软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(10):3309−3320 [doi: 10.13328/j.cnki.jos.005814]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

并行帧缓存设备:基于多核 CPU 的 Xorg 并行显示优化*

高 珑[1], 戴华东[2], 杨沙洲[1], 丁 滗[1]

[1](国防科技大学 计算机学院,湖南 长沙 410073)

[2](军事科学院 国防科技创新中心,北京 100091)

通信作者: 高珑, E-mail: longgao@nudt.edu.cn

摘 要: Xorg 图形服务器软件在帧缓存设备上采用单线程绘制模式,难以发挥多核 CPU 的性能.针对多核 CPU 上的帧缓存设备,设计了带有互斥操作的任务队列,并按照屏幕划分的方法,实现了 Xorg 的矩形填充操作在帧缓存设备上基于私有任务队列的多线程并行化,并实现了主从线程负载均衡.x11perf 测试结果表明,该算法在一台 4 核商用台式机上的加速比可以达到 2.06.

关键词: Xorg;帧缓存设备;嵌入式;并行算法;多核 CPU

中图法分类号: TP316

中文引用格式: 高珑,戴华东,杨沙洲,丁滗.并行帧缓存设备:基于多核 CPU 的 Xorg 并行显示优化.软件学报,2020,31(10):3309−3320 http://www.jos.org.cn/1000-9825/5814.htm

英文引用格式: Gao L, Dai HD, Yang SZ, Ding Y. Parallel frame buffer device: Graphics acceleration based on multi-core CPU for Xorg. Ruan Jian Xue Bao/Journal of Software, 2020,31(10):3309−3320 (in Chinese). http://www.jos.org.cn/1000-9825/5814.htm

Parallel Frame Buffer Device: Graphics Acceleration Based on Multi-core CPU for Xorg

GAO Long[1], DAI Hua-Dong[2], YANG Sha-Zhou[1], DING Yan[1]

[1](College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

[2](National Innovation Institute of Defense Technology, Academy of Military Sciences, Beijing 100091, China)

Abstract: Xorg server is running in single-threaded mode on frame buffer device, which is hard to obtain good performance on multi-core CPU. For frame buffer device on multi-core CPU, a task queue is designed with mutual-inclusion, screen is split into several sub-screens, and each sub-screen is attached with a thread to draw rectangles within that sub-screen. A private task queue for each thread is used to hold their own tasks to draw rectangles, and load balance is kept between the main thread and each sub-thread. Results of x11perf show that rectangles filling operation could reach a speed-up ratio of 2.06 on a 4-core DELL desktop computer.

Key words: Xorg; frame buffer; embedded; parallel algorithm; multi-core CPU

1 引 言

Xorg 图形服务器起源于 20 世纪 80 年代初,是 Unix/Linux 系统上最基本的图形交互系统[1],大致架构如图 1 所示.Xorg 图形服务器采用 Client/Server 设计思想,具备卓越的灵活性和强大的功能,成为 Unix/Linux 中缺省的主流图形交互环境.但是基于 Client/Server 的 X 协议的开销,使 Xorg 图形服务器也一直因为效率和性能方面的问题而饱受诟病[2].

* 基金项目: 国家核高基重大专项(2017ZX01038-104-002); 国家自然科学基金(61502510)

Foundation item: National Fundamental Science and Technology Foundation of China (2017ZX01038-104-002); National Natural Science Foundation of China (61502510)

收稿时间: 2017-03-24; 修改时间: 2018-10-02; 采用时间: 2019-01-15

针对 Xorg 图形服务器的效率和性能问题,有过不少改进的尝试,比较著名的有 WayLand[3−6],如图 2 所示. WayLand 改进了 Client/Server 的工作模式,转而采用 Client/Compositor 模式.但 WayLand 对于帧缓存设备并无更多改进,对于嵌入式设备上的帧缓存设备的性能并无改善.

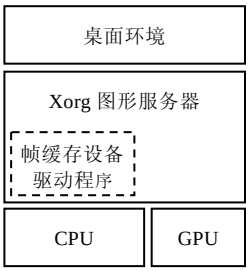


Fig.1 Xorg server

图 1 Xorg 图形服务器

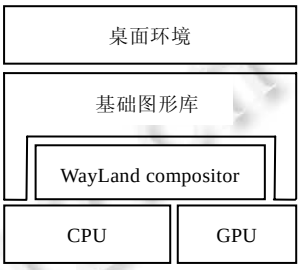


Fig.2 WayLand compositor

图 2 WayLand

资源受限的嵌入式操作系统常采用帧缓存设备作为 Xorg 的绘图设备.随着多核 CPU 的发展,目前嵌入式系统也逐步采用多核嵌入式 CPU.但是 Xorg 对于帧缓存设备仍然采用单线程串行的使用模式,导致在进行大量绘制操作时,容易造成 CPU 单核负载过高,而其他 CPU 核心处于相对空闲的状态,甚至影响到整个 Xorg 的交互和显示体验.对于 Xorg 的帧缓存设备进行并行化研究,不仅对于加速帧缓存设备上的绘制操作具有意义,也将有利于在 Xorg 上建立多用户交互通道的研究.

