学号 2018302110349

密级 _____

武汉大学本科毕业论文

面向非易失性内存的
栈空间损耗均衡方法研究

院（系）名　称：计算机学院

专　业　名　称：软件工程

学　生　姓　名：吴系风

指　导　教　师：李清安　副教授

二〇二二年五月

郑 重 声 明

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。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，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名：吴豪风　　日期：2022年5月11日

摘　　要

随着计算机系统处理的数据量迅猛增长，当前计算机系统以动态随机存储器为基础的主存系统在集成度、性能、功耗等方面越来越难以满足需求，以相变存储器为代表的非易失性存储器被认为是最有可能代替动态随即存储器的下一代存储器之一。然而，相变存储器耐久度过低的缺点尤其突出，必须将写操作尽可能均摊到存储器的所有字节上以提升其使用寿命。可以将程序中的循环转化为递归函数有效地实现损耗均衡，但该方法存在着创建的递归函数参数冗余、返回值的传递方式低效的问题。这使得生成的递归函数栈帧较大、指令数较多，在取得较好损耗均衡效果的同时容易引发栈溢出异常，也影响了程序运行的性能。

针对这些不足之处，我们在原有循环转递归方法 Loop2Recursion 的基础上，设计并实现了一种新的循环转递归方法 New-Loop2Recursion，减小生成的递归函数的栈帧，减少生成的递归函数的指令数，减少出现栈溢出异常的概率，提升程序运行的性能。由于调用迭代次数较多的循环转化生成的递归函数仍然容易因为递归过深而引发栈溢出异常，我们在 New-Loop2Recursion 中实现了限制递归深度、在递归中包含循环两种进一步优化栈内存使用的策略。最后，我们通过实验对 New-Loop2Recursion 和两种进一步优化栈内存使用的策略进行全面的评估。

实验结果表明，New-Loop2Recursion 在损耗均衡有效性整体上未受到很大影响的情况下，有效地减小了生成的递归函数的栈大小（平均为-44.68% ），减少了生成的递归函数的指令数（平均为-10.21% ），从而有效地减小了出现栈溢出异常的概率，提升了程序运行的性能。同时，实验也证实了在递归深度限制较大的情况下，限制递归深度可以有效进一步减小栈区内存使用量（平均为-44.05% ），且对损耗均衡有效性影响更小，是更优的进一步优化栈内存使用的策略。

关键词：非易失性内存；损耗均衡；循环；递归；LLVM

ABSTRACT

With the rapid growth of data processed by computer systems, DRAM-based main memory systems are struggling to keep up with the ever-increasing demands for integration, performance, and power efficiency. Non-Volatile Memory (NVM), represented by Phase Change Memory (PCM), is regarded as the most promising next-generation storage technology. However, PCM suffers from a severe write durability problem, and as a result, it is required to spread write operations evenly over all bytes to increase its lifespan. Converting loops in programs into recursive functions is a viable way of achieving wear balancing, however, such a method creates functions with redundant parameters, and return values are passed in an inefficient manner. This leads to large stack frame sizes and instruction counts in the generated recursive functions, which in turn has a significant performance penalty and is vulnerable to stack overflow exceptions.

We have designed and implemented New-Loop2Recursion that reduces the generated recursive functions' stack frame sizes and instructions, which in turn reduces the risk of stack overflows and improves the performance of the generated programs. As calling recursive functions converted from loops with a large number of iterations is still prone to stack overflows, we have implemented two schemes to further optimize stack memory usage, namely limiting recursion depth and including loops within recursive functions. Finally, we have assessed our methods through comprehensive experiments.

The experimental results show that New-Loop2Recursion significantly reduced both the stack sizes (on average -44.68%) and instructions (on average -10.21%) of generated recursive functions, thus effectively improving performance, while incuring a minimal overall cost on wear balancing. Furthermore, setting a large limit on recursive depth outperforms including loops within recursive functions, as it greatly further reduces stack sizes (on average -44.05%), and adds a lower penalty on wear balancing.

Key words: non-volatile memory; wear balancing; loop; recursion; LLVM

目　　录

图片索引

表格索引

1 绪论

1.1 研究课题的目的和意义

在大数据、人工智能的背景下，计算机系统处理的数据量迅猛增长，当前计算机系统以动态随机存储器（Dynamic Random-Access Memory，DRAM）为基础的主存系统在集成度、性能、功耗等方面越来越难以满足需求，越来越多的研究探索新的存储器件。其中，相变存储器（Phase Change Memory，PCM），既像动态随机存储器一样可字节寻址，又具有高集成度、低功耗、非易失等优点[1]。由于在现代计算机体系架构下控制主存系统的功耗尤为重要[2]，特别是对于深度神经网络[3]等数据密集、主存负载较重的应用场景，相变存储器具有十分广泛的应用场景[4]。

然而，作为一项新兴技术，相变存储器具有一些不足之处，如写延时是动态随机存储器的 4-8 倍，写耐久度（单个存储单元可写入次数）比动态随机存储器小 8 个数量级等[5]；其中写耐久度过低的缺点尤其突出，严重阻碍了相变存储器的广泛应用[6]。在写入均衡的情况下，相变存储器的使用寿命大概在 4-20 年之间，而在缺乏特殊的写策略的情况下，相变存储器的寿命甚至只有 1.1 月[7]。

1.2 国内外研究现状

为了解决相变存储器写耐久度过低的问题，研究人员提出了各种硬件级别和软硬件结合的方法。这些方法主要分为减少相变存储器存储单元的写入次数的方法，以及将写操作尽可能均摊到相变存储器的所有存储单元中的损耗均衡方法。

1.2.1 硬件级别的方法

1.2.1.1 减少相变存储器存储单元的写入次数的方法

在硬件级别上减少相变存储器存储单元的写入次数，可以采用基于 DRAM 和 PCM 的混合内存架构，将写操作尽可能分配在 DRAM 上而非 PCM 上[8–11]。除了减少相变存储器存储单元的写入次数之外，这一类方法还能在一定程度上缓解 PCM 写延迟的问题。

而在纯 PCM 的内存架构中，则可以在位的级别上减少写入。文献[12–14] 采用了将按位比较与按位写向结合的方法：在进行写操作的时候，将被写入某内存地址

的值与该内存地址当前的值进行按位比较，然后只针对不同的位进行写操作。在此基础上，文献[15]进一步提出了 Flip-N-Write 方法，在将被写入某内存地址的值与该内存地址当前的值进行按位比较后，计算需要写入的位数，并根据位数写入新值或新值取反后的值，保证每次写入的位数不超过数据总位数的一半。除此之外，在纯 PCM 的内存架构中，文献[14, 16]还提出可以通过对写入的数据进行压缩编码，达到减少写入的目的。

1.2.1.2 损耗均衡方法

在硬件级别上，可以通过转移或交换存储页、存储块、存储线等不同粒度下不同位置处的数据达到损耗均衡的目的。在存储页的粒度下，Ferreira 等人[17]提出对各存储页的写次数进行计数，当某一存储页的写次数达到一个阈值时，随机选取一个存储页与当前要写入的存储页进行交换。该方法的内存开销较小，但牺牲了损耗均衡的精确性。在存储块的粒度下，Zhou 等人[12]提出将内存划分为若干 1 MB 的存储块，对每个存储块的写次数进行计数，然后周期性地交换写次数较多的存储块和写次数较少的存储块的数据。相较于存储页的粒度下方法，该方法该方法的内存开销较高。而在存储线的粒度下，Qureshi 等人[10]提出了通过周期性地循环交换存储线，避免某个存储线写入次数过多的 Start-Gap 方法；Zhao 等人[18]提出在存储线内周期性地将第 i 位与第 $n-i$ 位交换，实现存储线内的损耗均衡。相较于存储页、存储块粒度下的方法，存储线粒度下的方法实现了更为精确的损耗均衡，但会引起更高的内存开销和性能开销。除了这些基于转移或交换的方法之外，文献[19]指出，可以在相变存储器的控制单元内维护将常被更新的热数据映射到写次数较少的存储单元的表，实现一种更为灵活的损耗均衡，而文献[6, 20]对该方法的性能和内存开销进行了优化。

1.2.2 软硬件结合的方法

1.2.2.1 减少相变存储器存储单元的写入次数的方法

由于硬件级别的方法往往需要定制化的硬件，且具有较高的硬件开销，很多研究人员也提出了软硬件结合的方法，减少相变存储器存储单元的写入次数。

在基于 DRAM 和 PCM 的混合内存架构下，可以在编译阶段的数据分配[21–24]，以及运行阶段操作系统级别的页面调度[25]，将读写频繁的数据分配在 DRAM 中，达到减少相变存储器存储单元的写入次数的目的。而针对纯 PCM 的内存架构，可

以在编译阶段对数据写回 PCM 再读取、重新计算数据的代价进行评估和比较，如果重新计算数据的代价更小，则不将数据写回 PCM 中，而在下次访问该数据时重新计算它[26]。

1.2.2.2 损耗均衡方法

软硬件结合的损耗均衡方法主要在操作系统级别和程序级别发挥作用。

在操作系统级别上，主要通过特殊的页面调度和页面置换算法达到损耗均衡的目标。Dhiman 等人[27] 将页面分为写入次数较多的“年迈”页面和写入次数较少的“年轻”页面，在发生页面请求时，优先分配“年轻”的页面。在此基础上，文献[28–30] 进一步在进行页面分配时将“年迈”的逻辑页面分配给“年轻”的物理页面，而 Chen 等人[31] 则提出了一种用于管理页面的数据结构，使得分配页面时总是分配“年轻”的页面，且当某个页面写入过热时，该方法会选择一个“年轻”的页面与这个页面进行置换。另一方面，Gotge 等人[32] 利用 Intel 的 PEBS 对页面进行损耗预测，并通过随机化的方式实现损耗均衡。同时，针对 DRAM 和 PCM 的混合内存架构，Liu 等人[33] 提出了根据页面的损耗程度组织页面的页管理架构 MH-BS、Wang 等人[34] 提出了实现损耗均衡的页面置换算法 CLOCK-RWRF。

而程序级别的损耗均衡方法主要通过优化内存分配和访问的模式将写操作均摊到程序的内存空间中。相较于硬件级和操作系统级的损耗均衡方法，程序级别的损耗均衡方法提供了一种更为经济灵活的损耗均衡方案，特别是在没有内存管理单元（MMU）和缓存，难以适配硬件级和操作系统级的损耗均衡方法的嵌入式系统上。

在堆区内存损耗均衡方面，Moraru 等人[35] 以及 Yu 等人[36] 提出了损耗感知的堆区内存分配器，避免大多数堆对象分配到同一地址。

而内存中不同区域的写次数极不均衡的情况在栈区尤为突出。在栈区，每当进行函数调用和返回时，通过调整栈指针寄存器，为存储函数信息（如函数的返回地址、参数、局部变量）的栈帧分配内存，或者释放栈帧的内存。在这种机制下，如果在循环体中出现函数调用，将可能导致在栈区的一些内存地址处累计分配大量的栈帧，而在另一些内存地址处几乎没有栈帧。针对这种弊端，受到堆区内存分配方法损耗均衡效果较好的启发，Li 等人[37] 提出了由编译器辅助的动态栈帧分配方法；在此基础上，Li 等人[38] 对该方法进行了改进。该方法虽然可以使得栈帧整体上更均匀地分布在内存中，但未考虑栈帧内部的损耗均衡。特别是一些经常被

更新的变量，会在它们所在的内存地址处产生大量的写操作。为了解决这个问题，考虑到循环通常会引入常被更新的热变量和在栈同一个位置处大量重复的函数调用，Li 等人[39] 提出了 Loop2Recursion，通过将循环转换为递归函数，将这些损耗均摊到程序的整个栈区中，实现栈帧内部的损耗均衡。然而，该方法存在创建的递归函数参数冗余、返回值的传递方式低效的问题，使得生成的递归函数栈帧较大、指令数较多，在取得较好损耗均衡效果的同时容易引发栈溢出异常，也影响了程序运行的性能。

