ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.7, July 2008, pp.1581−1589
DOI: 10.3724/SP.J.1001.2008.01581
© 2008 by Journal of Software. All rights reserved.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于插桩分析的Java虚拟机自适应预取优化框架*

邹 琼[1,2+], 伍 鸣[2], 胡伟武[2], 章隆兵[2]

[1](中国科学技术大学 计算机科学与技术系,安徽 合肥 230027)

[2](中国科学院 计算技术研究所 系统结构重点实验室,北京 100190)

An Instrument-Analysis Framework for Adaptive Prefetch Optimization in JVM

ZOU Qiong[1,2+], WU Ming[2], HU Wei-Wu[2], ZHANG Long-Bing[2]

[1](Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)

[2](Key Laboratory of Computer System and Architecture, Institute of Computing Technology, The Chinese Academy of Sciences, Beijing 100190, China)

+ Corresponding author: E-mail: zouqiong@ict.ac.cn

Zou Q, Wu M, Hu WW, Zhang LB. An instrument-analysis framework for adaptive prefetch optimization in JVM. *Journal of Software*, 2008,19(7):1581−1589. http://www.jos.org.cn/1000-9825/19/1581.htm

Abstract: Accessing to heap data brings main overhead for Java application. VM (virtual machine) researchers utilize prefetch or garbage collection to improve the performance, with the help of collected information of accesses to heap. The general methods to collect information are sampling and program instrumentation, however, they can't satisfy fine granularity and low overhead simultaneously. To satisfy these two requirements, this paper proposes an instrument- analysis framework for adaptive prefetch optimization in JVM, which instruments code to collect profiling information, and guide to dispatch code and perform prefetch according to feedback. The experimental results show that it achieves up to 18.1% speedup in industry-standard benchmark SPEC JVM98 and Dacapo on the Pentium 4, while the overhead is less than 4.0%.

Key words: instrument; dispatch; adaptive; prefetch optimization

摘 要: 对堆上数据的频繁访问是 Java 程序的主要开销,为此,研究者们通过虚拟机收集堆上数据访问的信息,而后采用预取或垃圾收集来改进内存性能.常用的收集方法有采样法和插桩法,但二者无法同时满足细粒度和低开销的要求.针对这两个要求,提出基于插桩分析的虚拟机自适应预取框架,该框架通过插桩收集信息,并根据程序运行时的反馈自适应地调整插桩并进行预取优化.实验结果表明,自适应预取优化在 Pentium 4 上对 SPEC JVM98 和 Dacapo 有不同程度的提高,最高的达到了 18.1%,而开销控制在 4.0%以内.

关键词: 插桩;反插桩;自适应;预取优化

中图法分类号: TP314 **文献标识码**: A

---

* Supported by the National Natural Science Foundation of China under Grant Nos.60673146, 60703017, 60736012, 60603049 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant Nos.2006AA010201, 2007AA01Z114 (国家高技术研究发展计划(863)); the National Basic Research Program of China under Grant No.2005CB321600 (国家重点基础研究发展计划(973)); the Beijing Natural Science Foundation of China under Grant No.4072024 (北京市自然科学基金)

Received 2006-12-11; Accepted 2007-03-29

© 中国科学院软件研究所 http://www.jos.org.cn

Java 是当今主流的程序开发语言之一,它具有易移植性、安全性、动态性等优点.在 Java 的世界中,Java 虚拟机起着重要的基础作用,是程序平台无关和拥有良好性能的关键.

Java语言完全面向对象,因此,对堆上数据的频繁访问是Java程序的主要开销.为了改善Java程序中内存访问的性能,虚拟机研究者们通过虚拟机收集堆上数据访问的信息,一方面,动态地对堆上数据进行预取[1,2],以减小load-use延迟.另一方面,通过垃圾收集将对象进行重排[3−5]以改善程序的空间局部性.

