一个基于部分求值的Java动态优化系统*)

高红雨　廖湖声　王　众
(北京工业大学计算机学院　北京100022)

A Java Dynamic Optimization System Based on Partial Evaluation

GAO Hong-Yu　LIAO Hu-Sheng　WANG Zhong
(Institute of Computer, Beijing University of Technology, Beijing 100022)

Abstract　A dynamic optimization system for Java programs based on partial evaluation is presented in this paper. It consists of four subsystems: Java program transformation, binding time analysis, program specialization generation and application interface generation.

Keywords　Partial evaluation, Specialization, Dynamic optimization

1. 引言

目前,可重用性、兼容性、可扩充性、可互操作性已经成为软件设计的发展方向,这使软件系统的通用性增强、适用范围扩大;与此同时,系统的结构变得越来越复杂,性能也受到负面影响。如果软件系统能够在运行时与具体应用环境紧密结合,排除无关的通用性计算,系统的性能将会得到提高。部分求值作为一种新型的软件自动化方法,是解决这一问题的有效手段。

部分求值技术是一种根据用户提供的不变量(invariant)信息对程序进行自动例化的程序变换技术。所谓例化(specialization)是指对程序进行专门化处理,即根据指定运行环境中不变量,将程序变换成基于这种环境状态的专用程序(即滞留程序,residual program)。利用使用者给定的不变量信息,部分求值系统完成相关的部分计算(前段计算),将应用程序变换为仅包含剩余计算的滞留程序(后段计算),从而获得较高的效率和性能。

经过近20年的研究,部分求值技术已取得较大发展,并在科学计算[4]、计算机图形学[5]、操作系统[6]、软件工程[7]等领域的一些实际应用中产生良好效果。但是目前部分求值技术的应用还不普及,其使用者往往就是部分求值系统的设计者[8];已有的部分求值系统多是针对过程型语言(主要是C语言)和函数型语言开发的。鉴于面向对象语言在实际中的广泛应用,有必要深入研究面向对象语言的部分求值技术。为此,我们研制了一个基于部分求值的Java程序动态优化系统。这是一个实现程序动态例化的软件工具,它使得Java程序能够根据当前运行环境实时地优化程序自身结构,从而实现程序优化,提高运行性能。该系统在绑定时间分析、例化等关键技术领域使用了新的方法,并在部分求值系统的易用性方面作了尝试。

2. 系统总体结构

基于部分求值的Java动态优化系统如图1所示。该系统由Java程序变换、绑定时间分析、程序例化生成和应用接口生成等四个子系统组成。下面详细介绍各个子系统。

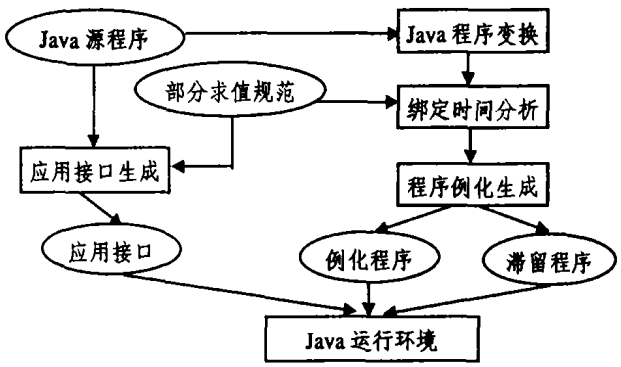

图1　基于部分求值的Java动态优化系统

3. Java程序变换

基于部分求值的Java动态优化系统的处理对象是Java语言源程序。为了降低处理工作的难度,有必要首先对Java语言源程序进行一定的变换,从而减少语言现象,简化语言结构。这一功能由Java程序变换子系统完成。通过程序变换,Java语言源程序被变换为等价的Java子集程序[2]。

