引用格式: 张子航, 罗雯, 陈钢. 使用函数式语言与命令式语言混合开发 EDA 程序的一次探索[J]. 微电子学与计算机, 2023, 40(5): 56-64. [ZHANG Z H, LUO W, CHEN G. An exploration of EDA program development with hybrid programming using functional language and imperative language[J]. Microelectronics & Computer, 2023, 40(5): 56-64.] **DOI:** 10.19304/J.ISSN1000-7180.2022.0492

使用函数式语言与命令式语言混合开发 EDA 程序的一次探索

张子航, 罗 雯, 陈 钢

(南京航空航天大学 计算机科学与技术学院, 江苏 南京 211106)

摘 要: 相较于命令式语言,函数式语言有两个明显的优点: 安全性好、开发周期短.但一般而言,函数式语言的代码性能不够好,妨碍了它的推广和实践应用,尤其是在性能要求很高的领域. 基于上述问题进行了一次函数式语言和命令式语言混合编程的探索,试图在同一个项目中结合两类编程语言混合编程,一方面用函数式语言 OCaml 快速编写复杂度较高的算法核心代码;另一方面,用 C 语言编写难度不大但是对性能影响比较大的代码,通过这种混合编程方式在较短的时间内可以实现一个结构比较复杂但在性能上接近 C 语言编写的同类代码的软件. 选用海量图形数据的高速区域化查询这一案例,该 EDA 问题对运算效率有较高的要求,所以在数据结构上选择四叉树结构来实现区域查询,因此是一个比较有代表性的使用高效数据结构来满足性能要求的问题. 实验结果表明,OCaml 和 C 的混合编程能将核心算法的研发周期明显缩短,同时性能与 C 语言编写的同类型的代码相仿,这也就说明了函数式语言和命令式语言的混合编程可以成为 EDA 软件开发的一个可行的方案.

关键词: 混合编程; 电子设计自动化; OCaml; 函数式语音; 敏捷软件开发; 四叉树

中图分类号: TP311.11; TN702 **文献标识码:** A **文章编号:** 1000-7180(2023)05-0056-09

An exploration of EDA program development with hybrid programming using functional language and imperative language

ZHANG Zihang, LUO Wen, CHEN Gang

(College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics Nanjing, Nanjing 211106, China)

Abstract: Compared with imperative languages, functional languages offer two main advantages: good security and a short development cycle. However, functional language programs are slower, which hinders their practical application, particularly in areas requiring high performances, such as exploratory data analysis (EDA). This study investigates a hybrid programming paradigm with both functional and command languages. Functional language OCaml develops the core code of an algorithm with high complexity. Meanwhile, C language is used to write codes that are not necessarily difficult but contribute significantly to the performance. An extremely complex algorithm can be implemented rapidly while achieving performance comparable to a handwritten C program using this method. A high-speed regional query of massive graphics data is used in the current experiment. Because EDA require a high computational efficiency of the algorithm, the quadtree data structure is used to accomplish an efficient binary region query. This representative problem uses an efficient data structure to fulfill the performance requirements.The experimental results show that the mixed programming of OCaml and C can shorten the development cycle of the core algorithm significantly, and the performance is similar to the code of the same type written in C language, which also shows that the mixed programming of functional language and imperative

收稿日期: 2022-08-17; **修回日期:** 2022-09-09

language can become a feasible program for EDA software development.

Key words: Mixed Programming; Electronic Design Automation(EDA); OCaml; Functional Language; Agile Software Development; Quadtree

1　引言

近年来,随着软件需求的的日益复杂,函数式编程[1-3] 凭借其优秀的安全性和开发过程的高效性在诸多行业开始流行,Facebook 使用 Haskell 来实现新闻的推送,Java 等主流语言也纷纷嵌入函数式模块,越来越多的用户发现了函数式语言的优势[4]. 其次函数式语言具有良好的数学特性和类型系统,这就为硬件设计提供了极大的遍历,随着 VHDL 和 Verilog 在工业上的广泛应用,自 1990 年以来开发的许多函数语言都配备了 VHDL 和 Verilog 的支持. 但函数式语言的效率问题却限制了它在部分领域的发展,命令式语言使用贴近冯·诺伊曼架构的方式进行程序编写,较为低阶,因此效率也相对较高,而函数式编程语言则使用较高阶的数学抽象,这就对其运行效率产生了影响[5]. 相较于大部分函数式语言来说,本文选用的 OCaml 语言在简洁、错误检测、性能等方面都有着很好的优势,但是即使 Ocaml 作为函数式语言的性能优于其余函数式语言,却依旧与 C 和 C++等命令式语言有一定的差距[6].

