内存异常数据的检测与恢复*)

朱智林[1,2] 李 航[1] 刘西洋[1] 陈 平[1] 张其林[2]
(西安电子科技大学软件工程研究所 西安710071)[1] (中国煤炭经济学院计算机系 烟台264005)[2]

Data Corruption in Memory Detection and Recovery

ZHU Zhi-Lin LI Hang LIU Xi-Yang CHEN Ping ZHANG Qi-Lin
(Institute of Software Engineering, Xidian University, Xi'an 710071)

Abstract For some certain applications, they require massive application data to be resident in main memory. We categorize the exceptional data resided in memory based on the analysis of corresponding reasons. Exploiting mature CRC technologies to build a redundancy code for each data unit in memory to achieve integrity and combining traditional log and checkpoint technologies, we present the basic methods of prevention for corrupted data propagation and directly static detection and recover of corrupted data.

Keywords Redundancy code, Data corruption, Detect and recovery, Main-memory data management, Data consistence

1. 引言

在计算机应用领域,存在一类应用,要求系统满足一定的实时性要求,较高的性能,较好的容错能力。这类应用的一个重要特征就是业务关键(mission critical),所涉及的数据量大,比如电信计费、军事指挥控制、银行业务处理等等。在集中式省级移动计费系统中,用户群庞大,话费数据不能及时反映用户帐户当前的实际情况,数据一般滞后数小时乃至二十四小时。随着硬件技术的发展,RAM 价格的下降,计算机配置数 GB 内存已经成为现实的情况下,用户的相关数据常驻内存,已经成为可能,从而可以提高系统的实时性。然而数据常驻在易失性介质中,虽然能够提高系统工作效率,但也容易造成数据丢失。

本文以移动记费为背景,针对大量数据常驻内存的情况下的内存数据管理中数据异常产生的原因进行了简单的分析和分类,主要讨论了基于校验码的技术在内存数据管理[1,6]中的应用,探讨了静态检测与预防数据异常发生以及进行简单的数据异常恢复的基本思想。

2. 数据异常产生的原因与简单分类

移动计费系统体系结构如图1所示,其中 BOOM (Business Object Organization and Management)[8]提供对完整的 ACID 语义事务模型的支持。

在应用系统中由于进程的异常中止等软件错误往往会引起数据状态的不一致,异常数据(data corruption)[3]就是其中的一类。事实上在软件错误中约有25%～30%属于地址错误[2],诸如数组下标越界、指针走飞、程序栈错误等等。对于这类数据状态的不一致情况仅仅依靠 ACID 无法解决,需要采用新的措施加以处理。

在 BOOM 数据管理中,约定包含在 BeginUpdate 和 EndUpdate 之间的数据更新是合法数据更新,其它一律为非法数据更新。非法的数据更新所产生的数据不一致状态称之为直接物理异常数据;而由用户错误输入或应用逻辑错误所导致的数据状态的不一致称之为直接逻辑异常数据;直接异常数据被其它进程作为操作依据时,所产生的新的数据状态的不一致则称之为间接异常数据。

异常数据会缓慢地导致数据状态的不一致进一步扩散,而这类数据状态不一致的发现比较滞后,进行修复代价也较高。预防异常数据产生,阻止其传播,及时发现并修复,可提高系统的性能和容错能力。

BOOM 的事务可恢复框架[4,8]如图2所示,系统数据区禁止应用程序访问,其中有系统的内核,日志,ATT 表等,应用

*)基金项目:可信软件工程(413150501)。朱智林 博士生,副教授,主要研究方向为软件工程,面向对象技术,数据库技术;李 航 博士生;刘西洋 博士,副教授;陈 平 教授,博导。

整个算法的时间复杂度为 $O(\max\{n^2 \cdot d \cdot \log_2 d, n^3\})$。

结束语 本文提出了一种基于贪心策略的最长 d 维箱嵌套问题的解决方法,并对算法的时间复杂度进行了分析。实验证明了方法的有效性,实际上,如果算法的输入为内部已经排好序的 n 个 d 维箱,那么算法的时间复杂度可降至 $O(n^3)$,这个复杂度是比较令人满意的。

