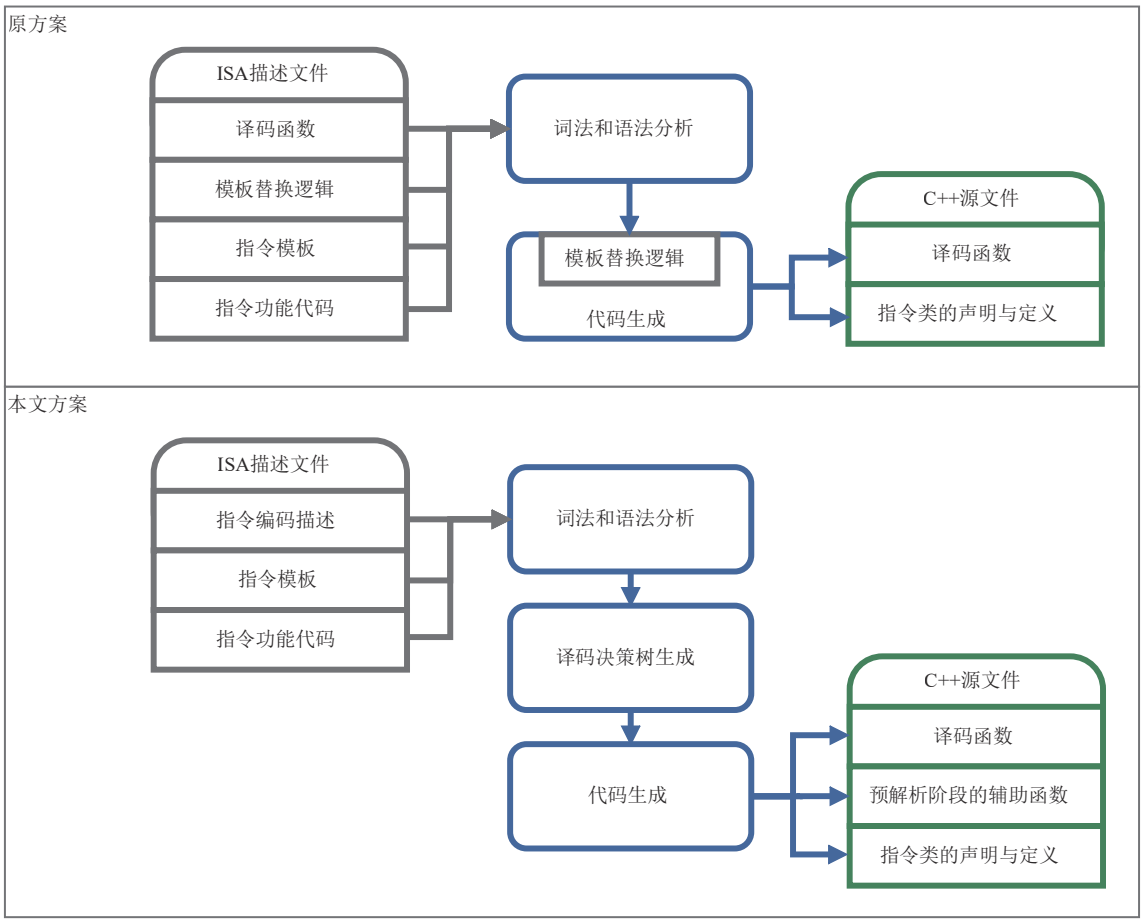


Fig. 1 Comparison of our scheme and original gem5 scheme

图 1 本文方案与 gem5 原方案的比较

程和指令功能的描述，也不易结合到现有的模拟器实现中. 本文方案主要基于 gem5 的指令集描述语言进行扩展，部分指令描述的设计参考了上述描述语言.

目前一些研究提出了构造决策树来优化译码分支的方法[9-11]，但在处理复杂指令结构时存在一些不能成功构建决策树的情况. Okuda 等人[12] 基于前人工作提出了对于不规则指令编码的译码分支优化算法，解决了复杂指令结构处理中的问题，可以生成平均高度低且内存占用小的决策树，并在 ARM 和 MIPS 指令集上取得了较好的结果. 此算法可以用于自动化构建处理器仿真[13]，此外还有研究分析了译码决策树的开销问题[14]. 本文基于此算法构造译码决策树，并针对 gem5 译码模块做出优化，构建多个决策子树来配合 gem5 的处理流程，提升译码模块性能.

指令集功能测试用于检测指令功能是否正确，例如检查单条指令执行前后的寄存器和内存的读写情况或者多条指令的处理器流水线情况[15] 等. RISC-V 提供了一套标准测试集[16]，通过模板构建测试用例，能够测试单条指令的功能. 本文搭建的指令功能测试框架中测试用例的设计参考了此测试集.

2 指令集描述

gem5 中已经实现了一个领域特定语言(domain specific language, DSL)，用于描述指令集中各个指令的功能及其译码函数，文件后缀为. isa. 在编译过程中，项目构建工具 SCons[17] 会调用 Python 脚本对所编译的目标架构的*. isa 文件进行词法和语法分析，从而生成包含指令定义和译码函数的 C++文件，最后 SCons 将这些生成的文件添加到编译任务中.

在实际使用中，我们发现该指令集描述语言存在 2 个问题：

1) 译码函数主要由开发者手动编写. 当指令数量较多且复杂时开发效率不高，并且手动编写的译码函数可能存在冗余，在执行效率方面有待优化.

2) 用于生成 C++代码的模板替换逻辑和待替换的数据没有分离. Python 脚本会使用函数*exec*()来处理这些字符串. 然而这些模板替换逻辑非常类似，这

种实现方式不仅增加了不必要的复杂性，而且不易维护.

本文参考了该指令集描述语言的实现，并提出了一种包含指令编码和指令功能信息的指令集描述语言，可以自动生成译码函数并自动完成指令功能代码与 C++代码模板的替换. 此外，对于不公开提供功能测试套件的指令集或是添加了自定义指令的情况，考虑到自行编写指令功能测试时很多所需的信息可以由指令集描述提供，本文方案允许标注操作数限制等指令测试所需的信息，支持生成指令的功能测试用例.

本文方案包括编码描述、功能描述和测试描述 3 部分. 这里以 ARM Cortex-M3 的 AND 指令为例来说明，如图 2 所示.

2.1 编码描述

编码描述用于说明指令编码的结构与限制，用于构造译码决策树. 本文方案将编码抽象为由固定位段和可变位段组成的结构，其中可变位段的取值可能存在一些限制. 下面结合实例来说明其具体实现.

指令编码是一串由 0、1 和小写字母组成的字符串. 其中，0 和 1 表示指令编码中取值确定的位，小写字母表示指令编码中取值可变的位. 同一小写字母必须相连，表示一个可变的位段. 例如，指令 t1_and_reg 的编码为 0100000000mmmddd，说明第 0 位和第 2~9 位为 0，第 1 位为 1，第 10~12 位为一个可变位段{*m*}，第 13~15 位为另一个可变位段{*d*}.