在硬件设计中使用函数式编程的想法早已被提出,Sheeran 曾在 Hardware Design and Functional Programming 一文中通过大量的实例验证了函数式语言做硬件设计的可行性以及优势[7],并且也有许多研究人员进行了尝试和探索工作,例如 Davy Wolfs 等人曾尝试使用函数式语言编写了一种在 VHDL 中自动生成密码硬件的电子设计自动化(Electronic design automation, EDA)工具[8-9],对于这项逻辑复杂、需要极高安全性且对效率有一定要求的工作,他们通过反复的优化探索,得到了较为优秀的成果,证明了使用函数式编程进行 EDA 开发的可行性.

数字芯片设计中的 EDA 工具设计流程主要由前端设计与后端物理实现两部分组成,其中,前端设计主要包括架构设计、RTL 设计、验证(Verification)、逻辑综合(Sythesis)、一致性验证(Formal)、可测性设计(Design For Test,DFT)等,而物理设计主要包括布局布线(Placement & Routing)、物理验证(Design Rule Check,DRC)等. 受语言特性的影响,在 EDA 应用方面,目前国际上所开展函数式语言开发工作主要在前端设计范围内,大部分工作集中在函数式硬件语言的设计和实现,比较著名的工作有 Lava 语言、Bluespec 语言、Chisel 语言等,EDA 后端工具很少有函数式语言的开发工作[10-11].

基于上述问题,本文尝试使用 OCaml 语言和 C++语言联合实现一个 EDA 算法,利用 OCaml 语言与 C 语言较为完善的通讯模块,把 OCaml 语言的良好的安全性和高开发效率同 C++语言优秀的运行效率结合起来. 本文使用 OCaml 语言完成 EDA 算法的核心部分,即数据处理等逻辑相对复杂的部分,完成后对 OCaml 实现的数据处理及查询部分预留接口,并使用 C++语言封装 OCaml 语言编写的接口函数,而后将完整的 OCaml 项目以及用于封装接口函数的 C++文件编译生成静态库,再使用 C++编写较为影响算法效率的部分功能,主函数在 C++部分实现,在主函数中可直接调用静态库中的接口函数,而后与封装好的静态库链接即可实现混合编程. 合理的语言安排结合高效的数据结构,为短时间内实现一个 EDA 算法奠定了基础,在此基础上尝试使用敏捷设计模式以达到快速开发可执行程序和优化算法效率的目的,敏捷方法主要是以迭代、增量方式进行软件开发,在整个生命周期中不断地改进、优化产生一系列的迭代版本[4]. 同时因为实验选取的 OCaml 语言拥有完善的嵌入模块用于实现与 C/C++的混合编程,因此本项尝试颇具实际意义.

2　实验内容与实现策略

2.1　问题描述

为了验证函数式语言和命令式语言的混合编程可以开发出高效安全的 EDA 软件,本文选择对海量图形数据的高速区域化查询这一常见 EDA 问题进行研究,实验题目及测试数据集均来自于第三届集成电路 EDA 设计精英挑战赛发布的文献 [12],实验问题的详细描述如下:

(1) 给定核心区域为一个由左下角坐标和右上角坐标所限定的一个矩形区域,如{{0,0}, {4 000, 2 000}}

(2) 核心区域内存在大量的矩形图形,每个矩形为一个小的坐标区域,由左下角和右上角坐标所描述,如{{10,10},{50,30}},{{511,320},{517,360}}…

(3) 定义查询区域也是一个矩形区域,如{{50,

50},{100, 120}},要求快速获取该区域内的所有矩形.

2.2 算法设计

区域查询问题国际上已经存在一系列研究,本文采用的四叉树算法是其中的一种比较高效的算法. 四叉树[13] 是一种层次空间数据结构,其中每一层都会对空间进行进一步的细化,通过递归对空间进行分解,是存储二维区域信息的常用结构[14],OCaml 语言作为经典的函数式语言,十分擅长通过递归的方式定义数据结构,例如对于图 1 所示二叉树,在函数式语言 OCaml 中可以使用 Node (Leaf 3, Node (Leaf 4, Leaf 5)) 来递归表述,而如若想要计算各个叶节点上的值的和也只需要使用 match...with...结构来匹配各个 Leaf 节点并记录 Leaf *n* 中 *n* 的值,随后求和即可实现[15],在函数式语言中模式匹配与递归类型的结合使用,实现了命令式语言中需要条件语句、循环语句加上指针构造所实现的算法,不仅安全性较高, 而且逻辑更加简单,代码也更加简短.

