Research and Implementation of Bytecode Instruction Folding on Java Card

Tianhua Liu, Dawei Zhang, and Yichen Jiang

School of Computer and Information Technology, Beijing Jiaotong University, Beijing (dwzhang@bjtu.edu.cn)

***Abstract*—How to improve the run speed of the interpreter on Java Card with limited resources is an important issue. An optimized instruction folding algorithm is given in this paper. At first, the optimization scheme based on off-card and on-card virtual machine is given. The optimized instruction roles model for folding is defined and the grammar rules for folding are designed. Secondly, the byte codes dependency in instruction folding groups is analyzed. The instruction folding algorithm and the state diagram of folding process are designed and implemented. At last, the optimized algorithm is tested on SLE66CLX800PE smart card. The results show that the speed performance gain of instruction groups is 2.28 and the gain of typical financial Applets is 1.24.The instruction folding algorithm speeds up the execution of interpreter.**

***Keywords*—Java Card, stack computer, instruction folding**

Java 智能卡指令折叠优化算法的研究与实现

刘天华　张大伟　蒋逸尘

北京交通大学计算机与信息技术学院，北京，中国

摘　要　如何在有限计算资源环境下提高解释器的执行速度是 Java 智能卡研究中的关键问题。本文提出了一种改进的指令折叠优化算法，通过消除字节码指令流中的部分堆栈操作来提高解释器的执行效率。首先，给出了基于卡内、卡外虚拟机结构的折叠优化总体方案，定义了改进的指令角色模型并据此设计了指令折叠的文法规则。其次，分析了指令流中字节码序列的相关特性，采用移进规约分析器设计实现了指令折叠算法，给出了处理过程的状态转换图。最后，在 SLE66CLX800PE 芯片上对优化算法和传统方法的执行速度进行了对比测试，结果表明指令折叠组的优化性能增益达到了 2.28，典型金融应用的优化性能增益达到了 1.24。解释器的执行速度得到了较好的提高。

关键词　Java 智能卡，堆栈计算机，指令折叠

1．引言

Java 智能卡技术是在智能卡技术的基础上发展起来的，是 Java 技术和智能卡技术的结合[1]。由于智能卡上有限的计算资源和芯片结构的限制，目前只能使用软件解释器来执行 Java 卡虚拟机 JCVM(Java Card Virtual Machine)的字节码。虚拟机解释器的执行过程由如下三个部分组成：取指令操作数；执行指令的指定功能；分支跳转到下一条指令。其中一和三项构成了解释器执行的额外开销(overhead)[2]。JCVM 是一个堆栈计算机[3][4]，所有指令的操作数均来自堆栈，因此在寄存器型体系结构的处理器中运行的字节码解释器在每次运算前必须相应的操作数压入堆栈中，操作数和运算结果在堆栈和寄存器间的传递过程明显增加了指令的执行时间。

指令折叠概念最早由 Sun 公司所提出，他们将字节码定义为 6 种类型和 9 种折叠模式，指令折叠单元检查相邻的字节码并按照折叠模式进行折叠处理[5][6]。相关的多项研究工作将这一方法应用到 Java 处理器的设计方面[7~10]。这一方法的问题在于字节码序列不能够匹配预先定义的模式时就顺序执行，因此折叠效率较低。为了进一步提高折叠效率，L.C. Chang 提出了 POC 折叠算法[11]。局限性在于只能够处理连续的可折叠指令。为此，A. Kim 又提出了 Advanced POC 折叠算法[12]。改进算法给出了 4 种非连续的

中央高校基本科研业务费支持（资助号：2011JBM228）

指令折叠序列模式。其后的一些研究工作又进一步优化了指令角色模型和硬件实现技术[13~15]。一些优化工作基于 Java 智能卡虚拟机的分立结构展开，如 Gemplus 的代码压缩工作[16]。他们在卡外完成复杂的代码分析和注释(annotations)的生成，在卡上处理生成的注释并产生压缩的字节码。国内也有基于分立结构的解析优化研究工作[17][18]。但目前还没有指令折叠相关工作在这方面展开。