本文将试图从优化 Xorg 图形服务器的帧缓存设备入手,研究如何在多核 CPU 上优化 Xorg 图形服务器的显示性能.本文以下部分提及的 Xorg 图形服务器简称为 X.

1.1 主事件循环

与大部分系统服务进程类似,X 中最主要的部分是一个名为 Dispatch 的无限循环,称为主事件循环.其伪算法采用单线程模式运行,可以简要描述如图 3 所示.

- 首先,在步骤①中,X 将睡眠等待,直到被系统通知唤醒(包括鼠标键盘等输入事件触发的 SIGIO 信号以及用户的请求等);
- 然后,在步骤②中,X 将输入信息处理成标准事件放入事件队列中,并将各个事件发送给对相应事件感兴趣的客户端程序;
- 最后,在步骤③中,X 完成客户端请求的服务,一般是请求图形绘制等服务.如无任何输入和服务请求发生,则 X 将继续睡眠等待.整个循环周而复始,直到 X 被异常条件终止为止.

```
PROCEDURE Dispatch
BEGIN
    WHILE (Exception=False){
        WaitForSomething(·)                    ①
        If (Input){                            ②
            Event=ProcessInput(Input)
            SendtoClient(Event)
        }
        If (Request)                           ③
            ProcessRequest(Request)
    }
END
```

Fig.3 Main loop dispatch of Xorg

图 3 Xorg 图形服务器主事件循环

可以看到:目前 X 对于用户输入、事件处理、响应用户请求等的处理依然采用串行处理方式,如果上一个

主事件循环中的客户端请求还没有处理完,X 就难以及时处理用户在下一个循环中的交互输入并做出响应.在 CPU 单核性能较弱或者系统重负载的情况下,这种串行处理方式将会使 X 的交互体验变得很差.如果在军事指挥控制等领域出现这种情况,将可能导致严重的后果.

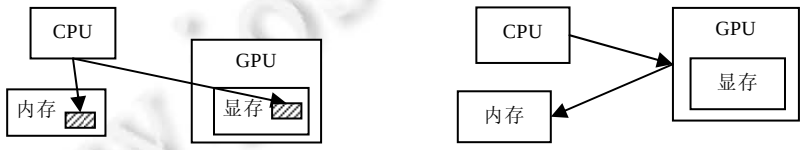
针对这种问题的并行化优化的思路主要有两个方向.

(1) 在步骤③中让 X 尽快并行处理完客户端程序的请求,如图形绘制请求等,本文主要按此方向展开讨论和研究;

(2) 多个子线程同时并行处理主事件循环,该方向思路将在第 5.3 节中简要地加以讨论.

1.2 帧缓存设备

帧缓存设备是从 Linux 内核 2.2 版本开始引入的,被广泛应用在嵌入式领域[7−13].帧缓存设备对应内存或者 GPU 显存的一部分存储空间,如图 4(a)中阴影部分所示.这部分空间内放置的数据恰好对应于屏幕上显示的图像,Xorg 通过向帧缓存设备写入数据完成绘制操作,进而画出各种图形,大量绘制时 CPU 负载较重.与之相对应,在 GPU 硬件加速模式中,CPU 则仅完成 GPU 指令和数据在内存的设置工作,随后由 GPU 完成剩余的图形绘制,CPU 负载较轻,如图 4(b)所示.



(a) 帧缓存设备 (b) GPU 硬件加速

Fig.4 Frame buffer device and hardware accelerated

图 4 帧缓存设备和 GPU 硬件加速

由于 CPU 性能的不断提高,帧缓存设备的性能也不断提高.同时,由于很多现代 CPU 指令集中也逐步加入了支持多媒体和图形图像处理的 SIMD 指令[14−19],例如 Intel 指令集中的 MMX 指令和 SSE 指令[20−22]、AMD 指令集中的 3DNow!指令[23,24]、ARM 指令集中的 NEON 指令[25,26]等,使得很多现代 CPU 在图形图像处理方面也有长足进步,CPU 逐步获得较高的图形绘制能力.例如:SSE 指令在 Geometry Processing 中可以获得接近 4 倍的性能提升[22];使用 ARM 的 NEON 指令优化后,不少应用的性能可以提高 4~8 倍[27],包括 X 的绘制函数 *fbCompositeSolidMask* 的速度也提高了 8 倍[27].由于在 pixman 函数库中通常已经包含对于 SSE 和 NEON 等 SIMD 指令的汇编指令级优化,所以 X 一般通过直接调用优化过的 pixman 图形函数库来利用这些 CPU 的 SIMD 指令.另外,一般由于业界的 GPU 大厂商,如 AMD、Nvidia、ARM 等都不完全开放 GPU 驱动源码和硬件接口协议,导致 Linux 桌面上的 GPU 驱动发展相对滞后,性能相对 Windows 的商业闭源 GPU 驱动低很多,在国产 CPU 领域尤其如此.所以在某些嵌入式场景中,X 使用帧缓存设备获得的性能甚至超过 X 使用开源 GPU 驱动所获得的性能[28,29].