Java变换子系统实现的具体任务包括:(1)减少控制流程语句种类:属于选择结构的switch-case语句由if语句实现;属于循环结构的for、do-while语句由while语句实现。(2)消除转移语句break、continue和return;其中,为了消除带有返回值的return语句,需要引入新的与返回值类型相同的类成员变量。(3)将多变量声明语句变换为若干单变量声明语句,同时消去其中变量初始化赋值表达式,另外生成新的赋值语句。(4)将多维数组变换为一维数组。(5)将赋值表达式、对象构造表达式分别变换为赋值语句、对象构造语句。(6)统计程序特征信息供BTA使用,如:各方法的参数和引用的类变量、各方法修改的类变量、各方法中调用的方法、递归调用的方法等。表1是一个是经过程序变换得到的Java子集源程序,方法

*)本研究得到国家自然科学基金(60173013)、教育部面向21世纪教学振兴计划和北京市自然科学基金(4982002)的资助。高红雨　讲师,主要从事软件方法、操作系统的研究和教学。廖湖声　教授,博士生导师,主要从事软件自动化、计算机语言和软件环境的研究与教学。王　众　讲师,主要从事程序变换和分析、数据结构的研究和教学。

power 用于求 m^n(m 和 n 是正整数)。

表1 一个 Java 子集源程序

```
public class MyCalc {
  public int rtn_power;
  public void power(int m, int n) {
    int pow;
    pow=1;
    while (n>0) {
      if ((n&1)!=0)
        pow*=m;
      m*=m;
        n>>=1;
      }
      rtn_power=pow;
    }
  }
```

4. 绑定时间分析

绑定时间分析(BTA,binding time analysis)是一种程序静态分析技术,用于程序设计语言部分求值的有效实现。本系统要求使用者编制部分求值规范文件,指明部分求值优化应用点信息,包括:被优化程序的类型,即本地应用程序、Applet 程序或远程方法调用(RMI,remote method invocation)程序;被优化方法,方法中哪些参数是不变量,引用了哪些不变量等。BTA 分析子系统根据部分求值规范中的应用点信息对 Java 子集程序进行分析,对于静态输入相关的表达式、语句、方法调用等各种程序结构标注为静态计算,也就是在部分求值阶段完成的计算;其余计算涉及的表达式、语句、方法调用将添加动态计算标注。BTA 子系统的输出为带有标注的 Java 子集程序。

为了更精确地进行 BTA 分析,本系统采用了一种适用于面向对象语言的 BTA 分析技术;和传统的过程型语言 BTA 分析相比,它不仅能够分析一般变量绑定时间的上下文敏感性(context sensitivity),而且能够分析数组变量、对象引用变量以及对象成员变量、数组元素等各种程序变量,从而将部分求值处理深入到复杂数据结构的内部,有效地扩大了部分求值的作用范围[2]。以上述 power 方法为例,设部分求值规范为:

MyCalc.power (int m = DYNAMIC, int n = STATIC) { }

这表示对 MyCalc 类的 power 方法进行优化,m 为动态参数,n 为静态参数;表2是 BTA 分析产生的标注程序,其中标注为 S 的部分在部分求值中完成,标注为 D 的部分代表剩余的计算保留在滞留程序中。

表2 一个带有 BTA 标准的源程序

```
public void power(int m^D, int n^S) {
  int pow^D;
  pow^D=^D1^S;
  while^S(n^S>^S0^S) {
    if^S(( n^S &^S1^S)!=^S0^S)
      pow^D*=^Dm^D;
    m^D*=^Dm^D;
    n^S≫=^S1^S;
  }
  rtn_power^D=^Dpow^D;
}
```

5. 程序例化生成

在 Java 动态优化系统的设计中,采用了一种新的程序例化技术与数据例化技术相结合的部分求值方法。这种方法以控制流图(control flow graph)的形式将前段计算的结果和优化后的控制转移信息保存在数据缓冲区内,后段计算依据数据缓冲区的控制流图进行工作,调用经过程序例化产生的程序模块,并直接引用前段计算的结果,完成其余计算。和传统的程序例化技术相比,这种方法不会带来程序膨胀;和传统的数据例化技术不同,这种方式能够完成控制转移计算的优化[1]。