通过上述分析，基于 JCVM 的分立结构，我们提出了卡内、卡外相结合的指令折叠优化新方案。

2．指令折叠优化的总体方案

指令折叠基本思想是将一组相关的运算指令合并成一条单独的虚拟机伪指令来执行，从而避免堆栈访问操作数的中间过程，减少解释执行的额外开销。两个基本定义：

指令折叠：将一组面向堆栈操作的相关指令合并成一条单独的复合指令的过程。

指令折叠组：一组可折叠的相关指令。

依据 JCVM 的分立结构特点，我们提出了卡内、卡外相结合的指令折叠优化方案，如图 1 所示。

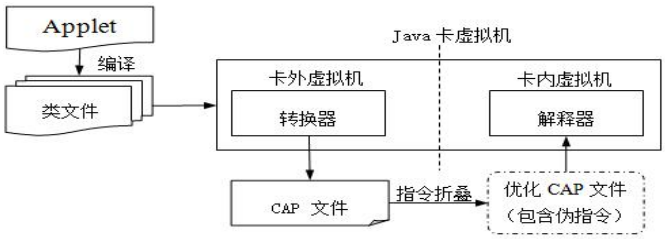

图 1 折叠优化的总体结构

虚线框内所示即为指令折叠的卡外部分，它将字节码序列(包含在 CAP 文件的方法组件中)中的一组相关的面向堆栈结构的可折叠指令(折叠组)折叠成一条单独的适于寄存器结构执行的两地址或三地制虚拟机伪指令。这些自定义的虚拟机伪指令在下载到卡上后由支持这些伪指令的解释器来进行解释执行。

在折叠优化总体方案基础上，采用软件方法设计并实现了指令折叠优化算法。

3．指令折叠的文法规则

我们扩展了可折叠指令的范围，增加了静态域和 this 对象访问指令。重新给出的 POC 模型的基本定义如下：

Java 卡虚拟机是一个堆栈计算机，因此根据对堆栈操作的相关特性可将字节码指令划分为三种基本角色类型：生产者 P、操作者 O 和消费者 C。它们的定义如下：

生产者 P：将常数、局部变量、静态域或 this 对象域压入操作数栈的指令，如 sconst_0，sload，getstatic_s，getfield_s_this。

操作者 O：从操作数栈栈顶获得操作数并执行相应操作的指令，其又可分为如下四类：

算逻运算 O_A：执行算逻运算并将结果写回操作数栈的指令。

控制转移 O_T：无条件控制转移类指令，如 jsr, athrow, invoke* ,goto, ret, sreturn, stableswitch。

改变堆栈状态类指令 O_S：无法加入指令折叠组的直接堆栈操作指令或其他改变堆栈状态类指令，如 pop，swap, dup，转换指令 i2s,s2i，非 this 实例域操作指令 getfield, putfield。

杂类 O_M：无法加入指令折叠组但不影响折叠判断的指令，如直接操作局部变量类指令 sinc。

消费者 C：从操作数栈弹出数据类指令，其又可分为如下两类：

变量存储 C_S：弹出栈顶数据存储到局部变量、静态域或 this 对象中，如 sstore，putstatic_s, putfield_s_this。

判断分支 C_B：弹出栈顶数据进行比较以进行程序控制分支，如 ifeq, ifnull。

指令折叠组中的指令经折叠处理后生成的虚拟机伪指令可分为三地制指令和两地址指令两种，其格式定义如下：

三地制折叠指令 I_3 的四元组表示为：I_3 = (O, S_1, S_2, T)，其中：O 为代表指令语义的操作码，S_1、S_2 为源操作数地址，T 为目的操作数地址。