目前在Java虚拟机中广泛使用的收集方法有采样法和插桩法[6].采样法开销小,适用于收集粗粒度的信息[7],比如方法调用次数的统计.但是,如果想得到细粒度的信息,此法是不可取的.插桩法与采样法相反,它可以帮助研究者们收集到细粒度的信息,但开销很大,有很多研究者采用插桩法来帮助离线的数据分析[8],也有一些研究者在程序开始运行的一段时间收集信息以降低开销[9].但是,程序在刚开始的行为不足以反映程序将来的行为[6],尤其是堆上的数据访问.如果可以根据当前收集的信息判断出程序的行为呈现出一定的规律性,此时便可以停止插桩,这种自适应地插桩的方法可以保证收集信息的质量,同时不会带来太大的开销.

因此,本文提出一种基于插桩分析的自适应预取优化框架来改善Java程序中内存访问的性能.该框架通过插桩收集信息,根据程序运行时的反馈自适应地调整插桩并进行预取优化.实验结果表明,自适应预取优化使得SPEC JVM98[10]和Dacapo[11]的性能有不同程度的提高,其中最高的达到 18.1%.

本文第 1 节是背景介绍.第 2 节是整体框架设计.第 3 节描述基于插桩分析的自适应预取优化算法.第 4 节是实验数据分析.第 5 节是结论和未来展望.

1 背景介绍

目前,主流的虚拟机有SUN JDK[12],BEA JRockit JDK[13],JIKES RVM[14]和DRLVM(dynamic runtime layer virtual machine)[15],其中,SUN JDK和BEA JRockit JDK都是商业化的产品,性能优秀,但是模块层次不够清晰,针对性太强,因此可研究性较差.JIKES RVM和DRLVM都是致力于研究用途的虚拟机,所不同的是,JIKES RVM采用Java语言编写而成,DRLVM采用C++和少量汇编语言编写而成,并且相对于JIKES RVM而言,它具有更好的结构层次和优秀的模块化等特征.因此,本文选择在DRLVM上展开研究.

图 1 中给出的是 DRLVM 的各个模块以及模块之间的接口.DRLVM 由 VM Core,Garbage Collector,Thread Manager,Porting Layer,Execution Manager 和 Execution Engine 这 6 部分组成.其中 Execution Engine 是 DRLVM 中的执行引擎,包括即时编译器和解释器,主要负责将 Java 程序编译成本地代码.本文提出的基于插桩分析的自适应预取优化框架实现在即时编译器中,这使得我们的工作更具挑战性,因为即时编译器生成的代码远远优于解释器生成的代码.

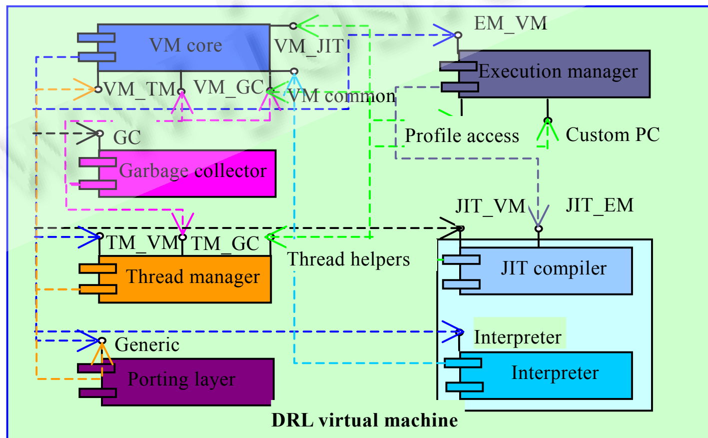

Fig.1 Major DRLVM components

图 1 DRLVM 的主要组成

2 整体框架设计

图 2 中描述的是整个自适应预取优化框架的结构图,其中白色的方框为框架的各个部件,主要包括 5 部分:Patcher、Trace 缓冲区、Trace 分析器、Dispatcher 和 Prefetcher.本小节描述各个组成部分的作用,框架的细节将在第 3 节中介绍.

