

一种链状数据结构细粒度自动加锁方法

张予现^{1,2}, 张 昱^{1,2}

¹ (中国科学技术大学计算机科学与技术学院, 安徽 合肥 230027)

² (中国科学技术大学苏州研究院软件安全实验室, 江苏 苏州 215123)

E-mail : yuzhang@ustc.edu.cn

摘 要: 细粒度锁在并行编程与并发软件设计中起着重要的作用。然而, 细粒度锁对程序员具有较高的要求, 通常在设计细粒度锁并发程序时会带来算法复杂, 编程困难, 程序易出错等问题。文中提出一种链状数据结构的细粒度加锁方法, 并实现自动转换含有形状和共享标注的程序为细粒度锁代码的翻译器, 为了使生成的代码具有无死锁, 引入了一致性加锁协议, 实验结果表明, 基于本文方法能够自动为链状数据结构操作添加细粒度锁, 生成的代码具有较高的并发性。

关键词: 并行编程; 并发软件; 细粒度锁; 链状数据结构; 翻译器

中图分类号: TP

文献标识码: A

文章编号: 1000-1220 (2009) 02--

An Automatic Approach of Fine-grained Locking for Linked Data Structure

ZHANG Yu-xian^{1,2}, ZHANG Yu^{1,2}

¹ (School of Computer Science & Technology, University of Science & Technology of China, Hefei 230027)

² (Software Security Lab., Suzhou Institute for Advanced Study, University of Science & Technology of China, Suzhou, 215123)

Abstract: Fine-grained locking is extremely important in multi-processor programming and concurrent software design. However, fine-grained locking is notoriously difficult for programmers to get right—mistakes can lead to data races, conservative protection placement, the program error-prone and other issues. This paper presents an automatic fine-grained locking for linked data structure technique. We have implemented a translator which takes a program annotated with shape declaration and share effect provided by programmer and automatically insert instrument the fine-grained locking. experimental results show that this method can automatically insert instrument fine-grained locking for linked data structure, making the program a higher concurrency.

Key words: parallel programming; concurrent software; fine grained; linked data structure; translator

1 引言

基于共享存储的并行编程是一种主流的并行编程模型^[1]。在这种模型下, 多个线程交错并发地访问共享数据, 程序员需要利用锁等设施来控制对共享数据访问的有效同步, 这不仅增加了编程的难度而且也常引起许多难以捉摸的错误。

为降低并行编程的难度, 近年来, 很多研究者从数据库领域引入了原子区^{[2][3]} (atomic) 的概念, 原子区是一种语言级别的抽象, 它要求原子区内的代码必须原子隔离地执行, 因此, 原子区提供了一种高级的规范描述同步语义。采用原子区确实简化了共享资源的访问控制, 但原子区的实现

必须由编译系统来保证。编译系统通过程序分析分析出共享资源的保护区间, 再通过程序变换生成基于锁^{[1][10][11]}或事务内存^{[4][5]}的代码, 当前, 这两类原子区的实现方法仍在研究中, 并且遇到一些难以解决的问题, 例如, 不在原子区内的共享资源如何保证它们的原子语义; 事务与非事务代码如何交互, 事务可否嵌套, 硬件实现的事务内存难以支持任意大小的事务, 软件实现的事务内存存在运行开销大、语义不明确等问题。

笔者所在的课题组正在研发一种小型并行语言SPC^[6], 它是一种基于共享存储的并行编程模型, 与现有的并行语言不同, 它的最大特点在于为程序员提供一种访问控制的高级抽象, 程序员不需要显式管理访问控制, 而只需要给出共享资源的使用声明, 由编译器辅助分析共享资源的保护区间,

称为**维持区间**，根据维持分析结果插桩访问控制代码，这种语言的优点在于抽象层次更高，更易于程序员编写并行程序。^[6]已经解决了整型上的并行编程，能够处理生产者消费者问题，^[7]提出了利用形状图对程序运行时的动态数据结构和指针变量关系建立抽象，通过形状图推导完成对共享单元的访问控制分析，完成了单链表上的访问控制分析。

但^[7]对指针类型支持还比较单一，支持的数据结构也较为简单，只给出了单链表上的程序分析，没有考虑可能的死锁问题，在^[7]中，含有**长访问路径**的语句如 $q=p \rightarrow next \rightarrow next$ 必须拆分为 $t=p \rightarrow next$; $q=t \rightarrow next$ 处理，增加了程序分析的遍数，同时引入了临时变量，增加了程序复杂度。本文针对^[7]不足，扩展了原有的形状图及其推导规则，给出了能处理如单链表、双链表的链状数据结构统一的处理方法；将维持语义划分为原子维持和隐式维持，保证了共享资源访问的正确性；引入了一致性加锁协议，避免了生成的代码发生死锁；完成了SPC翻译器与运行库的支持，保证了自动生成的细粒度代码的运行。