两地制折叠指令 I_2 的三元组表示为：I_2 = (O, S_1, T)，其中：O 为代表指令语义的操作码，S_1 为源操作数地址，T 为目的操作数地址。

依据前述的基本定义，指令折叠过程可表示为由 P、O、C 类指令所构成的指令折叠组折叠成一条单独的 I_3 或 I_2 折叠指令的过程。相应于此过程的指令折叠文法 F 四元组的定义：F = (V_N, V_T, G, S)，其中：终结符号集 V_T = { x | x 为字节码指令}；非终结符号集 V_N = { POC 模型中的标记符号 P、O_A、C_S、C_B }∪ {折叠指令标记 I_3、I_2 }∪{ S }，S∈ V_N，为文法的识别符号。产生式集 G = {S → I_3 | I_2, I_3 → PPO_AC_S | PPC_B | PPO_A, I_2 → PC_S| PC_B, P → < P 类字节码序列 >, O_A → < O_A 类字节码序列 >, C_S → < C 类字节码序列 >, C_B → < C_B 类字节码序列 >}。

集合 G 中的产生式 I_3、I_2 的右部既为可识别的指令折叠组，指令的折叠过程即为由产生式右部到左部的规约过程，折叠指令的生成规则如下：

- P 指令中的操作数即为折叠指令中的源操作数
- 含 CS 指令折叠组生成的折叠指令，其目的操作数为 CS 指令的操作数；含 CB 指令折叠组生成的折叠指令，其

目的操作数（跳转的目的地址）为 CB 指令中的偏移量；PPOA 折叠后的目的操作数地址为 OA 的目的地址 - 操作数栈栈顶。

- 若指令折叠组中含 O 类指令，则折叠指令的语义由其决定，否则由 C 类指令决定。操作语义结合操作数类型共同生成折叠指令的操作码。

4．指令折叠算法的设计与实现

依据指令折叠文法规则可知，指令的折叠过程实际上可认为是对字节码序列的一个依据折叠文法自下而上的分析过程，指令的折叠过程即为由产生式右部到左部的规约过程，所有可规约为文法识别符 S 的序列即为一个指令折叠组。因此指令折叠的过程可分为两个基本步骤：首先，是完成对输入字节码序列依据折叠文法规则的指令折叠组的识别。其次，判断相邻指令折叠组的类型(常规序列、类型 1、2 或 3)，依据前述给出的每种类型相应的折叠规则来进行处理，从而保证算法输出的字节码和折叠指令混合序列执行的正确性。

指令折叠组的识别由一个移进-规约分析器来完成。首先设置一个字节码符号栈，依据系统的当前状态来决定将输入缓冲区中适当的字节码移入符号栈中，一旦栈顶符号串与产生式的右部相匹配则可进行规约。在此过程中应遵循如下原则：

折叠过程在一个程序基本块(具有单一入口和单一出口的字节码序列)内进行，C_B 和 O_T 类指令标示着基本块的结束，因此在处理完此类指令应清空符号栈，重置系统空闲状态。

- O_S 类改变操作数栈状态的指令无法进行进一步的折叠判断，遇到此类指令应清空符号栈，重置系统空闲状态。
- 不改变操作数栈状态的 O_M 类指令不进入符号栈进行折叠处理。
- 当遇到移进-规约冲突时采用移进策略，既给较长的产生式以较高的优先权。避免了硬件实现在判断过程中由于可处理的连续字节码指令长度限制造成的折叠组丢失(未判断出)的问题[11]。

基于移进-规约分析器的指令折叠组识别算法的有限状态机模型 IFA 的数学表述如下：

IFA 为一五元组：IFA =(Q，Σ，f，q_0，E)，其中：Q 为移进-规约分析器状态的有限集合，除空闲状态和终止状态外，遇到 P、O、C 类指令系统转入相应的状态，因此状态集合如下：Q = { 空闲状态，P 状态，O_A 状态，C_B 状态， C_S 状态 }，Σ为有限输入符号集，在此即为文法 F 中的终结符号集 V_T。 f 为状态转换函数，定义域为状态集与输入符号集的笛卡尔乘积，值域为状态集：f：Q ╳Σ → Q，具体的图形表述请参见图 4。q_0∈Q 是状态机的初态，在此 q_0 = 空闲状态。E 为终止状态，表示一个基本块指令折叠的结束。图 2 采用 UML 建模语言中的状态图对 IFA 的状态转换进行了图形描述。