以 power 方法为例,程序例化生成子系统首先对 BTA 分析产生的标注程序进行基本块划分(图2(a));然后构造带有静态、动态标志的标注控制流图(图2(b));最后根据标注控制流图生成例化程序(表3)和滞留程序(表4)。

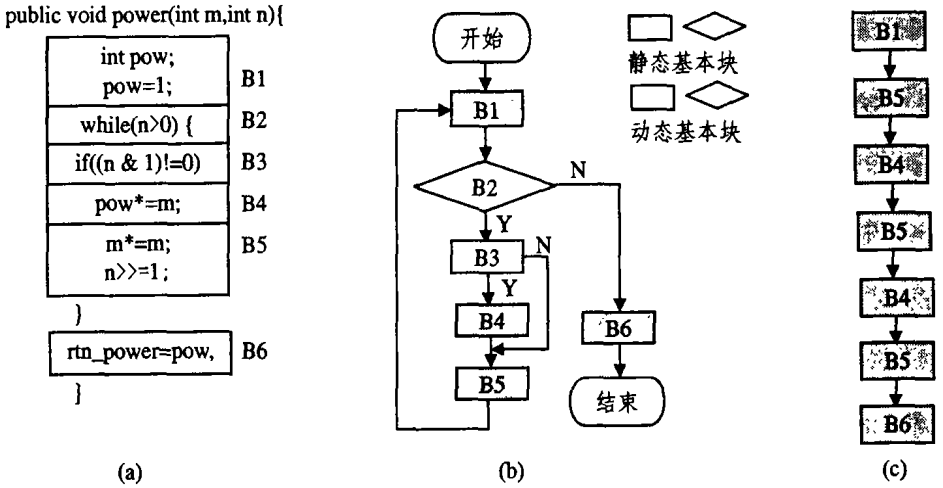

图2 方法的例化过程

例化程序先检查是否做过参数相同的例化,若是则返回已有结果(2～7行);否则,进行例化工作:完成静态基本块计算和静态转移控制(10、11、17行),根据给定的静态输入计算中间结果,将动态基本块加入到例化控制流图(8～9、12～13、15～16、19～20行),中间结果和例化控制流图被保存在数据缓冲区 Cfg 结构中。图2(c)是当 n 取值为6时,例化程序构造的例化控制流图。滞留程序包括所有动态基本块(各 case 语句下面的内容),它读取数据缓冲区里的中间结果和例化控制流图,并根据例化控制流图所规定的次序执行动态基本块。实际上,这种滞留程序构成了一个专用的虚拟机,例化控制流图表达的动态基本块执行顺序可以看作它的指令序列。

例化程序的构造与不变量的具体取值无关,它既可以在被优化程序的编译阶段执行,又可以在被优化程序运行时刻执行。因此这种例化方法具有动态优化的特点,应用范围广

泛。

表3 例化程序及其应用接口

```
1: public Cfg power_spec(int n){//例化程序
2:   n_power=new power_Name();
3:   n_power.n=n;
4:   Cfg cfg=Cfg.Lookup(n_power);
5:   if (cfg !=null)
6:     return cfg;
7:   cfg=Cfg.NewCfg(n_power);
8:   Block b1 = cfg.addBlock(1);
9:   cfg.Exit(b1.idx, b1.direct_next);
10:   while (n>0){
11:     if((n&1)!=0){
12:       Block b4 = cfg.addBlock(4);
13:       cfg.Exit(b4.idx, b4.direct_next);
14:     }
15:     Block b5=cfg.addBlock(5);
16:     cfg.Exit(b5.idx, b5.direct_next);
17:     n>>=1;
18:   }
19:   Block b6 = cfg.addBlock(6);
20:   cfg.Exit(b6.idx, b6.direct_next);
21: }
22: public void power_spec(int n) {//例化应用接口
23:   if (cfg1==null)//cfg1为类静态变量
24:     cfg1=power_spec(n);
25:}
```