在多核 CPU 上进一步提升帧缓存设备的性能,需在多核 CPU 上针对帧缓存设备开发线程级并行处理(thread level parallelism,简称 TLP)[30,31].即:在多核之间同时并行处理数千条以上规模的指令序列[32],才能有效弥补核间通信与同步的开销.

1.3 矩形填充

图形上下文 GC(graphics context)是 X 中所有绘制操作的核心数据结构,所有的基本图形绘制都要通过 GC 来完成.与文件系统读写操作类似,每一个 GC 都有一组相对应的 GCOps 函数指针来完成图形绘制.GCOps 包含画点、画线、画矩形、画椭圆、传输拷贝等 20 余种绘制操作函数指针.对于帧缓存设备来说,所有的绘制操作,都是通过 CPU 逐个改变像素点来实现的.对于其中任意一种绘制操作的并行优化,原理上对于其他绘制操作都是适用的.本文主要针对矩形填充操作进行优化,其他绘制操作可以用类似的方法来实现.

可绘制对象 Drawable 是 X 中所有可绘制对象的基本抽象结构,所有 GCOps 的操作都在某一种可绘制对象 Drawable 上完成.Drawable 又分为 Pixmap 和 Window 两种,一般认为,Pixmap 为单纯图像的位图,而 Window 则可能包含标题栏、菜单栏、按钮栏和窗体等组成部分.本文算法主要针对 Window 类型的 Drawable 对象实现.

矩形填充函数 *PolyFillRect* 是帧缓存设备的 GCOps 的绘图操作函数指针之一,专门用来在帧缓存设备上填充绘制由纯色填满的矩形.相应的参数包含 4 部分.

1) 指向 Drawable 对象的指针 *pDrawable*;
2) 指向 GC 的指针 *pGC*;
3) 该次绘制的矩形数量 *nrect*;
4) 该次调用需要填充的矩形结构链表的指针 *prect*.

PolyFillRect 一般随后会调用单个矩形绘制函数,例如 *fbFill*,逐个绘制链表 *prect* 所指向的所有矩形,*fbFill* 的前 2 个参数一般与 *PolyFillRect* 相同,后 4 个参数一般是矩形的左上顶点的坐标值(x,y)以及宽度 w 和高度 h.本文算法就是针对在帧缓存设备上的矩形绘制进行多线程优化的,在单个矩形绘制函数 *fbFill* 中生成任务并分发给多个子线程完成,其他绘制操作可以按照类似方法进行优化.

2 相关工作

针对 X 相关的多线程优化,已有的工作包括多调度队列算法、X 的输入线程化、CPU 和 GPU 协同多线程等.Linux 内核调度器的多调度队列算法打破了集中调度的局限性,采用分散的多调度队列,降低了全局调度开销.不过,针对其他具体应用场景,仍需要根据算法思想进行重新设计和优化.X 的输入线程化是指将处理鼠标键盘事件的过程作为单独的线程分离出来,但并未进行多线程处理,也不涉及帧缓存设备的优化.CPU 和 GPU 协同多线程优化主要涉及 CPU 端的数据准备和控制等方面的多线程优化,以便充分发挥高性能 GPU 的绘制性能起辅助功能,但不包含 CPU 端针对帧缓存设备的优化工作.

2.1 多调度队列

在并行计算中,单调度队列带来的性能开销往往都是 $O(n)$的,全局单一的任务调度队列往往会带来访问冲突、性能开销大等问题.例如:Linux 内核在 2.4 版本之前的调度器采用全局单一调度队列,调度的开销是 $O(n)$;而 2.6 版本之后引入的 $O(1)$调度器针对每个优先级分别设置私有的任务调度队列,将调度开销降低到 $O(1)$的常数时间,可以做到调度开销与系统中线程个数无关.CFS 调度器则为每一个 CPU 核心维护一棵以时间为顺序的红黑树,确保实现接近完全公平的进程调度[33].但针对其他具体应用场景仍需重新设计算法.