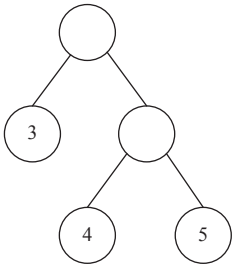

图 1 二叉树示意图

Fig. 1 Schematic representation of binary tree

OCaml 语言的这一优势同 EDA 算法的需求相契合,同时模式匹配机制也十分有利于四叉树相关操作的实现,因此选用基于四叉树的区域查询方法完成上述实验,并用 OCaml 语言编写项目的核心代码,但是由于 OCaml 语言本身在效率方面的短板,若单独使用它完成整个项目,那么代码在效率方面同 C/C++ 编写的软件相比差距比较大. 为了把 OCaml 的优势同 C++的性能相结合,采取的方法是将 OCaml 实现的核心算法嵌入到 C++语言实现的项目当中,对于实现相对困难的查询部分使用 OCaml 语言进行实现,而实现相对简单但十分影响算法运行效率的交互操作等部分则使用 C++进行实现,因此寻找到简洁通用的混合编程手段是本项研究的关键. 采用敏捷软件开发方法将开发过程划分为多个迭代版本,并进行频繁的集成测试,记录比较各个版本的性能表现为后续的代码开发提供了详细的思路. 在首个迭代版本中,当符合要求的矩阵占所有矩阵中的绝大多数时查询效率极低,比使用 C 语言编写的普通的哈希方式查找要慢百倍以上. 分析其原因主要有三点:

(1) 四叉树的各个节点在内存中的存放地址并不是连续的,这极大的影响了算法本身的查找效率.

(2) 如图 2 所示,当 b 节点所表示的区域空间符合要求时,此算法仍会对 f、g、h、i 节点及其子节点(若存在)进行遍历判断操作,这显然是多余的.

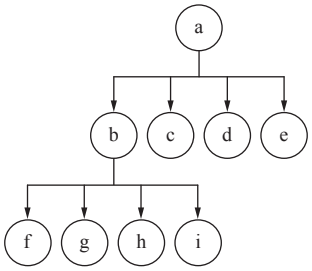

图 2 四叉树示意图

Fig. 2 Schematic diagram of a quadtree

(3) 由于语言本身的性质,OCaml 的打印输出 (Printf 函数) 速度相较于 C 语言要慢很多.

逐步解决上述三个问题,是进行版本迭代的核心也是整个开发过程及优化过程的核心.

2.2.1 基础结构

合理分配两种语言的编写部分才能最大化的发挥混合编程的优势,基于这个核心思想,数据结构使用 OCaml 实现. 将矩形连续的均匀划分为四份,最终划分结果就是一棵四叉树,这种数据结构通过将二维区域分割为多个子区域来为后续的操作提高便利[16]. 作为典型的函数式语言 OCaml 十分擅长使用递归结构解决问题,递归定义的数据结构在实现算法方面要比链表简洁的多,故这样的语言分配可以使得问题的解决过程大大化简[17].

四叉树的每一个节点都是一个由四个 int 型变量构成的组元,其中四个 int 变量两两一组分别表示二维平面中的两个坐标,整个组元代表一个矩阵. 对每一个二维区域进行划分的方式如图 3 所示.

对于 E,F,D(在单一区域内) 一类节点,会被放入子树类型中并划分至对应区域,而对于 A,C,J(横跨或竖跨多个区域) 一类节点,则会被划分至特殊的节点类型中. 建树的过程就是利用递归对区域进行划分,直至将测试集中的所有矩形块填入四叉树.

2.2.2 数据处理方式

数据处理的过程是从文件中读取数据而后递归建树的过程,因此在数据处理工作的实现过程中,首先需要遍历文件,从而获取核心区域的具体范围和数

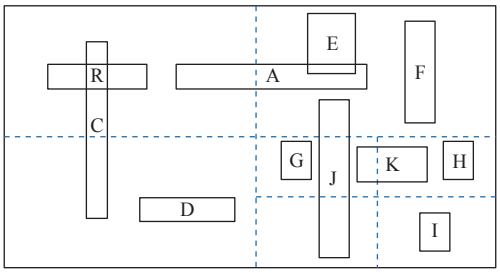

图 3　区域划分示意图

Fig. 3 Schematic diagram of area division

据总量,随后在此遍历文件,并在遍历的过程中通过递归的方式逐行读取文件内数据,并将矩形框的位置参数逐个插入至四叉树中,在插入过程中可根据几个输入参数进行判断,分析插入位置. 这一部分的实现逻辑相对复杂,但是使用 OCaml 的模式匹配结构进行开发,不仅可以降低开发难度,缩短开发周期,安全性也优于命令式语言,并且不会对算法效率造成较大的影响.

遵循敏捷开发的思想,通过多次版本迭代提高代码运行效率,建树的方式和树本身的存放方式是影响查询算法效率最重要的因素之一,因为一般的树结构是使用指针链接的方法实现在内存中的存储,这导致了在内存中各个节点的存放位置并不连续,相较于顺序存储而言,杂乱无序的在内存中存放会导致查询时间的浪费,因此可以针对上述问题进行迭代改进,首先,在建树时用连续的地址 (数组) 存储各个节点,即不再使用链表的结构进行存储而是改用存放地址连续的顺序表. 其次,意图通过对列表内元素进行排序以解决重复查询的问题,如对于图 2,当 b 节点区域均存在于查询区域内时,不希望再对它的子节点进行查询,而是可以直接将以 b 节点为父节点的整颗子树中的所有矩形框放入查询结果列表,这在排序前的树结构中是较难实现的.

