减少检查点开销的一种方法*

李凯原　杨孝宗
哈尔滨工业大学计算机科学与工程系　(哈尔滨 150001)

摘　要　设置检查点(checkpointing)是容错计算机系统进行故障恢复的重要手段。设置检查点的开销则是影响其性能的一个主要因素。文章提出了一种预先保存部分检查点数据的新方法。该方法不仅能够有效地减少检查点开销，而且具有比较短的检查点延迟。

关键词　容错　故障恢复　检查点开销　预先保存

A Method for Reduction of Checkpoint Overhead

Li Kaiyuan　Yang Xiaozong
(Dept. of Computer Science and Engineering, Harbin Institute of Technology, Harbin 150001)

Abstract: Checkpointing is one of the most important method for fault tolerant computer to recover from faults. The overhead of checkpointing is a major factor that has effects on the performance. This paper presents a new method, which pre-save some of checkpoint data. Not only can the new method reduce checkpoint overhead effectively, but also has a shorter checkpoint latency.

Keywords: fault tolerance, fault recovery, checkpoint overhead, pre-save

1　引言

设置检查点(checkpointing)是容错计算机系统进行故障恢复的重要手段。计算机系统通过周期性地设置检查点，把程序在运行时的正确状态保存到稳定存储器(stable storage)中。如果在随后运行过程中发生故障，那么系统进行卷回恢复(rollback recovery)，从稳定存储器中读出前一个检查点时的正确状态，从该点继续执行。这样就避免了由于故障而导致的程序从头重新执行，因而有效地减少了计算的损失。Duda 证明在故障条件下，如果不使用检查点，程序平均执行时间随其有效执行时间（假设不发生故障时的执行时间）呈指数增长，而使用固定间隔的检查点，则呈线性增长[1]。实际上，当故障率超过一定值，使两次故障的平均间隔时间小于程序的有效执行时间的时候，如果不使用检查点，程序几乎不能运行到结束。

设置检查点的开销是影响系统性能的一个主要因素。程序在故障条件下的运行开销有两个主要来源[2]。一个来源是系统在每个检查点都必须完成保存程序状态的操作，所以不可避免地要引入一定时间、空间开销。开销的另一个主要来源是因故障而引起的卷回，因为每次卷回都会损失已经完成的计算。减少设置检查点的开销，不仅可以直接减少为保存检查点而付出的额外代价，还能够使系统适当增加检查点的数量，减少检查点之间的时间间隔。这样在发生故障时，可以减少卷回的工作量。所以检查点开销的大小直接影响了系统的性能。

文章首先对可以直接用于减少检查点开销的现有方法做了简要的分析。在此基础之上，文章提出了一种预先保存部分检查点数据的新方法。该方法不仅能够有效地减少检查点开销，而且具有比较短的检查点延迟。

2　现有方法及分析

直接减少检查点开销的方法可分为两类：一类是减少设置检查点时需要保存的数据，主要包括增量式检查点设置 (Incremental Checkpointing)、程序员指导的内存排除(Programmer-Directed Memory Exclusion) 和检查点压缩(Checkpoint Compression)等；另一类则隐藏设置检查点的时间，主要包括检查点缓存(Checkpoint Buffering)和写时复制(Copy-on-Write) 等。有时两类方法可以结合在一起使用。

2.1　增量式检查点设置

增量式检查点设置是一种利用页式虚存管理，自动排除只读内存的技术，需要一定的硬件支持。每当完成一个检查点的设置后，程序空间内的所有页面都置为“只读”。如果程序在随后的运行中需要写一个只读的页面，就会产生“访问非法”错，这时由相应的处理程序记录下该页面，并将该页置为“读写”，使程序可以修改该页。当需要设置下一个检查点的时候，只需要保存处理器状态和所有被修改过的页面，而不用保存整个程序空间，所以减少了需要保存的数据量。使用这种方法，在大多数情况下都能很大程度地减少检查点的大小和时间开销[3]。

*文章受国家自然科学基金资助(69873013)。

作者简介：李凯原，男，1968 年生，博士生，主要研究方向为分布式系统容错和检查点与卷回恢复技术；杨孝宗，男，1939 年生，教授，博士导师，主要研究方向为多机系统与容错计算技术。

2.2 程序员指导的内存排除

增量式检查点设置虽然能够排除大量的“只读”页，但保存到检查点文件中的数据中仍然可能有一些与故障恢复无关。比如以后不再访问的临时变量，或在以后先进行写访问的变量。Plank 通过增加函数调用，使程序员在了解程序结构的条件下，用人工方法减少这一部分数据，取得了很好的效果[3]。同时这一方法也存在许多不足：破坏了检查点操作的透明性，程序员必须在程序中显式给出检查点包含或不包含的数据；仅对大型数组、内存块等效果好。如果处理单个的变量，辅助操作会引入更大的开销，得不偿失；人为因素也过多。不仅效果要依赖于程序员的经验，而且正确性也依赖于程序员。如果程序员错误地将必要的数据排除掉，则会造成检查点的不可恢复。

