第32卷 第24期 计 算 机 工 程 2006年11月
Vol.32 No.24 Computer Engineering December 2006
·软件技术与数据库· 文章编号：1000—3428(2006)24—0063—03 文献标识码：A 中图分类号：TP314

减小运行时优化开销的方法

郭振宇，刘 利，陈 彧，汤志忠
(清华大学计算机科学与技术系，北京 100084)

摘 要：运行时优化在程序运行期间，根据采集到的相关信息，确定程序的热点并进行优化，从而加速程序的执行。然而，运行时优化本身有一定的开销，有时候会抵消甚至超出优化得到的效果。该文设计和实现了一个基于 SMP/IPF(英特尔安腾系列)/Linux 架构的自适应二进制代码优化/编译框架，其中包含了运行时优化。分析了运行时优化的阶段和开销，并介绍了在设计和实现该框架的过程中，为减小这种开销所提出的思路和采用的方法。

关键词：运行时优化；优化开销；持续优化

Ways to Reduce the Cost of Runtime Optimization

GUO Zhenyu, LIU Li, CHEN Yu, TANG Zhizhong
(Dept. of Computer Science and Technology, Tsinghua University, Beijing 100084)

【Abstract】 Runtime optimization analyzes the runtime information it collects, identifies hot spots, applies optimization on them, thus speedups the execution of the programs. However, the system itself may consume critical resources, which sometimes counteracts or even outweighs the benefit it gains, and leads to the failure of the optimization. This paper implements an adaptive binary optimization/compilation framework on SMP/IPF (Intel Itanium processor family)/Linux, among which runtime optimization is included. It also analyzes the stage and cost of runtime optimization, introduces the ideas and ways which are conducted to reduce them, during the implementation of the framework.

【Key words】 Runtime optimization; Optimization cost; Continuous optimization

1 概述

运行时优化是在程序运行的过程中，采集和分析程序的运行时信息，对程序中执行频率较高的代码片断进行优化，从而加速程序运行。相比程序生命周期中其它时期的优化（编译、装载、链接、运行后），运行时优化可提供确切的程序运行事件统计和运行环境信息，从而针对当前特定的行为模式进行特定的优化。而其它时期的优化都只针对程序的平均行为模式，当程序运行模式偏离平均值较大时，优化的效果就会大打折扣，甚至导致程序的执行时间比没有优化之前还要长。正因为运行时优化的这种优势，它在近年来得到了较为广泛的研究[2~6,8]。

体系结构的发展[7]也促进了运行时优化的发展。首先是并行性方面。从单发射到多发射，从单核到SMP、超线程，再到最近的CMP，CELL，CPU可同时支撑的控制流数目越来越大，从而可以并行执行的任务数也越来越多。然而，软件的并行度并没有得到大的提高，使得CPU常常有多余的执行部件空闲。我们可以充分利用这些资源，运行运行时优化系统，在不阻塞原来程序运行的前提下提高程序的执行速度，以提高体系结构提供的并行度的利用率。另一方面，CPU本身集成了越来越多的收集运行时信息的性能监测部件，不仅仅是提供了传统的事件统计，而且还提供了很多细粒度信息，例如哪一条指令引用哪个内存地址的时候，发生了Cache缺失。这些细粒度的信息为运行时优化提供了高价值的优化线索。最后，典型的一机跑多操作系统的虚拟技术，如Intel的VanderPool等，得到了密切的关注，在不久的将来也会被广泛应用。而虚拟机技术本身，则包含了提供各种各样的运行时信息的可能性，同样丰富了运行时优化的素材。

然而，运行时优化有一个非常大的缺点，它本身要消耗一定的计算资源，当优化的效果不明显的时候，常常是不但没有加速程序的执行，反而会使程序的运行变慢。因此，运行时优化很重要的一个方面，就是想方设法降低运行时开销。在文献中，已经有很多运行时优化系统[2~6,8]，不过它们都注重于运行时优化的框架设计、优化的方法以及优化的效果。对于其中的开销，则提得很少。我们目前正在IPF/Linux下实现一个二进制代码优化/编译框架Betto，其中包含运行时优化。本文介绍了在设计和实现该框架的过程中，采取了怎样的方法来减小运行时优化的开销。本文的主要贡献在于：

(1)详细分析了运行时优化的几个阶段及其开销；

(2)提出了减小这些开销的2个思想，并根据这些思想设计和实现了一个新的运行时优化框架模型，大大减小了运行时优化的开销。

2 运行时优化的阶段和开销

程序的优化至少需要2个阶段：代码分析和代码转换[9]。前者负责程序的控制流和数据流的分析，生成控制流图(Control Flow Graph, CFG)和数据依赖图(Data Dependence Graph, DDG)，使得系统得以深入了解所要优化的程序的结构。后者在CFG和DDG的基础上生成优化后的相关代码片断。运行时优化又增加了2个阶段：运行时数据采集负责设置相关运行时信息采集软硬件，并收集采样数据；优化后代码挂入则在优化后的程序片断生成后，采用一定的方法使得在随后的时间里，程序的控制流可以到达这些片断。当然，这种划分并不是绝对的，但在过去的文献中，大多数运行时优化