1) Patcher:用来给应用程序进行插桩的工具,由虚拟机调用.
2) Trace 缓冲区:一个 64K 字节的缓冲区,缓冲区中每一项大小为 16 字节.缓冲区自顶向下写入,如图 3 所示,定义缓冲区的顶部为 trace_buf_top,底部为 trace_buf_bottom.缓冲区以下的页面标记为不可写,当应用程序往缓冲区中存入数据时,一旦溢出,便会产生段错误信号.
3) Trace 分析器:自适应预取优化框架的主要组成部分,它对 Trace 缓冲区中数据进行分析,如果分析的结果表明当前堆上访问存在一定的规律性时,那么它将唤醒 Dispatcher 和 Prefetcher.整个过程不需要用户的参与,根据程序的运行结果自动地进行分析和调整,是一种自适应的过程.虚拟机一旦接收到段错误的信号,便会唤醒该分析器.
4) Dispatcher:用来对应用程序解除插桩的工具,由 Trace 分析器根据程序运行的实际情况自适应地调用.
5) Prefetcher:用来对应用程序进行预取优化的工具,由 Trace 分析器根据程序运行的实际情况自适应地调用.

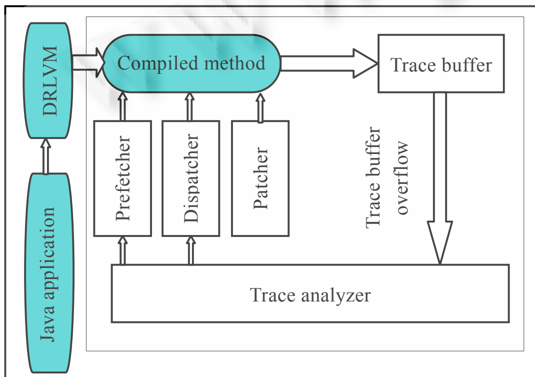

Fig.2 Framework of adaptive prefetch optimization
图 2 自适应预取优化框架图

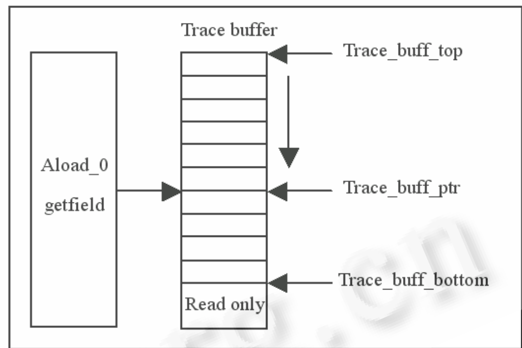

Fig.3 The structure of the Trace buffer
图 3 Trace 缓冲区的结构

3 基于插桩分析的自适应预取算法

传统的插桩法可以收集所有堆上数据的访问信息,这样做,极大地提高了预取和堆上数据重排的准确率.但是另一方面,此法会带来巨额的开销.因此,本文提出基于插桩分析的自适应预取算法,主要思想在于,根据程序运行时反馈的信息对插桩进行调整,这样可以降低开销,同时利用收集到的信息来指导 Prefetcher 进行预取优化.为了进一步降低算法的开销,我们将忽略程序启动时堆上的数据访问,因为这些访问不会给应用程序带来太大的影响.

3.1 算法面向的指令

堆上的数据访问可以分为 3 类:虚拟方法表、对象的域和数组元素,其中,对象的域和数组元素的访问超过 90%[16].本文中所指的堆访问仅指对象的域和数组元素,而插桩算法也仅面向对象和数组.Java程序中使用Java 字节码作为中间表示[17],其中可以用来表示对象和数组访问的指令有:getfield,putfield,aaload,aastore,baload, bastore等.本文称这些指令为对象相关的指令.当一个基本块中包含对象相关的指令时,该基本块为对象相关的基本块.

© 中国科学院软件研究所 http://www.jos.org.cn

3.2 算法使用的数据结构

首先给出规律的对象访问的定义:对于一条特定的对象相关的指令,它将以固定的步长去访问堆上的某个对象,同时称这条指令是规律的指令,该步长是热点步长.图 4 是本算法中使用的主要数据结构.