2.3 检查点压缩

通过压缩可以直接减少数据量，压缩方法可以选择标准的算法，比如 LZW[4]。不过使用压缩技术减少开销必须满足一定的条件。只有用于压缩的时间小于因数据量减少而节约的时间，使用检查点压缩才有意义。由于压缩比通常与具体的应用有密切关系，所以压缩方法的通用性不是很强。

2.4 检查点缓存

检查点缓存是通过隐藏检查点操作所消耗的时间来减少开销的一种最简单方法，其实质是使程序在保存检查点的过程中能与保存操作并行。首先程序被“冷冻”起来，防止其改变状态，再将整个程序空间的内容复制到缓冲区内。然后“解冻”应用程序，使之继续运行，同时完成将缓冲区的内容保存到稳定存储器中的操作[5]。

检查点缓存的主要缺点是如果检查点的数据量比较大，则需要很大的缓冲区。如果物理内存无法满足，部分程序空间有可能被操作系统对换到辅助存储器中，很大程度上会影响系统的性能。

2.5 写时复制

克服检查点缓存对缓冲区需求比较大的缺点，可以采用写时复制。设置检查点时，不必立刻将内容复制到缓冲区中，而是利用页式虚拟存储器的功能，将程序空间内的所有页都置为“只读”，然后允许应用任务继续运行。如果任务要修改某一个页面，则必然会产生“访问非法”，这时由相应的处理程序将该页面复制到缓冲区当中，再将该页设置为“读写”，允许应用程序修改。在随后的运行中，将缓冲区和主存中的内容存入稳定存储器中。与其他方法相比，写时复制在实验中的效果最好[6]。

3 预先保存

虽然写时复制在实验中的效果最好，但它存在一个问题：从开始设置检查点到其内容被完全保存到稳定存储器，要经历比较长的延迟(称为检查点延迟)。而某些情况下，过长的检查点延迟仍然会给系统引入很大的开销。比如事务处理系统常常要求快速提交(fast commit)，即在检查点设置完成后才能向外界输出结果。再比如分布式系统中，只有所有结点都完成检查点的保存后，系统才能确认为全局检查点，否则可能引起状态不一致。这些情况下，写时复制也无法有效地减少开销。

针对这一问题，文章提出预先保存的方法减少开销。

预先保存是基于这样一种想法：如果程序在运行过程中修改了某一个页面，并且直到下一个检查点设置时不再修改该页的内容，那么将该页内容预先保存到检查点文件中，可以减少检查点设置时需要保存的数据量。

为了减少需要保存的数据总量，作者在预先保存中结合了增量式检查点方法。

3.1 预先保存数据的确定

预先保存的关键问题是如何确定哪些数据应该被提前存入稳定存储器。如果不能准确地判定提前保存的数据，会导致被预先保存的数据在设置下一个检查点之前又被修改，这样不仅不会减少在设置检查点时的开销，还可能因为引入的预先保存操作使开销反而增大。

确定预先保存的数据，可以选择存储管理中的页面淘汰算法，将不再被访问的页面写入稳定存储器。但这种方法需要 CPU 和操作系统的支持，不仅不容易在应用程序级实现，而且可能引入比较大的开销。

文章采用如下方法确定预先保存的数据。先将两个检查点之间的时间间隔分为 n 个相等的子间隔。从第二个子间隔开始，每当子间隔结束时就检查上一个子间隔中修改过而当前子间隔没有修改的页面，将这些页面放入准备预先保存的队列中进行预先保存操作。如果在随后的运行中再次修改已加入预先比较队列中的页面，则将其从队列中删除。如果修改了已经保存的页面，则将该页标记为尚未保存。当第 n 个子间隔结束时，系统保存所有未能提前存储的页面。

虽然这种方法不能保证数据不被重复保存，但是适当选择子间隔的时间极有可能使重复保存的数据控制在允许的范围内。所以，在这种方法中 n 的选择非常重要。可以预见，如果 n 的取值过小，则可能使许多可以提前保存的数据留到了检查点设置的时候才保存，达不到预先保存的目的；如果 n 取值过大，则可能给系统引入过大的额外开销，并可能使数据重复传送的比率增高，增加 I/O 系统的负担。n 的确定与具体的应用有密切关系，文章将根据实验结果给出分析。

限于篇幅，文章不给出预先保存的完整算法描述。

4 实验结果

4.1 实验条件

该实验使用 IBM PC 兼容机，其 CPU 为 Pentium 133，内存 16M。作为稳定存储器的硬盘为 1.2GB 的 Seagate 硬盘。运行的操作系统为 Linux (Red Hat 5.1)。Pentium 机的页面大小为 4096 字节。

应用程序选择了大小为 1024×1024 的浮点矩阵乘方运

(下转 14 页)