作者简介：郭振宇(1981－)，男，硕士生，主研方向：并行编译等；刘 利，硕士生；陈 彧，博士生；汤志忠，教授

收稿日期：2005-12-23 **E-mail：**guozy03@mails.tsinghua.edu.cn

系统都可以分为这样 4 个阶段，如文献[2~6,8]等。由于运行时优化的开销非常关键，需要用最小的开销得到最大的收益，因此在几乎所有的运行时优化系统里，只有那些执行频率高的代码片断才得到优化。在文献中，这些代码称为热点区域(hot trace)，其起始点称为热点(hot spot)。运行时信息采集的另外一个任务，就是要识别出热点区域。

运行时优化的实时性决定了任何开销都会对程序最后的优化效果产生很大的影响。因此，对运行时优化各个阶段的开销进行较为详细的分析是非常必要和有意义的。

首先是数据采集。通常的采集方法有程序插桩(Instrumentation)、解释执行(Interpreted Execution)和硬件采样(Hardware Sampling)等。插桩的开销在于它要求系统执行额外的代码，当一个代码片断被插桩后，以后每次控制流经过这个片断，都需要执行那些额外的代码。解释执行的开销在于专用虚拟机的开销，它需要解析原有代码，生成新的代码，并利用本地指令去执行这些新生成的代码。硬件采样的开销是当硬件采样计数器溢出的时候，会发生中断，并进入用户指定的采样回调函数，从而引起一定的开销。在这三者之中，解释执行得到的信息是最完整和精确的，而插桩和硬件采样则分别和插桩数量和位置、采样事件和频率有关。研究表明[11]，基于统计的采样分析(Statistical Sampling)可以达到近似完全采样的效果。而从开销上来说，显然硬件采样的开销是最小的。综合效果和开销 2 个方面，硬件采样是最好的选择。对于第 1 阶段的另外一个任务：热点识别，这一般只需要在样本处理函数中设置计数器，进行累加和相关判断，因此并不引起额外的大的开销。

代码分析阶段的开销主要是 2 个方面：一是指令的解析，二是构建CFG和DDG的开销。前者的复杂度是线性的，和代码片断的大小有关。而后者的相关构建算法则达到了O(n*m*m)复杂度（其中n是最后生成的CFG中基本块的个数，m是基本块的平均指令条数)[12]。因此二者的开销都不容小觑。

代码转换和具体的优化方法有关，由于本文注重于基本的运行时优化系统的开销，因此对这个阶段的开销不作详细的分析。

优化后的代码挂入的开销主要在于各个版本的代码之间的切换。运行时优化系统的控制流在 3 个不同的代码域中相互切换，包括本地程序代码、优化后的程序代码以及运行时优化系统的代码。首先，为了把控制流从本地代码切换到优化后代码，需要对跳转目标地址在优化代码池中进行查找，找到对应的优化代码片断，这点在解释执行的优化系统中特别明显[4]。其次，为了防止代码域切换导致的副作用，需要进行控制流的上下文切换。当然，正如一些文献中所述[4]，这两部分的开销会随着热点区域的增多而逐渐下降，因为热点区域内之间的控制流切换无需查找(通过跳转指令即可)和代码域切换。不过，这里有 2 个问题：

(1)非热点区域的代码虽然执行频率较低，但是还是需要有查找操作；

(2)有研究[1]表明一次运行中程序的行为并不是只有一个模式，可能是分阶段的，因此这种一旦把代码转入优化代码池中就不再监测和优化的做法值得商榷。

3 Betto 的优化框架

我们正在实现中的 Betto 是一个针对可执行文件的持续优化框架；它采用了一个运行时/运行后(Post-runtime)交替优化的混合模型；它被设计成无缝集成到操作系统中，从而提供透明的程序优化服务。Betto 被划分为 4 个必要的子系统：BRS(Betto Runtime Service)，BES(Betto Embedded Service)，BMS(Betto Management Service)，BSDK(Betto Software Development Kit)，以及 2 个辅助子系统：BGS (Betto Graph Service) 和 BWS (Betto Web Service) 。运行时优化系统主要涉及 BRS 和 BMS，因此本文不再论述其它部分。Betto 的体系结构如图 1 所示。