经过上述两步的重排,图 2 所示四叉树将会按表 1 所示的顺序存放在内存中,此时的存放顺序可以节省在内存中搜索的时间,并且由于是按照子树的顺序存放,因此,可以方便的解决 2.2 节中提到的问题 2,即在查找时可以减少不必要的细化查询,因此效率相较于重排前会有显著的提升.

2.2.3　查询实现

数据查询工作是基于完成重排的四叉树顺序表进行的,基于上述合理的建树和重排过程所得的顺序表,数据的查询只需对树节点进行遍历判断,筛选出符合要求的节点,因此规范合理的存储形式可以大大简化查询步骤的难度.

表 1　内存存放示意

Tab. 1 Memory storage schematic table

虚拟地址	存放数据
0	a
1	b
2	f
3	g
4	h
5	i
6	c
7	d
8	e

虽然 OCaml 也为用户提供了 Printf 库用于格式化输出,但是由于语言本身在输出性能方面的局限,大批量打印所花费的时间远高于 C,因此在查询过程中使用一个全局数组来存储查询结果,通过混合编程程序接口使得 OCaml 和 C 语言的实现部分均可访问并修改这一数组,最终使用 C 语言遍历打印该数组, 来实现最终查询结果的快速输出.

同时还需要考虑到存储空间的节省优化,在 2.2.2 的建树过程中,两次的四叉树重排会分别开辟两个连续的列表空间,如今的全局数组可以对第一次优化重排时开辟的列表空间进行重复利用,也就是说直接覆盖已经使用完毕的乱序四叉树,从而实现对存储空间的节省优化.

此时数据处理、建树、查询等一系列工作均已在 OCaml 项目中实现,因此只需要将需要在 C 语言项目中被调用的 OCaml 函数封装为接口,即可将 OCaml 项目编译为静态库文件. 如何封装接口和如何在 C 语言项目中调用静态库接口将在下一章内介绍.

3　混合编程策略

在 OCaml 标准库目录下的 caml 子目录中,为使用 C 代码操作 OCaml values 提供了一系列的支持文件,而这些 C 语言头文件中提供的函数为混合编程的实现提供了极大的便利.

本文中核心算法部分使用 OCaml 实现,因此是将 OCaml 嵌入 C++程序,也就是 C++代码来完成主程序的角色,所以需要在 C++中调用 OCaml 函数,这就需要 C++部分提供一个 main 函数作为主函数,一

般情况下,它有一个 int 类型的参数 argc,和一个 char** 类型的参数 argv,而整个混合编程项目也会从这个 main 函数开始执行. 在这个 main 函数中首先（这里是指为了实现后续调用 OCaml 函数,如有其它功能需要首先实现,也可以在此之前编写）需要调用 caml_startup（argv）来初始化 OCaml 代码,其中 argv 参数是一个以 NULL 结尾的 C 语言字符串数组,其类型为 char**,它表示命令行参数,和前文提到的 main 函数的第二个参数是一样的,它用于初始化 Sys.argv. 在完成 OCaml 初始化后,会返回到调用它的 C++代码中继续执行,此时,就可以在 C++中通过回调机制调用 OCaml 函数了.

本文基于上述流程进行了一次混合编程的尝试,本文实现混合编程项目的完整流程以及项目的主文件结构如图 4 所示.

