

# WOB:一种新的文件检查点设置策略

裴 丹,汪东升,沈美明,郑纬民

(清华大学计算机系,北京 100084)

**摘 要:** 实现分布/并行系统容错的基础是单进程检查点设置和卷回恢复技术,而对进程活动文件状态进行保存和恢复则是这种技术的重要方面.本文提出的延迟写策略,实现了对用户文件的检查点设置,有效地解决了在发生故障时用户文件内容与进程全局状态的不一致问题.它对用户透明,并且通过优化设置内存缓冲区大小、时延隐藏等手段,使得这种策略在空间开销、正常运行时间、恢复时间等性能指标上优于其它方法.

**关键词:** 检查点设置;卷回恢复

**中图分类号:** TP302.8 **文献标识码:** A **文章编号:** 0372-2112 (2000) 05-0009-04

## WOB: A Novel Approach to Checkpoint Active Files

PEI Dan, WANG Dong-sheng, SHEN Mei-ming, ZHENG Wei-min

(Department of Computer Science, Tsinghua University, Beijing 100084, China)

**Abstract:** Checkpointing and rollback recovery of Unix process is the underlying technique of fault tolerance of distributed system and parallel environment. To save and restore the state of active file of the process is an important aspect of checkpointing and rollback recovery. This paper presents an approach called Write Operation Buffering to support this capability. WOB approach buffers all the write operations after a checkpoint until the next one, making all the operations between two checkpoints atomic as a whole. By choosing a suitable size dynamically for memory buffer, and by hiding the latency of flushing the buffer, the WOB approach achieves an overhead lower than other approaches.

**Key words:** checkpointing; rollback recovery

### 1 引言

近年来,性能/价格比高的工作站机群系统(Network of Workstations, NOW)被广泛地用作分布/并行系统平台. NOW系统的缺点是故障率高,因此有必要为其提供容错功能. 实现分布/并行系统容错的基础是单进程检查点设置和卷回恢复技术. 检查点设置和卷回恢复技术是指:在一个程序运行时,每隔一段时间就将这个程序运行状态保存到硬盘上;当程序发生故障时,从硬盘文件中读出所保存的程序状态,并从该状态重新开始执行. 在国际上, Condor<sup>[1]</sup>、libckpt<sup>[2]</sup>、libckp<sup>[3]</sup>等系统都提供了单进程检查点设置工具,并分别被一些并行容错系统作为底层技术.

为了使进程在卷回恢复后能够正确地继续执行,必须保证所保存的进程状态信息的完备性以及这些信息之间的一致性. 文献[1~3]中经过分析指出要保存的进程状态应包括:进程数据段、用户栈内容;程序计数器、处理机状态字寄存器、栈指针;信号屏蔽码、信号栈指针、信号处理函数、被挂起的信号;当前活动的用户文件信息,包括文件描述符、访问方式、文件大小、读写指针、文件内容等. 活动文件的内容是进程与外

界联系的重要手段,但是在目前已经报道的各种系统中,只有libckp<sup>[3]</sup>和libfcp<sup>[4]</sup>支持了对用户文件内容的恢复,但是它们的开销都比较大. 由作者研制的libesm实现了一种低开销的延迟写(Write Operation Buffering, WOB)策略,它能对用户活动文件内容进行检查点设置和卷回恢复. ChaRM<sup>[5]</sup>系统(基于检查点的并行程序卷回恢复和进程迁移系统)就是在libesm的基础上实现的.

### 2 恢复文件内容的必要性

用户文件内容与进程其它状态有很大的不同:其它状态往往随着应用程序发生故障而丢失,在恢复时把保存在硬盘上的状态写入进程空间就可以恢复这些状态;而用户文件本身就是保存在硬盘上,它并不随应用程序发生故障而丢失. 也就是说,如果不对文件内容进行特殊处理,当发生故障时,其它进程状态卷回恢复到上一个检查点所保存的状态,而用户文件内容仍然是发生故障时的状态,从而可能造成卷回恢复后用户文件内容进程全局状态不一致,进而导致程序运行错误.

```

checkpointing i
fd = open("example", O_WRONLY | O_APPEND);
write(fd, write_buf, BLOCKSIZE);
/* failure occurs, here */
checkpointing i + 1 /* should be here */