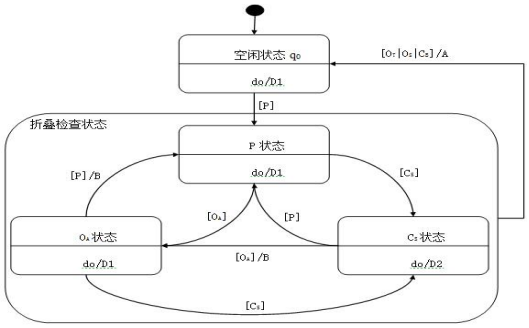

图 2 IFA 状态转换图

图中字母 A 代表依据折叠文法规则进行指令折叠组识别并清空符号栈的动作, B 代表指令折叠组识别的动作。D1 代表相符号栈中移入字节码的活动，D2 代表依据折叠文法规则进行指令折叠组识别的活动。为了表述的清晰，使用 UML 语言中的基状态的概念，为 Q 集合中的 P 状态、O_A 状态和 C_S 状态创立一个基状态“折叠检查状态”。Q 集合中的这些状态为“折叠检查状态”的子状态，继承了基状态的所有状态转换，即当遇到 O_T、C_B 和 O_S 指令时执行动作 A 并转入初始状态。

在完成指令折叠组的识别后，进行折叠过程中必须进行折叠组类型和数据相关的判断，依据相应的折叠规则来处理，并调整输出指令顺序以保证执行的正确性。因此在处理过程中设置折叠缓存队列 Q_F，将需进行相关性判断的折叠组加入此队列并在输出时移出。

5．指令折叠优化的性能增益

我们将经过折叠优化处理和未优化处理的 Java 智能卡 Applet 在 Infineon 公司 SLE66CLX800PE 智能卡上运行并进行比较，评估优化算法在字节码执行速度上的性能增益。

前述分析可知，取操作数和指令跳转是 Java 字节码解释器运行过程中的主要额外开销。因此，考虑指令折叠在这两方面所带来的性能增益。将属于不同指令折叠组 I_2 和 I_3 的字节码序列作为测试用例并将它们定义为五类折叠组：折叠组Ⅰ：可折叠为 I_2 的 PC_S 折叠组。折叠组 II：可折叠为 I_2 的|PC_B 折叠组。折叠组Ⅲ：可折叠为 I_3 的 PPO_AC_S 折叠组。折叠组Ⅳ：可折叠为 I_3 的 PPO_A 折叠组。折叠组 V:可折叠为 I_3 的 PPC_B 折叠组。测试用例取操作数和指令跳转的执行周期数作为测试结果，如表 1 和表 2 所示：

表 1 I2 指令折叠组取操作数和指令跳转优化前后的比较

I_2 Folding Group	Access arguments(cycles)		Instruction Dispatch(cycles)	
	un-optimized	optimized	un-optimized	optimized
I	70	30	30	15
II	35	15	30	15

表 2 I_3 指令折叠组取操作数和指令跳转优化前后的比较

I_3 Folding Group	Access arguments (cycles)		Instruction Dispatch(cycles)	
	un-optimized	optimized	un-optimized	optimized
III	105	44	60	15
IV	105	44	45	15
V	70	30	45	15

在智能卡芯片主频为 30MHz 的情况下，测试折叠组的全部可能组合在折叠前后的平均执行时间的比较如表 3:

表 3 优化前后的指令执行时间的比较

Folding Group	Ⅰ	II	Ⅲ	Ⅳ	V
Un-optimized(μs)	5.47	5.82	15.67	12.73	11.73
Optimized(μs)	2.79	3.01	5.86	5.86	5.02

定义指令折叠优化所带来的性能增益 Pg 为：

$$P_g = \frac{\text{优化前的指令执行时间}}{\text{优化后的指令执行时间}} = \frac{5.47+5.82+15.67+12.73+11.73}{2.79+3.01+5.86+5.86+5.02} = \frac{51.42}{22.54} = 2.28$$

同时，我们也选择了中国人民银行金融集成电路卡借记/贷记规范中的典型交易命令作为测试用例，测试比对结果如表 4 所示：

表 4 优化前后的命令执行时间的比较

Commands	IA	GPO	GAC	RR	VP	SA	SP
Un-optimized(ms)	132.1	38.7	102.5	9.3	45.5	20.2	27.1
Optimized(ms)	108.5	31.2	78.5	8.6	34.2	17.6	22.4