和已有研究工作相比，本文工作的主要特色和贡献在于：1、扩展了形状图，提供一个能够处理单、双链状数据结构的统一的程序分析框架，避免了长访问路径的拆分。2、维持语义细化为原子维持和隐式维持，程序分析更为简单，同时保证了所有共享变量访问的原子隔离性。3、提供一种无死锁的自动安插访问控制代码的机制，为原本只有共享标注的程序自动转换为细粒度锁代码。

本文其余部分安排如下：第2节给出问题描述与解决思路，第3节介绍内存抽象及共享单元维持，第4节重点介绍维持分析算法，第5节给出细粒度锁的访问控制代码生成，第6节给出实验结果，第7节进行小结。

2 问题描述与解决思路

2.1 共享性与形状声明

本文设计了强调指针类型的并行语言 SPC-II，图1(a)是 SPC-II 实现的双链表插入代码片段。图1(b)为自动生成的细粒度锁代码。

在图1(a)中，定义了双链表节点的结构体类型，标注“@DListNode(r,l)”声明数据结构形状，DListNode 表示双链表节点，(r,l)表示双链表节点的两个链域名。struct node * toshared p; 标注“toshared”声明 p 是指向共享单元的局部指针变量。图(b)是图(a)的程序经程序分析和程序变换得到的细粒度锁代码，其中，Lock(H)表示是对 H 指向的节点加锁，unlock(q)表示释放 q 指向的节点中的锁。

为了生成图(b)的细粒度锁代码，就需要对图(a)程序进行程序分析，分析出所有共享单元的保护开始和保护结束，这种共享单元的保护开始和保护结束在本文中称为**维持获得和维持释放**。而这种线程对共享单元保护的程序分析称为**维持分析**。在语句4，H是共享指针，它指向的单元是共享单元且第一被访问，所以，需要通过H维持H指向的共

享单元，在语句4之后，p和H别名，在语句6,7中，有对p的使用，所以，H维持的单元在语句6,7中不能释放，在语句8，q->r指向的共享单元第一次使用，需要通过q->r维持。在while循环第二次迭代时，q被重新定值，第一次迭代时，q指向的单元以后不再使用，需要在语句7之前通过q进行维持释放。所以，就有了语句4之前的维持获得Lock(H)，语句8之前的维持获得Lock(q->r)和语句7之前的维持释放unlock(q)。

```

struct node@DListNode(r, l) {
    int data;
    struct node* r,*l;
};
struct node* toshared p;
struct node* toshared q;
[p=H]4;
while([p!=NULL]6) {
    [q=p]7;
    [p=q->r]8;
}
[t=malloc(node_t)]10;
[t->r=NULL]11;
if([q!=NULL]12) {
    [q->r=t]13;
    [t->l=q]14;
}

struct node{
    int data;
    struct node* r,*l;
    void *lock;
};
struct node* p;
struct node* q;
Lock(H); p=H;
while(p!=NULL){
    unlock(q);q=p;
    Lock(q->r); p=q->r;
}
t=(struct node*)malloc(node_t);
t->r=NULL;
if(q!= NULL){
    q->r=t;
    t->l=q; unlock(q);
}

```

图1 SPC-II 语言实现的双链表插入

Fig 1. A doubly linked list insert implemented in SPC-II

2.2 SPC-II 语言的抽象文法

| | |
|----------|---|
| (Stmt): | $s ::= [p=a^l] s_1; s_2 \mid [p=\text{malloc}(\text{type})]^l [s_1] \dots \parallel s_n \mid [\text{free}(p)]^l \mid \text{if } [b]^l \text{ then } s_1 \text{ else } s_2 \mid \text{while } ([b]^l) s$ |
| (Lval) | $p ::= x \mid x \rightarrow f$ |
| (AExp) | $a ::= n \mid p \mid a_1 \text{ op}_r a_2 \mid \text{null}$ |
| (BExp) | $b ::= p \mid a_1 \text{ op}_r a_2 \mid b_1 \text{ op}_r b_2$ |
| (Types) | $\tau ::= \text{int} \mid x[\delta] : \{ f_1, \tau_1, \dots, f_n, \tau_n \} \mid x[\delta]^*$ |
| (Annote) | $\delta ::= \text{shared} \mid \text{toshared} \mid \text{List}$ |
| (Label) | $l \in L$ |
| (Field) | $f \in F$ |