参 考 文 献

1 Weisstein E W. Greedy algorithm from mathworld. http://mathworld.wolfram.com/GreedyAlgorithm.html 1999
2 Black P E. Greedy algorithm. http://www.nist.gov/dads/HTML/greedyalgo.html 2003
3 Yu Jiansheng. Greedy algorthm. http://icl.pku.edu.cn/yujs/papers/pdf/ComAlg10.pdf 2002
4 Sahni S. Data Structures, Algorithms, and Applictions in C++, McGraw-Hill, 1998
5 王晓东. 计算机算法设计与分析. 北京:电子工业出版社,2001

程序只能访问用户数据区。对于系统发生崩溃后的恢复[5,7]，通过对日志的扫描，根据日志方式的不同采用不同的恢复策略。

3. 异常数据的检测

借鉴数据通讯中的CRC技术思想，为内存中每个用户数据片断(data segment)，在系统数据区的校验码表中增加一个checkcode数据项，记录该片断的校验码。

通过合法数据更新方式，相关的校验码数据才被更新，否则该片断的校验码不会被更新。在每次访问该片断数据时，首先计算其校验码，并与系统中校验码表中的相应的校验码进行比较，如果匹配，则该数据是完整数据，否则该数据是异常数据，并终止相应的事务，调用相关的恢复进程，避免异常数据的进一步扩散。

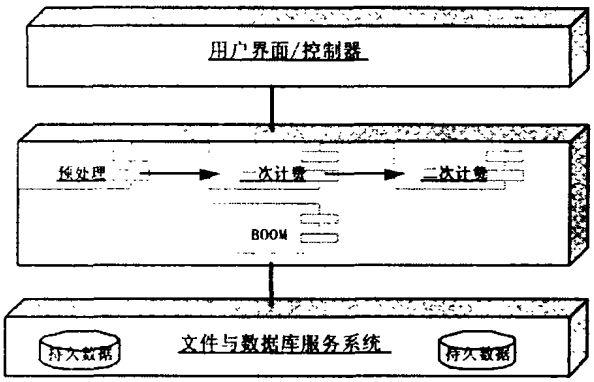

图1 计费系统体系结构示意图

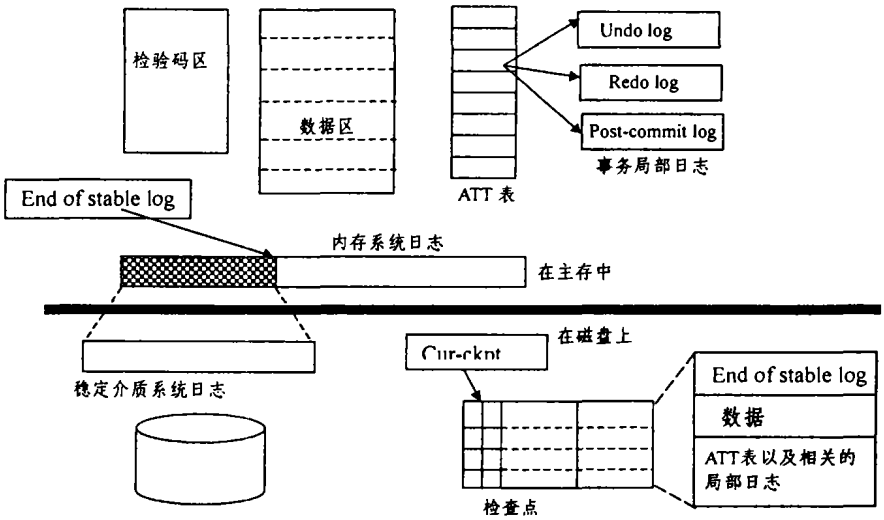

图2 事务恢复框架

3.1 校验码更新与异常数据预防

假设所更新数据片断为M，在更新前其校验码为CheckCodeOld，更新后的校验码为CheckCodeNew，M更新前后其校验码的变化记为ΔW，则CheckCodeNew = CheckCodeOld⊙ΔW，其中⊙是校验码的更新运算。在数据更新过程中，只要ΔW和CheckCod-eOld确定，则CheckCodeNew的计算就比较简单，不必对M从头扫描来计算新的校验字。