2.2 输入线程化

在通常的处理流程中,X 图形服务器在没有任何输入时一直保持睡眠状态,直到被鼠标键盘的输入事件激发的 SIGIO 信号唤醒.因为信号处理开销较大,也容易发生无效唤醒(lost wakeup)的问题,开源社区已在尝试放弃原先沿用的 SIGIO 信号,改用独立的单个线程处理输入事件.例如,Keith Packard 这几个月以来提交的 threaded input 补丁[34],现在已经合并到主分支加以发布.输入线程从主事件循环中独立出来,将有效地减小主事件循环中鼠标键盘的处理过程对图形绘制的影响,改造后,如图 3 所示,X 图形服务器根据用户请求绘制图形的过程(process request)将不必再等待鼠标键盘处理过程(processInput)完成,而可以立即执行.但处理鼠标键盘的输入线程仍为一个,没有考虑多用户的情况,具体可以参考第 5.3 节的讨论.

2.3 CPU和GPU协同多线程

即使在图 4(b)所示的 GPU 绘制模式中,CPU 也具有重要作用:CPU 将协助 GPU 准备所需的数据并进行控制.由于现在很多 GPU 性能越来越高,所以不少情况下 CPU 端的计算也会成为系统的瓶颈,此时进行 CPU 的多核优化将有助于消除这种绘制过程中的瓶颈.例如,最新的 Windows 平台上的图形开发库 DirectX 12 和 Linux 的下一代 OpenGL Vulkan 都已经实现了针对多核 CPU 的优化,图形渲染性能成倍提高[35,36].

3 帧缓存并行显示优化算法

为了实现帧缓存设备的并行化,并最大程度地避免不同子线程之间的数据相关性,我们对屏幕进行划分,划分后的每一块子屏幕对应一个子线程,仅绘制位于该子屏幕中的图形,如图 5 所示.本文中仅针对矩形填充操作 *SolidFill* 进行了实现,其他图形绘制操作的实现与此类似.

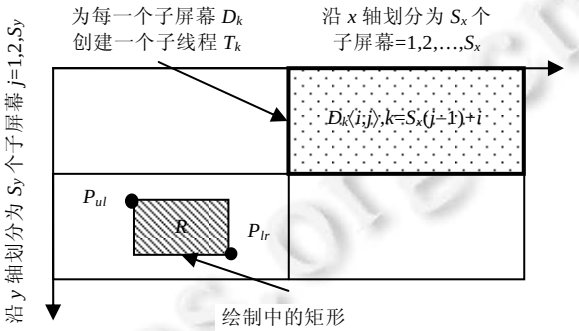


Fig.5 Drawable screen partition

图 5 Drawable 分屏

3.1 屏幕划分

本文第 1.3 节中已经介绍过可绘制对象 Drawable,X 中的每一个 Drawable 都对应屏幕上某一块矩形可绘制区域,即窗口对象 Window 或者位图对象 Pixmap 在屏幕上所占据的区域.图 5 中的最大矩形就代表某 Drawable 对应的可绘制区域,由横坐标 x 轴和纵坐标 y 轴标示.一般使用矩形的左上顶点 P_{ul} 和右下顶点 P_{lr} 的像素坐标值即可唯一确定该矩形的位置和大小.

如图 5 所示,在并行帧缓存算法中,将帧缓存设备上的每一个 Drawable 对象对应的矩形屏幕按照 x 轴和 y 轴分别平均分成 S_x 和 S_y 份,这样每一个 Drawable 对象对应的矩形屏幕就被均分成互不相交的 $S_x \cdot S_y$ 个子屏幕,每一个子屏幕用 $D_k\langle i,j\rangle$ 来表示,其中,i 和 j 分别代表 x 轴和 y 轴上从 1 开始的区间编号,其中,$k=S_x(j-1)+i$,易知 k 的范围 $1\leqslant k\leqslant S_x \cdot S_y$.根据子屏幕的划分方法,有:

分屏规则.给定矩形 R 的左上顶点 P_{ul} 以及子屏幕 D,$R\subset D$,当且仅当 $P_{ul}\in D$.

也就是说:仅当矩形的左上顶点 P_{ul} 属于某个子屏幕 D 时,该矩形才属于该子屏幕 D.我们为所有的子屏幕 D_k 创建一个子线程 T_k,将所有属于子屏幕 D_k 的矩形 $R\subset D_k$ 的填充绘制任务交给与子屏幕 D_k 绑定的子线程 T_k 完成.对于右下顶点 P_{lr} 可能超出其所在子屏幕 D 的矩形,我们将在第 5.1 节中给出讨论.