如图 4 所示,该数据结构主要由两部分组成:一部分是 Trace 缓冲区,Trace 缓冲区中的每一项分别包括被记录信息的指令地址、该指令访问对象的基地址以及偏移,总共有 4 096 项;另一部分是步长访问表,该表是一个哈希表,采用指令地址作为键值对其进行索引,数组中记录的是该条指令所使用的各种步长的频率.倘若其中某个步长出现的频率超过 Trace 缓冲区长度的一半,我们就认为该指令是规律的指令,相应的步长为热点步长.整个数据结构中需要权衡考虑的因素是步长数组的长度,如果增加数组的长度,则可以在一个更大的范围内选择热点步长,但同时也会带来空间和时间上的开销,第 4.3 节中将会对此进行讨论.

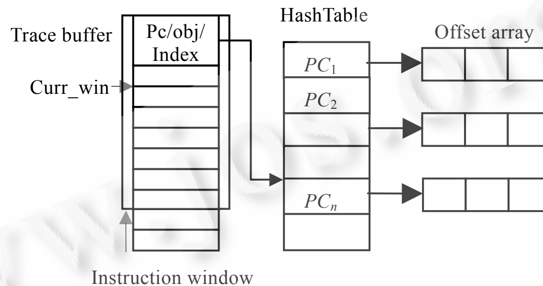

Fig.4 Data structure of adaptive instrumentation algorithm

图 4 自适应插桩算法中使用的数据结构

3.3 自适应预取优化框架工作流程图

第 2 节给出了自适应预取优化的框架图,整个框架的工作流程描述如下:

```
profiled code for objects:
    push edx
    mov %edx, trace_buff_ptr
    mov (%edx), objptr_reg
    mov 4(%edx), ip
    mov 8(%edx), index_reg
    add %edx, 4
    mov trace_buff_ptr, edx
    pop edx
```

Fig.5 Instrumented code for collecting the information of objects

图 5 统计对象信息的代码

1) 在编译过程中,虚拟机调用 Patcher 对对象相关的指令进行插桩,插桩的代码如图 5 所示;
2) 应用程序在运行的过程中,将收集到的对象信息写入到 Trace 缓冲区中;
3) Trace 缓冲区溢出,发出用户段错误例外,虚拟机接收到信号,唤醒 Trace 分析器;
4) Trace 分析器调用自适应预取优化算法;
5) 回到步骤 2)继续执行.

这是一个循环执行的过程:运行、收集信息、分析信息、反插桩和预取优化、运行、收集信息、...,程序在运行过程中不断地将相关信息反馈,反插桩和预取优化又将引起程序行为的变化.

3.4 算法描述

基于插桩分析的自适应预取算法主要实现了 4 个方面的功能:1) 计算一条指令的访问步长;2) 更新步长访问表中该步长出现的频率,并检测该步长是否成为热点步长;3) 通知 Dispatcher 取消对规律指令的插桩;4) 指导 Prefetcher 对规律的对象访问进行预取优化.

Trace 缓冲区中的每一项对应一个指令窗口,该窗口用来帮助计算访问步长,窗口中的第 1 条指令采用 *Curr_wind* 来标记,大小为 12(窗口大小设置为 12 时,探测热点步长的精度达到 95%).算法流程如图 6 所示,具体描述如下:

© 中国科学院软件研究所 http://www.jos.org.cn

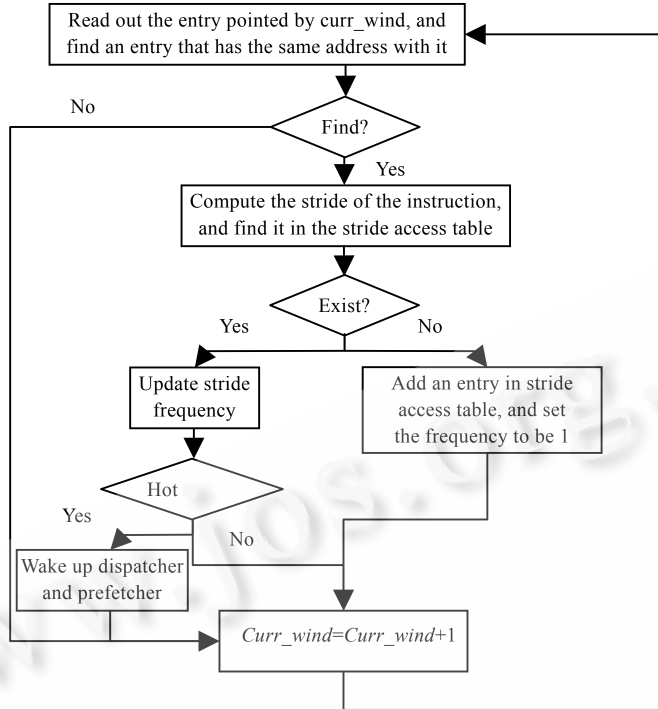

Fig.6 Adaptive prefetch optimization algorithm description

图 6 自适应预取优化算法流程图

1) 读取 *Curr_wind* 所指向的项中指令地址,记为 ip0,在指令窗口中搜索指令地址为 ip0 的项,如果找到跳至步骤 2),否则跳至步骤 4);
2) 将两项中指令访问对象的偏移相减得到步长,然后由 ip0 索引到步长访问表,若不存在,则跳至步骤 3);否则,更新相应步长的频率,同时判断该步长是否热点步长,若是,则跳至步骤 5),否则,跳至步骤 4);
3) 在步长访问表中添加 PC 为 ip0 的项,同时将该步长频率设置为 1,跳至步骤 4);
4) *Curr_wind*←*Curr_wind*+1,跳至步骤 1);
5) Dispatcher 根据指令的地址停止对当前指令的插桩,跳至步骤 6);
6) Prefetcher根据指令的地址和热点步长计算出预取的数据地址,指定对象的地址为addr,热点的步长为stride,那么,最后的预取指令为*prefetch*(*addr*+*d*×*stride*),其中*d*为预取距离[1].

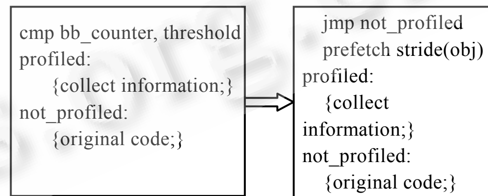

Fig.7 Code transition

图 7 代码的转换

经过步骤 5)、步骤 6)优化后的代码如图 7 所示,最后跳至步骤 4).

4 实验评估

本节主要对两方面进行评估,一方面是插入预取指令所带来的性能的提高,另一方面是框架的开销,并将其与传统的插桩方法进行对比.

4.1 实验方法

本文采用SPEC JVM98 和Dacapo中的部分程序作为测试程序,并且在SPEC JVM98 上运行最大的数据集

© 中国科学院软件研究所 http://www.jos.org.cn

(100),因为在运行较小的数据集(1,10)时,收集到的数据不具有代表性.表 1 中列出了实验中所使用的测试程序.运行环境是 2.6GHz. Pentium IV PC[18],512MB内存,256KB,8 路组相联 2 级Cache,其中,行大小为 128 字节,一级Cache大小是 8KB,4 路组相联,行大小为 64 字节,操作系统是Linux Fedora 4.Java编译器使用的是SUN JDK1.5.

Table 1 Description of the benchmarks in our experiment

表 1 实验中使用的测试程序

Program	Description
Compress	Modified Lempel-Ziv method
Jess	Java expert shell system
db	Database access program
Jack	Java parser generator
Bloat	A bytecode-level optimization and analysis tool for Java
Fop	An output-independent print formatter
Ps	A postscript file interpreters
pmd	A source code analyzer for Java

4.2 预取的效果评估

4.2.1 预取距离

Trace 分析器一旦发现规律的对象访问,便会唤醒 Prefetcher,同时将相应的指令地址和指令访问的热点步长传递给它,Prefetcher 插入预取指令以减小 load-use 延迟.

Prefetcher接收到Trace分析器传递来的指令地址和指令访问的热点步长,还要知道预取距离才能得到最终应该插入的预取指令.预取距离一个通用的公式是[17]:$D=l/w$,其中,D是预取要提前的循环次数,l是Cache的失效开销,单位是处理器时钟周期数,w是循环体的一次执行时间的估计,也可以认为是一条指令两次执行之间时间的估计,单位也是处理器时钟周期数.在我们的运行环境中,l=80,w=5×2.我们采用的预取指令是prefetchnta.