```

图1 FARE 故障

```

fd = open("example", O_RDWR);
checkpoint i
read(fd, read_buf, BLOCKSIZE);
lseek(fd, 0, SEEK_SET);
write(fd, write_buf, BLOCKSIZE);
/* failure occurs, rollback */
checkpoint i + 1 /* should be here */

```

图2 FARW 故障

由于直接在检查点中保存用户文件的全部内容的方法显然不可行,所以大多数系统只是在设置检查点时,保存进程当前用户文件的活动信息;在卷回恢复时,重新打开文件,并根据活动信息将相应文件截成原来的长度并将文件指针指向原来的位置。但是,通过分析应用程序中由各种系统调用和读写方式构成的不同文件操作流程对文件内容的影响,可以发现上述方法有时会导致卷回恢复后文件内容与进程全局状态不一致。

在图1中,设置完检查点*i*之后,文件 example 被以添加、只读方式打开,并写入一段内容。由于故障是在检查点*i+1*之前发生,而且文件的长度没有保存在检查点*i*中,所以在卷回恢复时,文件 example 无法被截成正确的长度。因此,缓冲区 write\_buf 中的内容实际上被添加了两次,引起执行错误。同样,如果先以文件尾作为起始地址调用系统调用 LSEEK,之后再行写操作,也会产生类似的错误。这两种错误的原因都是写操作的内容和方式依赖于某些实时事件(在这里是读取文件长度),并且在该写操作之后尚未设置过检查点时发生故障。因此,此类故障被称为 FARE 故障(Failure After Real-time Event)。

在图2中,先以读写方式打开文件 example,在设置完检查点*i*后,对同一文件区域先后进行读和写操作。写操作完毕后系统发生故障并卷回到检查点*i*处重新执行,此时欲读的文件区域的内容已被修改且未随卷回而得到恢复,从而导致文件内容与进程全局状态不一致。这次的重新执行就会因从该区域读出的内容是修改后的新内容而引起执行错误。这种故障被称为 FARW 故障(Failure After Reading and Writing the same area)。

作为更明显的情形,如果在设置完检查点*i*之后,进程读文件后调用系统调用 UNLINK 将它删除,当系统发生故障并卷回到检查点*i*处重新执行时,将会因读一不存在的文件而产生更为直接的程序错误。这种故障被称为 FAD 故障(Failure After Deleting)。

FARE, FARW, FAD 这三种典型故障在仅保存文件活动信息时,将导致卷回恢复后的程序执行错误。虽然通过限制用户所能进行的文件操作可以在一定程度上避免上述故障,但这显然会影响检查点设置工具的可用性和透明性。因此有必要

对活动文件内容进行保存和恢复,以支持用户程序任意的文件操作。

### 3 延迟写策略

#### 3.1 基本思想

与传统的检查点设置和卷回恢复技术类似,WOB 策略也是利用了用户文件内容保存在硬盘上并且不随程序发生故障而丢失的特点。WOB 策略的基本思想是使得相邻两个检查点之间的所有文件写操作(包括删除操作)具有原子性,即这些操作或者全都正确执行完毕,或者由于发生了故障全部放弃。具体而言,WOB 策略是使得在检查点*i*到*i+1*之间的写操作并不真正执行,而是将欲写区域的始址、大小、以及要写入的新内容记录到一个缓冲区中,直到建立检查点*i+1*时才真正执行那些延迟的写操作,并清空缓冲区。WOB 策略使得用户文件内容在设置下一个检查点之前不会被改变,实际上是对文件内容做了检查点设置。在每个检查点间隔内,硬盘上的检查点文件和用户文件的内容都是一致的,从而 WOB 策略使得上述的三种故障都不能再引起卷回后程序错误了。