3.2 私有任务队列

并行帧缓存设备算法参照单生产者多消费者模型[37]实现.在经典的生产者消费者模型中仍然存在一些竞争冲突和遍历开销等问题.为了改进这些问题,并行帧缓存设备算法中的每个消费者都采用了独立的私有任务队列作为缓冲区,如图 6(b)所示.矩形绘制任务将由生产者按照分屏规则分发到消费者各自的私有队列中,再由消费者从各自的私有队列中取出完成.由于分屏规则计算量不大,所以生产者分配引入的串行开销在可以接受的范围内.

并行帧缓存设备算法如图 6(b)所示,T_0 代表生产者主线程,$T_k(1\leqslant k\leqslant S_x \cdot S_y)$代表绑定子屏幕 $D_k(1\leqslant k\leqslant S_x \cdot S_y)$ 的消费者子线程.生产者算法如图 7 所示,消费者算法如图 8 所示,生产者和消费者公用的处理子函数 *Process*(q) 如图 9 所示.

图 6(b)中就绪队列 Q_k 长度 N 以及运行队列 q_k 长度 M,实际上已成为并行帧缓存设备算法的两个主要参数,在第 4 节的实验中可以看到不同的 M 和 N 的取值对于算法性能的影响.

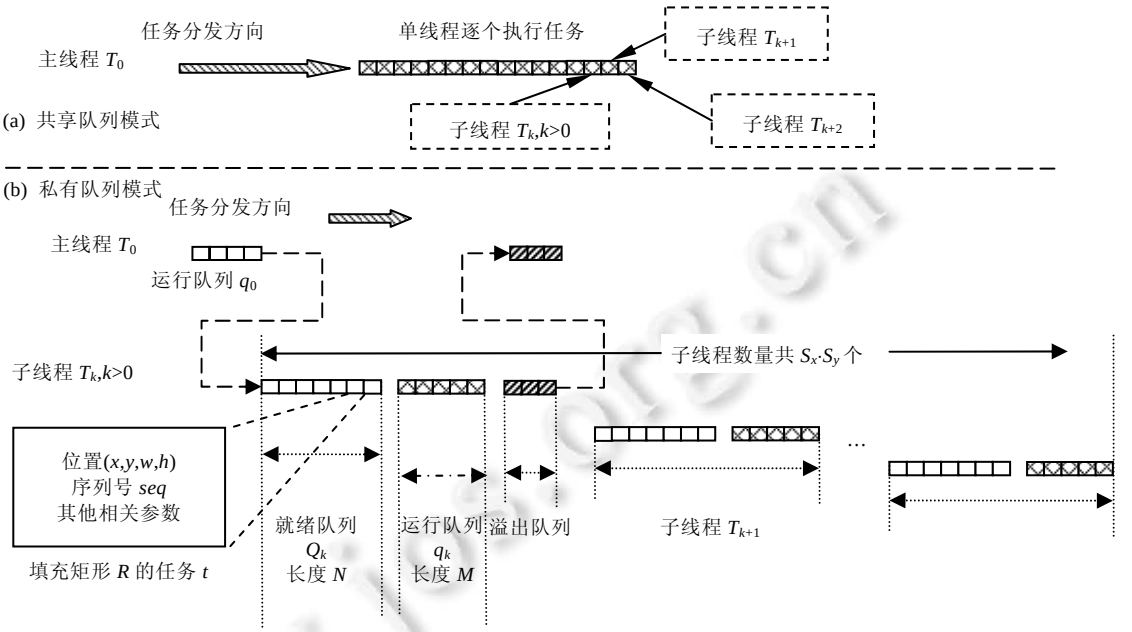


Fig.6 Multi-task queues

图 6 多任务调度队列

PROCEDURE *Master*(*pDrawable*,*pGC*,*x*,*y*,*w*,*h*)
INPUT:Drawable 指针 *pDrawable*,GC 指针 *pGC*,矩形坐标 *x*,*y*,矩形大小 *w*,*h*;
OUTPUT:子线程就绪队列 Q_k.
BEGIN
    建立 S_x·S_y 个子线程
    新建任务 *R*(*x*,*y*,*w*,*h*,*seq*++)
    计算 *R* 应当归属的子屏幕和子线程序号 *k*
    将任务 *R* 加入子线程 T_k 的就绪队列 Q_k,并唤醒所有等待 Q_k 的子线程
    IF (Q_k 的长度>*N*) **DO**{
        取出 Q_k 的最多前 *M* 个任务加入到主线程 T_0 的运行队列 q_0
        Process(q_0)
    }
END

Fig.7 Master T_0 thread

图 7 主线程 T_0 算法

PROCEDURE *Worker*(Q_k)
INPUT:子线程就绪队列 Q_k;
BEGIN
    DO{
        IF (就绪队列 Q_k 的长度>0){
            从 Q_k 中取出包含最多 *M* 个任务放入子线程 T_k 的运行队列 q_k
            Process(q_k)
        } **ELSE**
            睡眠等待 Q_k
    } **WHILE** (True)
END