1.3 研究课题的主要内容

针对原版循环转递归 Loop2Recursion 的不足，我们基于 LLVM 13.0.1，以对源代码编译生成的 LLVM IR 进行变换的 LLVM Pass 的形式，设计并实现了一种新的循环转递归方法 New-Loop2Recursion，减小生成的递归函数的栈帧，减少生成的递归函数的指令数，从而减小出现栈溢出异常的概率，提升程序运行的性能。由于调用迭代次数较多的循环转化生成的递归函数仍然容易因为递归过深而引发栈溢出异常，我们针对 New-Loop2Recursion，适配了限制递归深度、在递归中包含循环两种进一步优化栈内存使用的策略。在此基础上，我们使用来源于 MiBench[40] 的源代码，从耗均衡的高效性和有效性两方面对我们的方法进行评估，包括比较 New-Loop2Recursion 和原版 Loop2Recursion，以及探究并比较限制递归深度、在递归中包含循环两种进一步优化栈内存使用的策略的效果。

我们研究课题的主要内容概括如下：

- 我们实现了改进的循环转递归方法 New-Loop2Recursion，对生成的递归函数的栈帧和指令数进行优化；
- 我们在 New-Loop2Recursion 中实现了限制递归深度、在递归中包含循环两种进一步优化栈内存使用的策略；
- 我们通过实验对 New-Loop2Recursion 和两种进一步优化栈内存使用的策略进行了全面的评估。

2 背景知识

2.1 静态单赋值形式

静态单赋值形式（Static Single Assignment Form，SSA）是编译器中间表示的一种属性[41]。在静态单赋值形式下，源代码中的变量（如 x、y、z）被分割成许多不同的版本（如 x_1、x_2、y_1、y_2、z_1、z_2），其中每个版本仅被赋值一次。

对于如下所示的源代码：

```
y = 1
y = 2
x = y
```

下面是一种满足 SSA 形式的中间表示：

```
y_1 = 1
y_2 = 2
x_1 = y_2
```

在静态单赋值形式下，可能会出现与分支相伴的不同版本的变量的合并问题。如下所示的代码：

```
x = 5
x = x - 3

if x < 3:
    y = x * 2
    w = y
else:
    y = x - 3

w = x - y
z = x + y
```

在满足静态单赋值形式的中间表示下，会出现如下所示无法表示的 y?：

```
x_1 = 5
x_2 = x_1 - 3

if x_2 < 3:
    y_1 = x_2 * 2
    w_1 = y_1
else:
```

```
    y_2 = x_2 - 3

w_2 = x_2 - y?
z_1 = x_2 + y?
```

为了解决这个问题，在静态单赋值形式中增加一个特殊的描述，被称为Φ（Phi）函数。Φ 函数根据程序运行的路径选择值，如 `Φ(y_1, y_2)` 将会根据程序经过了给 `y_1`、`y_2` 的哪一个赋值的路径，选取 `y_1` 或 `y_2` 的值。使用 Φ 函数，我们可以在 `if-else` 后面通过 `y_3 = Φ(y_1, y_2)` 选取 `y_1` 或 `y_2` 的值，再将下面的 `y?` 表示为 `y_3`，解决上述问题。如下所示：

```
x_1 = 5
x_2 = x_1 - 3

if x_2 < 3:
    y_1 = x_2 * 2
    w_1 = y_1
else:
    y_2 = x_2 - 3

y_3 = Φ(y_1, y_2)

w_2 = x_2 - y_3
z_1 = x_2 + y_3
```

许多编译优化算法，如常量传播、消除无用的代码、寄存器分配，在 SSA 形式下更容易完成。因此，LLVM、V8 引擎、安卓运行时（Android Runtime，ART）等多个不同后端均采用 SSA 形式的中间表示。

2.2 LLVM

LLVM[42] 是一套编译器基础设施项目，起源于 2000 年伊利诺伊大学厄巴纳-香槟分校维克拉姆·艾夫（Vikram Adve）与克里斯·拉特纳（Chris Lattner）的研究。它是一个用于开发编译器前端、后端的 C++ 库，同时包含了一组基于它们开发，模块化程度高，完成生成中间表示、优化中间表示、生成机器码等任务的可执行文件。相较于 GCC 等传统编译器，LLVM 具有模块化程度高、可扩展性强、中间表示可读性好等优势，得到了苹果、谷歌等众多大企业背书和广泛使用。

2.2.1 LLVM IR

LLVM IR 是 LLVM 内部采用的中间表示。它是一种强类型的、静态单赋值形式的精简指令集，抽象了目标平台函数调用约定等底层细节，并使用满足 SSA 形式、数量不限的中间结果存储值。LLVM IR 具有三种不同的等价形式：适合编译器前端生成和优化器操纵的对象表示形式、适合存储在磁盘上的字节码形式，以及适合人阅读的纯文本汇编代码形式。其中，字节码形式和汇编代码形式可以利用命令行工具 `llvm-as`、`llvm-dis` 相互转换，而 `clang` 等 LLVM 前端以及 LLVM 的代码优化器 `opt` 可以将对象表示形式序列化为字节码形式、汇编代码形式，或者将字节码形式、汇编代码形式反序列化为对象表示形式。

2.2.2 LLVM IR 中的循环

LLVM 的代码优化器 `opt` 可以通过运行 `LoopInfoWrapperPass`，从 LLVM IR 描述的控制流中检测循环。在 LLVM 中，循环是满足如下性质的基本块集合：

- 这些基本块的诱导子图是一个强连通子图。这意味着其中任意一个基本块从任意其他基本块出发都是可达的。
- 所有从集合外的基本块出发指向集合内的边，均指向集合中的同一个基本块。该基本块被称为 `Header`，它支配该集合的所有基本块。
- 这个集合是极大的。

在 LLVM 中，有如下的与循环相关的术语：

- `Entering` 基本块。循环外有一条指向 `Header` 的出边的基本块（可能还有指向循环外其他基本块的出边）。
- `Latch` 基本块。循环内有一条指向 `Header` 的出边的基本块（可能还有指向循环内其他基本块的出边）。
- `Exiting` 基本块。循环内有一条或多条指向循环外基本块的出边的基本块（可能还有指向循环内其他基本块的出边）。
- `Exit` 基本块。循环外有一条或多条来自 `Exiting` 基本块的入边的基本块（可能还有来自循环外其他基本块的入边）。

LLVM 循环的结构如图2.1所示。

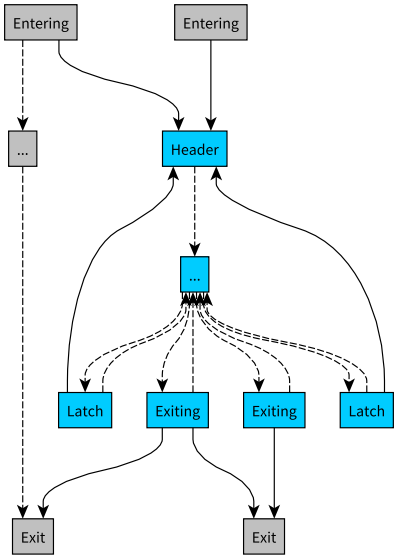

图 2.1　LLVM 循环的结构

而 `opt` 通过运行 `LoopSimplify`、`LCSSA`，还可以把循环转换为 Loop Simplify 形式、Loop Closed SSA 形式。

其中，Loop Simplify 形式的循环引入了以下限制：

- 循环只有一个 `Entering` 基本块，称为 `Preheader` 基本块。
- 循环只有一个 `Latch` 基本块。
- `Exit` 基本块没有来自循环外其他基本块的入边。

Loop Simplify 形式的循环结构如图2.2所示。

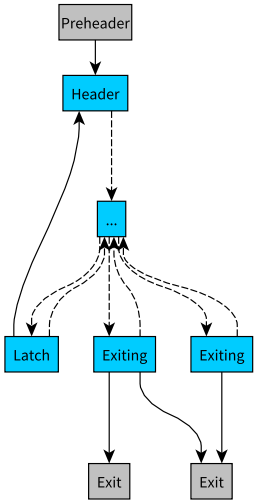

图 2.2　LoopSimplify 形式的循环结构

而 Loop Closed SSA 形式则确保所有在循环内定义的值只在循环内使用。对于

所有原本在循环内定义、在循环外使用的值，将会在各个 `Exit` 基本块插入相应的 Φ 函数，捕获这些值。

Loop Simplify、Loop Closed SSA 两种形式大大方便了对循环进行分析和变换。我们的循环转递归方法 New-Loop2Recursion 就处理经过转换同时满足 Loop Simplify、Loop Closed SSA 两种形式的循环，从而用一种统一的方式处理源代码中形态各异的循环。

3 循环转递归的基本流程及优化

3.1 循环转递归的基本流程

在理论上，由于循环与尾递归的等效性[43, 44]，我们可以将任意循环转化为递归函数；而在实践中，将循环转递归设计为一个可以被 LLVM 的代码优化器 `opt` 运行的 LLVM Pass，在 LLVM IR 的层面上工作，对源代码编译生成的 LLVM IR 进行变换，可以非常方便地将形形色色的循环转化为递归函数。

- 首先，作为众多编程语言的编译产物，LLVM IR 使得循环转递归适用于多门语言。
- 其次，通过运行相关的前置 pass，LLVM 的代码优化器 `opt` 可以将 LLVM IR 中满足一定约束条件的控制流图识别为循环，并将循环转化为 Loop Simplify、Loop Closed SSA 的标准形式形式，这使得我们只需关注如何通过修改控制流图完成循环转递归，而无需关注编程语言源代码中纷繁芜杂的循环类型（for/while/do-while 等）和循环内的控制语句（break/continue/return 等）。
- 最后，相较于汇编语言，LLVM IR 与平台无关，这使得循环转递归适用于各种不同的硬件平台。

对于一个循环而言，转换过程可以分为三部分：获取与循环相关的信息、生成递归函数、将原来的循环替换为调用递归函数。

3.1.1 获取与循环相关的信息

循环转递归的第一步是获取与循环相关的信息，包括循环外定义循环内使用的局部变量、循环内定义循环外使用的局部变量等。

3.1.1.1 获取循环外定义循环内使用的局部变量

一个循环通常使用在循环外定义的一些局部变量。将循环转换为递归函数之后，这些变量在递归函数中不可被访问。因此，我们需要找到所有这样的局部变量，将它们作为参数传递给递归函数，以使递归函数可以访问它们。

在静态单赋值形式下，这些局部变量可以分为两类：在循环的每一次迭代被更新的可变参数（如循环变量），以及在整个循环的过程中都不被修改的不变参数。这两种不同类型的参数可以通过两种不同方式获得。

可变参数在循环的第一次迭代时被赋予一个初始值，而在循环的每一次后续迭代都被赋予一个新值。在 LLVM IR 中，由于从循环外进入循环到达 `Header` 基本块，从循环的一次迭代进入下一次迭代也从循环的 `Latch` 基本块到达 `Header` 基本块，因此，我们可以通过 `Header` 基本块的 Φ 函数获取可变参数。通过 Φ 函数，我们可以确定这些可变参数第一次调用递归函数的初始值，以及后续调用递归函数的新值。

而对于不变参数，我们可以遍历循环内各指令（`Header` 基本块的 Φ 函数除外），并检查它们的操作数。若某一个操作数在循环外定义，则该操作数为不变参数。

3.1.1.2 获取循环内定义循环外使用的局部变量

与一些局部变量在循环外定义、在循环内使用相对应，一些局部变量也可能在循环内定义、在循环外使用。为了能在将循环转换为递归函数之后，在递归函数外使用这些局部变量，我们可以将这些局部变量作为递归函数的返回值。