4.2.2 预取带来的性能提高

图 8 给出的是预取带来的性能提升.从图中可以看出,采用了预取优化的程序性能提升了 18.1%~1.5%.提升最多的程序是 ps,达到了 18.1%,其次是 jess 和 fop,也有 13.8%和 8.3%的提高.在程序中,规律的对象访问主要来自 3 个方面:一是循环内不同的指令;二是循环间相同的指令;三是频繁执行的小方法.我们对这些程序收集到的堆访问信息进行分析发现,这些访问主要是来自后面两点,因为自适应预取算法只是针对同一条指令进行预取.如果程序中规律的对象访问来自于循环内不同的指令,那么本文中的算法不会有显著的效果.db 就是这样一个程序,它有 85%的时间运行在一个循环中,并且该循环针对较大的记录结构进行排序,因此,在不同的指令之间存在规律的对象访问,而相同指令的规律访问显得较少.所以,自适应预取优化对 db 只有 1.5%的提高.

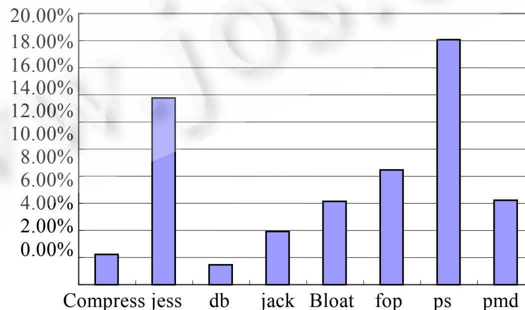

Fig.8 Performance speedup using prefetch

图 8 预取带来的性能提升

4.2.3 Cache 失效和 TLB 失效

本节着重比较插入预取指令后与原始的程序(图 9 中的baseline列)的内存存储的性能.我们采用的性能分析器是Intel VTune Performance Analyzer[19] Version 8.0,度量指标是MPI,即平均每条指令的失效率.图 9 给出了一

© 中国科学院软件研究所 http://www.jos.org.cn

级数据Cache、二级数据Cache和数据TLB的MPI.

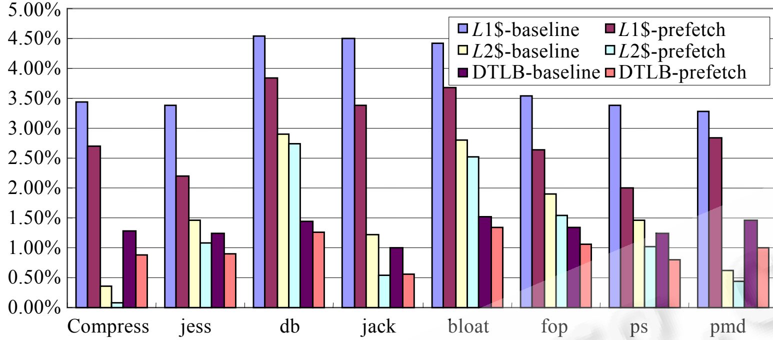

Fig.9 Memory performance comparison: Baseline vs. prefetch

图 9 预取前后内存的性能比较

Cache 和 TLB 失效的原因在于:连续的访存指令所访问的不同内存块地址恰好映射到 Cache 或者是 TLB 的同一表项上.预取不能减少 Cache 或者是 TLB 替换的次数,它只能减少 Cache 或者是 TLB 失效事件的发生.我们用来考察的指标是 MPI,插桩增加了最终提交的指令数,表 2 是最终插入的预取指令的条数,远远小于提交的总指令数,所以增加的指令数可以忽略不计.

Table 2 Number of instrumented prefetch instructions

表 2 插入的预取指令条数

Compress	56
jess	37
db	26
jack	21
bloat	41
fop	63
ps	24
pmd	48

从图 9 中可以看出,预取优化不同程度地降低了大部分测试程序的一级数据 Cache、二级数据 Cache 和数据 TLB 的 MPI.对于 compress 和 db 这两个程序,预取减小了它们的数据 Cache 的 MPI,同时却增加了它们的数据 TLB 的 MPI,数据 TLB 失效率的增加掩盖了数据 Cache 命中率下降所带来的好处,这也是两个程序性能提高不多的原因,由此可见,减小数据 TLB 失效率也是提高性能的一个重要因素.