Fig.8 Worker T_k thread

图 8 子线程 T_k 算法

```
PROCEDURE Process(q_k)
INPUT:子线程运行队列 q_k
BEGIN
    WHILE (队列 q_k 不为空){
        弹出 q_k 头部任务 R
        绘制任务 R
    }
END
```

Fig.9 *Process* function

图 9 *Process* 子函数

3.2.1 任务封装

首先,单个矩形 $R(x,y,w,h,seq)$被封装成为独立的矩形填充任务 t,如图 6(b)所示,其中包含矩形的坐标$\langle x,y\rangle$、宽 w、高 h、标记时间顺序的序列号 seq 等内容,这样便于将多个矩形填充任务加入队列.序列号 seq 代表任务生成时的时间戳,如图 5 所示算法,每生成一次序列号 seq 加 1.seq 采用长整型类型 long int,一般假定 seq 在 Drawable 生存周期中不会溢出.因此,如果某任务的序列号较小,那么该任务的生成时间也较早.

3.2.2 条件等待

每一个任务队列都设有加入任务和弹出任务两个条件等待变量,分别用于控制向队列中加入任务和弹出任务.当任务队列 l 为空时,所有从队列 l 中弹出任务的线程,将自动进入睡眠状态等待,直到有线程向队列 l 中压入任务后,将自动唤醒所有等待任务的线程,继续从队列中弹出任务.同样地,若队列 l 的长度超出阈值后,所有向队列 l 中增加任务的线程将获得一个错误返回值,表明队列长度已经超过限定值;此时可以采取相应措施应对,本算法采取的是主子线程负载均衡的措施,具体见第 3.3 节中的讨论.图 7 和图 8 所示算法中,都隐含了操作队列时的唤醒和睡眠动作.

3.2.3 私有缓冲区

在标准的单生产者多消费者模型中[37],多消费者共享的公用缓冲区容易产生拥塞.如果采用共享的公用缓冲区,如图 6(a)所示,则将导致两个问题:(1) 共享的公用缓冲区将带来竞争冲突,某一个消费者在读写访问公用缓冲区时,其他消费者只能等待;(2) 每一个消费者遍历共享的公用缓冲区的开销较大,遍历开销与缓冲区的长度 N 成正比,如果共享的公用缓冲区较大,还将导致线程间的竞争开销急剧增长.

在并行帧缓冲算法中,每个消费者都具有私有的队列缓冲区.主线程既是生产者,也要负责将任务分配到适当的子线程中去.主线程将按照分屏规则把每一个矩形放入相应的消费者子线程的私有队列缓冲区中去.由于主线程分配任务时处于串行模式没有竞争,可以较快完成,所以这样既避免了消费者遍历共享的公用缓冲区挑选任务的开销,也避免了多消费者在访问共享的公用缓冲区时可能产生的竞争冲突.具体如图 6(b)所示,在并行帧缓存设备算法中,每个消费者子线程 T_k 都拥有两个相对独立的私有任务队列:就绪队列 Q_k 和运行队列 q_k.生产者主线程 T_0 产生矩形填充任务后,根据第 3.1 节中的分屏规则,将任务放入与子屏幕 D_k 绑定的子线程 T_k 的就绪队列 Q_k 中.每个消费者子线程 T_k 每次从自身的就绪队列 Q_k 中弹出不多于 M 个任务,放入自身的运行队列 q_k 中,再根据运行队列 q_k 中各个任务的内容逐个完成矩形填充的绘制工作.

3.2.4 就绪和运行双队列

与传统的生产者消费者模型不同,如图 6(b)所示,并行帧缓存设备算法中的每个消费者子线程 $T_k(1\leqslant k\leqslant S_x\cdot S_y)$都有两个分离的队列:就绪队列 Q_k 和运行队列 q_k,分别存放等待绘制的任务和正在绘制的任务.这样,当生产者线程独占就绪队列 Q_k 并向 Q_k 中添加产品时,消费者线程仍然可以独占运行队列 q_k 来消费产品,而不会产生互斥竞争;反之亦然.另一方面,消费者每次最多从就绪队列 Q_k 中弹出 M 个任务到运行队列 q_k,相比每次仅弹出 1 个任务就进行处理而言,也减少了消费者互斥访问就绪队列 Q_k 的频率,降低了与生产者之间发生私有任务队列访问冲突的概率.

3.3 主子线程负载均衡

在经典的生产者消费者模型中,当有限产品缓冲区满时,生产者都是处于空闲等待状态,等待消费者消耗缓

冲区里面的产品之后[37],生产者才能加入新的产品.但空闲的生产者可能浪费系统软硬件资源.