对于不符合要求的编码情况，使用 EXCLUSION 语句列出可变位段的所有例外表示，将其作为该指令编码的排除条件. 例如，指令 t1_and_imm 的编码排除了{*n*}为 13 到 15 和{*d*}为 13 或 15 的情况.

2.2 功能描述

功能描述用于说明指令功能的实现和标注信息，用于生成指令功能代码和提供仿真所需的控制信息和计数信息. 本文方案将功能抽象为由特定功能代码和模板组成的结构. 下面结合实例来说明其具体实现.

代码块由{{和}}来表示，在代码块中可变位段可

```
1   INST(AND) : PredProcessOp{
2       t1_and_imm : DataImm('11110i00000snnnn0jjjddddkkkkkkkk') {
3           EXCLUSION({d}): ['1111', '1101']
4           EXCLUSION({n}): ['1111', '1110', '1101']
5           {{
6               rd = @REG@{d};
7               rn = @REG@{n};
8               setFlags = {s};
9               imm = ({i} << 11) | ({j} << 8) | {k};
10          }}
11          FORMAT([0, 3])
12          TEST([True, True, Rx, Rx, Constant]) # AND{S}{cond} Rd, Rn, #constant
13      }
14
15      t1_and_reg : DataReg ('0100000000mmmddd') {...}
16      t2_and_reg : DataReg ('11101010000snnnn0iiidddjjttmmmm') {...}
17
18      SRC {N, Z, C, V, Rn}
19      DEST {Rd, N, Z, C}
20      CODE {
21          0: {{
22              expanded = thumbExpandImm_c(imm, C);
23              Rd = Rn & expanded;
24          }}
25          1: {{...}}
26          2: {{...}}
27          3: {{
28              if (setFlags) {
29                  N = bits(Rd,31);
30                  Z = ! Rd;
31              } else {
32                  C = xc -> tcBase() ->readCCRegFlat(CCREG_C);
33              }
34          }}
35      }
36  }
```

Fig. 2 An example of instruction description

图 2 指令描述示例

以作为数值来使用, REPLACE_MAP 中定义替换词可以在前后加上@作为代码片段来使用.

指令构造函数代码以代码块形式定义在指令编码描述结束后, 用于为指令基类定义的操作数变量根据实际编码来赋值. 例如, 指令 t1_and_imm 的基类为 DataImm, 其中定义了操作数寄存器的序号、立即数、其他参数和功能函数等.

指令功能函数代码存放在一个字典结构中, 代码块所对应的序号会作为参数传入 FORMAT 语句. FORMAT 语句用于处理 C++模板替换过程, 定义在构造函数描述之后. 例如, CODE { 0: {{ int *a*=0; }} }说明指令功能代码为 int *a*=0, 其对应序号为 0.

FORMAT 语句会根据指令类型的格式从 FORMAT_MAP 中得到对应的 C++代码模板和参数格式, 之后通过字符串替换生成相应的指令类. 例如, 指令类型 AND 的格式为 PredProcessOp, 其 C++代码模板为 BasicDeclare, BasicConstructor, PredProcessExecute, 分别用于指令类的声明、构造函数和功能函数的生成, 其参数格式为 ['*process_code*']. 指令 t1_and_imm 将 [0, 3] 传入 FORMAT 语句, 即将序号为 0 和 3 的代码拼接后的值作为 *process_code*.

2.3 测试描述

测试描述用于说明指令功能测试的参数要求, 用于辅助生成指令功能测试. 本文方案将功能测试抽象为由测试参数和测试模板组成的结构. 下面结合实例来说明其具体实现.

指令测试的所需的信息由 TEST 语句负责处理, 用于生成该指令的功能测试用例. TEST 语句的参数对应于该指令的汇编表示, 例如指令 t1_and_imm 的汇编表示包括可选的标记S和条件*cond*, 以及 2 个寄存器类型 Rx 和 1 个立即数类型 Constant. 类型 Rx 和类型 Constant 定义于 TEST_MAP 中, 设置了其取值范围和函数名, 在测试用例生成时会调用类型所对应的函数来生成数值.

3 译码决策树生成

本节说明了译码决策树生成过程中涉及的基本概念, 并给出了译码决策树生成算法的具体描述. 本文方案对前人所提出的算法[12] 进行了改进, 并结合 gem5 的特性实现了针对性优化.

3.1 基本概念

本节定义决策树的输入与输出形式, 以表 1 中的译码项来说明基本概念. 令指令集中的 1 条指令编码对应 1 个译码项, 算法的输入为 1 个译码项集合, 算法的输出为由该译码项集合生成的 1 个译码决策树.

Table 1 An Example of Decoding Entries

表 1 译码项示例

名称 m	编码 d	排除条件 C
A	00*XXXX*00	
B	0100*X*0*XX*	
C	010001*XX*	
D	010011*XX*	
E	01010100	
F	01010101	
G	01110111	
H	1000*XX*00	
I	10100*XXX*	({6, 7}, 00) ({6, 7}, 01)
J	10100*X*00	
K	10100*X*01	
L	11000*XXX*	
M	11100*XXX*	

3.1.1 译码项

译码项e包括名称m、编码d和排除条件集合C, 记为$e=(m,d,C)$. 其中, 译码项的名称和编码是唯一的, 排除条件可以为 0 或多个, 译码项集合记为E, $E=\{e\}$.

本文方案将编码定义为$d\in\{0,1,X\}^n$. 其中, X表示该位可以与 0 或 1 相匹配, 用于表示可变位段, 编码长度为n位. 给定一个位串$s\in\{0,1\}^n$, 定义位串s与编码d的匹配规则为$\forall i\in\{0,1,\cdots,n-1\}$, 当且仅当$s[i]=d[i]$或$d[i]=X$时, 位串$s$与编码$d$匹配, 记为$s\in d$. 若该编码所对应的译码项$e$无排除条件, 则位串$s$与译码项$e$相匹配, 记为$s\Rightarrow e$. 例如, 位串 00001100匹配编码 00*XXXX*00.

3.1.2 译码项中的排除条件

为了能够编码更多指令, 在设计指令集时某些编码被添加了排除条件, 即对编码中的可变位段设置了取值限制. 若位串的某些可变位段等于特定常数时, 则与该编码不匹配.

一个排除条件包括一个下标集合I_{ex}和一个排除项p_{ex}, 记为(I_{ex},p_{ex}). 其中, 下标是从小到大排列, 且与排除项的各位按序一一对应, 即将应排除的值的第i位记为$p_{ex}[i']$. 本文方案定义位串s使得排除条件(I_{ex},p_{ex})成立的判定规则为: $\forall i\in I_{ex}$, 当且仅当$s[i]=p_{ex}[i']$时, 位串s使得排除条件(I_{ex},p_{ex})成立, 记为*exclude*$(s,(I_{ex},$

p_{ex})). 例如，译码项的排除条件($\{6,7\},00$)表示应排除第 6 位和第 7 位都为0的位串.