在数据更新过程中，需要更新的数据不仅有用户数据区数据，而且还有相应的校验码数据，其中校验码的更新是原子操作，具体的过程如下：

BeginUpdate：对要更新数据片断所对应的校验码加锁，记录该数据片断的Undo日志，同时在Undo中记录校验码更新标志(CheckCodeFlag)为False。

EndUpdate：记录Redo日志，比较相应的Undo和Redo日志，计算ΔW和新的校验码，即CheckCodeNew = CheckCodeOld⊙ΔW，同时更新该数据片断的校验码，置Undo日志中的校验码更新标志(CheckCodeFlag)为True，最后释放该数据片断的校验码上的锁。

Undo Update：与正常撤销事务一样，记录Redo日志，对校验码也做相应更新撤销处理。在具体撤销处理过程中，对Undo中每一条记录，根据CheckCodeFlag的状态，采取不同的撤销措施。当Undo中的CheckCodeFlag为True时，表明校验码更新已完成，要撤销该更新操作。当CheckCodeFlag为False时，表明校验码未更新，无需撤销。对于其它撤销动作的执行与CheckCodeFlag状态无关。

Read：数据读取时，首先计算该数据片断的检验码，并与存储在系统数据区中相应的检验码进行比较，如果一致，则继续，否则启动相关的恢复进程，进行修复。

3.2 异常数据静态检测与恢复

读时进行校验，能够防止异常数据的传播，但滞后。如果直接物理异常数据已产生，那么要及时发现并加以恢复，尤其是要防止导致间接异常数据的产生，间接异常数据的修复代价极高。

静态检测与恢复思想是指在检测时停止接收新的事务，等到当前所有活动事务提交或者中止，并在日志中写入了COMMIT或ABORT之后，才开始进行检测与恢复。在BOOM中，引入了一致性检测和恢复的守护进程，负责对数据进行周期性的静态检测。

在系统日志刷出到磁盘之后，假定那些被刷出的日志中所涉及的检验码信息和校验码所对应的数据内容是一致的，则静态检测与恢复的具体伪码如下，其中end_of_stable_log为本次要刷出的系统日志起点；Begin_Audit和End_Audit分别为系统日志中需要进行恢复的起点和终点；check函数完成对第index项日志相关事务的数据片断的校验码计算和比较，如果比较结果不一致，则返回BAD；recover是一个事

务恢复函数。

```
index=end_of_stable_log;
flag=0;
while(index<= system_log_length)
{
    status=check(system_log[index])
    if(status==BAD&&flag==0)
    {
        begin_audit= index;
        end_audit =index;
        flag=1;
    }
    else if(status=BAD&&flag==1)
      end_audit=index;
    index ++;
}
recover(needed_audit_segment, begin_audit, end_audit)
```

直接的逻辑异常数据，由于与应用程序的逻辑结构以及用户的交互输入有关，对这类异常数据的检测比较困难，一旦发现这类不一致状态，可以通过对日志进行分析，追溯产生不一致产生的根源，然后将与此相关的事务进行撤销或者重做处理，消除间接异常数据的影响。

结论 校验码技术是一种检测数据状态不一致性的手段，读时比较可以避免异常数据的传播。直接物理异常数据的静态检测与恢复，能及时修复部分异常数据，但采用静态的方式，进行检测恢复处理时，需要等待活动事务结束，这对只有短事务的应用系统而言是很有效的方法，但对存在长事务的应用系统而言，势必会影响系统的性能，这也是今后工作中需要进一步研究解决的问题。

参 考 文 献

1 Lehman T J, Shekita E J. An Evaluation of Starburst's Memory Resident Storage Component [J]. IEEE Transactions on knowledge and data engineering, 1992, 4(6): 555～566

2 Sullivan M, Stonebraker M. Using Write Protected Data Structures to Improve Software Fault Tolerance in Highly Available Database Management Systems [A]. In: Proc. of the 17th VLDB conf. [C], 1991. 171～180