在 WOB 策略中,写缓冲区的基本方式是在缓冲区末尾添加,但是当对同一文件区域进行第二次写操作时则直接写到第一次写入时的位置,即同一检查点间隔内对同一文件区域进行的写操作要进行合并,保证缓冲区中保存着最新的文件修改效果。在进行读操作时,如果要读的文件区域在缓冲区中就直接从中读出,否则该区域在本检查点间隔内一定未被修改过,则直接从文件中读出所需内容。缓冲区分为两部分,一部分在内存中,这一部分的大小可以改变;另一部分则在硬盘中。在写缓冲区时,如果当前内存缓冲未满载则写入内存缓冲区,否则写入硬盘缓冲区。

#### 3.2 性能优化

WOB 策略的性能在一定程度上依赖于内存缓冲区的大小。当内存缓冲区足够大以至于不必使用硬盘缓冲区时,该策略性能最佳,甚至有可能提高速度。这是因为无论是写操作本身,还是之后对其进行的读操作,由于只是内存之间的拷贝,其速度比不施加 WOB 策略要快;由于对用户文件的写入集中在设置检查点时进行,是一种化整为零的写入方式,其速度也比不施加 WOB 策略快。但当内存缓冲区不足时,其性能将有所下降。使用硬盘缓冲区时,用户进行的一次写操作将对应着同样规模的两次写操作和一次读操作:将该写操作内容写入硬盘缓冲区中,检查点设置时将该写操作内容从硬盘缓冲区中读出,并写入用户文件。因此,当内存缓冲区远不能满足需求时,WOB 引入的开销还是无法忽略的。因此,有必要在上述基本思想的基础上对 WOB 策略进行性能优化。

##### 3.2.1 合理选择内存缓冲区大小

由于主存空间大小有一定限制,而程序占用内存随时间变化,所以有必要根据主存剩余空间的大小为进程分配合适大小的内存缓冲区。但是,由于在一个检查点间隔内进程所需的缓冲区的总大小不仅与具体应用程序有关,而且与检查点的设置间隔长短有关,所以内存缓冲区大小的选择是一个比较复杂的问题。因此,这种优化就是在保证内存剩余空间的大

小不低于一个阈值的前提下,考虑上述各种因素,在一个检查点间隔内,动态地选择一个合适大小的内存缓冲区。

### 3.2.2 时延隐藏

由上面的分析可知,使用硬盘缓冲区将导致检查点设置时的额外开销。通过采用时延隐藏优化技术,可以把这些额外开销隐藏起来:把文件检查点设置与程序的计算并行进行,使得 WOB 策略下的文件检查点设置并不直接影响程序的运行时间。

### 3.3 数据结构

为了保存和恢复用户文件状态,要在进程数据段中开辟一个活动文件信息表,为每一个活动文件分配一个表项,信息表结构如表 1 所示。表中主要域有:wasClosed 和 wasDeleted 用于标记该文件是否被关闭、删除了;FilePointer 和 FileSize 分别为文件指针和文件大小。AMLHead 指向该文件对应的地址映射链表 AML (Address Mapping List) 的表头。AML 表的节点结构如表 2 所示。表中的 Start、End 以及 BufferStart、BufferEnd 分别表示该区域在原文件中以及缓冲区中的起始和终止地址;MemOrDisk 用于表明该区域的缓冲区的位置。

表 1 活动文件信息表结构

FileName	Fd	OpenMode	FilePointer	FileSize
Duplist	WasDeleted	WasClosed	AMLHead	
MemBufferStart	MemBufferPointer	MemBufferSize	MemBufferFull	
DiskBufferFd	DiskBufferPointer			

表 2 AML 表节点结构

Start	End	MemOrDisk	BufferStart	BufferEnd	Next
-------	-----	-----------	-------------	-----------	------

### 3.4 实现

实现 WOB 策略的关键在于正确维护 AML 表,其中最主要的工作是根据所读写的区域的起始和终止地址访问 AML 表,从而确定所读写区域中哪部分是原 AML 表中有的,哪部分是原 AML 表中没有的,并进行相应的操作。用符号  $(s, e)$  代表 AML 表中的一个节点,其中  $s, e$  分别表示 Start 域和 End 域,AML 表中的节点始终以  $(s, e)$  的升序排列。设当前 AML 表的结构如下:

$$(s_1, e_1) \quad (s_2, e_2) \quad \dots \quad (s_n, e_n),$$

$$(\text{其中 } s_1 < e_1 < s_2 < e_2 < \dots < s_n < e_n)$$

要读写的区域起始和终止地址分别为 min 和 max,若有

$$\min < s_i < e_i < \dots < s_j < e_j < \max$$

则对于区域  $(\min, s_i - 1), (e_i + 1, s_{i+1} - 1) \dots (e_j + 1, \max)$  (区域类型 1):在写操作时,将内容写到相应的缓冲区的末尾,创建新的 AML 表项并插入在适当位置以确保 AML 表按  $(s, e)$  的升序排列;在读操作时则直接从原文件中读出内容。而对于区域  $(s_i, e_i), (s_{i+1}, e_{i+1}) \dots (s_j, e_j)$  (区域类型 2):在写操作时,将内容直接写到相应的缓冲区中,覆盖原有区域;在读操作时则从缓冲区的相应区域读出内容。

在许多实际应用中,程序常常是反复调用 WRITE 系统调用,每次将一定大小的数据顺序写入用户文件中的一段连续区域。这种连续多次的写操作,在 WOB 的缓冲区中对应一段连续的区域,在 AML 表中只对应着一个节点。这种处理可以

减小 AML 表的维护开销,也可以加快文件检查点设置的度。

```

WRITE()
{
    计算所写区域的起始地址和终止地址;
    在 AML 表中搜索文件区域类型 1 或 2;
    while(未到 AML 表末尾){
        if(如果区域属于类型 1){
            将该区域的内容写到缓冲区末尾;
            if(能合并)
                合并,修改原节点的 BufferStart, BufferEnd 域;
            else
                创建新节点并插入 AML 表中;
        }else/* 区域属于类型 2 */
            将该区域的内容写到缓冲区相应位置;
        在 AML 表中搜索文件区域类型 1 或 2;
    }/* end while */
    更新文件信息表中的 FilePointer 域;
}/* end WRITE */

```

图 3 封装后的 WRITE 流程

```

Checkpoint()
{
    if(fork() == 0)
    {
        /* 专门用于设置检查点的子进程 */
        遍历 AML 表,将缓冲区内容写入原文件;
        删除 wasDeleted 域为 TRUE 的文件;
        关闭 wasClosed 域为 TRUE 的文件;
        释放内存缓冲区,清空硬盘缓冲区;
        捕获其它状态,写入检查点;
        重新分配内存缓冲区;
    }else/* 父进程 */
        return;
}

```

图 4 优化的 WOB 文件检查点设置

为了实现上述 WOB 策略,还必须对 OPEN、READ、WRITE、CLOSE、LSEEK、UNLINK 等文件操作系统调用进行封装。也就是说,提供与原系统调用接口完全相同的函数。在这些函数中截获用户所要进行的文件操作,并根据 WOB 策略维护文件信息表和 AML 表。这一封装工作对用户来讲是透明的,即用户不必为利用 WOB 策略对源程序作任何修改。封装后的 WRITE 的流程如图 3 所示。

封装后的 OPEN 要完成在文件信息表中添加相应项、设置初值,以及创建 AML 表和缓冲区等工作;封装后的 CLOSE 不进行真正的关闭操作,而是将文件信息表中对应文件的 wasClosed 域设为 TRUE;封装后的 UNLINK 不进行真正的删除操作,而是将文件信息表中对应文件的 wasDeleted 域设为 TRUE;封装后的 LSEEK 先要根据文件信息表中的 FilePointer 域和 FileSize 域进行修正,再进行所要求的操作;封装后的 READ 通过查找 AML 表,将文件内容从内存缓冲区、硬盘缓冲区、或硬盘文件中读出。

采用了性能优化措施后的检查点设置流程如图 4 所示。

进行了文件检查点设置后,文件内容与全局状态达到一致.

#### 4 相关工作

libckpt<sup>[3]</sup>在打开文件时把文件大小存于硬盘,当某段文件区域即将被修改时或文件即将被删除时,对整个文件进行备份.当发生 FARE 故障时,把文件截成原来的大小;当发生 FARW 或 FAD 故障时,使用备份文件来恢复用户文件内容.