因此，本文方案定义位串s与具有排除条件的译码项$e_c=(m_c,d_c,C_c)$的匹配规则为：当且仅当$s\in d_c$且$\forall(I_c,p_c)\in C_c,\neg exclude(s,(I_c,p_c))$时，位串$s$与译码项$e_c$相匹配，记为$s\Rightarrow e_c$. 例如，位串10100000虽然与名称为 I 的译码项和名称为 J 的译码项的编码匹配，但由于名称为 I 的译码项的排除条件($\{6,7\},00$)成立，所以该位串是与名称为 J 的译码项匹配.

3.1.3 译码决策树

译码过程是通过逐步查询位串中特定位段的值来获得该位串所匹配的译码项. 本文将这个过程用译码决策树来表示，记为$T=(V_T,E_T)$. 其中，V_T表示节点的集合，E_T表示边的集合. 每个节点有唯一标识符，记为n_{id}. 本文将节点分为内部节点和叶节点. 内部节点表示译码选择分支，包含一个特定位的下标集合I和一个元组集合，每个元组由标签和子节点组成，记为(p,v_{child}). 其中，下标集合中的元素从小到大排列，且与标签的各位按次序一一对应，l为下标集合的长度，标签 $p\in\{0,1\}^l$为叶节点，其表示译码查询所匹配的最终结果，包含一个译码项.

译码决策树示例如图 3 所示. 内部节点由包含其标识符n_{id}和下标集合I的方框表示，边框为弧线表示该节点在优化过程中被修改过. 叶节点由包含其标识符n_{id}、译码项名称m和译码项编码d的方框表示. 每条从内部节点所出的边的标签 p 和所指向的子节点 v_{child}由该内部节点中的各个元组(p,v_{child})得出.

3.2 算法描述

本节详细说明了构造译码决策树的算法. 首先，分析了前人的工作[12]并提出了本文方案的译码决策树构造算法，扩展了前人对于内部节点的构造方法并增加了 2 个优化策略. 此外，本文方案还针对 gem5 的译码过程对译码决策树和生成的代码做了优化. 最后，给出一个示例来说明每种优化策略所针对的情况和优化效果.

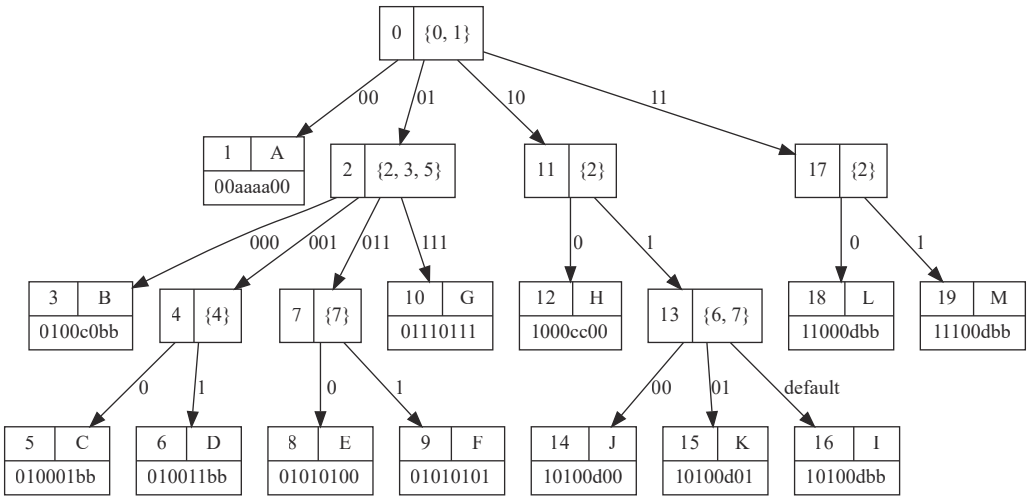
3.2.1 基础算法

Okuda 等人[12]的算法主要包含 3 个过程：1) 过程*MakeTree*根据给定的译码项集合E递归创建译码决策树的节点及其子节点. 2) 过程*MakeNode*创建一个无排除条件的内部节点，并将译码项集合 E 划分为多个子译码项集合，为每个子译码项集合创建子译码决策树. 3) 过程*MakeConditionNode*创建一个有排除条件的内部节点，并将译码项集合根据是否符合排除条件划分为 2 个子译码项集合，分别为这 2 个子译码项集合创建子译码决策树. 此外，该算法是通过与指令长度相等的编码格式来划分子译码项集合. 例如，使用以 0 开头和以 1 开头的 4 位编码格式来划分包含编码为 0101 和 1 101 的译码项集合.

Okuda 等人[12]的算法存在 2 个问题：

1）过程*MakeConditionNode*在划分子译码项集合时仅根据其是否符合排除条件分为 2 类. 对于不符合排除条件但也可被区分的子译码项来说，可能会再次处理该相同位段，导致译码决策树的高度增加.

2）算法根据与指令长度相等的编码格式来划分



注：内部节点左框为标识符n_{id}，右框为下标集合I；
叶节点左上框为标识符n_{id}，右上框为译码项名称m，下框为译码项编码d；
边指向子节点，边的标签p为下标取值.

Fig. 3 An example of a decoding decision tree

图 3 译码决策树示例

译码项集合, 不便于添加额外的优化策略, 例如处理个别位的合并或移除操作.

本文方案的算法基于 Okuda 等人[12]的算法做了扩展和优化. 在构造内部节点时, 本文方案使用下标集合 I 所确定的多个标签来划分译码项集合, 每个译码项根据其编码在该下标集合I处的取值情况划分到不同的标签 p下. 本文方案为每个标签的译码项集合构造译码决策树, 并将其作为该内部节点的子树. 此外, 本文方案修改了过程*MakeConditionNode*的实现, 使其能够根据排除条件的下标集合I_{ex}来区分多个子译码项集合. 并且, 本文方案以能够区分出最多子译码项集合的下标集合 I 来创建节点, 这样可以避免重复判断相同位段的值并减少译码决策树的高度.

3.2.2 扩展算法

本文方案的算法对过程*MakeNode*和过程 *MakeConditionNode*做了扩展和优化, 见算法 1.

算法 1. 译码决策树构建算法.

输入: 译码项集合E、初始译码决策树T;

输出: 原地更新后的译码决策树T.

过程$MakeTree(E, T)$

① 若$|E| = 1$, 则创建叶子节点并返回;

② $(result, node) \leftarrow MakeNode(E, T)$;

③ if $result$ = FAILED

④ return $MakeConditionNode(E, T)$;

⑤ else

⑥ return $node$;

⑦ end if

过程$MakeNode(E, T)$

① $I_{sig} \leftarrow GetSignificantBits(E)$;

② 若$I_{sig} = \varnothing$, 则返回(FAILED, nil);

③ $\left(isOpt, I_{sig}, Info\right) \leftarrow SelOptBits\left(I_{sig}, E\right)$;

④ 根据I_{sig}更新待处理的下标集合I_{unproc};

⑤ $(t_1, T_{child}) \leftarrow MakeChild\left(Info, I_{unproc}\right)$;

⑥ if t_1 = True

⑦ $(t_2, I_{re}, Info_{re}) \leftarrow ReselOptBits\left(I_{sig}, E\right)$;

⑧ if t_2 = True

⑨ $isOpt \leftarrow$ True;

⑩ $I_{sig} \leftarrow I_{re}$;

⑪ $(t_3, T_{child}) \leftarrow MakeChild\left(Info_{re}, I_{unproc}\right)$;

⑫ end if

⑬ end if

⑭ $node \leftarrow CreateNode\left(I_{sig}, T_{child}, isOpt\right)$;

⑮ 根据T_{child}更新T;

⑯ return (OK, $node$).