表4 滞留程序及其应用接口

```
1: public void power_run2nd(int m, Cfg cfg){//滞留程序
2:   int p=0, pow;
3:   while (true){
4:     Block b = cfg.blocks [p];
5:     switch(b.label){
6:       case 1:pow=1;
7:              p=b.next;
8:              continue;
9:       case 4:pow*=m;
10:             p=b.next;
11:             continue;
12:      case 5: m*=m;
13:             p=b.next;
14:             continue;
15:      case 6:rtn_power = pow;
16:             p=b.next;
17:             continue;
18:    }
19:    break;
20:  }
21: }
22: public int power_run2nd(int m){//滞留应用接口
23:   power_run2nd(m, cfg1);
24:   return rtn_power;
25: }
```

6. 应用接口生成

应用接口生成子系统按照部分求值规范为被优化方法生成例化应用接口(表3中22～25行)和滞留应用接口(表4中22～25行),并将应用程序中对被优化方法的引用变换为对例化应用接口和滞留应用接口的引用(如表5)。

表5 对被优化方法引用处的变换

变换前	MyCalc myObj = new MyCalc();
	int [] a = new int [20];
	for (int i = 0; i <= 10; i++)
	  a [i] = myObj.power(i, 6);

变换后	MyCalc myObj = new MyCalc();
	int [] a = new int [20];
	for (int i = 0; i <= 10; i++) {
	  myObj.power_spec(6);
	  a [i] = myObj.power_run2nd(i);
	}

结束语 如果一个程序中由静态输入决定的计算和控制结构越多,那么采用部分求值技术进行优化所取得的效果可能越好;相反地,如果一个程序的计算几乎都取决于运行时刻动态输入,则部分求值优化收效甚微。但是,具体的应用通常包含一定数量的静态输入,开展部分求值优化具有实际意义。例如,按照上述方法,我们在一个绘制星图的应用程序[8]中选取了3个有关坐标计算的方法作为应用点进行优化,使其运行速度提高了20%。

滞留程序的性能代表了基于部分求值技术优化的效果。滞留程序可以看作是执行动态基本块的虚拟机。通过优化动态基本块的结构,可以提高滞留程序虚拟机的运行效率。首先,有些动态基本块之间的转移关系是确定的,不受静态输入的影响,因此可以将这样的基本块合并为一个大的块,从而减轻块间转移的负担。其次,对于经过程序例化生成处理之后结构完全相同的动态基本块,可以化简为一个基本块,减少了滞留程序中基本块的数目和代码的长度。采用这些策略,我们对快速傅立叶变换(FFT)、矩阵乘法(Matrix)、龙贝格积分法(Romberg)、三次样条插值法(Csi)和切米雪夫多项式(Cheb)的计算程序进行了优化,测试了优化前后的运行速度,结果如表6所示。

表6 速度测试

测试程序	速度比
FFT(M=32)	1.67
Matrix(N=25)	1.79
Romberg(N=8)	1.33
Csi(N=10)	1.69
Cheb(N=5)	2.13

下一步,我们将尝试直接生成字节码形式的滞留程序,减少解释例化控制流图的开销,达到更好的优化效果。

参考文献

1 廖湖声.基于流程图的数据例化与程序例化.计算机学报,2001,24(9)

2 廖湖声,童兆丰,王众.面向对象程序设计语言的一种绑定时间分析技术.软件学报,2003,14(3)

3 Le Meur A-F, Lawall J L, Consel C. Towards bridging the gap between progamming language and partial evaluation. In: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02), 2002

4 Berlin A, Surati R. Partial evaluation for scientific computing: The supercomputer toolkit experience. In: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, 1994

5 Andersen P. Partial evaluation applied to ray tracing. In: W. Mackens, S. Rump, eds. Software Engineering in Scientific Computing, Vieweg, 1996. 78～85

6 Pu C, et al. Optimistic incremental specialization: streamling a commercial operating system. In: Proc. of the Fifteenth ACM Symposium on Operating System Principle, Copper Mountain, CO, Dec. 1995

7 Marlet R, Thibault S, Consel C. Efficient implementations of software architectures via partial evaluation. Journal of Automated Software Engineering, 1999, 6(4): 411～440

8 Ladd S R 著,辛运帏,等译. Java 算法. 电子工业出版社, 1998