3.3.1 溢出队列

与经典生产者消费者模型不同,本算法设立了私有任务队列的溢出队列,用于主子线程间的负载均衡[38,39].如图 6 中的溢出队列所示:当某个子线程 T_k 的就绪队列 Q_k 长度大于队列长度预设阈值 N 时,生产者主线程 T_0 将暂停生产任务,并协助任务已满的子线程绘制矩形.具体就是从 Q_k 中弹出等待最久的若干任务形成溢出队列,并加入到主线程 T_0 的运行队列 q_0 中,由主线程 T_0 尽快地将溢出的任务绘制完成.当生产者主线程 T_0 完成所有子线程的溢出任务后,生产者主线程 T_0 才再次重新转换回生产者角色,继续生产任务.

这种主子线程间的负载均衡主要有以下好处.

1) 防止子线程任务队列长度的无限快速增长.如图 7 所示算法,如果不进行负载均衡,主线程 T_0 将任务放入恰当的子线程 T_k 的就绪队列 Q_k 后,并不需要做任何实际的矩形填充绘制工作即可返回,所以从调用主线程 T_0 的应用程序来看,矩形填充操作都将会以异常快的速度完成,而此时的任务实际上仍然在某个子进程的私有缓冲区中,并没有实际绘制完成.如果不对生产产品的速度加以限制,这将可能导致过长的就绪队列 Q_k.当 Drawable 所属的整个进程退出时,相应子线程将不得不耗费非常长的时间来处理并清空这些任务队列;
2) 防止图形界面进入无响应的等待状态.因为如果 Xorg 主线程 T_0 发现任务生产过多后,不进行负载均衡而是立即进入睡眠状态,这将导致 Xorg 进入无响应的等待状态,停止响应用户的鼠标键盘,图形交互系统也将失去响应,当前窗口将变灰无法操作.

值得注意的是:因为没有线程给生产者主线程 T_0 分配任务,所以 T_0 的就绪队列 Q_0 一直保持为空.

4 测试和结论

4.1 测试平台和用例

测试平台采用一台商用 DELL OPTIPLEX 3010 台式机,4 核心,4G 内存,操作系统采用 Ubuntu 15.10,内核为 4.2.0.测试用例采用通用的 x11perf -rect100 测试用例,如图 10 所示.

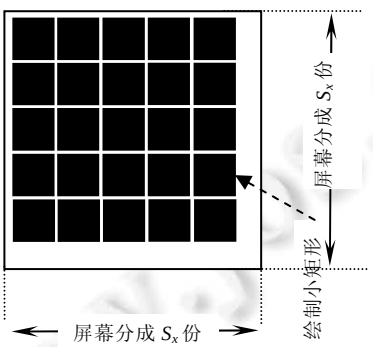


Fig.10 x11perf -rect100

图 10 x11perf -rect100 测试用例

上述的测试平台在单线程的普通帧缓存模式下,x11perf -rect100 的得分约为 454 000.在并行帧缓存算法中,将把图 10 所示的测试窗口按照 x 轴和 y 轴分别均分成 S_x 和 S_y 份,共被分为 S_x·S_y 个互不相交的子屏幕.测试结果如图 11~图 19 所示.