LLVM 可以将循环转换为 Loop Closed SSA 形式。在该形式下，对于所有在循环内定义、在循环外使用的值，将会在各个 `Exit` 基本块插入相应的 Φ 函数，捕获这些值。因此，我们可以先将循环转换为该形式，然后遍历 `Exit` 基本块的 Φ 函数，确定递归函数有哪些返回值。而在将循环转换为递归函数之后，我们要将 Exit 基本块的 Φ 函数替换为递归函数的这些返回值，使调用递归函数后的代码能正确使用这些值。

3.1.2 从循环创建递归函数

在获取了与循环相关的信息后，可以从循环创建递归函数。

3.1.2.1 递归函数的名称

考虑到在待处理的源代码中，一个函数可能包含若干个循环结构，同时循环结构可以彼此嵌套，我们可以通过如下的方案命名生成的递归函数：

- 若函数 `f` 内有 m 个循环，则第 i 个循环对应的递归函数命名为 `f$i`。
- 若函数 `f` 内第第 i 个循环有 n 个嵌套循环，则第 j 个嵌套循环对应的递归函数命名为 `f$i$j`。

3.1.2.2 递归函数的结构

在原来的循环中，通过循环内的 Latch 基本块回到 Header 基本块，进入循环的下一次迭代；通过循环内的 Exiting 基本块到达某一个 Exit 基本块，退出循环，如图3.1所示。

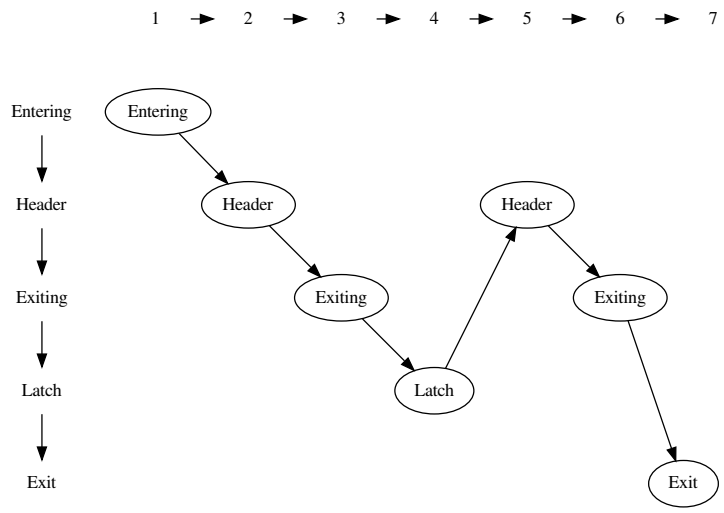

图 3.1 原来的循环结构案例

而在生成的递归函数中，应通过 Latch 基本块进入递归函数的下一次调用；而应通过 Exiting 基本块，一级级向上返回，完成所有的递归调用；在此之后，需要到达原应到达的 Exit 基本块。为了实现这样的效果，在递归函数内部，除了包含原循环的所有基本块外，我们可以插入如下的基本块：

- 插入一个 RecursivelyCallFunction 基本块，在该基本块内递归调用函数，然后返回。将 Latch 基本块指向 Header 基本块的出边改为指向它。
- 插入与原来各个 Exit 基本块对应的一系列 ReturnFromRecursion 基本块，在这些基本块内写返回信息结构体，并返回。将 Exiting 基本块指向 Exit 基本块的出边改为指向相应的 ReturnFromRecursion 基本块。

生成的递归函数的结构如下图3.2所示：

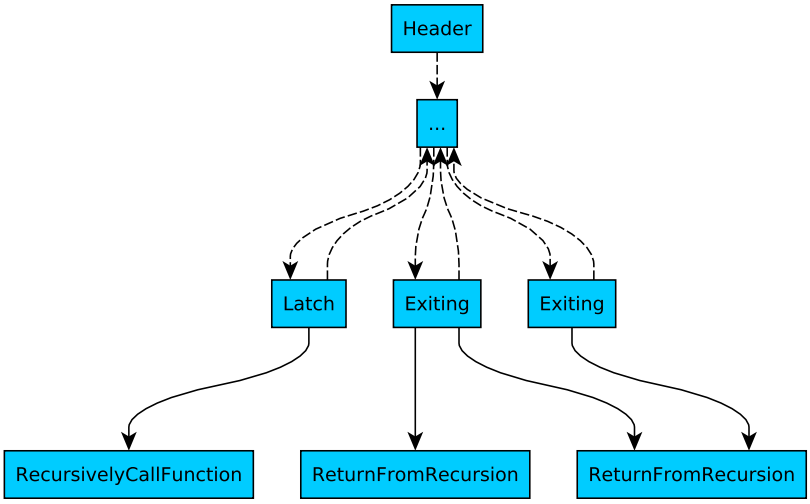

图 3.2　递归函数的结构

由于不论从递归函数的哪个基本块返回都会回到相同的调用函数的基本块，为了实现此时可以到达原应到达的 `Exit` 基本块，我们需要给所有的 `Exit` 基本块编号，将原应到达的 `Exit` 基本块的编号加入递归函数的返回值中，并在从递归函数获得该返回值后，根据该返回值跳转至相应的 `Exit` 基本块。

3.1.3　调用递归函数

完成了递归函数的创建后，还要从递归函数外调用递归函数，完成原来循环完成的功能。

在 Loop Simplify 形式下，循环外有唯一的 `Preheader` 基本块作为循环的唯一入口，且有一个或若干个 `Exit` 基本块作为循环的出口。同时，`Preheader` 基本块只有循环的 `Header` 基本块一个后继；每个 `Exit` 基本块只有循环的一个 `Exiting` 基本块作为前驱。从循环外进入循环、从循环的一次迭代进入下一次迭代、从循环退出，都通过跳转完成。

而生成递归函数之后，则通过函数调用从递归函数外面开始递归调用和进入下一次递归调用，并通过逐级返回从递归调用中退出。开始递归调用时，需要初始化不可变参数、返回值结构体并将可变参数以函数参数的形式传递给结构体；进入下一次递归调用时，需要继续以函数参数的形式传递可变参数；而在从递归调用中退出后，需要从返回值结构体中获取返回值，并在有多个 `Exit` 基本块时，根据返回的 `Exit` 编号，跳转至相应的 `Exit` 基本块。

因此，为了有效调用递归函数，我们需要做如下的调整：

- 在调用递归函数的函数中,插入 InitializeContext、CallRecursiveFunction 两个基本块。
- 将 Preheader 的后继设置为 InitializeContext。
- 在 InitializeContext 中创建返回值结构体、初始化存储在静态区的不变参数，并跳转至 CallRecursiveFunction。
- 在 CallRecursiveFunction 中调用递归函数。

接下来，如果有多个 Exit 基本块：

- 插入一个 BranchToExit 基本块。
- 将 CallRecursiveFunction 的后继设为 BranchToExit。
- 在 BranchToExit 中根据返回值结构体中的 Exit 基本块编号，跳转至相应的 Exit 基本块。

有多个 Exit 基本块的情况下，调用递归函数如图3.3所示：

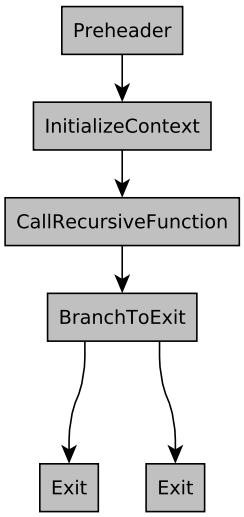

图 3.3　调用递归函数

如果只有一个 Exit 基本块，则直接将 CallRecursiveFunction 的后继设为那个 Exit 基本块。

最后，在各 Exit 基本块中，从返回值结构体中获取需要的返回值，替换原本完成从循环获取返回值的 Φ 函数。

3.2　优化措施

在循环转递归基本流程的基础上，我们可以从不变参数的传递方式、返回值的返回方式和嵌套循环的转化顺序三方面采取优化措施，减小生成的递归函数的

栈帧，减少生成的递归函数的指令数，从而减小出现栈溢出异常的概率，提升程序运行的性能。

3.2.1 优化不变参数的传递方式

由于不变参数在整个循环的过程中都不被修改，因此每次调用循环转化生成的递归函数时，这些参数的值都是相同的，这为我们提供了优化空间。

原版 Loop2Recursion 将可变参数和不变参数都实现为递归函数的参数。而我们在 New-Loop2Recursion 中，在第一次调用递归函数之前，将不变参数存储在静态区，而非作为函数的参数。如此以来，可以缩小递归函数的栈大小，而仍然可以在生成的递归函数中直接访问不变参数。

3.2.2 优化返回值的返回方式

对于递归函数的返回值，原版 Loop2Recursion 在只有一个返回值时，从最后一次调用递归函数开始，逐级返回该值。在有多个返回值的情况下，定制一个包含所有返回值的结构体，在调用递归函数之前为这样的结构体分配空间，而将结构体指针作为函数的一个参数，在从递归函数返回时通过该指针填充结构体。

而我们在 New-Loop2Recursion 中，在只有一个返回值时，也使用用返回值结构体；同时，返回值结构体的指针作为不变参数，在第一次调用递归函数之前存储在静态区，而非作为函数的参数。如此以来，消除了返回值对递归函数栈帧大小的影响。同时，虽然通过该指针填充结构体指令较多，但该操作只需在最后一次调用递归函数时完成一次，相较于逐级返回递归函数的返回值，指令要少很多。

3.2.3 优化嵌套循环的转化顺序

一段代码中的循环往往不止一个，一个循环内还可能有嵌套循环。对于嵌套循环，原版 Loop2Recursion 按照从内到外的顺序转换各级循环。而我们在 New-Loop2Recursion 中按照从外到内的顺序转换；对于每一级循环转换生成的递归函数，我们将再次检测其中是否包含循环；若包含，则会将它们转换为新的递归函数。

相较于从内到外的转换顺序，从外到内的转换顺序可以识别外层和内层循环共享的不变参数，从而可以在内层循环复用外层循环的不变参数；在我们不变参数的传递方式下，这样可以减小不变参数占用的静态区空间，减少与传参相关的指令数。

3.3 实验研究

我们基于 LLVM 13.0.1，以 LLVM Pass 的形式实现了 New-Loop2Recursion。为了评估我们的优化措施，我们将 New-Loop2Recursion 与原版 Loop2Recursion 进行对比，具体的实验内容如下所示：

- 评估由栈区内存的使用量、指令数所反映的损耗均衡的高效性，这是我们改进的重点。
- 评估损耗均衡的有效性，这是我们需要兼顾的目标。由于相变存储的寿命由最热内存地址的损耗决定，我们可以通过栈上最热内存地址的写次数，量化损耗均衡的有效性[39]。

在实验中，我们使用来源于 MiBench 的源代码[40]。我们将使用 `clang` 前端，以`-O2` 的优化级别编译所有源代码生成 LLVM IR，作为 New-Loop2Recursion 加工处理的对象；在完成循环转递归后，我们将从经过处理的 LLVM IR 生成最终的可执行文件。

为了获取我们所需的实验数据，我们基于 Intel Pin[45] 设计了一个对生成的可执行文件进行插桩的工具。该工具在运行可执行文件期间，将所有的函数调用与此时的栈指针、函数返回与此时的栈指针，以及写操作和被写的内存地址写入日志文件。通过分析这样的日志文件，我们可以还原程序运行期间栈帧的情况，得到栈内存的使用量、栈上最热内存地址的写次数、一个内存地址上方栈帧数的最大值和所有的栈帧中最热内存地址的写次数。同时，我们也基于 Intel Pin 设计了统计程序运行期间执行的指令总数的工具。

3.3.1 损耗均衡的高效性

3.3.1.1 栈区内存的使用量

原版 Loop2Recursion 和我们的 New-Loop2Recursion 生成的可执行文件在运行期间栈区内存的使用量如表3.1所示。可以看到，除 `pbmsrch_small` 持平外，对所有的测试用例而言，New-Loop2Recursion 生成的可执行文件在运行期间栈区内存的使用量均发生了非常明显的下降，其中平均下降程度为-44.68% 。