图2 SPC 抽象语法

Fig. 2 SPC abstract syntax

为了便于形式地讨论本文的分析方法，图2给出SPC-II的抽象语法，其中忽略了函数调用和除整型外的其他简单类型和运算。(为简便起见，后文的SPC都指SPC-II)。在SPC中，指针类型的变量只能用于赋值、相等与否比较、存取指向对象等运算以及作为函数的参数，禁止指针算术和取地址运算(&)。

在 SPC 中，每条赋值语句和 if、while 语句中的条件表达式都被认为是原子执行的，简称**原子命令**。l 为原子命令的标签，为了方便程序分析而引入。在声明变量时，可以加上 δ 标注。 δ 标注有两类，一类声明形状，如图1(a) @DListNode(r,l)定义双链表，单链表节点如@ListNode(r);

另一类声明共享性，它们有不同的作用域，shared 表示共享变量，作用域是全局的或过程内，全局变量默认是 shared 的，在 main 函数内，且在并行语句外部的变量也可以声明 shared 的，因可被多个线程同时访问而看作是共享的，toshared 表示可以指向共享单元的局部变量，作用域是并行语句中的每个并行块内，因此，表达式 $x \rightarrow f$ 中，如果 x 是全局的，或者 shared 或者 toshared，则 $x \rightarrow f$ 指向的单元是共享的，否则是私有的。

由于指针别名的存在，会给共享单元的维持分析带来困难。如图 1 (a) 程序， $[p=q \rightarrow r]^8$ 简记为原子命令 A_8 ，变量 p 在 A_8 中重新定值，在不考虑别名的情况下， p 指向的共享单元在 A_8 入口不再需要维持；但由于 A_7 执行后导致 p 和 q 指向同一个单元，而 q 在 A_8 中被使用，所以， p 指向的共享单元不能在 A_8 入口处维持释放。

3 内存抽象及共享单元维持

为了解决指针别名带来的问题，本节将利用形状图抽象内存单元，先给出利用形状图抽象内存单元的方法，然后给出共享单元维持。

3.1 内存抽象

将内存单元抽象为形状图上的节点，通过对节点的维持达到对共享单元保护的目的。具体思想是抽象堆分配的数据，将堆上分配的数据抽象成形状图上的节点，在维持分析过程中，以节点为中心，可以确保在不同的程序点用不同的变量维持获取和维持释放的是同一个节点。

为了表示程序中无限的堆位置，^[8]中提出了形状图的表示方法，本文对^[8]的形状图进行了扩展，本文的形状图如下：

- $G = \langle N^+, E, \mu_K, \mu_P, \mu_S, \mu_H \rangle$ 表示形状图，是一个六元组。
- $N^+ = N \cup \{n_N, n_D\}$ ， N^+ 由集合 N ，表示空和悬空的节点 n_N, n_D 组成，其中， N 由结构节点和抽象节点组成。结构节点代表实际链表上的一个节点，抽象节点代表实际链表上的 0 个或多个连续节点，抽象节点有一个 length 属性记录其代表的结构节点的个数，该属性在 5.2 节的避免死锁还会用到。
- $E = E_P \cup E_F$ ，表示边由指向边和域边组成，指向边 $E_P = \{(v, n) | v \in V, n \in N^+\}$ ，域边 $E_F = \{(n_1, f, n_2) | n_1, n_2 \in N^+, f \in F\}$ ，其中， F 为域名集合。
- $\mu_K : N \rightarrow K, K = \{a, s\}$ 表示节点与其类型的映射，节点类型有结构节点和抽象节点。
- $\mu_P : N \rightarrow Nat$ ，表示节点的形状，0 表示单链表节点，1 表示双链表节点，为了以后支持更多的数据结构，可以扩展节点的形状属性。
- $\mu_S : N \rightarrow B, B = \{T, F\}$ 表示节点与其共享性的映射， T 表示共享， F 表示私有。

$\mu_H : N \rightarrow B, B = \{T, F\}$ 表示节点与其维持性的映射， T 表示已维持， F 表示其它情况（包括尚未维持和不需要维持），初始值默认为 F 。

形状图的结构如图 3 所示。

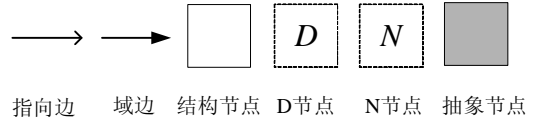


图 3 形状图结构

Fig.3 holding analysis graph

形状图的展开规则是对抽象节点 n 的一步展开。当抽象节点 n 为空节点时，节点类型由抽象节点改变为 n_N ；当 n 不为空时，可以展开为一个结构节点 n 和一个新的抽象节点 n' ， n' 的属性从 n 拷贝，但 n' 维持属性为初始默认值 F 。