4.3 自适应插桩算法的评估

4.3.1 步长数组的评估

图 10 给出各种步长出现频率的分布图,可以看出,值为–1,0,1,2 的步长占了所有步长的 90%以上.因此,我们将步长数组长度设为 20,大于 10 或小于–9 的步长将被忽略.

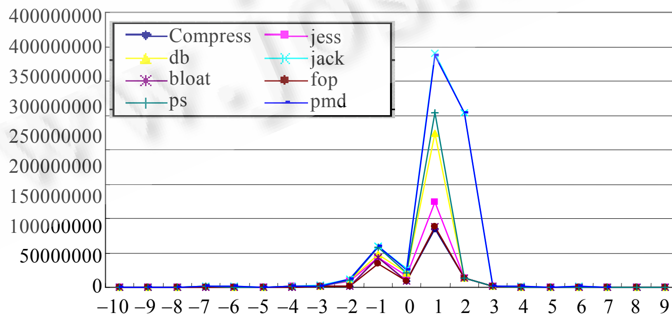

Fig.10 Stride of array access distribution

图 10 各种步长的分布图

4.3.2 自适应预取优化算法的开销

图 11 给出了自适应预取算法的开销与传统的插桩方法的开销对比,其中,自适应预取算法的基本块阈值设为 0,传统的插桩方法阈值设为 15 000.从图中可以看出,传统的插桩方法开销不菲,其中,compress 的开销达到了

© 中国科学院软件研究所 http://www.jos.org.cn

55.6%,fop 的开销有 40%,最少的 jess 开销也有 6%,如此大的开销说明,传统的插桩方法是不可取的.相比之下,自适应预取算法的开销大为降低,compress 开销为 3%,fop 为 2.6%;相反地,原先开销并不是很大的 jess 和 jack,其开销降低的幅度远不如 compress.这说明,这两个程序中规律的对象访问数目不多,却很集中.参看表 1,jess 中规律的对象访问有 37 个,jack 有 21 个,而 compress 和 fop 中规律的对象访问分别有 56 和 63 个,是 8 个程序中最多的一个,这也是它们的开销降低最多的原因.

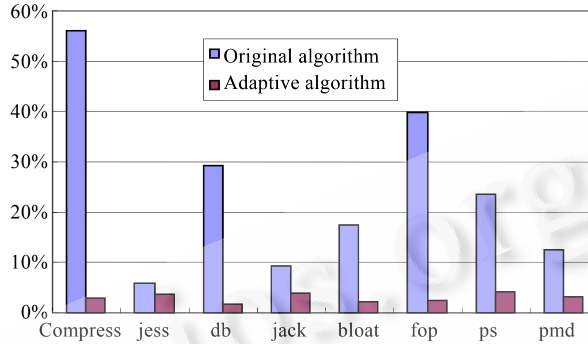

Fig.11 Overhead comparison between original algorithm and adaptive algorithm

图 11 自适应预取优化算法开销与传统插桩法开销的对比

自适应预取算法的开销来自于 3 个操作:一是填写 Trace 缓冲区;二是分析 Trace 缓冲区;三是插入预取指令.表 3 给出这 3 类操作的开销占所有开销的百分比.可以看到,填写 Trace 缓冲区的开销所占比重为 65.4%,因为在 3 类操作中它最为频繁.分析 Trace 缓冲区占的比重达到 25.3%,预取指令占的比重为 9.3%.3 类开销的分布在所有的测试用例中是保持一致的.

Table 3 Distribution of adaptive algorithm's overhead

表 3 自适应预取算法开销分布

Name	Percentage (%)
Fill the trace buffer	65.4
Analyze the trace buffer	25.3
Prefetch Instructions	9.3

5 结论和未来展望

本文提出一个基于插桩分析的自适应预取优化的框架,其中,自适应地消除插桩和指导预取的方法有效地降低了开销,指导预取,使得 SPEC JVM98 和 Dacapo 的性能有 1.5%~18.1%的提高,并且将开销控制在 4%以内.