表 3.1　Loop2Recursion 和 New-Loop2Recursion 栈区内存的使用量表

测试用例	Loop2Recursion	New-Loop2Recursion	New-Loop2Recursion 增长
average	8.13e+06	4.49e+06	-44.68%
basicmath_small	4.98e+04	3.38e+04	-32.16%
bf	1.14e+06	2.62e+05	-77.10%
bitcnts	6.00e+06	4.80e+06	-19.99%
crc_32	8.76e+07	4.38e+07	-50.00%
dijkstra_small	7.28e+04	5.48e+04	-24.79%
fft	4.59e+05	1.97e+05	-57.07%
patricia	1.05e+06	5.26e+05	-49.85%
pbmsrch_small	1.08e+04	1.08e+04	0.15%
qsort_small	8.32e+06	8.00e+06	-3.85%
rawcaudio	2.30e+05	1.87e+05	-18.68%
rawdaudio	1.82e+05	1.55e+05	-14.81%
sha	9.02e+04	1.79e+04	-80.19%
susan	4.17e+05	3.81e+05	-8.55%

3.3.1.2　指令数

运行原版 Loop2Recursion 和 New-Loop2Recursion 生成的可执行文件时，执行的指令数如表3.2所示。

表 3.2　Loop2Recursion 和 New-Loop2Recursion 指令数表

测试用例	Loop2Recursion	New-Loop2Recursion	New-Loop2Recursion 增长
average	6.56e+07	5.89e+07	-10.21%
basicmath_small	5.01e+07	4.97e+07	-0.89%
bf	7.85e+07	5.62e+07	-28.43%
bitcnts	1.15e+08	1.04e+08	-10.26%
crc_32	6.17e+07	5.07e+07	-17.76%
dijkstra_small	1.30e+08	1.17e+08	-10.54%
fft	3.11e+07	3.05e+07	-1.94%
patricia	7.33e+07	7.24e+07	-1.23%
pbmsrch_small	1.40e+05	1.44e+05	3.29%
qsort_small	1.48e+07	1.46e+07	-0.95%
rawcaudio	1.02e+08	1.01e+08	-1.35%
rawdaudio	8.19e+07	7.98e+07	-2.51%
sha	2.46e+07	2.26e+07	-8.31%
susan	8.94e+07	6.87e+07	-23.15%

可以看到，除 `pbmsrch_small` 略微增长外，对所有的测试用例而言，New-

Loop2Recursion 生成的可执行文件在运行期间执行的指令数发生了明显的下降，其中平均下降程度为-10.21% 。

总的来说，我们不变参数的传参方式、递归函数返回值的传递方式的两点优化，通过减少传参和返回相关指令，有效地实现了减少指令数的预期目标。而 pbmsrch_small 指令数略微增长，则主要是因为其中的函数 main$0$3 需要访问 <2 x i64> 类型、被转化为全局变量的不可变参数 @"main$0$3"。

该函数内反复出现若干次如下所示的

```
%i2 = load <2 x i64>, <2 x i64>* @"main$0$3$7", align 16
store <2 x i64> %i2, <2 x i64>* %i1, align 16, !tbaa !1
```

在生成的可执行文件中，对应的汇编指令如下：

```
movdqa 0x6c1ad0,%xmm0
movdqa %xmm0,0x6c1250(,%rdi,8)
```

可以看到，LLVM IR 将 <2 x i64>* 类型的全局变量 @"main$0$3$7" 存储在%i1 地址处的操作被翻译为了先将全局变量的内容移动至寄存器中，再将寄存器的内容移动至指定地址处。这是因为 x86 平台完成双浮点数移动的 movdqa 指令只能完成寄存器——寄存器、内存——寄存器或寄存器——内存的数据转移，而无法实现内存——内存的数据转移；相比于将作为不变值的两个双精度浮点数作为函数参数存储在寄存器中，将该不变值作为全局变量确实会在完成该功能时多一条指令。事实上，虽然因为这个原因该函数的指令数增幅较大，但 pbmsrch_small 总指令数增长幅度相当有限；而其他各测试用例的指令总数均下降。这是因为大部分全局变量为单个整型或浮点型变量，而移动单个整型或浮点型变量的 mov 指令可以直接实现内存——内存的数据转移，相比于将不变量作为函数参数，指令数不会增加，反倒会因为没有将不变量传递给函数而减少；同时，即便存在较大全局变量在内存中的转移，我们的优化通过减少传参和返回相关的指令，也可以很好地弥补因此增加的指令开销。

3.3.2 损耗均衡的有效性

3.3.2.1 栈上最热内存地址的写次数

首先，我们对比原版 Loop2Recursion 和 New-Loop2Recursion 生成的可执行文件栈上最热内存地址的写次数，如表3.3所示。

表 3.3　Loop2Recursion 和 New-Loop2Recursion 栈上最热内存地址的写次数表

测试用例	Loop2Recursion	New-Loop2Recursion	New-Loop2Recursion 增长
average	1.37e+05	1.38e+05	1.00%
basicmath_small	1.84e+04	3.31e+04	80.22%
bf	3.12e+05	3.12e+05	0.00%
bitcnts	6.60e+01	6.90e+01	4.55%
crc_32	1.37e+06	1.37e+06	0.00%
dijkstra_small	1.02e+04	1.00e+04	-1.78%
fft	3.32e+02	4.30e+02	29.52%
patricia	2.18e+04	2.18e+04	0.00%
pbmsrch_small	4.70e+01	4.50e+01	-4.26%
qsort_small	1.19e+04	1.19e+04	0.07%
rawcaudio	6.66e+02	7.25e+02	8.86%
rawdaudio	6.87e+02	7.33e+02	6.70%
sha	2.44e+04	2.44e+04	0.00%
susan	6.25e+03	9.19e+03	47.12%

虽然对几乎所有的测试用例而言，栈上最热内存地址的写次数均出现了增长，其中测试用例 `basicmath_small`、`fft`、`susan` 还出现了较大增长，但总体上增幅不大，平均仅有 1.00% 。这表明，New-Loop2Recursion 在提升损耗均衡的高效性的同时，在整体上很好地控制了对损耗均衡有效性的影响。

另一方面，由于调用迭代次数较多的循环转化生成的递归函数仍然容易因为递归过深而引发栈溢出异常，因此我们在实际中使用循环转递归时，需要进一步优化栈内存使用。后续实验结果表明，我们限制较大的递归深度，栈上最热内存地址的写次数要远小于限制较小的递归深度，而栈区内存使用量则只略微高于限制较小的递归深度。而在原版 Loop2Recursion 中限制递归深度，栈区内存使用量与栈上最热内存地址的写次数之间存在明显的负相关性，进一步优化栈内存使用意味着栈上最热内存地址的写次数将迅猛增长[39]。我们为了进一步优化栈内存使用而限制较大的递归深度，栈上最热内存地址的写次数反而将具备优势。

由于栈上最热内存地址的写次数受栈帧在栈区内存的分布、写操作在栈帧的分布共同影响，为了探究栈上最热内存地址的写次数出现增长的原因，我们将对二者进行度量。对于前者，我们度量栈上一个内存地址上方栈帧数的最大值；而对于后者，我们度量所有的栈帧中最热内存地址的写次数。

3.3.2.2 栈帧中最热内存地址的写次数

实验所测得原版 Loop2Recursion 和 New-Loop2Recursion 生成的可执行文件栈帧中最热内存地址的写次数，如下表3.4所示。我们可以看到，对所有的测试用例而言，New-Loop2Recursion 栈帧中最热内存地址的写次数，相较于原版 Loop2Recursion，均持平。由此可见，New-Loop2Recursion 未因为影响写操作在栈帧内的分布而导致栈上最热内存地址的写次数出现增长。

表 3.4 Loop2Recursion 和 New-Loop2Recursion 栈帧中最热内存地址的写次数表

测试用例	Loop2Recursion	New-Loop2Recursion	New-Loop2Recursion 增长
average	1.36e+05	1.36e+05	0.00%
basicmath_small	1.80e+04	1.80e+04	0.00%
bf	3.12e+05	3.12e+05	0.00%
bitcnts	1.60e+01	1.60e+01	0.00%
crc_32	1.37e+06	1.37e+06	0.00%
dijkstra_small	1.00e+04	1.00e+04	0.00%
fft	3.80e+01	3.80e+01	0.00%
patricia	2.18e+04	2.18e+04	0.00%
pbmsrch_small	1.20e+01	1.20e+01	0.00%
qsort_small	1.00e+04	1.00e+04	0.00%
rawcaudio	7.00e+00	7.00e+00	0.00%
rawdaudio	7.00e+00	7.00e+00	0.00%
sha	2.44e+04	2.44e+04	0.00%
susan	7.00e+00	7.00e+00	0.00%

3.3.2.3 栈上一个内存地址上方栈帧数的最大值

而栈上一个内存地址上方栈帧数的最大值，如表3.5所示。可以看到，相对于原版 Loop2Recursion，New-Loop2Recursion 栈上一个内存地址上方栈帧数的最大值出现了较大幅度的增长，平均增幅为 28.30% 。

表 3.5　Loop2Recursion 和 New-Loop2Recursion 栈上一个内存地址上方栈帧数的最大值表

测试用例	Loop2Recursion	New-Loop2Recursion	New-Loop2Recursion 增长
average	5372.23	6892.54	28.30%
basicmath_small	41164.00	50972.00	23.83%
bf	154.00	347.00	125.32%
bitcnts	54.00	59.00	9.26%
crc_32	16.00	16.00	0.00%
dijkstra_small	7623.00	9190.00	20.56%
fft	354.00	484.00	36.72%
patricia	121.00	216.00	78.51%
pbmsrch_small	48.00	46.00	-4.17%
qsort_small	11952.00	11981.00	0.24%
rawcaudio	688.00	695.00	1.02%
rawdaudio	688.00	695.00	1.02%
sha	683.00	5713.00	736.46%
susan	6294.00	9189.00	46.00%

一个内存地址上方栈帧数的最大值之所以会增长，与不同栈帧体积缩小的程度不同有关。程序运行期间，每当遇到一个返回指令之前，程序可能进行了若干次连续的函数调用，并为这些函数分配了栈帧，这些函数的栈帧在内存中是连续的。由于程序的执行过程不变，而不同栈帧体积缩小的程度不同，这些连续的栈帧在内存中的分布变得更加紧凑，相邻两次分配的连续的栈帧重合程度增大。因此，一个内存地址上方栈帧数的最大值将会增长。

但即便在一个内存地址上方栈帧数的最大值出现这样的增幅之下，栈上最热内存地址的写次数总体上增幅也相当有限（平均增幅仅为 1.00% ），且各测试用例一个内存地址上方栈帧数的最大值的增幅、栈上最热内存地址的写次数的增幅二者之间相关系数 r 为-0.1477 ，可以看到二者之间的相关性较弱。

除了保持栈帧中最热内存地址的写次数不变值外，我们的优化减小了栈帧的体积，与返回值和不变参数函数相关、栈帧体积缩小的部分的写操作也得到了消除。因此，虽然一个内存地址上方栈帧数的最大值明显增长，由栈上最热内存地址的写次数所反映的损耗均衡有效性在整体上仍未受到很大的影响。

3.3.3 总结

相比原版 Loop2Recursion，New-Loop2Recursion 生成的可执行文件在运行期间栈区内存的使用量、执行的指令数均发生了明显下降，平均降幅分别为-44.68%、-10.21%，由此证明我们的优化措施有效地减小了生成的递归函数的栈帧，减少了生成的递归函数的指令数，提升了程序运行的性能，减小了出现栈溢出异常的概率。虽然栈上最热内存地址的写次数发生了增长，但总体上增幅相当有限（1.00%）。由此可见，我们的优化措施在提升程序运行的性能的同时，由栈上最热内存地址的写次数所反映的损耗均衡有效性在整体上仍未受到很大的影响。