形状图的折叠规则是对多个结构节点的折叠。当有两个相邻节点无声明指针指向，且它们维持属性都为 F 时，可将它们折叠为一个抽象节点。

3.2 共享单元维持

函数 $access([A])$ ：返回原子命令 $[A]$ 内访问的变量对应形状图上的节点集合。

定义 1 (原子维持) 在原子命令 A_i 的入口，如果共享单元 n 满足 $n \in access(A_i) \wedge \mu_S(n) = T \wedge \mu_H(n) = F$ ，则该共享单元需要在原子命令 A_i 中进行保护，这种原子命令自身对所访问的共享单元的保护即为原子维持。

定义 2 (隐式维持) 在线程 T 的连续原子命令序列 $[A_1], \dots, [A_n]$ 上有，共享单元 n 满足 $n \in access(A_i) \wedge n \in access(A_j) \wedge \mu_S(n) = T$ ，则该共享单元在 $[A_i]$ 到 $[A_j]$ 的保护即为隐式维持，它是原子命令之间的维持。

隐式维持非形式的描述为在原子命令 A 中，存在一个定值点，如果该定值语句所引用的共享单元第一次被使用，则在该语句之前开始维持该共享单元，如果该共享单元是最后一次引用，则在此语句之前释放该共享单元的维持。

定义 3 (维持获得与释放) 无论是原子维持还是隐式维持，把对共享变量维持的开始称作维持获得，对共享变量维持的结束称作维持释放。

例如，原子命令 $[p=q \rightarrow next \rightarrow next]^1$ ，则 $access(l) = \{n_p, n_q, n_{q \rightarrow next}, n_{q \rightarrow next \rightarrow next}\}$ ， n_p 形式表示 p 指向的节点，下面内容中次形式表示相同含义。 $q \rightarrow next \rightarrow next$ 指向的节点是新访问的节点属于隐式维持，而 $q, q \rightarrow next$ 指向的节点若只在该原子命令中访问，属于原子维持，这样，将维持分为原子维持和隐式维持，就可以保证所有共享单元都在维持下进行访问，保证了程序的正确性。

4 维持分析算法

维持分析旨在分析 SPC 程序中的共享单元以获取线程对共享单元的维持。维持分析算法分两步获取线程对共享单元的维持：首先针对每条原子命令有相应的处理方法，获取每个原子命令的维持信息；然后通过收集循环迭代产生的维持信息进行合并，就是每个线程对共享单元的维持。

4.1 原子命令上的维持分析

在维持分析过程中，需要时刻记录当前的形状图和形状图上维持获得和维持释放，维持信息以<访问路径，节点>二元组的形式存放，这样做的好处是能够迅速的找到要维持的节点和维持节点的访问路径。

G : 当前语句的形状图。

M_{acq} : 当前语句前的维持获得集合。

M_{rel1} : 当前语句前的维持释放集合。

M_{rel2} : 当前语句后的维持释放集合。

M_{atom} : 当前语句中所有访问路径到指向节点的映射集合。

M_{pre} : 当前语句中被定值的声明变量到其指向节点的映射集合。

每条原子命令的处理可以分为五个阶段：

(1) 预处理。在初始形状图或前驱语句传入的形状图上找到当前语句中所有访问路径可以访问到的节点，如果没有该节点，就需要展开抽象节点，并将当前语句中的所有访问路径到节点的映射写入 M_{atom} 。

(2) 维持获得分析，依据 M_{atom} 中记录的要访问的节点，如果节点是共享的且其还没有被维持，则它是维持获得，记录到 M_{acq} 中。

(3) 形状图变换，将语句中对指针的修改情况反映到对形状图的变换。

(4) 维持释放分析，如果确定在的形状图上的节点不再使用，则它是维持释放，记录到 M_{rel1} 中。

(5) 后处理，合并原子维持获得和隐式维持获得，记录到 M_{acq} 中，合并语句后维持释放和原子维持释放，记录到 M_{rel2} ，为了使形状图不会随着循环迭代无限膨胀，对形状图 G 调用折叠规则。