过程 $MakeConditionNode(E, T)$

① $(I_{ex}, Info) \leftarrow SelConditionBits(E)$;

② 根据I_{ex}更新待处理的下标集合I_{unproc};

③ $T_{child} \leftarrow MakeChild\left(Info, I_{unproc}\right)$;

④ $node \leftarrow CreateConditionNode(I_{ex}, T_{child})$;

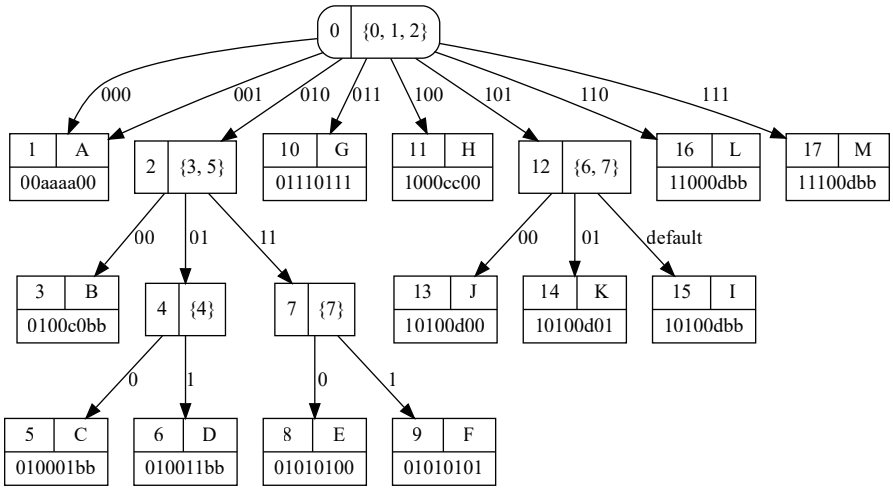
⑤ 根据T_{child}更新T;

⑥ return $node$.

过程*MakeNode*用于创建一个不使用排除条件的内部节点. 首先, 过程*GetSignificantBits*根据译码项集合E找出一个下标集合I_{sig}, 使得所有译码项可以用标签区分. 过程*SelOptBits*根据下标集合划分各个标签对应的子译码项集合, 将结果记为子节点信息 $Info$. 之后检查是否可以通过扩增下标集合以减少译码决策树高度, 如果可以优化, 则更新下标集合I_{sig}和子节点信息 $Info$. 过程*MakeChild*会调用过程*MakeTree*并根据子节点信息和尚未处理的下标集合来递归创建子节点译码决策树, 将结果记为T_{child}. 如果创建的子节点译码决策树中存在已优化的节点, 则需要通过过程 *ReselOptBits*再次更新下标集合I_{sig}、子节点信息$Info$和子节点译码决策树 T_{child}. 最后, 过程*CreateNode*创建此内部节点, 记录其下标集合I_{sig}和子节点译码决策树T_{child}, 并设置节点类型是否为已优化节点. 图 4 展示了优化后的效果, 将下标 2 增加到根节点的判断中, 减少了名称为 G 的译码项和名称为 M 的译码项所在的层数.

过程*MakeConditionNode*用于创建使用排除条件的内部节点. 首先, 过程*SelConditionBits*根据译码项集合E从各个译码项的排除条件中找出一个排除条件(I_{ex}, p_{ex}), 使得根据此下标集合I_{ex}划分得到的标签数量最多, 将结果记为子节点信息 $Info$. 这里没有使用过程*MakeNode*中的优化方法是因为使用排除条件所得到的标签中一定包含 default, 之前的优化不适用于这种情况. 之后执行过程 *MakeChild*, 将结果记为T_{child}. 最后, 过程*CreateConditionNode*创建此内部节点, 记录其下标集合I_{ex}和子节点译码决策树T_{child}, 并设置节点类型是否为使用排除条件的节点. 图 5 说明了 Okuda 等人[12]的算法对于条件节点的处理, 仅判断是否满足排除条件, 未考虑相同下标对应多个可区分编码的情况, 导致冗余判断. 本文使用下标集合来划分子译码项集合, 因此在确定下标集合后, 其划分方式与常规内部节点相同, 如图 3 所示.

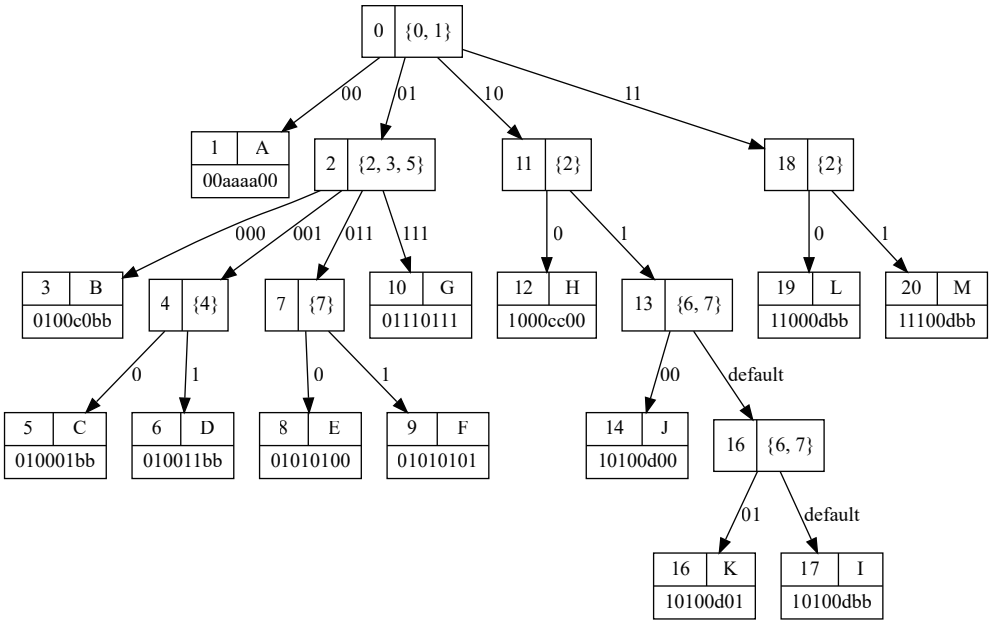
此外, 本文方案新增了过程 *OptimizeTree*、过程*OptimizeNode*、过程*FixNode*和过程*FixTree*对译码决



注：内部节点左框为标识符n_{id}，右框为下标集合I，已优化节点边框为弧线；
叶节点左上框为标识符n_{id}，右上框为译码项名称m，下框为译码项编码d；
边指向子节点，边的标签p为下标取值.