4 对栈内存使用的进一步优化

相比于循环，递归函数可以有效地实现损耗均衡；但与此同时，使用递归将极大地提升栈内存的使用量。虽然我们的优化措施减小了生成的递归函数的栈帧，但如果不加限制，较深的递归仍然容易将程序的栈空间消耗殆尽，引发栈溢出异常。针对这种情况，可以通过限制递归深度、在递归中包含循环，进一步减少调用递归函数使用的栈空间。

4.1 限制递归深度

Li 等人[39] 提出，可以允许程序员指定一个递归深度的上界，当递归深度达到该上界时，从递归函数返回，并循环调用递归函数，直到递归的总次数等于原循环的迭代次数。我们对这种进一步优化栈内存使用的策略进行了复现，具体的实现方法如下。

4.1.1 在递归的过程中记录当前递归深度

这种限制递归深度的策略需要我们在递归的过程中记录当前的递归深度。为此，我们在递归函数的参数列表中加入了一个整型的递归深度计数器参数。每一次从递归函数外调用递归函数时，将该参数的值设置为 0。

而在递归函数内部，我们可以进行如下的调整：

- 加入 `IncrementDepthCounter`、`SaveRecursionState` 两个基本块。
- 将不限制递归深度时各 `Latch` 到达 `RecursivelyCallFunction` 的出边，调整为到达 `IncrementDepthCounter`。
- 在 `IncrementDepthCounter` 中，递增参数列表中的递归深度计数器，并将递增后的递归深度计数器与递归深度限制进行比较。根据比较结果，跳转至完成下一次递归调用的 `RecursivelyCallFunction` 基本块，或者保存递归状态并返回的 `SaveRecursionState` 基本块。

在限制递归深度的情况下递归函数内各基本块之间的关系如图4.1所示：

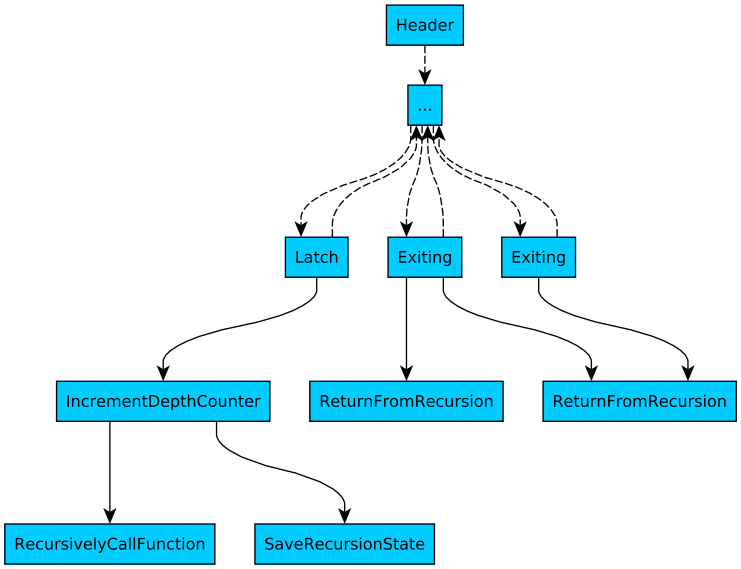

图 4.1　在限制递归深度的情况下递归函数内各基本块之间的关系

4.1.2　循环调用递归函数，并在该过程中保存递归状态

在循环调用递归函数的过程中，我们需要保存在循环的每一次迭代被更新的可变参数、表示所有的递归调用是否都已经完成的状态信息。为此，我们可以定制与原循环配套的“递归状态结构体”，存储这些状态信息。

第一次在递归函数外调用递归函数之前，我们可以在 `InitializeContext` 基本块为该结构体分配空间。

为了实现可以在该结构体中存储状态信息，我们可以将该结构体的指针作为生成的递归函数的一个参数。在递归函数内的 `SaveRecursionState` 基本块，通过指针向递归状态结构体写入下一次调用递归函数时可变参数的值和代表递归尚未完成的布尔值“假”，然后从递归函数返回。而在完成最后一次递归调用时到达的 `ReturnFromRecursion` 基本块，除了写返回信息结构体外，还向递归状态结构体写入代表所有递归已完成的布尔值“真”，然后从递归函数返回。

而为了实现在递归函数外循环调用递归函数并获取下一次调用递归函数需要提供的可变参数的值，我们可以在递归函数外进行如下的调整：

- 加入 `ExtractNextValues` 基本块，完成从递归状态结构体提取下一次调用递归函数时可变参数的值，并跳转至 `CallRecursiveFunction` 基本块。
- 在 `CallRecursiveFunction` 末尾从递归状态结构体提取代表所有递归是否都已完成的布尔值，根据布尔值的真假决定跳转至 `CallRecursiveFunction` 原先的后继基本块，还是 `ExtractNextValues` 基本块。

- 在 CallRecursiveFunction 开头加入根据控制流来自 InitializeContext 还是 ExtractNextValues 选择可变参数的值的 Φ 函数。

在限制递归深度的情况下调用递归函数如图4.2所示。

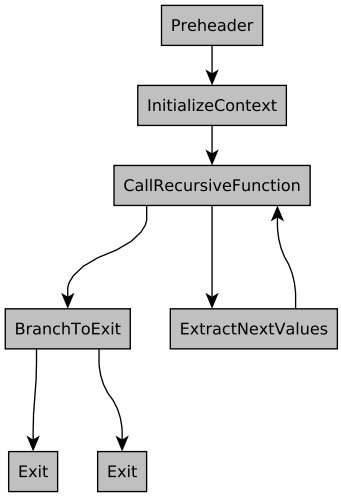

图 4.2　在限制递归深度的情况下调用递归函数

4.2　在递归中包含循环

除了限制递归深度，还可以通过在生成的递归函数中保留一部分原有的循环，进一步优化栈内存使用。与限制递归深度的思路类似，可以允许程序员指定一个循环迭代次数的上界，当循环迭代次数达到该上界时，完成递归函数的下一次递归调用。为此，我们可以通过如下的方式，修改递归函数，在递归函数中保留一部分循环：

- 加入一个基本块 PreheaderInRecursion，作为递归函数的第一个基本块。将 Header 基本块作为 PreheaderInRecursion 的后继基本块。
- 在 Header 基本块的开始，加入一个与循环计数器相对应的 Φ 函数，如果控制流来自 PreheaderInRecursion 选择常数 0。
- 加入一系列与各 Latch 基本块对应的 IncrementLoopCounter 基本块。将各 Latch 基本块到达 RecursivelyCallFunction 的出边，改为到达与该 Latch 对应的 IncrementLoopCounter 基本块。
- 在每个 IncrementLoopCounter 基本块内，递增循环计数器。根据递增后的循环计数器是否小于限定值，选择跳转至 Header 或 RecursivelyCallFunction。
- 修改与循环计数器相对应的 Φ 函数，如果控制流来自 IncrementLoopCounter，选择递增过的循环计数器。

在递归中包含循环的情况下递归函数内各基本块之间的关系如图4.3所示：

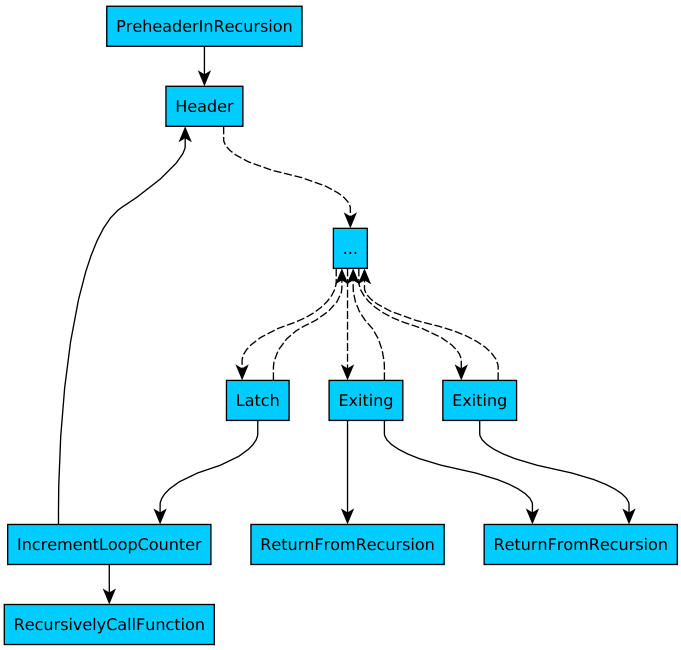

图 4.3　在递归中包含循环的情况下递归函数内各基本块之间的关系

4.3　实验研究

我们在 New-Loop2Recursion 中实现了限制递归深度、在递归中包含循环两种进一步优化栈内存使用的策略。通过实验，我们将探究限制不同的递归深度、在递归函数中保留不同的循环迭代次数的影响，并探究限制递归深度、在递归中包含循环二者中何者为更优的进一步优化栈内存使用的策略。

4.3.1　限制递归深度

4.3.1.1　损耗均衡的高效性

（1）栈区内存的使用量

限制不同的递归深度时，栈区内存的使用量如图4.4、表4.1所示。

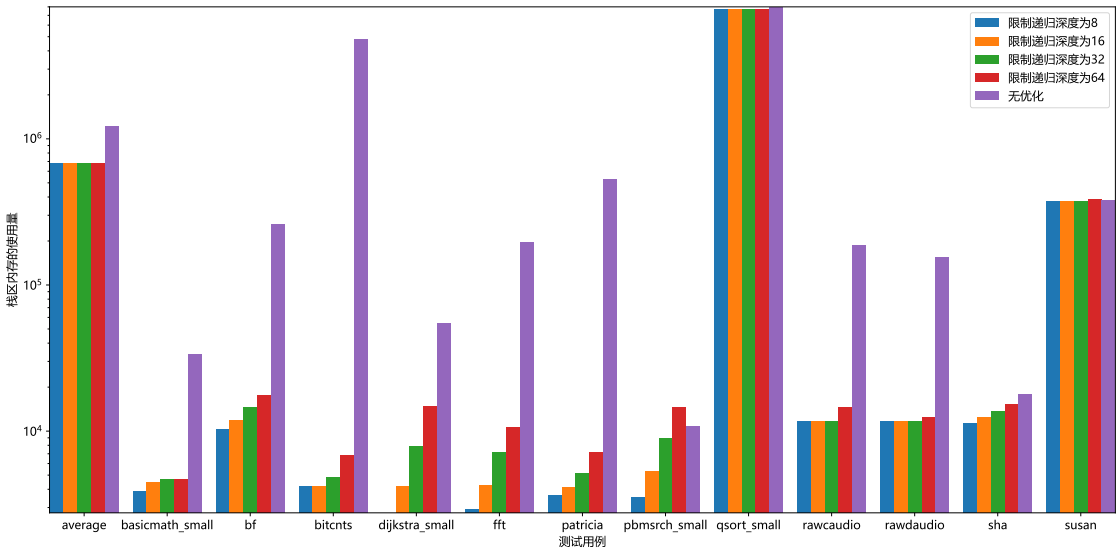

图 4.4　限制不同的递归深度时栈区内存的使用量图

表 4.1　限制不同的递归深度时栈区内存的使用量表

测试用例	递归深度为 8 增长	递归深度为 16 增长	递归深度为 32 增长	递归深度为 64 增长
average	-44.50%	-44.44%	-44.31%	-44.05%
basicmath_small	-88.61%	-86.86%	-86.13%	-86.13%
bf	-96.04%	-95.51%	-94.43%	-93.31%
bitcnts	-99.91%	-99.91%	-99.90%	-99.86%
dijkstra_small	-94.96%	-92.30%	-85.52%	-72.90%
fft	-98.52%	-97.85%	-96.34%	-94.58%
patricia	-99.31%	-99.21%	-99.02%	-98.63%
pbmsrch_small	-67.65%	-51.11%	-18.02%	33.68%
qsort_small	-3.99%	-3.99%	-3.99%	-3.97%
rawcaudio	-93.77%	-93.77%	-93.77%	-92.24%
rawdaudio	-92.49%	-92.49%	-92.49%	-91.98%
sha	-37.04%	-30.59%	-23.11%	-13.97%
susan	-2.36%	-2.36%	-1.84%	0.58%