图 4 的辅助定义给出了形状图六元组某些元素变化后的形状图简记，以及对维持记录集合进行元素的更新（包括添加）和删除操作。

$$G \stackrel{\text{def}}{=} (N, E, \mu_K, \mu_P, \mu_S, \mu_H)$$

$$G|_{\mu'_H} \stackrel{\text{def}}{=} (G.N, G.E, G.\mu_K, G.\mu_S, \mu'_H)$$

$$G|(E', \mu'_S) \stackrel{\text{def}}{=} (G.N, E', G.\mu_K, \mu'_S, G.\mu_H)$$

$$\text{dom}(M\{a \rightarrow b\}) \stackrel{\text{def}}{=} \begin{cases} \text{dom}(M) & \text{if } a \in \text{dom}(M) \\ \text{dom}(M) \uplus \{a\} & \text{otherwise} \end{cases}$$

$$(M\{a \rightarrow b\})(x) \stackrel{\text{def}}{=} \begin{cases} M(x) & \text{if } x = a \\ M(x) \uplus \{b\} & \text{otherwise} \end{cases}$$

$$M(x) - \{a \rightarrow b\} \stackrel{\text{def}}{=} \begin{cases} M - \{b\} & \text{if } a \in \text{dom}(M) \\ M(x) & \text{otherwise} \end{cases}$$

图 4 辅助定义

Fig.4 auxiliary definition

在以下各个阶段，语句 s 有不同的类型，有些阶段没有列出语句或者没有列出所有语句，说明语句 s 不影响分析结果或其它类型 s 在该阶段不用处理。

4.1.1 预处理

$$f_{pre}(G, s) = (G', M_{atom})$$

f_{pre} 表示对形状图 G 中在 s 中访问到的抽象节点进行展开，得到形状图 G' ，使得 s 中访问路径访问的节点在形状图 G' 中可以找到，并将访问路径到节点的映射记录到 M_{atom} 中。

例如，在图 1 (a) 中，在语句 8 前， q 和 p 指向同一个抽象节点，语句 8 为 $p=q \rightarrow r$ ， $q \rightarrow r$ 指向的节点不在形状图中，展开 q 指向的抽象节点，使语句 8 中的访问路径在形状图中可以找到， $M_{atom}(8) = \{ \langle p, n_p \rangle, \langle q, n_q \rangle, \langle q \rightarrow r, n_{q \rightarrow r} \rangle \}$ 。

4.1.2 维持获得分析

根据维持语义，维持获得也按隐式维持获得和原子维持获得这两种类型进行收集。

a) 隐式维持获得分析

$$f_{impl}(G, s, M_{atom}) = (G', M_{acq}, M_{atom})$$

f_{impl} 表示对形状图 G ，如果 s 右部 e 指向的节点 n_e 维持属性为 F 且共享性为 T ，则 n_e 是隐式维持获得节点，记录到 M_{acq} 中，并更新 n_e 维持属性为 T ，得到形状图 G' ，**形状图 G' 相对于 G 仅 μ_H 改变。**

$$\text{case } [l_1 = e]^t :$$

$$n_e \in M_{atom}(e), \mu'_H = \mu_H \{ n_e \rightarrow T \} (x),$$

$$M_{acq} = M_{acq} \{ e \rightarrow n_e \} (x), G' = G|_{\mu'_H}$$

$$\text{where } \mu_H(n_e) = F \wedge \mu_S(n_e) = T$$

b) 原子维持获得分析

$$f_{atom}(G, M_{atom}) = (G', M'_{atom})$$

f_{atom} 表示对 M_{atom} 中的节点 n ，如果 n 的维持属性为 F ，更新为 T ，得到形状图 G' ，**形状图 G' 相对于 G 仅 μ_H 改变。**

对任何类型语句 s ，推理规则都如下：

$$G' = G|_{\mu'_H}$$

$$\forall r \in \text{dom}(M_{atom}), n \in M_{atom}(r), \mu'_H = \mu_H(n \rightarrow T)$$

$$M'_{atom} = M_{atom} \{ r \rightarrow n_r \} (x)$$

c) 维持释放预记录

$$f_{rel}(G, s) = (G, M_{pre})$$

f_{rel} 表示如果 s 是对声明变量 l_1 的定值，则将 l_1 到它指向节点的映射记录到 M_{pre} 中，这是因为对 l_1 重新定值，会导致在该语句之后释放 l_1 指向的节点时，没有变量指向，所以，需要事先记录指向该节点的指针。**形状图 G 没有发生改变。**