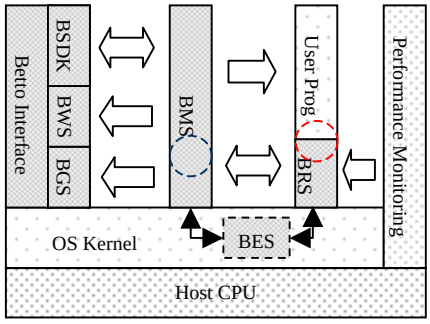

图 1 Betto 的体系结构

BRS 负责了 Betto 的运行时优化，包括采样的设置和收集、代码片断的分析、代码转换以及优化后代码的挂入。BMS 负责 Betto 的运行后优化，同样包含了代码分析和代码转换这 2 个阶段。Betto 不间断地监测目标程序的每一次运行，交替调度 BRS 和 BMS，进行运行时和运行后的优化，从而达到程序越跑越快的效果。图 2 给出了理想状况下，同一程序多次运行的执行时间。

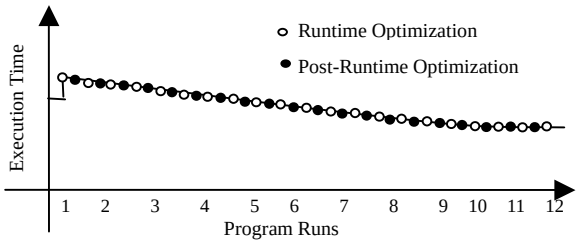

图 2 理想状况下的混合/持续优化

4 降低运行时开销的方法

Betto 降低运行时开销的主要思想包括 2 点：(1)把运行时的工作尽量搬到离线状况下来做；(2)结合以前多次的优化，用比较少的代价达到相同的效果。前者主要来源于 Betto 的混合优化模型，而后者则来源于持续优化的好处。同样，下面将分阶段讨论 Betto 中运行时优化的开销。

注意到第 3 节论述Betto的优化框架时，运行时优化和运行后优化都包括了代码分析和转换 2 个阶段，其中代码分析是可以共享的（实际上，Betto中代码转换也是自适应共享的）。Betto运行时优化时代码分析的开销几乎为 0，这是通过导入运行后优化的代码分析结果实现的。在运行后优化情况下，开销并不是关键问题，因此程序的ICFG(Inter-procedure Control Flow Graph)得以生成，并且存储到可执行文件的末端（通过改写可执行文件）。下一次该程序运行时，对应的ICFG会被自动导入到内存中。这样运行时优化器在程序一开始就可以得到整个程序的控制流图，而不需要作代码的实时分析。另外，通过修正可执行文件的内存映射表（Program Header Table），把ICFG串行化(Serialization & Deserialization)的开销（因为要存储到文件中，并且重新导入到内存）从O(n)(其中

n是基本块的数量)到O(1)。这是通过实现专用的动态存储分配器(Dynamic Storage Allocation System)实现的。基本的方法是使ICFG在运行阶段载入内存时，其内存虚地址和原来在离线状况下的内存虚地址相同，这样就保证了数据结构中包括指针在内的数据的有效性，从而略去了串行化的操作。这样做的缺点在于运行时所需要的内存比原来的方法要多，而且并不是所有的数据，如那些不是热点区域的CFG，都会在随后的优化中用到。Betto通过 2 个方面来解决这个问题：(1)诸如Linux这样的现代操作系统都提供了Lazy-Allocation的机制。在分配了很多内存，但是不去引用它们的时候，相关内存页并不会真正被创建，唯一的消耗就是内存空间被无端浪费了一部分；(2)上面提到的专用动态存储分配器采用了偏离空闲链表法(segregated free list)[10]，这种方法主要用于几种固定大小的数据结构的分配（这里是基本块、弧以及函数等），并且保证得到的外部碎片最少，因此ICFG最后占用的内存大小几乎将是最少的。

在数据采集阶段，Betto基于IPF自身集成的运行时性能监测部件，因此属于硬件采样，开销本来就不是太大。另外，Betto降低了运行时信息采集的频率，从而达到降低开销的目的。这样做的问题在于得到的信息过少，不足以确证程序运行的特征。不过，由于Betto的持续优化模型，Betto得以通过部分继承以前的运行时信息的方法，来有效地解决这个问题。设当前的实际采样数据为向量V(n)，最终需要用到的累计采样数据为$V_a(n)$，以前程序运行的累计采样数据为$V_a(n-1)$，那么

$$V_a(n) = \alpha * V_a(n-1) + (1-\alpha) * V(n)$$

其中α是V(n)和$V_a(n-1)$之间的相似因子，$0 < \alpha < 1.0$。α可以是一个经验数值，也可以通过相似函数[2]计算V(n)和$V_a(n-1)$的相似度得到。对于热点识别，通常是用计数的方法得到。当计数值超过预设的阈值的时候，就认为一个热点被找到了。Betto同样通过上述公式去部分继承以前的热点识别计数结果，从而加速热点识别过程。