可以看到，限制递归深度对减小栈区内存使用量起到了很好的效果，即便在递归深度的限制最大时，各测试用例栈内存使用量也平均减小了-44.05% 。而随着对递归深度的限制逐渐由大到小，虽然各测试用例栈区内存的使用量均有减小趋势，但是栈区内存使用量相对不限制递归深度减小的程度总体上变化不大，其平均值仅发生极其略微的增大。由此可见，限制递归深度对栈区内存使用量的优化与递归深度的限制关系不大。

（2）**指令数**

限制不同的递归深度时，运行执行的指令数如图4.5、表4.2所示。

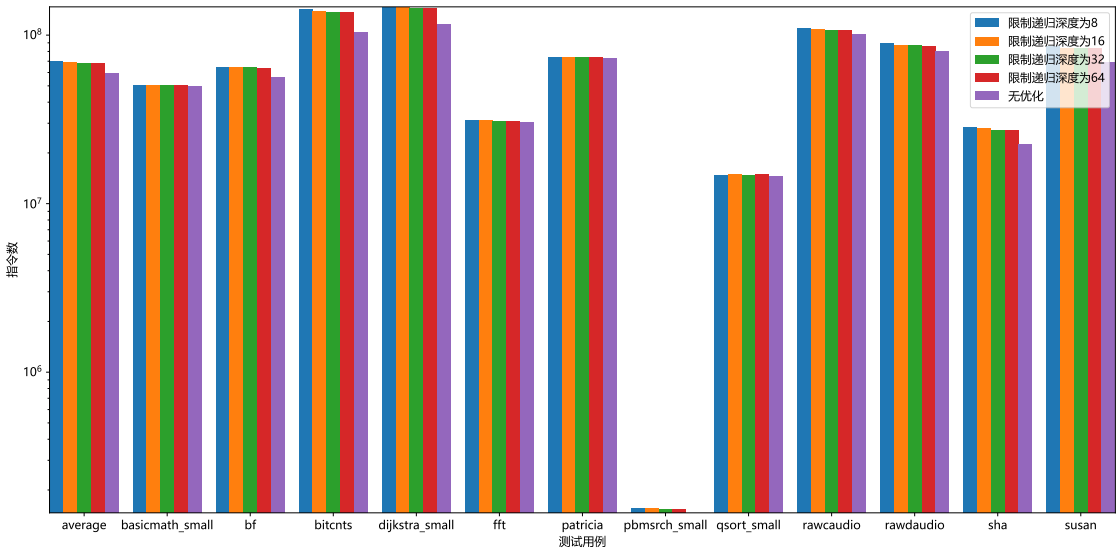

图 4.5　限制不同的递归深度时的指令数图

表 4.2　限制不同的递归深度时的指令数表

测试用例	递归深度为 8 增长	递归深度为 16 增长	递归深度为 32 增长	递归深度为 64 增长
average	16.94%	15.24%	14.52%	14.30%
basicmath_small	1.57%	1.46%	1.42%	1.42%
bf	15.45%	14.53%	14.07%	13.61%
bitcnts	36.43%	32.99%	31.24%	31.12%
dijkstra_small	26.06%	24.52%	23.75%	23.30%
fft	1.84%	1.73%	1.68%	1.66%
patricia	1.64%	1.44%	1.50%	1.50%
pbmsrch_small	5.72%	5.67%	5.66%	5.66%
qsort_small	1.21%	2.15%	1.12%	2.10%
rawcaudio	9.09%	7.62%	6.88%	6.50%
rawdaudio	11.46%	9.60%	8.67%	8.19%
sha	25.89%	23.25%	20.76%	20.75%
susan	25.52%	21.64%	21.64%	21.64%

可以看到，在限制递归深度的情况下，运行可执行文件执行的指令数有所增大。这源于调用递归函数时需要额外提供递归深度计数器的参数、每一次递归调用时需要递增并比较递归深度计数器、递归深度达到上限时需要存储循环变量并在下一次调用递归函数时获取它们。由于减小递归深度的限制会导致循环调用递归函数的次数增多，因此递归深度达到上限时存储循环变量并在下一次调用递归函数时获取它们的指令增多；但因此而导致的总指令数增长趋势总体上不明显。

4.3.1.2 损耗均衡的有效性

（1）栈上最热内存地址的写次数

限制不同的递归深度时，栈上最热内存地址的写次数如图4.6、表4.3所示。

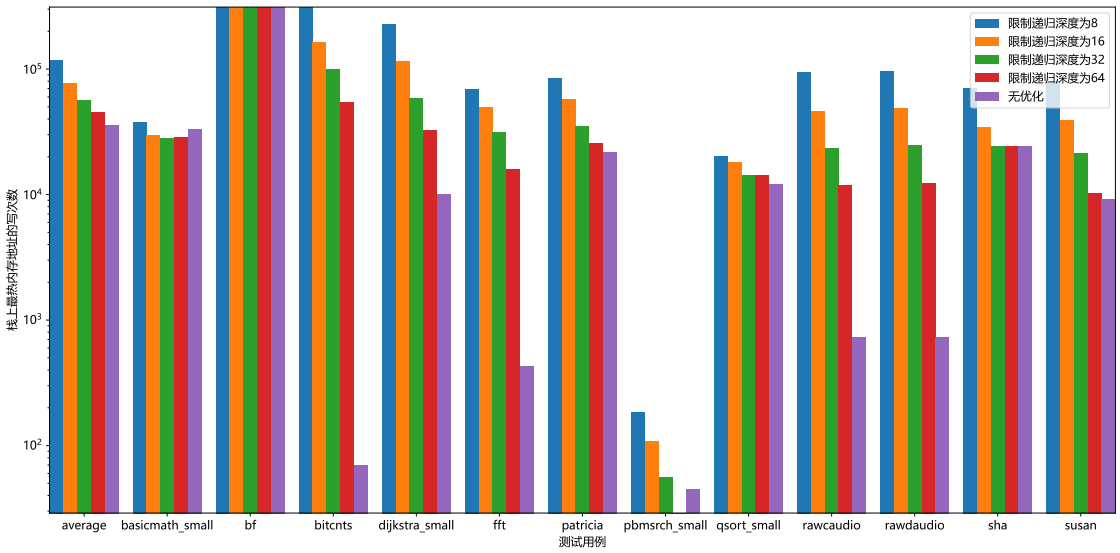

图 4.6　限制不同的递归深度时栈上最热内存地址的写次数图

表 4.3　限制不同的递归深度时栈上最热内存地址的写次数表

测试用例	递归深度为 8 增长	递归深度为 16 增长	递归深度为 32 增长	递归深度为 64 增长
average	229.34%	115.67%	57.99%	27.57%
basicmath_small	13.53%	-10.35%	-15.88%	-13.46%
bf	0.00%	0.00%	0.00%	0.00%
bitcnts	448,284.06%	237,692.75%	142,579.71%	78,189.86%
dijkstra_small	2,159.31%	1,055.29%	480.17%	225.60%
fft	15,880.93%	11,380.93%	7,105.12%	3,600.00%
patricia	287.42%	164.89%	61.29%	16.67%
pbmsrch_small	306.67%	140.00%	24.44%	-35.56%
qsort_small	67.78%	51.91%	17.93%	19.51%
rawcaudio	12,840.69%	6,280.83%	3,138.21%	1,516.83%
rawdaudio	13,038.34%	6,521.69%	3,263.44%	1,557.71%
sha	184.91%	41.44%	0.00%	0.00%
susan	764.41%	326.35%	131.67%	11.23%

在限制递归深度的情况下，由于当递归深度达到上限时，将从递归函数中逐级返回，然后再从头开始调用递归函数，递归函数的栈帧内的写操作将被重复，因此，栈上最热内存地址的写次数将增大。同时，我们可以看到，在递归深度限制减小的情况下，栈上最热内存地址的写次数快速增长。这是因为递归函数的总调用

次数不变，在递归深度限制减小的情况下，循环调用递归函数的次数将增加，从而递归函数的栈帧内的写操作数将增加。故而限制递归深度应控制递归深度上限，避免其过低，以免对损耗均衡的有效性造成较大的负面影响。

4.3.2 在递归中包含循环

4.3.2.1 损耗均衡的高效性

（1）栈区内存的使用量

在递归函数中包含不同迭代次数的循环时，栈区内存的使用量如图4.7、表4.4所示。可以看到，随着循环迭代次数的增多，栈区内存的使用量先是快速下降，后逐渐趋于稳定。

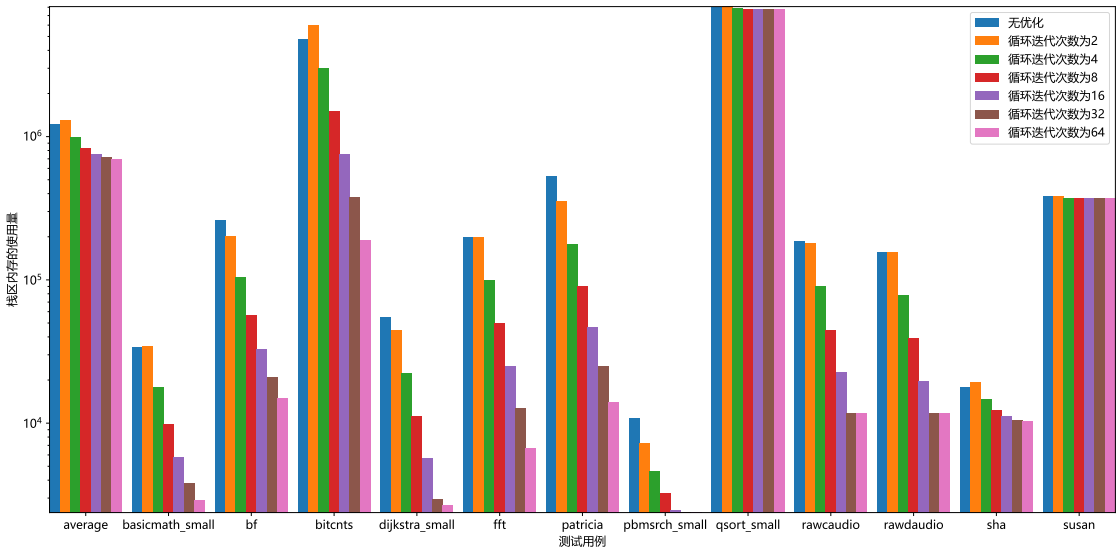

图 4.7 在递归函数中包含不同迭代次数的循环时栈区内存的使用量图

表 4.4 在递归函数中包含不同迭代次数的循环时栈区内存的使用量表

测试用例	循环次数为 2 增长	循环次数为 4 增长	循环次数为 8 增长	循环次数为 16 增长	循环次数为 32 增长	循环次数为 64 增长
average	7.00%	-18.92%	-31.84%	-38.30%	-41.52%	-43.05%
basicmath_small	1.87%	-47.25%	-70.93%	-82.86%	-88.73%	-91.48%
bf	-23.67%	-60.16%	-78.35%	-87.43%	-91.98%	-94.29%
bitcnts	25.00%	-37.50%	-68.75%	-84.37%	-92.18%	-96.08%
dijkstra_small	-19.35%	-59.56%	-79.54%	-89.54%	-94.59%	-95.11%
fft	0.00%	-49.90%	-74.82%	-87.29%	-93.52%	-96.61%
patricia	-33.14%	-66.27%	-82.83%	-91.12%	-95.26%	-97.33%
pbmsrch_small	-32.79%	-57.61%	-70.01%	-77.10%	-78.06%	-78.06%
qsort_small	1.00%	-1.50%	-2.75%	-3.37%	-3.69%	-3.84%
rawcaudio	-4.30%	-52.13%	-76.03%	-87.91%	-93.78%	-93.78%
rawdaudio	-0.03%	-50.00%	-74.97%	-87.37%	-92.50%	-92.50%
sha	8.02%	-18.41%	-30.94%	-37.12%	-40.98%	-42.23%
susan	-0.14%	-2.45%	-2.45%	-2.45%	-2.45%	-2.45%