4.1.3 形状图变换

$$f_{trans}(G, s) = G'$$

f_{trans} 表示将语句 s 中对指针的修改情况反映到对形状图的变换，得到形状图 G' ，**形状图 G' 相对于 G 仅 E 和 μ_S 改变。**由于篇幅有限，具体规则见^[9]的技术报告。

$$G' = G|(E', \mu'_S)$$

4.1.4 维持释放分析

a) 原子维持释放分析

$$f'_{atom}(G, M_{atom}) = (G', M_{atom})$$

f'_{atom} 表示对形状图 G 中 M_{atom} 指向的任一节点 n ，更新维持属性为 F ，得到形状图 G' ，形状图 G' 相对于 G 仅 μ_H 改变。

$$\begin{aligned} G' &= G|_{\mu'_H} \\ \forall r \in dom(M_{atom}), n &= M_{atom}(r) \\ \mu'_H &= \mu_H \{n \rightarrow F\}(x) \end{aligned}$$

b)隐式维持释放分析

$$f'_{rel}(G, M_{pre}, s) = (G', M_{rel1})$$

f'_{rel} 表示对形状图 G 进行隐式维持释放。隐式维持释放分两种情况 (1) 如果存在维持属性为真的节点，而该节点无声明变量指向，则它是因为 M_{pre} 中变量被重新定值导致，则释放 M_{pre} 中的节点。(2) 如果形状图上存在被维持的节点 n 既无共享声明变量指向，又无共享节点的域边指向，说明该节点不再共享，如果某个节点不再共享或者其指向边的声明变量不再活跃，对节点 n 进行隐式维持释放，记录到 M_{rel1} 中，并更新 n 的维持属性为 F ，得到形状图 G' 。这里需要利用活跃变量分析^[8]结果，它是一种普通的数据流分析，如果变量 x 在某个程序点 p 之后不再被使用，就说变量 x 在点 p 不再活跃。形状图 G' 相对于 G 只有 μ_S, μ_H 改变。

$$\begin{aligned} case \ [l_1 = e]^l : \\ case \ [l_1 = \text{malloc}(type)]^l : \\ case \ [\text{free}(l_1)]^l : \\ G' &= G|_{(\mu'_H, \mu'_S)} \\ (1) \ \forall r \in dom(M_{pre}), n_r \in M_{pre}(r), \mu_H(n_r) &= T \wedge \\ \exists unref(G, n_r), M_{rel1} &= M_{rel1} \{r \rightarrow n_r\}(x) \\ (2) \ \mu_H(n_r) &= T \wedge (\mu_S(n_r) = F) \wedge \neg islive(r), \\ M_{rel1} &= M_{rel1} \{r \rightarrow n_r\}(x) \end{aligned}$$

4.1.5 后处理

计算维持获得： $M_{acq} = M_{acq} \sqcup M_{atom}$

计算语句后维持释放： $M_{rel2} = M_{rel2} \sqcup M_{atom}$

为了保证迭代能够达到稳定状态，在后处理部分，会折叠一些节点，使得形状图不会随着循环迭代无限膨胀。

4.2 数据流等式与转换方程

本文采用正向数据流分析，数据流集合的元素是一个四元组 $(G, M_{acq}, M_{rel1}, M_{rel2})$ 。语句入口处的数据流值由它前驱语句出口的数据流值合并得到。语句出口处的数据流值由4.1节的五个阶段构成的函数 f_{trans} 得到。

数据流方程

$$\begin{aligned} H_{in}([s]^l) &= \begin{cases} (G_0, \emptyset, \emptyset, \emptyset) & \text{if } l = \text{init}(S) \\ \cup H_{out}(l') \mid (l', l) \in \text{flow}(S) & \text{otherwise} \end{cases} \\ H_{out}([s]^l) &= \\ \{f_{tran}(h, s) \mid h \in H_{in}([s]^l), h = (G, M_{acq}, M_{rel1}, M_{rel2})\} & \end{aligned}$$

图5 维持分析数据流方程

Fig.5 The data flow equation of holding analysis

在每次迭代过程中，会不断展开抽象节点，对于符合条

件的节点，会折叠成抽象节点，当前迭代产生的形状图和上一次迭代产生的形状图相同时，说明迭代达到稳定状态。

将所有形状图上的维持信息进行合并，即可得到所有程序点维持和释放维持。

4.3 实例分析

图1(a)程序的维持分析结果如表1所示(表中的 qr 为 $q \rightarrow r$ 的简写)， M_a, M_{r1}, M_{r2} 分别表示 $M_{acq}, M_{rel1}, M_{rel2}$ ，并且只给出二元组的第一元。维持分析在三次迭代后达到稳定状态，虽然每个迭代的形状图不同(迭代产生的形状图可见^[9]的演示文档)，但都可以通过统一的访问路径表示要维持的节点。