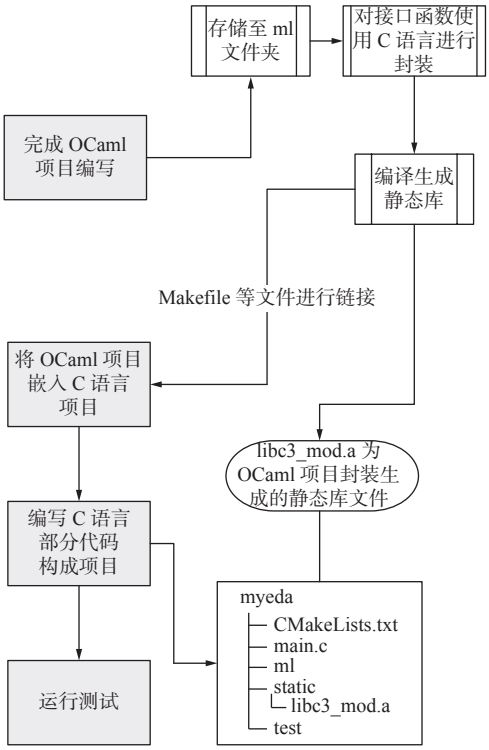

图 4 实现混合编程项目的总流程

Fig. 4 Mixed programming project implementation process

3.1 混合编程设计

合理的分配两种语言的实现部分是实现有效混合编程的基础,这一过程围绕“提高运行效率、缩短开发周期”的思想进行展开. 首先在在 C 语言项目中调用在 OCaml 部分编写的接口函数读取文件并针对文件内的数据构造内部树（C 语言文件无法访问该四叉树）,OCaml 中实现的建树的过程包括对于四叉树的两次重排,而后 C 语言项目会读入表示查询区域的四元组,并调用用于查询的接口函数将上述四元组传输到 OCaml 项目中,从而在 OCaml 中实现四叉树的搜索查询,完成搜索后查询结果被存放在全局数组中,在 C 语言项目中调用接口函数即可获取全局数组的指针,随后在 C 语言部分循环遍历该数组即可完成结果的打印工作. 综上所述,逻辑复杂的数据处理和查询部分选用 OCaml 语言实现,而结果打印等部分选用 C 语言实现这些部分代码简单,命令式语言实现起来并不困难,却能保证较快的运行效率.

结果的输出函数会在查询函数中调用,从而实现查询完后直接打印输出结果,为此在项目中定义了一个全局的数组用于存储最终的输出结果,其中在 OCaml 项目中定义并封装好的函数可以帮助 C 语言项目返回在 OCaml 项目中声明定义的数组的头节点,这样就使得两个项目中可以对同一数组内容进行访问. 在输出时只需要调用函数返回对应的头节点到相应类型的指针,并调用函数返回总矩形数,最后循环输出全局列表中的所有矩形框即可.

3.2 详细技术实现

混合编程的实现核心是 C 语言和 Ocaml 语言之间通信的实现,用 C 和 OCaml 编写的程序各部分之间的通信是通过创建包含这两个部分的可执行文件来完成的,这些部分可以单独编译,后续的链接阶段负责在 OCaml 函数名和 C 函数名之间建立连接,并创建最终的可执行文件. 为此,程序的 OCaml 部分包含描述此连接的外部声明. 在 OCaml 中编写的函数需要在 OCaml 程序中进行封装并在 C 语言中进行声明方可在项目的 C 语言部分直接使用,同理在 OCaml 中调用 C 语言函数也需要进行外部声明. 如图 5 中函数 f_c 与类型为 int -> int -> int -> int 的 OCaml 函数相关联,它是一个 C 语言函数,有三个参数,最终会返回三个参数的和[18].

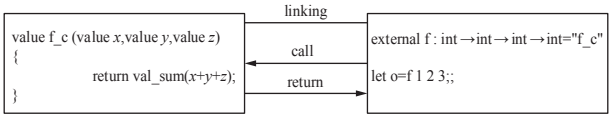

图 5 通信演示[18]

Fig. 5 Communication demo

为了实现混合编程,首先需要将 OCaml 文件进行封装,生成一个可以被 C 语言直接调用的静态库文件或是直接得到一个可执行文件,其次对 CMakeList 等编译文件进行修改,加入对静态库或可执行文件的链接调用,在完成这些改动,并且在 C 语言主程序中加入 caml_startup 函数完成 OCaml 代码的初始化后,

就可以在 C 语言文件中直接实现对 OCaml 语言编写的函数的调用. 实现链接有多种链接模式,但是本机代码编译器、字节码编译器的链接命令并不相同,更多的链接方式和详细的链接代码读者可参阅文献 [18].

在本次尝试中选用了将 OCaml 项目进行封装并在 C 语言项目中调用的方式,在生成静态库前需要将 OCaml 中的接口函数用 C 语言代码对其进行封装,也就是在 OCaml 项目中新建一个.c 文件,在其中把需要在 C 部分中调用的 OCaml 函数进行逐一封装,由于封装过程是在 OCaml 项目中也就是生成静态库前实现的,因此 C 语言部分在调用静态库的接口时调用的已是 C 语言函数.

而上述封装过程实际是使用 OCaml 的闭包来实现的. OCaml 为实现混合编程提供了一个注册机制,也就是可以将一个 OCaml 函数注册至一个全局的名字来供其它语言的实现部分调用. 首先需要在 OCaml 中使用 Callback.register s v 为函数 v 注册全局名称 s,而后在封装接口的 C 文件中使用 caml_named_value (s) 获取对应 value 的指针,除此之外,还需要在封装函数中使用 callback 获取要调用的 OCaml 函数的闭包,图 6 展示了一个完整的定义 OCaml 函数 f 和将其封装为 C 接口 call_f 的例子,此时的 call_f 函数即是一个纯粹的 C 语言函数,可以在 C 语言算法中被直接调用.

```
let f x = print_int x
let _ = Callback.register "packaging" f
```

封装

```
void call_f(int arg)
{
    static value *f_closure = NULL;
    if (f_closure == NULL) {
        f_closure =
caml_named_value("packaging");
    }
    caml_callback(*f_closure, Value_int(arg));
}
```