上述方法改进版本是一个单独的库 libfcp<sup>[4]</sup>.它采用的是 *inplace update with undo logs* 的策略:在设置完一次检查点之后,当某块文件区域即将被修改时,先要根据写操作的内容和当时文件的内容组织出对应的 UNDO 操作,并将其保存到硬盘上的 UNDO 日志文件中.一旦进程发生故障并卷回到上一检查点重新执行,则按照从后向前的顺序逐一执行记录在日志文件中的 UNDO 操作,将从上一检查点开始到故障前一刻被修改的文件内容恢复.这种策略使得每次正常的写操作必须引入额外读操作和写操作各一次,而且必须等组织完 UNDO 操作并写入 UNDO 日志文件之后才能对原文件进行下一次写操作,其带来的正常执行开销是相当大的.

与 libckpt 和 libfcp 的方法相比,本文提出的 WOB 策略具有如下优点:(1)空间开销最小.由于一个检查点间隔内,用户文件中可能只有很小一部分内容发生改变,libckpt 中对整个文件进行备份的方法空间开销很大.由于只对要写入用户文件的内容进行了缓存,WOB 引入的空间开销比 libckpt 的开销要小得多.由于对同一区域的多次写操作只占用一块缓冲区,WOB 的空间开销也比 libfcp 小.(2)正常运行时间开销最小.在 libckpt 中,每个检查点间隔内的第一次写操作必须在整个文件备份完毕后才能进行,这引入了较大的正常运行时间开销;libfcp 为每次正常的写操作引入的额外开销也无法隐藏.而 WOB 策略由于采用了动态内存缓冲区分配和时延隐藏技术,给正常运行带来额外时间开销较小,甚至有可能提高程序运行速度.即使在 WOB 的最差情形(将内存缓冲区大小设为零并且不进行时延隐藏),即一次写操作也引入额外的读操作和写操作各一次,其开销也不大于 libfcp 的开销.(3)恢复时间最小.在发生故障卷回恢复时,只需清空硬盘上的缓冲区,而不需对用户文件做任何改动.这所花费的时间极少,远比 libfcp 耗费 CPU 资源的逆序恢复开销小得多,也比 libckpt 的磁盘文件拷贝的开销小.

#### 5 结论和进一步的工作

本文提出的 WOB 策略,实现了对用户活动文件的检查点

设置,支持用户进程进行任何文件操作,有效地解决了在发生故障时用户文件的卷回恢复的问题.它对用户透明,并通过优化设置内存缓冲区大小、时延隐藏等手段,使得这种策略在空间开销、正常运行时间、恢复时间等性能指标上优于其它方法.

在今后的工作中,我们准备对 WOB 策略做进一步的性能优化.首先是设计出更高效的缓冲区大小动态选择算法,另外还要使文件检查点设置与其它状态的保存工作并行进行,以减小开销.

#### 参考文献

- [1] T. Tannenbaum and M. Litzkow, The Condor Distributed processing system. Dr. Dobbs' Journal, Feb. 1995, (2): 40 ~ 48
- [2] J. S. Plank, M. Beck, G. Kinsley. Libckpt: Transparent Checkpointing under Unix. in Usenix Winter 1995 Tech. Conference, Jan. 1995: 213 ~ 223
- [3] Y. M. Wang, Y. Huang, et al. Checkpointing and Its Applications. Proceedings of IEEE 25th Symposium on Fault-Tolerant Computing, June 1995: 22 ~ 31
- [4] P. E. Chung, Y. Huang, et al. Checkpointing in CosMic a User-level Process Migration Environment. Pacific Rim International Symposium on Fault-Tolerant Systems, Dec. 1997: 187 ~ 193
- [5] 汪东升,沈美明,郑纬民,裴丹. 一个基于检查点的并行程序卷回恢复和进程迁移系统. 软件学报, Jan. 1999, 10(1): 68 ~ 73



裴 丹 1973 年生,硕士研究生,1997 年获清华大学工学学士学位.主要研究兴趣为分布/并行处理、容错计算.



汪东升 1966 年生,博士,副教授.主要从事分布/并行处理、容错计算等方面的教学和科研工作.近年来在国内外刊物上发表论文 30 余篇.