表1 双链表插入实例分析结果

Table 1 The analysis results of list insert program

| 语句 | Iter1 | | Iter2 | | Iter3 | |
|----|-------------|------------------|-------------|------------------|-------------|------------------|
| | M_a | $M_{r1} M_{r2}$ | M_a | $M_{r1} M_{r2}$ | M_a | $M_{r1} M_{r2}$ |
| 4 | {H} | \emptyset | {H} | \emptyset | {H} | \emptyset |
| 7 | \emptyset | {q} \emptyset | \emptyset | {q} \emptyset | \emptyset | {q} \emptyset |
| 8 | {qr} | \emptyset | {qr} | \emptyset | {qr} | \emptyset |
| 14 | \emptyset | {q} \emptyset | \emptyset | {q} \emptyset | \emptyset | {q} \emptyset |

5 访问控制代码生成

5.1 细粒度锁语句插桩

根据维持分析结果和程序的控制流图，即可按下面的算法为源程序添加细粒度锁(维持结果的合并见^[9]技术报告)，进而得到图1(b)的细粒度锁代码。

for($[A]^l \in S$) do

(1.1) for($x \in \text{Ha}(l)$) do /*Acquire lock operation*/

for($l', l \in \text{flow}(S)$) do

insert lock(x) between l' and l

(1.2) for($x \in \text{Hr}(l)$) do /*Release lock operation*/

for($l', l \in \text{flow}(S)$) do

insert unlock(x) between l' and l

(1.3) for($x \in \text{Hr}(l)$) do /*Release lock operation*/

for($l, l' \in \text{flow}(S)$) do

add unlock(x) between l and l'

end for

5.2 避免死锁

图6是两个线程操作双链表的细粒度锁代码片段，线程T1从链表头部遍历，线程T2从链表尾部遍历。在某个时刻，线程T1获得节点a锁，申请节点b锁，线程T2获得节点b锁，申请节点a锁。两线程在第03行死锁。

程序中的共享单元已被抽象为形状图上的节点，形状图上的每个节点都有唯一标号id，id满足自然数上的全序关系 $<$ ，节点id和节点锁一一对应，当线程T在申请节点锁失败时，如果该节点的id小于线程T占有锁的某个节点的id，说明该线程获取锁次序逆序，且与其它线程冲突，线程T

需要释放之前持有的锁, 重新进行获取锁操作, 否则, 线程 T 等待该锁。

```

T1:{
01 lock(p);
02 while (p!=tail){
03   lock(p->next);
04   p=p->next;
05   ....
06 }
07 .....
}

T2:{
01 lock(p);
02 while (p!=head){
03   lock(p->prior);
04   p=p->prior;
05   ....
06 }
07 .....
}

```

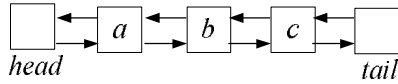


图 6 双链表操作死锁

Fig 6. Deadlock example

虽然形状图上的节点与实际链表节点并不一一对应, 但每个抽象节点都维护了一个 id 列表 list<id>, 形状图折叠时, 被折叠节点的 id 记录在 list<id>中, 形状图展开时, 新产生的节点仍使用原有 id, 这就保证了实际链表上节点锁的有序性, 不会产生死锁。

图 6 中 T1 会等待节点 b 锁, T2 申请锁节点 a 锁失败, T2 会释放节点锁 b, 并重做, 此时 T1 可以成功获取节点 b 锁, 之后, T2 成功获取锁, 不会出现死锁。

6 实验结果和分析

笔者在 SUIF2^[14]编译框架上, 实现了过程内的维持分析算法, 应用到了 SPC 翻译器^[9]中。为了测试 SPC 翻译器生成的细粒度锁代码的正确性和并发性, 本文比较了手工粗粒度锁代码, 事务内存代码 (采用 TL2^[4]类库), 串行代码与 SPC 翻译器生成的细粒度锁代码之间的性能。

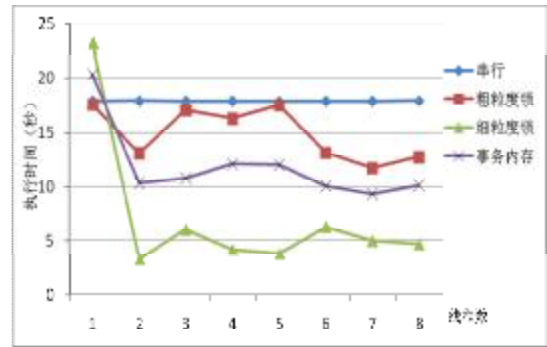
程序测试用例是两种链状数据结构, 单链表和双链表, 它们是长度为 2000 的有序链表, 节点 data 域的值 1 到 2000, 工作任务是链表的插入, 查找和删除的混合操作, 它们的比例是 10%的链表插入和删除, 80%链表查找。