本文提出的算法也存在一定的局限性,它只能用于探测同一指令的固定步长访问,而不能探测不同指令间的固定步长访问,因此,推广该算法是我们的工作之一.目前有很多研究者通过收集对象信息来指导堆上数据的布局,这些工作很好地改善了空间局部性.如何把这些工作加入本文的框架,也是将来要做的工作.

References:

[1] Inagaki T, Onodera T, Komatsu H, Nakatani T. Stride prefetching by dynamically inspecting objects. In: Crocker R, ed. Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation. New York: ACM Press, 2003. 269−277.

[2] Adl-Tabatabai AR, Hudson RL, Serrano MJ, Subramoney S. Prefetch injection based on hardware monitoring and object metadata. In: Pugh W, Chambers C, eds. Proc. of the ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation. New York: ACM Press, 2004. 267−276.

[3] Huang X, Blackburn SM, McKinley KS, Moss JEB, Wang Z, Cheng P. The garbage collection advantage: Improving program locality. In: Vlissides J, Schmidt D, eds. Proc. of the 19th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications. New York: ACM Press, 2004. 69−80.

© 中国科学院软件研究所 http://www.jos.org.cn

[4] Chilimbi TM, Larus JR. Using generational garbage collection to implement cache-conscious data placement. In: Peyton S, Jones R, eds. Proc. of the 1st Int'l Conf on Int'l Symp. on Memory Management. New York: ACM Press, 1998. 37−48.

[5] Chilimbi TM, Davidson B, Larus JR. Cache-Conscious structure definition. In: Ryder BG, Zorn BG, eds. Proc. of the ACM SIGPLAN'99 Conf. on Programming Languages Design and Implementation. New York: ACM Press, 1999. 13−24.

[6] Arnold M, Hind M, Ryder BG. Online feedback-directed optimization of Java. In: Lbrahim M, Matsuoka S, eds. Proc. of the 17th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications. New York: ACM Press, 2002. 111−129.

[7] Holzle U, Ungar D. Reconciling responsiveness with performance in pure object-oriented languages. ACM Trans. on Programming Languages and Systems, 1996,18(4):335−400.

[8] Pettis K, Hansen RC. Profile guided code positioning. In: Fischer BN, ed. Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation. New York: ACM Press, 1990. 16−27.

[9] Paleczny M, Vick C, Click C. The Java hotspot server compiler. In: Wold S, ed. Proc. of the Java Virtual Machine Research and Technology Symp. USENIX Association, 2001. 1−12.

[10] Standand performance evaluation corporation (SPEC). JVM Client'98 (SPECjvm'98). 1998. http://www.spec.org/osg/jvm98

[11] Dacapo. http://www-ali.cs.umass.edu/DaCapo/

[12] Java hotspot virtual machine 2.0. 2000. http://java.sun.com/javase/technologies/hotspot/

[13] BEA JRockit 5.0. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/jrockit/

[14] Jikes research virtual machine (RVM). http://www.ibm.com/developerworks/oss/jikesrvm

[15] Dynamic runtime layer virtual machine (Drlvm). http://harmony.apache.org/subcomponents/drlvm/index.html

[16] Wu Y, Breternitz M, Quek J, Etzion O, Fang J. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In: Vernon M, ed. Proc. of the 2001 ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems. New York: ACM Press, 2001. 194−205.

[17] Arnold K, Gosling J, Holmes D. The Java Language Specification. 3rd ed., New York: Prentice Hall PTR, 2000.

[18] Intel Corporation. Intel Pentium 4 processor optimization reference manual. http://www.intel.com/design/pentium/manuals/248966.pdf

[19] Intel Corporation. VTune performance analyzer. http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm

邹琼(1982－),女,安徽合肥人,博士生,主要研究领域为编译技术,计算机体系结构.

胡伟武(1968－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为高性能计算机系统结构,并行处理,VLSI 设计.

伍鸣(1978－),男,博士生,主要研究领域为计算机体系结构,操作系统,JAVA 虚拟机,内存管理.

章隆兵(1974－),男,博士,副研究员,主要研究领域为高性能计算机系统结构,并行处理.

© 中国科学院软件研究所 http://www.jos.org.cn