Fig. 4 The decoding decision tree optimized by process *MakeNode*

图 4 使用过程*MakeNode*优化的译码决策树



注：内部节点左框为标识符n_{id}，右框为下标集合I；
叶节点左上框为标识符n_{id}，右上框为译码项名称m，下框为译码项编码d；
边指向子节点，边的标签p为下标取值.

Fig. 5 The condition node of Okuda et al's algorithm

图 5 Okuda 等人所提算法的条件节点

策树进行整体优化和调整, 见算法 2.

算法 2. 译码决策树优化算法.

输入: 译码决策树T、译码决策树节点N;

输出: 原地更新后的已优化译码决策树T.

过程$OptimizeTree(N, T)$

① 若N是叶子节点, 则返回;

② $OptimizeNode(N, T)$;

③ for all $N_{\mathrm{child}} \in N$的子节点集合

④ $OptimizeTree(N_{\mathrm{child}}, T)$;

⑤ end for

过程$OptimizeNode(N, T)$

① $I_{\mathrm{opt}} \leftarrow GetOptimizableBits(N, T)$;

② 若I_{opt}为空, 则返回;

③ $T_{\mathrm{child}} \leftarrow \varnothing$;

④ for all $N_{\mathrm{leaf}} \in N$的叶子子节点集合

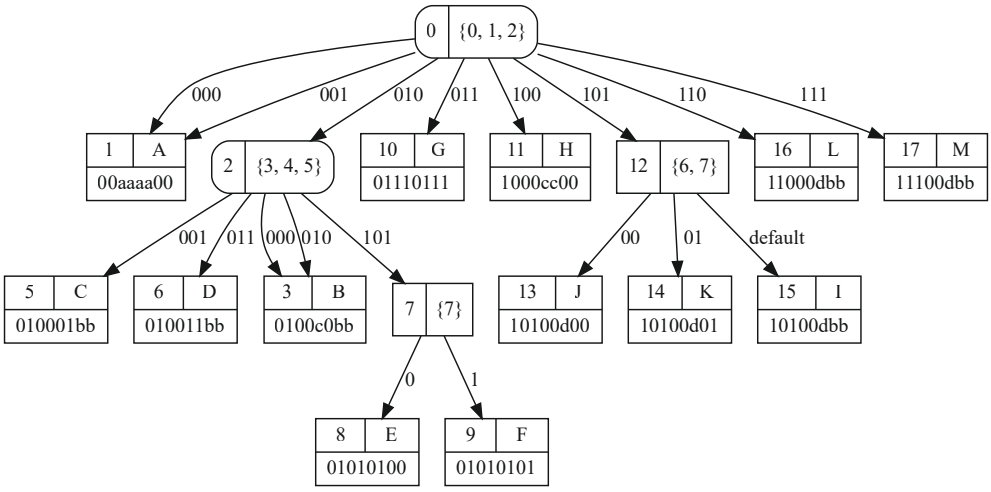
⑤ $(t_3, T_{\mathrm{res}}) \leftarrow GetLeafPattern(N_{\mathrm{leaf}})$;

⑥ 若t_3 = True, 设置N_{leaf}为多标签节点;
⑦ 根据T_{res}更新T_{child};
⑧ end for
⑨ for all $i \in I_{opt}$
⑩ for all $N_{inner} \in N$的内部子节点集合
⑪ $(t_4, T_{res}) \leftarrow GetInnerPattern(N_{inner}, i)$;
⑫ 若t_4 = True, 将N_{inner}从T中移除;
⑬ 根据T_{res}更新T_{child};
⑭ end for
⑮ end for
⑯ $UpdateOptimizableNode(T, N, I_{opt}, T_{child})$.

过程$FixTree(N, T)$
① 若N是叶子节点, 则返回;
② $FixNode(N, T)$;
③ for all $N_{child} \in N$的子节点集合
④ $FixTree(N_{child}, T)$;
⑤ end for

过程$FixNode(N, T)$
① 在N的所有多标签子节点中查找满足合并条件的N_{multi};
② 若不存在N_{multi}, 则返回;
③ 修改N到N_{multi}的分支标签并更新T.

过程*OptimizeTree*和过程*OptimizeNode*用于前序遍历译码决策树中的节点并做优化. 首先, 过程*GetOptimizableBits*检查节点N的内部子节点的下标集合I_{sig}是否仅包含1个元素, 将满足条件的下标集合取并集作为待选下标集合. 遍历待选下标集合, 检查其他各个内部子节点下的所有译码项是否在该下标处取值相同. 如果存在这样的下标, 则将其加入到I_{opt}中. 之后, 对于节点N的每个叶子子节点N_{leaf}, 使用过程*GetLeafPattern*修改指向此N_{leaf}的标签, 将结果更新到T_{child}中. 遍历下标集合I_{opt}, 对节点N的每个内部子节点N_{inner}用过程*GetInnerPattern*确认是否需要移除此内部节点, 并修改指向此N_{inner}或其子节点的标签, 将结果更新到T_{child}中. 最后过程*UpdateOptimizableNode*修改节点N的下标集合为I_{opt}, 更新子节点译码决策树T_{child}, 并设置节点类型是否为已优化节点. 图6展示了优化后的效果, 将下标4增加到2号节点的判断中, 减少了名称为C的译码项和名称为D的译码项所在的层数.



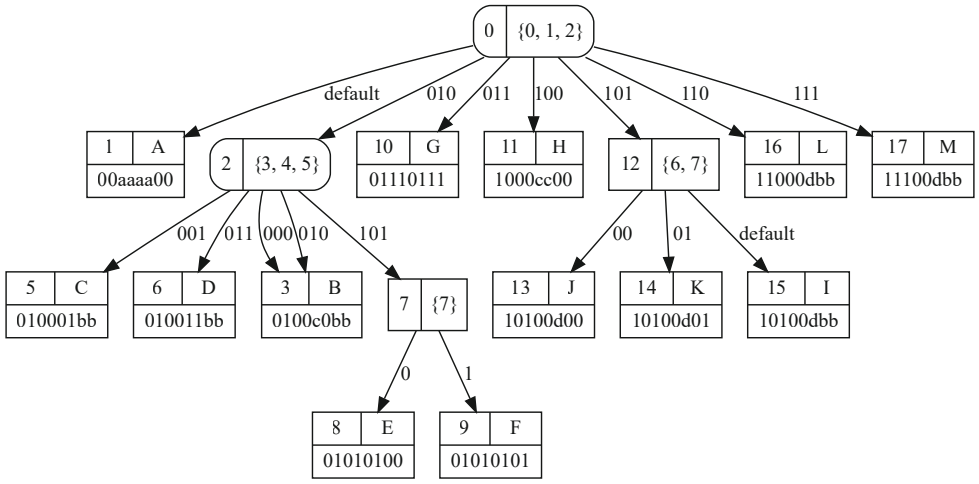
注: 内部节点左框为标识符n_{id}, 右框为下标集合I, 已优化节点边框为弧线;
叶节点左上框为标识符n_{id}, 右上框为译码项名称m, 下框为译码项编码d;
边指向子节点, 边的标签p为下标取值.