循环迭代次数的增多导致栈区内存的使用量的机理在于其减少了递归函数的调用次数，从而减少了因递归而消耗的栈空间。而在循环迭代次数达到一定值之后，递归函数的调用次数随循环迭代次数的增加而减小的程度变得不明显，此时栈区内存的使用量逐渐达到一个下限。

（2）指令数

在递归函数中包含不同迭代次数的循环时，运行时执行的指令数如图4.8、表4.5所示。可以看到，不论循环的迭代次数为多少，执行的指令数均有所增长；增长程度随循环迭代次数的增长缓慢减小，并趋于稳定。

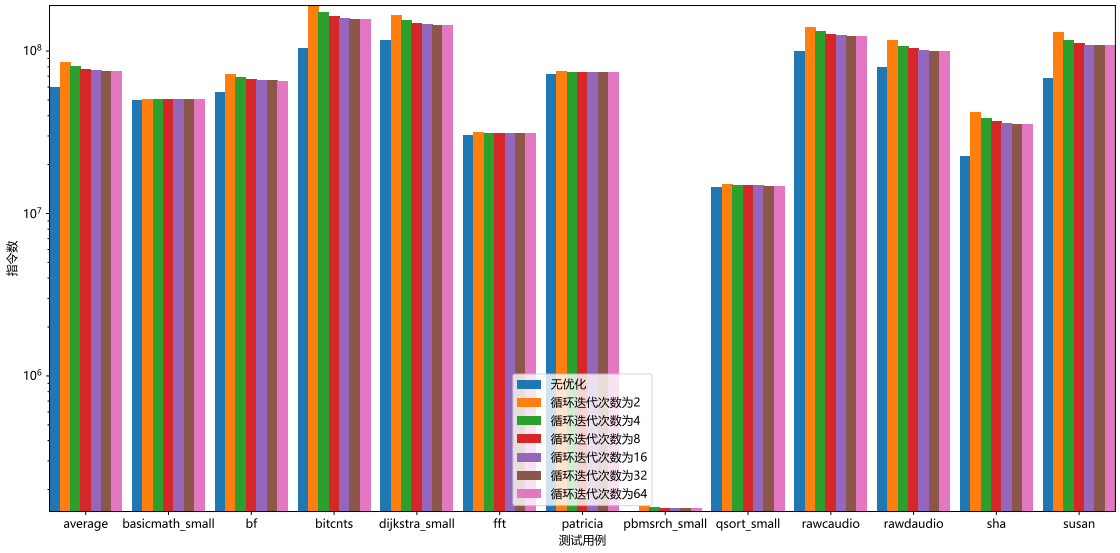

图 4.8　在递归函数中包含不同迭代次数的循环时的指令数图

表 4.5　在递归函数中包含不同迭代次数的循环时的指令数表

测试用例	循环次数为 2 增长	循环次数为 4 增长	循环次数为 8 增长	循环次数为 16 增长	循环次数为 32 增长	循环次数为 64 增长
average	44.08%	34.86%	30.24%	27.94%	26.97%	26.64%
basicmath_small	2.54%	2.05%	1.81%	1.68%	1.63%	1.63%
bf	29.14%	22.86%	19.72%	18.47%	17.84%	17.21%
bitcnts	83.94%	67.51%	58.63%	54.19%	51.93%	51.78%
dijkstra_small	41.75%	32.57%	28.09%	25.84%	24.72%	24.07%
fft	4.11%	3.06%	2.76%	2.62%	2.55%	2.52%
patricia	3.51%	2.74%	2.44%	2.11%	2.16%	2.22%
pbmsrch_small	7.51%	5.83%	5.15%	5.08%	5.06%	5.06%
qsort_small	3.37%	2.12%	2.04%	1.95%	1.07%	1.05%
rawcaudio	40.46%	31.44%	26.94%	24.70%	23.58%	23.00%
rawdaudio	45.41%	35.12%	29.97%	27.42%	26.15%	25.49%
sha	86.14%	70.14%	63.54%	60.24%	57.19%	57.18%
susan	89.38%	71.15%	62.04%	57.49%	57.48%	57.48%

在递归函数中保留循环需要在每一次调用递归函数时初始化循环计数器，并在每一次循环完成后递增并比较循环计数器。随着递归函数中循环次数的增多，调

用递归函数的次数减少，调用递归函数、初始化循环计数器的指令减少；但递增并比较循环计数器的指令不变。在这种情况下，指令总数从一个最大值开始逐渐减小。而在循环迭代次数达到一定水平之后，递归函数的调用次数随循环迭代次数的增加而减小的程度变得不明显，此时指令总数逐渐达到一个下限。

4.3.2.2 损耗均衡的有效性

（1）栈上最热内存地址的写次数

在递归函数中包含不同迭代次数的循环时，栈上最热内存地址的写次数如图4.9、表4.6所示。

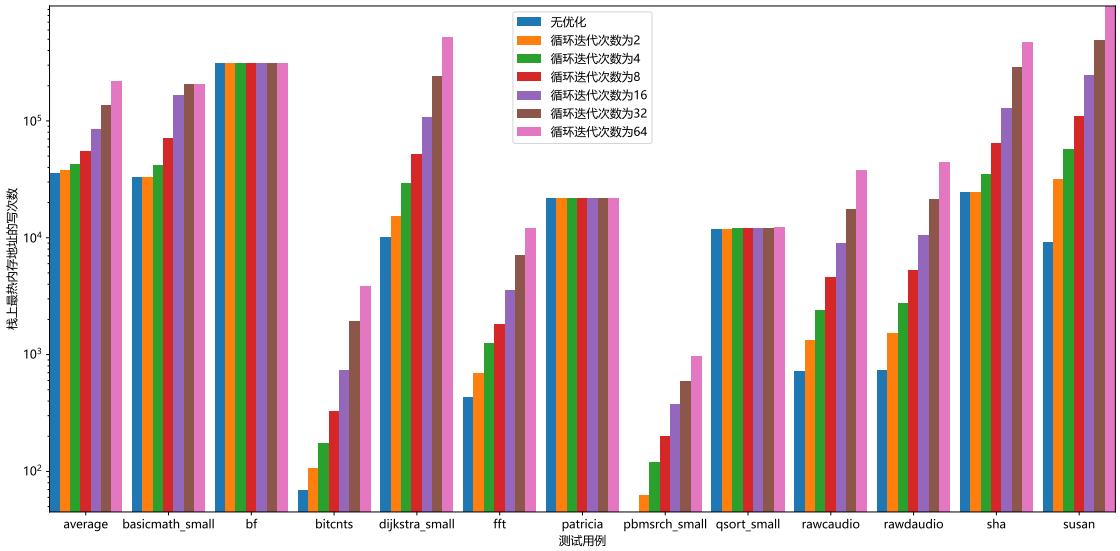

图 4.9 在递归函数中包含不同迭代次数的循环时栈上最热内存地址的写次数图

表 4.6 在递归函数中包含不同迭代次数的循环时栈上最热内存地址的写次数表

测试用例	循环次数为 2 增长	循环次数为 4 增长	循环次数为 8 增长	循环次数为 16 增长	循环次数为 32 增长	循环次数为 64 增长
average	6.95%	21.32%	54.64%	139.22%	281.96%	513.53%
basicmath_small	0.06%	25.66%	113.80%	398.50%	526.87%	521.47%
bf	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
bitcnts	55.07%	153.62%	378.26%	962.32%	2,717.39%	5,434.78%
dijkstra_small	51.85%	195.06%	422.08%	966.41%	2,289.80%	5,118.34%
fft	59.77%	187.91%	320.47%	723.72%	1,547.44%	2,678.37%
patricia	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
pbmsrch_small	37.78%	164.44%	340.00%	733.33%	1,211.11%	2,060.00%
qsort_small	-0.03%	0.05%	0.22%	0.55%	1.24%	2.57%
rawcaudio	81.24%	230.90%	530.90%	1,131.86%	2,328.69%	5,074.48%
rawdaudio	108.32%	274.76%	618.96%	1,329.33%	2,791.81%	5,921.96%
sha	0.00%	42.14%	165.85%	425.32%	1,077.64%	1,819.00%
susan	245.63%	516.78%	1,102.31%	2,572.06%	5,262.94%	10,382.66%

随着循环迭代次数的增多，栈上最热内存地址的写次数快速增长。因此，为了避免损耗均衡的有效性受到影响，在递归函数中保留循环时，需要尽可能减少的

迭代次数。

4.3.3 总结

通过上述实验，我们可以得到如下的实验结论：

- 限制递归深度可以有效地减少栈区内存的使用量，且受递归深度限制影响极小；与此同时，指令数受递归深度限制影响也较小，而栈上最热内存地址的写次数会随递归深度限制的减小而快速增大。因此，限制递归深度的最佳策略为限制较大的递归深度。
- 递归中保留循环时，随着循环迭代次数的增多，栈区内存的使用量将下降并趋于稳定；与此同时，指令数受循环迭代次数影响较小，而栈上最热内存地址的写次数会随循环迭代次数的增多而快速增大。因此，在递归函数中保留一部分循环时，需要权衡栈区内存的使用量和栈上最热内存地址的写次数。

我们可以获取这两种策略在参数取不同值时，栈区内存的使用量、栈上最热内存地址的写次数针对所有测试用例的平均数据，绘制散点图4.10。

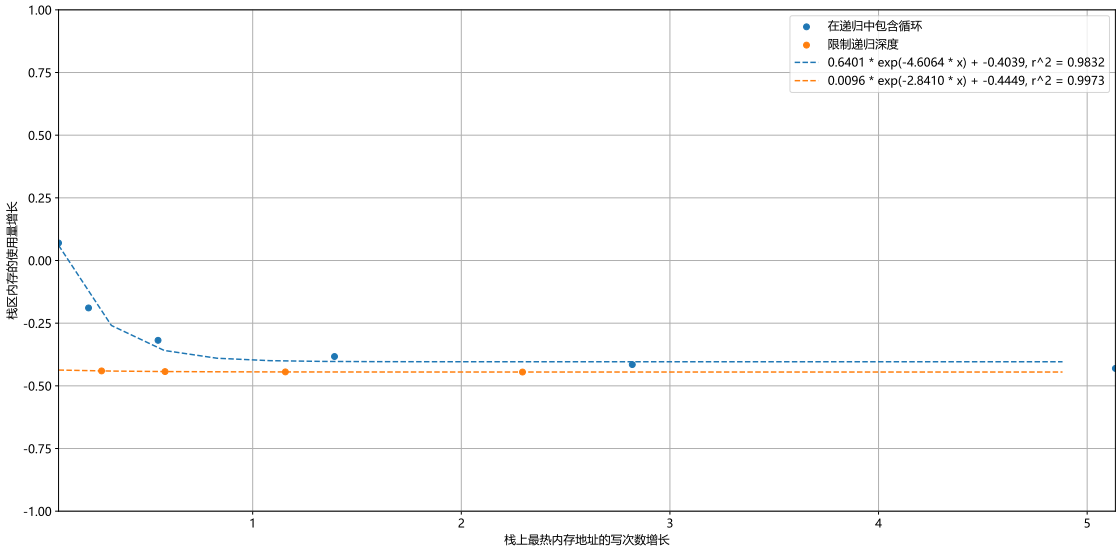

图 4.10　两种策略栈区内存的使用量、栈上最热内存地址的写次数散点图

从散点图中我们可以看出，改变参数的取值，在递归中保留循环时，栈区内存的使用量增长量、栈上最热内存地址的写次数存在明显的负相关性，而限制递归深度这种相关性非常弱。因此，在不牺牲栈区内存，亦即不牺牲损耗均衡的高效性的情况下，我们可以调整制递归深度的参数（提高递归深度限制），减少栈上最热内存地址的写次数，提升损耗均衡的有效性。而对在递归中保留循环而言，减少栈上最热内存地址的写次数意味着要牺牲栈区内存，从而牺牲损耗均衡的高效性。