定义指令折叠优化所带来的性能增益 P_g 为：

$$P_g = \frac{\text{优化前的指令执行时间}}{\text{优化后的指令执行时间}} = \frac{132.1+38.7+102.5+9.3+45.5+20.2+27.1}{108.5+31.2+78.5+8.6+34.2+17.6+22.4} = \frac{375.4}{301} = 1.24$$

6．结论

本文提出了一种改进的指令折叠优化算法以减少解释执行中堆栈操作带来的额外开销。测试结果表明，指令折叠通过减少堆栈操作和指令合并的方法，有效地提高了取操作数和指令跳转执行效率，优化了卡片运行性能。

参考文献(References)

[1] Z. Chen, R. Di Giorgio. Understanding Java Card 2. 0. 1998. http://www.javaworld.com/javaworld/jw-03-1998/jw-03-javadev .html.

[2] M. Anton Ertl. Stack caching for interpreters. Proc of the ACM SIGPLAN conf on Programming language design and implementation. New York: ACM, 1995:315-327

[3] Sun Microsystems Inc. Java Card 3.0 Virtual Machine Specification. 2008. http://java.sun.com/javacard/3.0.1/specs.jsp H. Simpson, *Dumb Robots*, 3rd ed., Springfield: UOS Press, 2004, pp.6-9.

[4] P. Koopman. Stack Computers: The new wave. 1989. http://www.ece.cmu.edu/~koopman/stack_computers/contents.html

[5] M. O'Connor, M. Tremblay. PicoJava-I: The Java Virtual Machine in Hardware. IEEE Micro, 1997, 17(2):45–53

[6] Sun Microelectronics. PicoJava-II in an FPGA. Proc of the 5th Int workshop on Java technologies for real-time and embedded systems. New York: ACM, 2007:213-221

[7] S. Liu, Z. Mao, Y. Ye. A Study on Java Card and Its Implementation. Microelectronics. 2000, 30(6):402~405

[8] T. Wang, Z. Mao, Y. Ye. The study and realization of instruction folding in Java processor. Journal of computer research & development. 2000, 37(1): 66~72.

[9] X. Tang, X. Yang. The design of an embedded Java microprocessor core for smart cards. Microelectronics, 2000, 30(6):382～386.

[10] T. Wang, Z. Mao, Y. Ye. Research on a dedicated CPU architecture for Java IC card. Acta electronica sinica, 2000, 28(11): 77~80.

[11] L. C. Chang，Stack Operations folding in Java Processor. IEEE Proc of Computers Digital Techniques, 1998, 145(5): 333~340.

[12] A. Kim. Advanced POC Model-Based Java Instruction Folding Mechanism. Proc of the 26th Euromicro Conf. Piscataway, NJ:IEEE, 2000: 332~338.

[13] Kim. An Advanced Instruction Folding Mechanism For A Stackless Java Processor. Proc of the 2000 IEEE Int Conf on Computer Design: VLSI in Computers & Processors. Piscataway, NJ: IEEE, 2000: 565~567.

[14] T. Yiyu, Y.C. Hang. Design and Implementation of a Java Processor. IEEE Proc of Computer and Digital Techniques, 153(1), 2006:20–30.

[15] T. Yiyu. An Instruction Folding Solution to a Java Processor. LNCS 4672: Proc of IFIP Int Conf on Network and parallel computing 2007, Berlin: Springer, 2008:415-424.

[16] G. Bizzotto, G. Grimaud. Practical Java Card bytecode compression. Proc of RENPAR14 / ASF / SYMPA, 2002.

[17] W. Jin, Q. Chang, C. Li, et al. JCVM resolving optimization design and implementation. Journal of Beijing University of aeronautics and astronautics. 2004, 30(12): 1204-1208.

[18] D. Zhang, W. Ding. Optimization of resolution on Java card. Journal of Beijing University of aeronautics and astronautics. 2009, 35(1): 78-81.