图 6　封装函数的实现

Fig. 6 Encapsulate the implementation of functions

在 Makefile 中使用 ar 命令生成.a 后缀的静态库,并使用 cp 命令将生成的静态库复制存放到指定目录,存放位置如图 4 所示. 本文中静态库命名为 libc3_mod.a,放于 static 文件夹中. 随后需要在 C 语言项目的 CMakeLists 中链接编译刚刚生成的静态库. CMakeLists 中的文件链接通过 cmake_minimum_required、include_directories、LINK_DIRECTORIES、add_executable、target_link_libraries 的依次使用来实现,完成链接后,在 C 语言项目中可以直接调用静态库的接口,也就是封装好的 C 语言函数,不过在调用前还需要需使用 extern 关键字对函数进行声明.

OCaml 本身有着十分优秀的多语言联合程序设计的基础,除了本文选用的 C++之外,OCaml 与 C、C#、F#等语言都有着实现混合编程的基础支持,例如文献 [15] 中的第七章就提供了一个 C#、Access、F# 和 OCaml 多语言多工具联合开发的参考案例.

4　敏捷开发方法与实验结果分析

4.1　迭代与增量开发过程

迭代开发是构建软件的一种方法,即在软件开发的整个生命周期中不断进行改进和优化从而产生一系列的迭代版本,最后一次迭代的版本即为最终完成的成果. 增量开发是指针对各迭代版本的集成测试结果进行分析,并根据需求依次递增地添加软件功能,开发初期优先实现基本功能,在此后的版本中不断进行优化升级[19].

利用敏捷软件开发的快速迭代特征进行频繁的集成测试,从而高效编写简洁可用的高质量代码[20],相较于其他只在最后交付整个应用程序的开发方法,本次试验使用敏捷软件开发方法会得到多个功能完整且可独立运行的代码版本,并且在整个开发过程中,使用敏捷软件开发可以对各个代码版本进行测试、分析和改进,而不用等到最终成果确定后才能进行测试,这使得在开发的过程中可以及时进行测试并且发现代码问题,为之后的版本迭代奠定基础[21]. 下面是这次实验过程中得到的代码版本:

(1) 版本一:完成了基本功能,使用四叉树实现区域查询,得到查询正确率 100% 的算法.

(2) 版本二:更改了节点和树的可变性,使得初始化期间可以更新树和节点.

(3) 版本三:使用通用的一维整数数组对所有矩形的列表进行存储.

(4) 版本四:对四叉树存储进行了重排,将树节点存储在内存地址连续的列表中,并且将各个子树集中存放在顺序表中,这一次迭代实现了 2.2.2 中提到的关键的数组重排过程.

(5) 版本五:实现了用于存储结果的全局列表,用 C 语言中对全局列表进行打印.

根据上述增量开发的过程,对核心算法和混合编程模式进行开发和优化,各迭代版本的算法性能表现如表 2 所示,三组测试数据来源于同一测试集的三个测试案例,它们的总矩阵数据一致,但是选区的查询区域不同,即符合要求的矩形框占总矩形框的比例也存在较大的跨度. 在输出矩形框数量较多时 (test3) 第五个迭代版本的表现远优于前继版本,很好地验证了本次尝试的合理性,通过对算法实现难度和实现效率的分析,合理采用不同的语言完成算法不同步骤的实现,来得到符合需求的最终算法.

4.2 与标准运行时间对比

为了验证本文提出的混合编程方法满足算法性能要求,还需将混合编程实现的算法与标准算法运行效率进行比对,其中标准算法是 C++语言编写的哈希查找算法,测试用例为赛题项目组提供的两组(总矩形框数据集相同)共六个(查询区域不同)测试案例,具体对比如表 3 所示,在时间效率方面,本文的混合编程实现在大多用例中都要优于完全由 C 语言编写的哈希查询算法的效率,但是在数据集较小且符合要求的矩形占总矩形的比例较大时,效率会略低于哈希查询算法的效率,导致这一问题的原因有很多种,如函数式语言本身的性能. 但是对于大多的数据集,混合编程项目的查询效率表现的非常优秀.