Fig. 6 The decoding decision tree optimized by process *OptimizeTree* based on fig. 4
图6 在图4的基础上使用过程*OptimizeTree*优化的译码决策树

过程*FixTree*和过程*FixNode*与前2个过程类似, 用于修正指向译码决策树中叶子节点的边, 将重复分支的标签合并为default. 图7展示了优化后的效果, 由于根节点下的分支数已达到最大且仅有名称为A的译码项对应多个标签, 所以本文方案将这些标签合并为default.

3.2.3 gem5中的译码过程

在gem5的实现中, CPU在译码阶段会先将取指阶段得到的数据传给译码模块, 然后调用译码模块来解析指令并得到一个基类指针, 在执行阶段会调用其所指向的具体指令对象的执行函数, 具体处理流程如图8所示.



注：内部节点左框为标识符n_{id}，右框为下标集合I，已优化节点边框为弧线；
叶节点左上框为标识符n_{id}，右上框为译码项名称m，下框为译码项编码d；
边指向子节点，边的标签p为下标取值.

Fig. 7 The decoding decision tree optimized by process *FixTree* based on fig.6

图 7 在图 6 的基础上使用过程*FixTree*优化的译码决策树

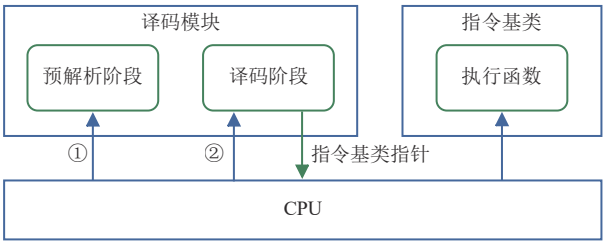


Fig. 8 The decode and execute processes in gem5

图 8 gem5 中的译码和执行流程

译码模块中的指令解析过程分为 2 个阶段：预解析阶段（图 8 ①）和译码阶段（图 8 ②）. 首先，预解析阶段的函数 *moreBytes*() 会根据大小端和指令长度对取指阶段得到的数据块进行处理，得到符合译码要求的具体指令位串. 之后，译码阶段会调用译码函数对该指令位串进行解析. 其中，译码函数的源文件是根据指令集描述语言在编译过程中生成的，其函数体为一个多层嵌套的 switch-case 结构.

指令长度是根据位串中特定位的值来判断，这个过程与之后译码函数中的操作存在重复，即预解析阶段和译码阶段都会对同一段指令编码进行判断. 以 Cortex-M3 指令集为例，16 位 Thumb 指令和 32 位 Arm 指令是由指令编码的前 5 位来区分的，因此预解析阶段和译码阶段都会判断这前 5 位. 此外，条件指令 IT 因为其条件执行功能涉及对译码模块的操作，所以会在预解析阶段中被完整解析，但在之后的译码阶段仍会被再次解析以获得一个指向该指令对象的指针.

为了解决上述问题，本文方案将原方案的译码函数拆分为多个译码函数，并使用函数指针优化调用过程. 具体来说，本文方案的算法会根据指令集描述文件中设置的指令长度下标集合及其取值情况来构造多个译码决策树，从而生成多个译码函数. 此外，本文方案的算法会自动生成预解析阶段所需的指令长度判断函数，以便根据判断结果选择对应的译码函数.

4 测试用例生成

本文方案搭建了一个对指令进行功能测试的框架，能够根据指令集描述为每条指令生成功能测试用例，并在 gem5 上运行所生成的测试程序得到测试报告. 该框架主要包括 3 部分：操作数数据生成、期望值数据计算和测试程序模板.

以 ARM Cortex-M3 的 AND 指令为例，其操作数生成和期望值计算见算法 3. 该指令的汇编表示有 2 种，即 AND{S}{*cond*}*Rn*, #*constant*和 AND{S}{*cond*}*Rn*, *Rm*, *shift*. 其中，S表示是否更新状态位，*cond*表示执行的条件，*Rn*表示寄存器，#*constant*表示满足特定格式的立即数，*Rm*和*shift*表示寄存器及其移位信息.

算法 3. 功能测试用例生成算法.

输入：指令测试描述*info*、测试范围*range*和测试用例数量*num*；

输出：测试用例列表G.

过程*GenData*(*info*, *range*, *num*)

① 根据$info$查找各操作数的生成函数，按操作数顺序将生成函数添加到$opFuncList$;
② 初始测试用例的操作数列表$opsList$;
③ for all $f \in opFuncList$
④ 将$f(range, num)$生成的一组具体操作数添加到$opsList$;
⑤ end for
⑥ for $ops \in opsList^T$
⑦ $expects \leftarrow GenExpAND(info, ops)$;
⑧ 将$(expects, ops)$添加到G;
⑨ end for
⑩ return G.

过程$GenExpAND(info, ops)$
① $(S, apsr, op1, op2) \leftarrow ParseOps(info, ops)$;
② $expExec \leftarrow Check(apsr)$;
③ $expRes \leftarrow op1 \& op2$;
④ if S = True
⑤ $expApsr \leftarrow SetNewApsr(apsr, expRes)$;
⑥ end if
⑦ return $(expExec, expRes, expApsr)$.

过程$GenData$用于生成操作数数据，根据指令测试描述$info$、测试范围 $range$ 和测试用例数量 num 来随机生成操作数，并在边界附近多生成一些值，之后调用过程$GenExpAND$生成期望值 $expects$，最后返回测试用例列表 G.

过程$GenExpAND$用于生成期望值数据，根据指令测试描述$info$和操作数ops来计算期望值 $expects$，最后返回结果.

此外，本文方案准备了一系列测试程序模板，以宏的方式组织汇编指令，能够为每条指令生成相应的汇编文件作为测试程序. 每个用例以宏的方式呈现，例如 TEST_AND_R_OP2C($testnum$, $inst$, $expExec$, $expRes$, $expApsr$, $Apsr$, Rn, $Constant$) 是用于 AND 指令的第 1 种汇编表示的宏，参数包括用例编号、被测指令、期望值和操作数.

5 实验结果

本文实验环境的 CPU 为 Intel Core i5-10400 @ 2.90 GHz，内存为 32 GB，系统为 Linux Mint 20.1 Kernel 5.4.0-89-generic. 本文的 gem5 配置为系统调用仿真（system call emulation，SE）模式，CPU 模型为 AtomicSimpleCPU，编译版本为 fast. 本文方案中的指令集描述语言使用 SLY[18] 作为词法和语法分析工具①，并且修改了 gem5 的编译脚本，将本文方案整合到项目的构建过程中.