3 Jagadish H V, Lieuwen D. Dali: A High Performance Main Memory Storage Manager [A]. In: Proc. of the 20th VLDB Conf. [C], 1994. 48～59

4 Delis A, Kanitkar V, Kollios G. Database Architectures [M]. New York, NY, Feb. 1999

5 Garcia-Molina H, Ullman J D, Widom J. 数据库系统实现[M]. 机械工业出版社, 2001

6 Liu Yun-Sheng, et al. Data Organization and Management of Real-Time Main Memory Database [J]. Computer Research & Development, 1998, 35(5): 469～473

7 王意洁，胡守仁. 面向对象数据库中的故障恢复[J]. 计算机科学, 1999, 26(2): 35～38

8 张振华. 业务对象中恢复机制研究[D]: [西安电子科技大学2002年硕士论文]

（上接第160页）

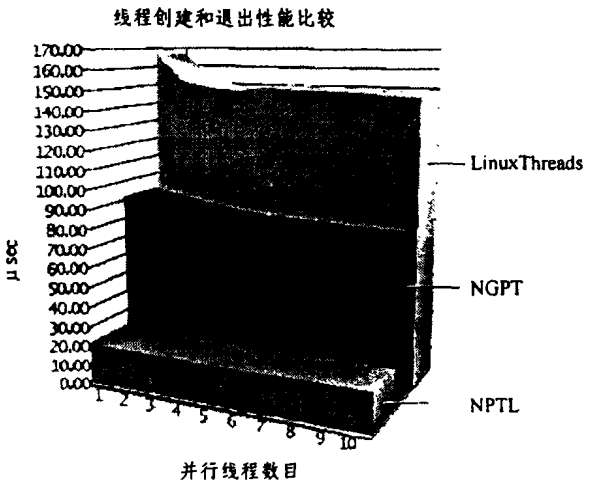

图1

Generation POSIX Threading）和 NPTL（Native POSIX Thread Library）。它们采用不同的设计思想来增强 Linux 对线程的支持。NGPT 使用 M:N 模型，开发者认为 M:N 模型能完全地使用多 CPU 的能力，同时又使开销尽可能少。通过将线程调度主要放在用户态完成，使核心态的任务切换变得很少。当然，由于需要使用核心态和用户态两个调度器，使 NGPT 实现起来很复杂，信号处理也很复杂，当一个线程被阻塞时需要避免阻塞其他线程（它对 Linux 核心做的修改非常小）。NPTL 仍然使用1:1模型，但是在核心做了很大的改变。开发者认为以下事实使1:1模型非常有吸引力：1个调度器，信号处理完全由核心完成，不存在阻塞问题。本文所述对核心的修改一部分是同时针对 NGPT 和 NPTL 的，另外很大部分则是专门针对 NPTL 的。

就测试的结果来看，NGPT 目前性能比 NPTL 稍逊一筹。这可能是由于 NGPT 没有完全利用内核最新的进展所造成的，也有可能是因为 M:N 模型的复杂性造成了它的性能稍差。由于 NGPT 相对于 NPTL 有自身的优点，而且更加成熟，我们下一步的计划是对 NGPT 做深入的研究，尽可能地发挥出它的潜力。

图1中纵坐标是 NPTL、NGPT 和 LinuxThreads 做10000次线程创建和退出操作，平均每一次需要的时间。横坐标是并行的线程数。

无论如何 NGPT，NPTL 两者相对于 LinuxThreads 都有着非常大的性能提升，同时它们都很好地实现了 PThreads 标准。两者最后谁会成为下一代 Linux 标准线程库还很难说。可以预计，当它们最终替代了 LinuxThreads 以后，Linux 作为一个企业级的操作系统必然又迈出了坚实的一步。

参 考 文 献

1 Lewis Bil, Berg D J. Threads Primer - a Guide to Multithreaded Programming. SunSoft Press, 1996

2 Richter J. Windows NT 高级编程技术. 清华大学出版社, 1994

3 Franke H, Russell R. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In: Proc. of The Ottawa Linux Symposium, 2002

4 Drepper U. ELF Handing For Thread-Local Storage. http: // people. redhat. com/drepper/tls. pdf