代码转换由于和具体优化相关，这里同样不作分析。

如前文所述，代码挂入阶段的开销包括目标跳转地址的查找和跨不同代码域执行时的上下文切换。前者的深层次的原因在于优化系统不知道程序的整体结构，因此当它挂入某个优化后的代码片断时，无法同时修改那些跳转到该优化片断的指令。Betto 由于在运行时拥有整个程序的控制流图，因此可以通过回溯指向当前优化代码段的弧来确定那些需要修改的跳转指令。虽然间接跳转的存在导致无法定位所有的相关跳转指令，但是这已经能够解决大多数的跳转，从而大幅减少了消耗在目标跳转地址的查找上的时间。另一方面，由于优化后的代码和原来的代码之间的切换更多地通过跳转指令直接完成，而不是通过优化系统进行衔接，因此也大大减少了跨不同代码域执行时的上下文切换的次数。

由于框架的复杂性，我们没有能够实现 2 套不同的试验环境，因此也没能得到可以对比的试验数据。不过，从前面的分析依然可以看到，Betto 通过转移运行时的必要工作到离线状况下去完成，以及部分继承以前的数据，从而达到了大幅度降低运行时优化开销的效果。

5 相关工作

在过去的文献中，对运行时优化系统的介绍挺多[2~6,8]，不过根据笔者的检索结果，专门分析运行时开销的还没有。

如前所述，Betto采用硬件采样的方法进行运行时信息的采集，而文献中还有很多其它的方法。如加州大学的Thomas Kisler[2]对整个运行时环境做了全新的设计，要求被优化的程序本身有很强的反射能力。在其运行时环境里面，每一个程序自己都有一个信息采集器，负责跟踪所有的函数调用及数据访问等。因此，它在运行时级别提供了运行时信息。Dynamo[4]在运行时对所有的指令进行了解析，从而得到了运行时信息。M. C. Merten[3]实现了Dynamo的硬件版本。斯坦福的Morph[5]系统则通过一个固件层进行相关的信息采集，不过它需要程序对应的中间代码来辅助实现这一功能。微软的Mojo[8]则采用了插桩的方法，它要求用户在他们的程序中使用Mojo提供的一些API，以进行相关的信息采集。明尼苏达大学的ADORE[6]则和Betto相同，利用了安腾处理器提供的运行时性能监测部件，通过采样的方法来收集运行时信息。这些方法各有各的特点，比较来说，Betto的方法最为高效，但是需要专用运行时性能监测硬件的支持。庆幸的是，正如本文在概述中所总结的，这种支持正成为体系结构方面的一种趋势。

另外，上述的运行时优化系统都在运行时分析热点区域的控制流结构，而 Betto 则因为其交替优化模型，得以充分利用运行后优化得到的分析结果。

持续优化在过去几年里也是一个热点[2,5]，不过它们在两点上有所缺憾。一是它们一般都只考虑了持续地运行时优化，而没有考虑交替执行的模式。二是它们没有专门分析持续优化中的运行时开销，也没有充分利用以前的运行时信息去降低这些开销。

笔者调查了十多个近年的运行时优化系统，总的来说它们都各有各的特点，这让我们获益良多。同时，Betto 由于其持续/交替的优化模型，和以前的运行时优化系统相比，其运行时开销得以大幅降低。

6 总结和工作展望

本文详细分析了一般运行时优化的阶段和开销，并介绍了在设计和实现一个自适应的二进制代码优化/编译框架中，提出了怎样的思想，采用了怎样的方法去降低这种开销。目前我们的框架还在继续完善中，对于运行时开销方面，还有以下几个方面的工作要继续深入下去：

(1)运行时信息采样的内容和频率的确定策略上，还需要做更多的调查，使得可以用更少的代价，得到更有效的数据。

(2)需要建立有效的相似函数模型，去计算不同的程序运行期的运行时信息的相似度，从而最有效地继承以前的运行时数据。

(3)继续探索如何在单线程/多线程环境下，进行动态插桩，从而达到优化后代码的安全、高效挂入。

参考文献

1 Wu Y F, Breternitz M, Quek J, et al. The Accuracy of Initial Prediction in Two Phase Dynamic Binary Translators[C]. Proc. of the International Symposium on Code Generation and Optimization, 2004.

2 Kistler T, Franz M. Continuous Program Optimization: A Case Study[J]. ACM Transactions on Programming Languages and Systems, 2003, 25(4): 500-548.

3 Merten M C, Trick A R, Barnes R D, et al. An Architectural Framework for Runtime Optimization[J]. IEEE Transactions on Computers, 2001, 50(6): 567-589.

4 Bala V, Duesterwald E, Banerjia S. Dynamo: A Transparent Dynamic Optimization System[C]. Proceedings of the ACM SIGPLAN, 2000.

(下转第 120 页)