实验选择 Cortex-M3 指令集作为待描述的指令集，该指令集在 gem5 项目中未做实现. 实验分别使用 gem5 原方案与本文方案实现该指令集，并比较这 2 种方案的性能以及所生成的代码的运行性能. 此外，在实验前使用第 4 节的功能测试框架为其生成了指令功能测试用例并确保其能正确执行，即保证所实现的 Cortex-M3 指令集的译码和执行过程是功能正确的.

5.1 方案性能比较

指令集中每条指令编码的查找路径的长度是在译码决策树中根节点到其所对应叶节点的边数，即叶节点v在译码决策树T中的高度，记为$H_T(v)$. 将各译码决策树中叶节点高度的平均值记为$\overline{H}_T$，将译码决策树集合中所有叶节点高度的平均值记为$\overline{H}$.

比较 gem5 原方案与本文方案所生成的译码决策树高度，结果如表 2 所示. Cortex-M3 指令集共有 226 个指令编码. 原方案生成的译码决策树中叶节点的最大高度为 10，最小高度为 3，总平均高度为 5.52. 本文方案共生成 4 个译码决策树和 1 个 IT 指令的译码函数，总平均高度为 2.87. 其译码范围是根据预解析阶段所处理的指令前 5 位取值以及是否为 IT 指令来划分的，表 2 中标明了各个译码决策树负责处理的译码范围. 其中，16 位指令的译码决策树包含指令编码数量最多，32 位指令（以 11101 开头）的译码决策树的平均高度$\overline{H}_T$最大. 与依赖开发者手动构造译码函数的原方案相比，本文方案在译码决策树高度方面的优化效果较好，并且很大程度上提升了开发效率.

此外，统计这 2 种方案生成译码决策树及其 C++ 源文件的时间和编译后的可执行文件 gem5.fast 的代码大小，结果如表 3 所示. 原方案需要开发者手动编写译码决策树，并且需要将模板替换逻辑与指令定义写在同一指令集描述文件中，在解析的同时处理替换过程，耦合度较高. 本文方案仅将指令定义写在指令集描述文件中，之后根据编码信息自动生成译码决策树和 C++源文件，所用的总时间比原方案减少了约 64%，代码大小减少了约 407 KB.

① gem5 原词法和语法分析工具为 PLY（Python Lex-Yacc），SLY（Sly Lex-Yacc）是其新版本.

Table 2 Comparison of the Height of the Generated Decoding Decision Tree Between Two Methods Under Cortex-M3

表 2 在 Cortex-M3 指令集下 2 种方案所生成的译码决策树高度的比较

方案	译码范围	指令编码数量	max$H$	min$H$	$\overline{H}_T$	$\overline{H}$
gem5 原方案	所有指令	226	10	3	5.52	5.52
本文方案	16 位指令	79	4	1	2.23	2.87
	32 位指令（以 11101 开头）	40	5	2	3.37	
	32 位指令（以 11110 开头）	45	5	2	3.11	
	32 位指令（以 11111 开头）	61	6	1	3.23	
	IT 指令	1	0	0	0	

Table 3 Comparison of the Performance Between Two Methods Under Cortex-M3

表 3 在 Cortex-M3 指令集下 2 种方案的性能比较

方案	指令集描述文件解析时间/s	译码决策树生成时间/s	源文件生成时间/s	总时间/s	代码大小/B
gem5 原方案	不能分阶段统计			0.282	13 238 308
本文方案	0.073	0.022	0.006	0.101	12 831 270

5.2 在 gem5 中的运行情况

在 gem5 中运行一些测试程序并比较 2 种方案的运行性能. 实验使用的测试程序来自 Embench[19], 这是一个面向嵌入式设备的开源测试集, 它包含 22 个测试程序, 支持对真实机器和模拟器的测试. 该测试集原本是通过远程调试器来获取测试函数前后的指令周期数来评估处理器速度, 而本文所研究的译码模块优化不影响模拟器中的周期数, 所以我们改为获取测试函数前后的实际时间来评估模拟器译码模块的性能, 即通过 shell 命令获取宿主机时间. 为减少额外操作对时间的影响, 我们将默认的测试循环次数由 1 改为 10,

每个用例用时大约在 10 s 左右. 此外, gem5 原本使用了一个译码表来缓存相同地址或相同指令编码的译码结果. 由于实验所用的测试程序都依赖于循环, 并且在开始计时前会先运行几轮来预热缓存, 导致 gem5 实际运行时 2 种方案几乎没有区别. 这与本文实验目的不符, 因此我们在测试时关闭了译码缓存功能.

测试结果如图 9 所示, 原方案平均用时 10.24 s, 本文方案平均用时 8.91 s, 比原方案减少了大约 13%.

5.3 方案效果分析

本文方案的译码决策树生成算法采用划分编码位段的方式, 通过计算位段的匹配情况和使用多个优化策略来构建树的节点, 从而减少译码过程的时间开销.

译码过程的搜索时间开销与译码决策树的高度

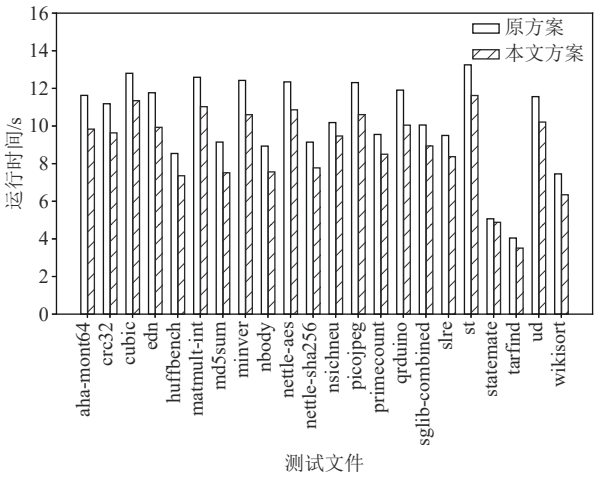


Fig. 9 Execution performance of two methods in gem5 Under Cortex-M3

图 9 在 Cortex-M3 指令集下 2 种方案在 gem5 中运行性能

呈正相关. 树的每个内部节点所处理的位数n决定了该节点的子节点数量上限2^n. 对于相同的指令数量, 树中内部节点指向新子节点的分支数量越多, 树的高度越低. 因此在树的构造过程中需要充分考虑指令长度和编码限制条件, 处理节点之间的合并优化.

为进一步说明改进效果, 我们使用本文方案为 RV64GC 指令集生成了译码决策树并与 gem5 原方案已实现的译码函数做比较, 结果如表 4 所示. RV64GC 指令集的编码设计较为规整, 较少使用编码的排除条件, 因此其译码决策树的高度整体较为平均. RV64GC 实验结果与 Cortex-M3 类似, 本文方案有效降低了译码决策树的最大高度并且优化了平均高度.