实验环境为 Intel Xeon 八核 CPU, 单核的主频 1.60GHz, Cache 为 4096KB, 主存为 4GB。实验数据如图 7 所示, 横坐标表示线程数量, 纵坐标表示程序执行时间 (单位是秒), 图 (a) 是单链表测试, 图 (b) 是双链表测试, 从两个性能图可以看出不同并发控制完成相同工作的执行时间相差很大, 其中, 多线程方式比串行方式性能要高, 事务内存比粗粒锁性能要高, SPC 翻译器生成的细粒度锁代码比粗粒度锁和事务内存的并发性能都高, 这种情况随着线程数量的增加而更加明显。

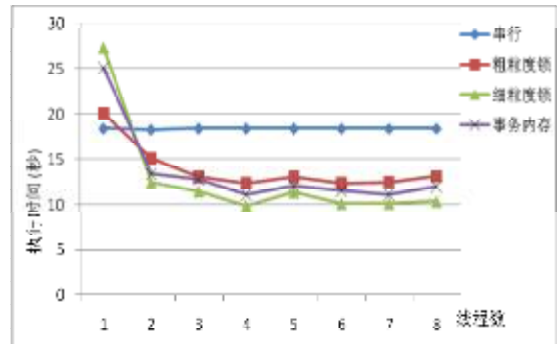
7 结束语

文中提出了一种链状数据结构细粒度加锁方法, 利用程序员提供的形状和共享标注, 编译器通过程序分析分析出共

享单元的维持, 然后自动安插细粒度锁代码。通过实验发现, 这些翻译器生成的细粒度锁代码具有较高的并发性。本文不足之处在于只是过程内的分析, 没有完成过程间的分析, 所能支持的数据结构还较为简单。过程间分析, 以及支持树, 跳表等更为复杂的数据结构是我们接下来工作的主要内容。



(a) 单链表



(b) 双链表

图 7 不同并发控制的链表操作性能

Fig 7. The performance of different concurrent list operations

References:

- [1] Michael Emmi, Jeffrey S. Fischer et al. Lock Allocation [C]. In Proceedings of the ACM Symposium on the Principles of Programming Languages, Jan. 2007, 291-296
- [2] Vijay Saraswat. Report on the experimental language X10. Version 2.01. [EB/OL]. <http://x10-lang.org/>, Jan. 2010.
- [3] Jonathan Lee and Jens Palsberg. Featherweight X10: A core calculus for async-finish parallelism [C]. In Proceedings of 15th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'10), Bangalore, India, Jan. 2010, 25-36.
- [4] Dave Dice, Ori Shalev, and Nir Shavit, Transactional Locking II [C]. In Proceedings of the 20th International Symposium on Distributed Computing (DISC), Stockholm, Sweden, Sep. 2006.
- [5] Kunal Agrawal, Angelina Lee and Jim Sukha. Safe open-nested transactions through ownership [C]. In Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'09), Raleigh, North Carolina, USA, Feb. 2009.

- [6] Wang Chen, Zhang Yu, Fu Xiao-peng, et al. A parallel programming language with shared variable holding declaration. accepted by Journal of Chinese Computer Systems, May 2010.
- [7] Zhang Wei, Zhang Yu, Wang Chen et al. An analysis approach to concurrent access control for shared mutable data [J]. Journal of University of Science and Technology of China, 2011, 41(2): 164-172.
- [8] F Nielson, HR Nielson, C Hankin. Principle of program analysis [M]. 1999.
- [9] Zhang Yu. A Parallel Language with Shared Resource Declaration for Mutable Data Structure [EB/OL]. <http://kyhcs.ustcsz.edu.cn/pplsrs>, 2011.
- [10] Sigmund Chorem, Trishul Chilimbi, Sumit Gulwani. Inferring locks for atomic sections [C]. In Proceedings of ACM Conference on Programming Languages Design and Implementation(PLDI'08), Tucson, Arizona, June 2008, 304-315.
- [11] Bill McCloskey, Feng Zhou et al. Autolocker: Synchronization Inference for Atomic Sections [C]. In Proceedings of the ACM Symposium on the Principles of Programming Languages, 2006, 346-358.
- [12] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections [C]. In ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, June 2006.
- [13] Benjamin Hindman, Dan Grossman. Atomicity via source-to-source translation [C]. In ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, October 2006.
- [14] R. Wilson, R. French, C. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers[R]. In ACM SIGPLAN Notices, December 1994, 29(12): 31-37.

附中文参考文献:

- [6] 汪晨, 张昱, 付小鹏, 等. 一种共享变量维持声明的并行程序语言[J]. 小型微型计算机系统, 2010年5月
- [7] 张伟, 张昱, 汪晨, 等. 一种动态共享数据结构的并发访问控制分析方法[J]. 中国科学技术大学学报, 2011, 41(02): 164-172