与此同时，在栈上最热内存地址的写次数相同的情况下，限制递归深度的栈空间较小。故而，相对在递归中保留循环，限制较大的递归深度是在损耗均衡的高效性和有效性两方面更优的策略。

5 结论

以相变存储器为代表的非易失性存储器被认为是最有可能代替动态随即存储器的下一代存储器之一。针对相变存储器耐久度过低的缺点，将程序中的循环转化为递归函数，可以有效地实现损耗均衡，提升其使用寿命。

原版 Loop2Recursion 存在创建的递归函数参数冗余、返回值的传递方式低效的问题，容易引发栈溢出异常，同时影响了程序运行的性能。为此，我们设计并实现了一种新的循环转递归方法 New-Loop2Recursion，优化了生成的递归函数的栈帧大小和指令数，减少出现栈溢出异常的概率，并提升了程序运行的性能。同时，针对较深的递归容易引发栈溢出异常的缺点，我们实现了限制递归深度、在递归中包含循环两种进一步优化栈内存使用的策略。

实验结果表明，New-Loop2Recursion 有效地减小了生成的递归函数的栈大小（平均为-44.68%），减少了生成的递归函数的指令数（平均为-10.21%），同时栈上最热内存地址的写次数总体上增幅相当有限（平均为 1.00%），在提升程序运行的性能、减少出现栈溢出异常的概率的同时，损耗均衡有效性在整体上仍未受到很大的影响。同时，实验结果表明，由于有效地减小了栈区内存的使用量（平均为-44.05%），且在栈区内存的使用量相同情况下较在递归中保留循环栈上最热内存地址的写次数更优，限制较大的递归深度是在损耗均衡的高效性和有效性两方面更优的进一步优化栈内存使用的策略。

参考文献

[1] LERSCH L. Leveraging non-volatile memory in modern storage management architectures[Z]. 2021.

[2] KIM M, CHANG I J, LEE H J. Segmented tag cache: A novel cache organization for reducing dynamic read energy[J]. IEEE Transactions on Computers, 2019, 68 (10): 1546-1552.

[3] KIM B, LEE S H, KIM H, et al. Pcm: precision-controlled memory system for energy efficient deep neural network training[C]//2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2020: 1199-1204.

[4] LEE H, KIM M, KIM H, et al. Integration and boost of a read-modify-write module in phase change memory system[J]. IEEE Transactions on Computers, 2019, 68 (12): 1772-1784.

[5] WONG H S P, RAOUX S, KIM S, et al. Phase change memory[J]. Proceedings of the IEEE, 2010, 98(12): 2201-2227.

[6] LEE H, JUNG H, LEE H J, et al. Bit-width reduction in write counters for wear leveling in a phase-change memory system[J]. IEIE Transactions on Smart Processing & Computing, 2020, 9(5): 413-419.

[7] QURESHI M K, KARIDIS J, FRANCESCHINI M, et al. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling[C]//2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO). IEEE, 2009: 14-23.

[8] LEE B C, IPEK E, MUTLU O, et al. Architecting phase change memory as a scalable dram alternative[C]//Proceedings of the 36th annual international symposium on Computer architecture. 2009: 2-13.

[9] WU X, LI J, ZHANG L, et al. Hybrid cache architecture with disparate memory technologies[J]. ACM SIGARCH computer architecture news, 2009, 37(3): 34-45.

[10] QURESHI M K, SRINIVASAN V, RIVERS J A. Scalable high performance main memory system using phase-change memory technology[C]//Proceedings of the 36th annual international symposium on Computer architecture. 2009: 24-33.

[11] LIU T, ZHAO Y, XUE C J, et al. Power-aware variable partitioning for dsps with hybrid pram and dram main memory[J]. IEEE transactions on signal processing, 2013, 61(14): 3509-3520.

[12] ZHOU P, ZHAO B, YANG J, et al. A durable and energy efficient main memory using phase change memory technology[J]. ACM SIGARCH computer architecture news, 2009, 37(3): 14-23.

[13] XU W, LIU J, ZHANG T. Data manipulation techniques to reduce phase change memory write energy[C]//Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design. 2009: 237-242.

[14] ZHANG W, LI T. Characterizing and mitigating the impact of process variations on phase change based memory systems[C]//2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2009: 2-13.

[15] CHO S, LEE H. Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance[C]//Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. 2009: 347-357.

[16] SUN G, NIU D, OUYANG J, et al. A frequent-value based pram memory architecture[C]//16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011). IEEE, 2011: 211-216.

[17] FERREIRA A P, ZHOU M, BOCK S, et al. Increasing pcm main memory lifetime [C]//2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010). IEEE, 2010: 914-919.

[18] ZHAO M, SHI L, YANG C, et al. Leveling to the last mile: Near-zero-cost bit level wear leveling for pcm-based main memory[C]//2014 IEEE 32nd International Conference on Computer Design (ICCD). IEEE, 2014: 16-21.

[19] KIM S, JUNG H, SHIN W, et al. Had-twl: Hot address detection-based wear leveling for phase-change memory systems with low latency[J]. IEEE Computer Architecture Letters, 2019, 18(2): 107-110.

[20] KIM M, LEE H, KIM H, et al. Wl-wd: Wear-leveling solution to mitigate write disturbance errors for phase-change memory[J]. IEEE Access, 2022, 10: 11420-11431.

[21] HU J, XUE C J, ZHUGE Q, et al. Towards energy efficient hybrid on-chip scratch pad

memory with non-volatile memory[C]//2011 Design, Automation & Test in Europe. IEEE, 2011: 1-6.

[22] LI Q, ZHAO Y, HU J, et al. Mgc: Multiple graph-coloring for non-volatile memory based hybrid scratchpad memory[C]//2012 16th Workshop on Interaction between Compilers and Computer Architectures (INTERACT). IEEE, 2012: 17-24.

[23] SHAO Z, LIU Y, CHEN Y, et al. Utilizing pcm for energy optimization in embedded systems[C]//2012 IEEE computer society annual symposium on VLSI. IEEE, 2012: 398-403.

[24] BATHEN L A, DUTT N. Havoc: A hybrid memory-aware virtualization layer for on-chip distributed scratchpad and non-volatile memories[C]//DAC Design Automation Conference 2012. IEEE, 2012: 447-452.

[25] LI Y, CHEN Y, JONES A K. A software approach for combating asymmetries of non-volatile memories[C]//Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design. 2012: 191-196.

[26] HU J, XUE C J, ZHUGE Q, et al. Write activity reduction on non-volatile main memories for embedded chip multiprocessors[J]. ACM Transactions on Embedded Computing Systems (TECS), 2013, 12(3): 1-27.

[27] DHIMAN G, AYOUB R, ROSING T. Pdram: A hybrid pram and dram main memory system[C]//2009 46th ACM/IEEE Design Automation Conference. IEEE, 2009: 664-669.

[28] YU L, CHEN T, WU J. A software-hardware collaborating framework for wear leveling on phase change memory[C]//2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems. IEEE, 2012: 1360-1367.

[29] PAN C, XIE M, HU J, et al. Wear-leveling for pcm main memory on embedded system via page management and process scheduling[C]//2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications. IEEE, 2014: 1-9.

[30] AGHAEI KHOUZANI H, XUE Y, YANG C, et al. Prolonging pcm lifetime through energy-efficient, segment-aware, and wear-resistant page allocation[C]//Proceedings of the 2014 international symposium on Low power electronics and design. 2014:

327-330.

[31] CHEN C H, HSIU P C, KUO T W, et al. Age-based pcm wear leveling with nearly zero search cost[C]//Proceedings of the 49th Annual Design Automation Conference. 2012: 453-458.

[32] GOGTE V, WANG W, DIESTELHORST S, et al. Software wear management for persistent memories[C]//17th USENIX Conference on File and Storage Technologies (FAST 19). 2019: 45-63.

[33] LIU R, JIN P, WU Z, et al. Efficient wear leveling for pcm/dram-based hybrid memory[C]//2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, 2019: 1979-1986.

[34] WANG H, SHEN Z, ZHAO M, et al. Clock-rwrf: a read-write-relative-frequency page replacement algorithm for pcm and dram of hybrid memory[C]//2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, 2020: 189-196.

[35] MORARU I, ANDERSEN D G, KAMINSKY M, et al. Consistent, durable, and safe memory management for byte-addressable non volatile main memory[C]// Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems. 2013: 1-17.

[36] YU S, XIAO N, DENG M, et al. Walloc: An efficient wear-aware allocator for non-volatile main memory[C]//2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC). IEEE, 2015: 1-8.

[37] LI Q, HE Y, CHEN Y, et al. A wear-leveling-aware dynamic stack for pcm memory in embedded systems[C]//2014 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2014: 1-4.

[38] LI W, SHUAI Z, XUE C J, et al. A wear leveling aware memory allocator for both stack and heap management in pcm-based main memory systems[C]//2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2019: 228-

233.

[39] LI W, WU L, YUAN M, et al. Loop2recursion: Compiler-assisted wear leveling for non-volatile memory[C]//2020 IEEE 38th International Conference on Computer Design (ICCD). IEEE, 2020: 581-588.

[40] GUTHAUS M R, RINGENBERG J S, ERNST D, et al. Mibench: A free, commercially representative embedded benchmark suite[C]//Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538). IEEE, 2001: 3-14.

[41] ROSEN B K, WEGMAN M N, ZADECK F K. Global value numbers and redundant computations[C/OL]//POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: Association for Computing Machinery, 1988: 12–27. https://doi.org/10.1145/73560.73562.

[42] LATTNER C, ADVE V. Llvm: A compilation framework for lifelong program analysis & transformation[C]//International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, 2004: 75-86.

[43] LIU Y A, STOLLER S D. From recursion to iteration: what are the optimizations? [C]//Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation. 1999: 73-82.

[44] MURNANE J. To iterate or to recurse?[J]. Computers & Education, 1992, 19(4): 387-394.

[45] LUK C K, COHN R, MUTH R, et al. Pin: building customized program analysis tools with dynamic instrumentation[J]. Acm sigplan notices, 2005, 40(6): 190-200.

致谢

感谢李清安老师，他在本科期间开阔了我的视野，在各方面给予了我很大的帮助，教育了我很多学习、科研的方法和为人处世的道理。虽然学术研究的道路漫长而坎坷，但我希望能一直坚持下去，不辜负李老师对我的教导！

感谢祝园园老师和叶茫老师，他们多次带领我参加科研、竞赛方面的实践，在这个过程中给了我宝贵的教育和指导，也在出国方面给予我很大的指引，让我受益匪浅。

感谢武汉大学所有教授过我课程的老师，他们丰富了我的知识，提升了我的能力，为我之后在计算机领域发展奠定了坚实的基础。

感谢武汉大学计算机学院的所有领导和老师，他们为我的学习和生活提供了无微不至的关心和保障。

感谢武汉大学计算机学院叶文涛、杨枫、唐锦燃、李雨昕、花子涵、骆荟州等2018级软工班的同学，赖思奇、谭瑞锋、李光华、周燊、杨雨卓、龚艳等2018级计科班的同学，张钊为、王威卫等2019级的同学，以及2020级作为班助所带的各位同学；武汉大学经济与管理学院义力、马世昌、苏雯婕、李骏麒等同学；武汉大学弘毅学堂韩晨等同学；武汉大学化学与分子科学学院王京润等同学（以上排名不分先后）。他们与我一起参加了很多竞赛和活动，也在学习中、生活中给了我很多帮助和指导，开阔了我的视野，拓展了我的思维。愿友谊长存，也希望之后有机会多多合作！

感谢父母一直以来对我无条件的理解、支持和鼓励。我会继续努力，不辜负你们的期盼！

新的征程即将开始。愿我们在未来的岁月里，万事顺遂，心想事成！