Table 4 Comparison of the Height of the Generated Decoding Decision Tree Between Two Methods Under RV64GC

表 4 在 RV64GC 指令集下 2 种方案对于所生成的译码决策树高度的比较

方案	译码范围	指令编码数量	max$H$	min$H$	$\overline{H}_T$	$\overline{H}$
gem5 原方案	所有指令	197	5	2	3.48	3.48
本文方案	16 位指令（以 00 开头）	7	1	1	1.00	2.08
	16 位指令（以 01 开头）	8	2	1	1.47	
	16 位指令（以 10 开头）	12	3	1	1.58	
	32 位指令	160	3	1	2.24	

6 讨 论

本文方案支持使用 gem5 中提供的接口函数和类型定义(如向量操作),可以用于实现其他指令集架构,例如 MIPS 和 Cortex-A 等. 从 RV64GC 的实验结果可以看出,自动化生成和优化的预解析阶段的辅助函数和译码决策树可以在一定程度上降低译码决策树的平均高度,提升执行效率. 此外,本文方案的指令集描述文件更易被编写与修改,开发效率较高.

本文方案的编码描述和译码决策树生成算法具有通用性,能够应用于指令译码过程,而功能描述和代码生成阶段则与模拟器的具体实现关系紧密. 若要将本文方案推广到其他模拟器,则还需从实现角度考虑模拟器是否可以采用这种译码和执行流程,以及这种模板替换策略能否与模拟器其他模块适配.

7 结 论

本文方案解决了模拟器的指令集实现及其功能测试的开发效率问题并提升了 gem5 译码模块的性能,为添加新指令集或自定义扩展指令的开发与测试提供了一套完整方案. 本文方案设计的指令集描述中定义了指令集的编码描述、功能描述和测试描述,用于生成模拟器的译码模块代码、指令集实现代码和指令功能测试用例. 本文提出了一个针对 gem5 优化的译码决策树构造算法和一个指令功能测试框架. 该算法使用指令的特定位和排除条件位来判断指令编码,并且允许根据 gem5 指令预解析阶段的条件划分各个决策树的范围,从而降低译码决策树的平均高度并且减少了原方案中的重复判断,提升执行性能. 此外,该指令功能测试框架可以根据指令集描述生成指令功能测试用例,提升开发效率.

作者贡献声明:赵紫微为论文工作的主要完成人,负责实验设计与实施、论文撰写;涂航和刘芹审阅论文的内容并提出意见;余涛协助论文功能测试的软件实现;李莉对论文提出修改意见,完善论文思路和实验设计,负责论文审校.

参 考 文 献

[1] Liu Yuchen, Wang Jia, Chen Yunji, et al. Survey on computer system simulator[J]. Journal of Computer Research and Development, 2015, 52(1): 3−15 (in Chinese)
(刘雨辰, 王佳, 陈云霁, 等. 计算机系统模拟器研究综述[J]. 计算机研究与发展, 2015, 52(1): 3−15)

[2] Bellard F. QEMU, a fast and portable dynamic translator[C] //Proc of the 16th USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2005: 41–46

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

[4] Krishna R, Austin T. Efficient software decoder design[C] //Proc of the 1st Workshop on Binary Translation. New York: ACM, 2001: 21–30

[5] Harry W. From high level architecture descriptions to fast instruction set simulators[D]. Edinburgh: University of Edinburgh, 2015

[6] Lockhart D, Ilbeyi B, Batten C. Pydgin: Generating fast instruction set simulators from simple architecture descriptions with meta-tracing JIT compilers[C] //Proc of the 16th Int Symp on Performance Analysis of Systems and Software. Piscataway, NJ: IEEE, 2015: 256−267

[7] Zivojnovic V, Pees S, Meyr H. LISA-machine description language and generic machine model for HW/SW co-design[C] //Proc of the 5th Workshop on VLSI Signal Processing. Piscataway, NJ: IEEE, 1996: 127–136

[8] Hartoog M, Rowson J, Reddy P, et al. Generation of software tools from processor descriptions for hardware/software codesign[C] //Proc of the 34th Design Automation Conf. New York: ACM, 1997: 303−306

[9] Theiling H. Generating decision trees for decoding binaries[J]. ACM SIGPLAN Notices, 2001, 36(8): 112−120

[10] Wei Qin, Malik S. Automated synthesis of efficient binary decoders for retargetable software toolkits[C] //Proc of the 40th Design Automation Conf. New York: ACM, 2003: 764–769

[11] Fournel N, Michel L, Pétrot F. Automated generation of efficient instruction decoders for instruction set simulators[C] //Proc of the

26th Int Conf on Computer-Aided Design. Piscataway, NJ: IEEE, 2013: 739−746

[12] Okuda K, Takeyama H. Decision tree generation for decoding irregular instructions[C] //Proc of the 19th Design, Automation & Test in Europe Conf & Exhibition. Piscataway, NJ: IEEE, 2016: 1592−1597

[13] Okuda K, Chiba S. Domain-specific programming assistance in an embedded DSL for generating processor emulators[C] //Proc of the 36th Annual ACM Symp on Applied Computing. New York: ACM, 2021: 1256–1264

[14] Tadros L. A cost model for decoder decision trees[C] //Proc of the 1st European Symp on Software Engineering. New York: ACM, 2020: 142–147

[15] Mishra P, Dutt N. Functional coverage driven test generation for validation of pipelined processors[C] //Proc of the 8th Design, Automation & Test in Europe Conf & Exhibition. Piscataway, NJ: IEEE, 2005: 678−683

[16] RISC-V International. riscv-tests[EB/OL]. [2022-01-14]. https://github.com/riscv-software-src/riscv-tests

[17] SCons Foundation. SCons: A software construction tool [EB/OL]. [2022-01-14]. https://scons.org/

[18] Beazley D. SLY (Sly Lex-Yacc) [EB/OL]. [2022-01-14]. https://github.com/dabeaz/sly

[19] Free and Open Source Silicon Foundation. Embench: A modern embedded benchmark suite [EB/OL]. [2022-01-14]. https://www.embench.org/

Zhao Ziwei, born in 1998. Master candidate. Her main research interest includes embedded system security.

赵紫微, 1998 年生. 硕士研究生. 主要研究方向为嵌入式系统安全.

Tu Hang, born in 1975. PhD, associate professor. Member of CCF. His main research interests include IoT security, embedded system security, cryptography and network security.

涂　航, 1975 年生. 博士, 副教授. CCF 会员. 主要研究方向为物联网安全、嵌入式系统安全、密码学和网络安全.

Liu Qin, born in 1978. PhD, associate professor. Member of CCF. Her main research interests include IoT security, embedded system security, and cloud computing security.

刘　芹, 1978 年生. 博士, 副教授. CCF 会员. 主要研究方向为物联网安全、嵌入式系统安全和云计算安全.

Li Li, born in 1976. PhD, associate professor. Member of CCF. Her main research interests include embedded system security, IoT security, and mobile security.

李　莉, 1976 年生. 博士, 副教授. CCF 会员. 主要研究方向为嵌入式系统安全、物联网安全和移动安全.

Yu Tao, born in 1997. Master candidate. His main research interest includes embedded system security.

余　涛, 1997 年生. 硕士研究生. 主要研究方向为嵌入式系统安全.