断言和软件测试技术来验证软件的安全特性[2]。再如使用错误注入分析法来分析程序在异常环境(如非期望的输入)下的行为。

5.2 基于系统审计信息的检测

这种方法主要是在系统遭受攻击或入侵后基于审计信息进行检测。如基于规则和专家系统检测,即对审计信息运用机器学习方法生成(或人工建立)一个规则库,再构造一个专家系统,并基于这个规则库来推导新的审计信息中是否包含入侵。再如基于统计信息检测,即建立主体(用户或系统)的运行轮廓(由一组统计参数构成),然后在运行中基于运行轮廓来检测系统的异常,同时学习并更新主体的运行轮廓。

作者在 C2 审计系统上(如 Solaris 的 BSM)利用 UNIX 进程执行踪迹结合神经网络分类的方法来检测对 suid 程序的栈溢出攻击。首先为所有要监测的 suid 程序建立一个包含全部可能的某个长度的系统调用序列的"正常"数据库和"异常"数据库,然后用它们来训练一个 MLP 网络(BP 算法)。再用此网络来检测新产生的系统调用序列,可获得较好的检测效果[3]。

6 结论

Internet 为人们提供了一个传输和共享信息的平台,但其安全问题也日益突出。缓冲区溢出是黑客常用的攻破远程系统并获取超级权限的工具。UNIX 上缓冲区溢出隐患存在的根源是:C 语言无边界检查、UNIX 进程和内存管理、suid root 程序。对缓冲区溢出攻击的防范方法有:修改 suid root 程序的源代码、修改 C 编译器及去除 OS 内核堆栈执行特权。检测技术主要有基于软件工程的分析技术、基于审计信息的检测和基于神经网络的检测技术。

(定稿日期:1999 年 8 月)

参考文献

1.Aleph One. Smashing the Stack for Fun And Profit. Phrack, 1996;7(49)

2.G.Fink,M.Bishop. Property-based testing: A new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes, 1997;22(4)

(上接 5 页)

算。选择矩阵乘方作为应用程序的主要原因是既可以获得比较长的运行时间,又可以获得比较大的数据量,以防止偶然性因素带来的影响。对于每种测量数据,文章采用 10 次的平均值。

4.2 实验结果

实验结果如表 1 所示。由于结合了增量式检查点方法,预先保存方法的平均大小与增量式检查点的大小相同。虽然预先保存方法的每个检查点开销大于写时复制,但其平均检查点延迟则是最小的。

表 1 几种检查点设置方法比较

实现方法	检查点平均大小(KB)	检查点平均开销(ms)	检查点平均延迟(ms)
完全检查点	8229.2	518.5	509.4
增量式检查点	2326.8	166.7	151.0
写时复制	8229.2	88.6	539.4
预先保存(n=10)	2326.8	132.5	117.3

预先保存的每个检查点平均开销与子间隔的取值有密切关系,图 1 给出了二者之间的关系。使预先保存开销最小的 n 值,应该在 10~20 之间。

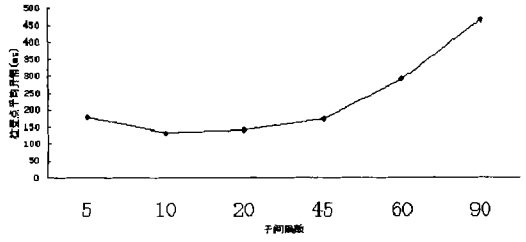

图 1 不同子间隔下的检查点平均开销

5 结论及今后工作

文章在简要分析了可以直接用于减少检查点开销的现有方法的基础上,提出了一种预先保存部分检查点数据的新方法。实验结果显示,该方法不仅能够有效地减少检查点开销,而且具有比较短的检查点延迟。

今后的工作主要包括使用其它应用程序进一步判定子间隔数的取值与检查点开销的关系,以及如何选取适当子间隔数等等。(定稿日期:1999 年 8 月)

参考文献

1.A.Duda. The Effects of Checkpointing on Program Execution Time. Information Processing Letters,1983;16(4): 221-229

2.A. Ziv,J. Bruck. Performance Optimization of Checkpointing Schemes with Task Duplication. IEEE Trans. Computers,1997; 46 (12): 1381-1386

3.J. S. Plank, M. Beck,G. Kingsley. Libckpt: Transparent Checkpointing under Unix. 1995 USENIX Technical Conference, 1995:213-223

4.T. A. Welch. A Technique for High-performance Data Compression. IEEE Computer,1984;17(6):8-19

5.J. S. Plank,K. Li. Ickp.A Consistent Checkpointer for Multicomputers. IEEE Parallel and Distributed Technology, 1994;2(2): 62-67

6.E. N. Elnozahy, D.B. Johnson,W. Zwaenepoel. The Performance of Consistent Checkpinting. The 11th Symposium on Reliable Distributed Systems,1992;(8):39-47