表 2 各版本运行效率分析

Tab. 2 Analysis of operating efficiency of each version

版本	符合查询要求的矩形框数(最终输出的矩形框数)	时间占用/ms	空间占用/Mib
版本一	26	0.195	411.84
版本二	26	0.269	410.64
版本三	26	0.323	438.94
版本四	26	0.095	466.18
版本五	26	0.076	465.05
版本一	17 409	22.914	412.04
版本二	17 409	16.545	412.58
版本三	17 409	39.245	438.83
版本四	17 409	19.124	464.06
版本五	17 409	2.429	467.04
版本一	83 935	92.595	412.16
版本二	83 935	70.373	410.59
版本三	83 935	114.335	438.79
版本四	83 935	69.094	464.12
版本五	83 935	7.495	466.97

表 3 与标准算法对比

Tab. 3 compared with standard algorithms

符合查询要求的矩形框数	总矩形框数	标准算法运行时间/ms	混合编程算法运行时间/ms	混合编程算法空间占用	标准算法空间占用
26	233 778	4.409	0.076	375.14 Mib	465.05 Mib
17 409	233 778	5.028	2.429	374.85 Mib	467.04 Mib
83 935	233 778	5.255	7.495	375.22 Mib	466.97 Mib
2009	9 289 725	71.908	0.918	1.55 Gib	4.96 Gib
22 925	9 289 725	79.247	6.775	1.55 Gib	4.96 Gib
2 553 557	9 289 725	131.892	189.399	1.6 Gib	4.96 Gib

4.3 时间效率的详细分析

对比 4.2 节中的测试数据,可以明显观察到第一个迭代版本的运行效率远远逊色于标准算法,而版本迭代的过程是解决前继版本问题的过程. 表 2 前五行所对应的测试用例符合要求的矩阵数只有 26 个,数据量较少因此运行时间的增加和减少并不明显,因此后续分析的趋势主要针对剩余测试数据. 版本二相较于版本一效率有小幅度提升,是由于建树更加规范灵活,减少了查询时所需要的的时间,版本三相较于版本二效率有些许降低,是因为版本三是更改存储所有矩形框的列表所用方式的中间版本,弃用 List 结构改选 Bigarray 结构,主要是为后续使用 Bigarray.Array1 存储所有矩形的列表做准备,是实现版本四前的关键尝试,这个版本的效率降低也说明了在使用敏捷开发方法进行版本迭代的过程中,并不是总能得到令我们满意的版本, 偶尔出现的反向优化的也是正常现象,只要是合理且为后续版本迭代做出了贡献的迭代版本,都是成功的迭代过程,版本四是效率提升

最关键的一环之一,在这次迭代过程中实现了 2.2.2 中介绍的四叉树的两次重排,由于减少了在内存中的搜索时间,并且不再对符合要求的子树进行重复查询,所以效率大幅提高,版本五更改了结果的打印方式,改用 C++来实现结果的打印,根据表 2 可知,对于符合要求的矩形框较多,也就是需要打印的内容较多时,这一改动使得运行时间缩短了接近十倍,这次迭代过程体现了混合编程的巨大优势,它规避掉了函数式语言的硬性短板,使得 C++优秀的运行效率在混合编程项目中得以发挥.

版本的迭代和增量开发过程遵循的基础是软件的需求,算法性能的提升是这次混合编程尝试的主要需求目标,以第三个测试集的结果为例,经过五个版本的迭代,算法运行时间缩短了 12.5 倍. 这一结果印证了本次尝试中使用的方法具有一定的可行性以及推广价值.

4.4　空间效率的详细分析

空间占用率在最终版本要比官方的测试数据(官方测试数据是使用 C 语言编写的哈希查找算法运行得到的标准参考数据)大很多,这是由于两个重排数组都占用了不小的内存. 虽然在迭代过程中为了节省空间做了一些尝试,比如存储最终查询结果的数组是覆盖了重排过程中产生的废弃数组,但是最终版本也依旧在空间效率方面的表现依旧逊色于标准查询方法,不过从整体表现上看,为时间性能提升而做出的空间性能牺牲是值得的.

5　结束语

函数式语言的优势在近年来被越来越多的开发人员所认可,同样在 EDA 领域也有着巨大的发展潜力,却因为运行效率问题受到了一定的限制,本文提出了使用混合编程的手段来弥补函数式语言在目标代码性能上的不足,为加快 EDA 软件的研发开辟了一条新的思路. 而合理的混合编程策略是使用混合编程手段进行开发的基础,这需要清晰认识各个语言的优势,并合理分析软件开发过程中不同部分实现的复杂度和对整体性能的影响,针对每个部分都尽可能使用最合适的编程语言[15]. 在本次尝试中选用 C++与 OCaml 语言进行混合编程最终得到的迭代版本能够较好的满足 EDA 问题在运行效率方面的需求,这种综合各个语言的优势的编程策略,可以实现快速又高效的软件研发,而多语言联合程序设计也并不局限于这两种语言,着眼于需求选择合适的语言进行混合编程是后续研究工作中关注的问题.

参考文献:

[1] HUGHES J. Why functional programming matters[J]. The Computer Journal, 1989, 32(2): 98-107. DOI: 10.1093/comjnl/32.2.98.

[2] WADLER P. The essence of functional programming [C]//Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Albuquerque: ACM, 1992: 1-14. DOI: 10.1145/143165.143169.

[3] HUDAK P. Conception, evolution, and application of functional programming languages[J]. ACM Computing Surveys, 1989, 21(3): 359-411. DOI: 10.1145/72551.72554.

[4] HU Z J, HUGHES J, WANG M. How functional programming mattered[J]. National Science Review, 2015, 2(3): 349-370. DOI: 10.1093/nsr/nwv042.

[5] 蔡学镛. 思考函数式编程[J]. 程序员, 2008(7): 96-98.
CAI X Y. Think about functional programming[J]. Programmer, 2008(7): 96-98.

[6] MINSKY Y. OCaml for the masses[J]. Communications of the ACM, 2011, 54(11): 53-58. DOI: 10.1145/2018396.2018413.

[7] SHEERAN M. Hardware design and functional programming: a perfect match[J]. Journal of Universal Computer Science, 2005, 11(7): 1135-1158. DOI: 10.3217/jucs-011-07-1135.

[8] WOLFS D, AERTS K, MENTENS N. Design space exploration for automatically generated cryptographic hardware using functional languages[C]//Proceedings of the 22nd International Conference on Field Programmable Logic and Applications. Oslo: IEEE, 2012: 671-674. DOI: 10.1109/FPL.2012.6339174.

[9] WOLFS D, AERTS K, MENTENS N. Implementing an electronic design automation tool for cryptographic hardware using functional languages[C]//Proceedings of Symposium on Trends in Functional Programming. St Andrews, 2012.

[10] CHEN G. A short historical survey of functional hardware languages[J]. International Scholarly Research Notices, 2012, 2012: 271836. DOI: 10.5402/2012/271836. DOI: 10.5402/2012/271836.

[11] HUANG G Y, HU J B, HE Y F, et al. Machine learning for electronic design automation: a survey[J]. ACM Transactions on Design Automation of Electronic Systems, 2021, 26(5): 40. DOI: 10.1145/3451179.

[12] 杨帆. 集成电路EDA设计精英挑战赛[EB/OL]. https://eda.icisc.cn/file/cacheFile/0314c02158c34a3a9189703f371ee713.pdf.
YANG F. Integrated circuit EDA elite challenge [EB/OL]. https://eda.icisc.cn/file/cacheFile/0314c02158

c34a3a9189703f371ee713.pdf.

[13] SAMET H. The quadtree and related hierarchical data structures[J]. ACM Computing Surveys, 1984, 16(2): 187-260. DOI: 10.1145/356924.356930.

[14] D'ANGELO A. A brief introduction to quadtrees and their applications[C]//Proceedings of the Style File from the 28th Canadian Conference on Computational Geometry. Canada, 2016.

[15] 陈钢, 张静. OCaml语言编程基础教程[M]. 北京: 人民邮电出版社, 2018.
CHEN G, ZHANG J. OCaml language programming foundation tutorial[M]. Beijing: People's Post and Telecommunications Press, 2018.

[16] 王汝传. 用四叉树对二维图形进行处理的算法[J]. 南京邮电学院学报, 1997, 17(1): 83-86.
WANG R C. Algorithm of applying quadtree in 2-dimension graphic processing[J]. Journal of Nanjing Institute of Posts and Telecommunications, 1997, 17(1): 83-86.

[17] 萧柯. 图象的四叉树结构及其应用的研究与发展[J]. 遥感技术与应用, 1991, 6(1): 54-62. DOI: 10.11873/j.issn.1004-0323.1991.1.54.
XIAO K. Research and development of image quad tree structure and its application[J]. Remote Sensing Technology and Application, 1991, 6(1): 54-62. DOI: 10.11873/j.issn.1004-0323.1991.1.54.

[18] INRIA. Communication between C and Objective CAML[EB/OL].https://caml.inria.fr/pub/docs/oreilly-book/html/book-ora114.html.

[19] 林鑫瀚. 敏捷方法与小型团队的软件开发[J]. 软件导刊, 2009, 8(9): 29-31.
LIN X H. Agile approach with software development for small teams[J]. Software Guide, 2009, 8(9): 29-31.

[20] 马佳俊, 罗翊坤. 敏捷软件开发的三重迭代模型[J]. 电子技术与软件工程, 2017(6): 52-54.
MA J J, LUO Y K. Triple iterative model for Agile software development[J]. Electronic Technology & Software Engineering, 2017(6): 52-54.

[21] GHEORGHE A M, GHEORGHE I D, IATAN I L. Agile software development[J]. Informatica Economica, 2020, 24(2): 90-100. DOI: 10.24818/issn14531305/24.2.2020.08.

作者简介:

张子航 女,(2000-). 研究方向为软件工程、EDA.

罗 雯 女, (2000-). 研究方向为软件工程.

陈 钢(通讯作者) 男,(1958-),教授. 研究方向为形式化工程数学、COQ 定理证明、函数式语言、类型系统、形式化方法、控制系统. E-mail: gangchensh@nuaa.edu.cn.