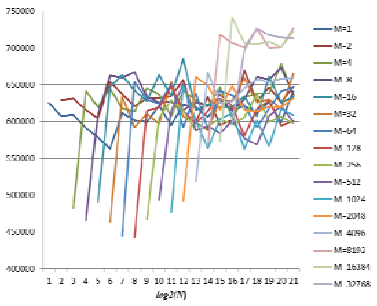


Fig.11 $S_x \cdot S_y$=1×1

图 11 $S_x \cdot S_y$=1×1

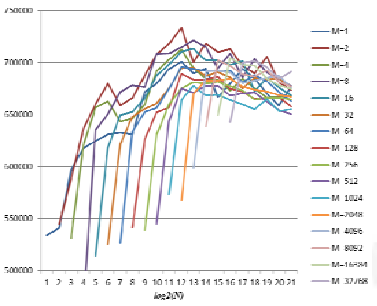


Fig.12 $S_x \cdot S_y$=2×1

图 12 $S_x \cdot S_y$=2×1

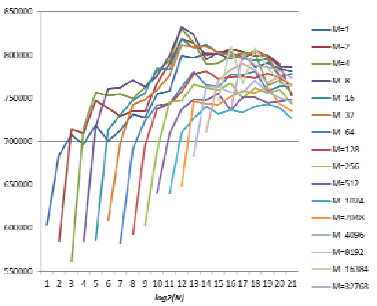


Fig.13 $S_x \cdot S_y$=1×2

图 13 $S_x \cdot S_y$=1×2

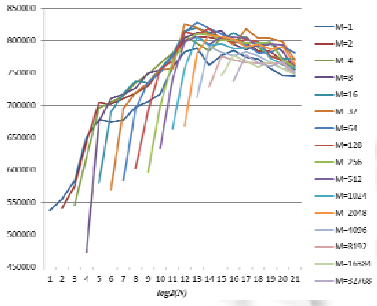


Fig.14 $S_x \cdot S_y$=3×1

图 14 $S_x \cdot S_y$=3×1

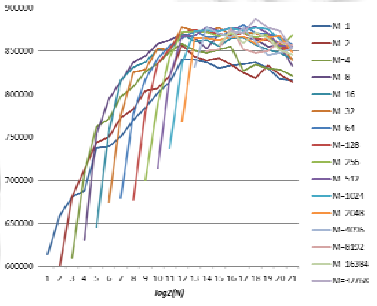


Fig.15 $S_x \cdot S_y$=1×3

图 15 $S_x \cdot S_y$=1×3

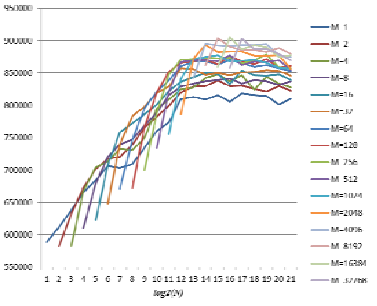


Fig.16 $S_x \cdot S_y$=2×2

图 16 $S_x \cdot S_y$=2×2

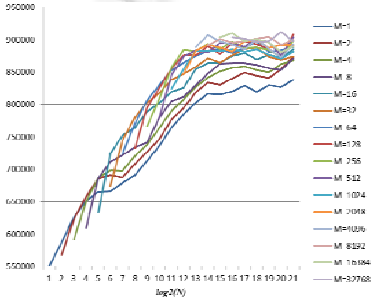


Fig.17 $S_x \cdot S_y$=3×2

图 17 $S_x \cdot S_y$=3×2

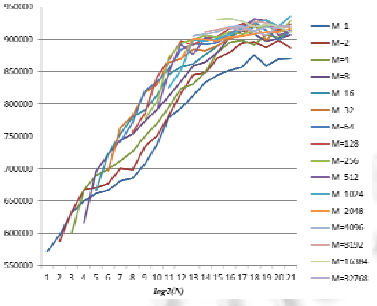


Fig.18 $S_x \cdot S_y$=2×3

图 18 $S_x \cdot S_y$=2×3

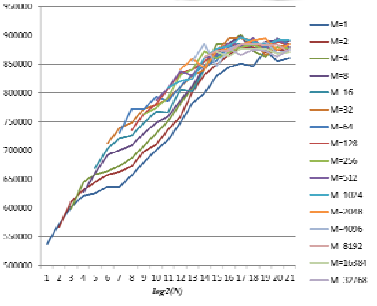


Fig.19 $S_x \cdot S_y$=3×3

图 19 $S_x \cdot S_y$=3×3

4.2 加速比分析

实验测试曲线如图 11~图 19 所示,因为私有就绪队列长度 N 数量级变化很大,所以采用 $\log_2(N)$表示图中横坐标.纵坐标为 x11perf 得分(得分值越高,性能也越高).不同颜色的曲线对应不同的私有运行队列长度 M 值.

从图 11~图 19,每一个图都代表一种不同的分屏方案.不同的分屏方案(S_x 和 S_y)之间对比,一般分屏数量越多,性能值越高,但超过 4 个分屏后,性能增加明显减少,甚至下降;每一个图内,每一条由相同的运行队列长度 M 对应的曲线,一般都随着 N 的增大其性能逐步提高,但都在大约 $\log_2(N)$=15 之后便不再明显上升反而有所下降;每一个图内,在任意 N 值附近,一般 M 越大性能越高,但当 M 超过 1 024 后,性能反而有所下降.

从图 11~图 19 可见:随着屏幕越分越细密(S_x 和 S_y 增大)以及子进程私有队列越来越长(M 和 N 越来越长),矩形绘制性能逐步提高,并且测试曲线逐步接近直线,加速比也逐步趋于线性.分析其原因,主要是由于随着屏幕划分的增多,可以创造出更多的线程,为充分发挥多核 CPU 的性能提供了可能;另外,M 和 N 的增加,也使多线

程同步和互斥的相对开销所占比例逐步降低,使得加速比曲线更加趋近线性.

但曲线也并非一直增长,大约在 $\log_2(N)=15$ 附近取得极值.考虑到本实验中 CPU 仅有 4 物理核心,所以创造出多于 4 个物理核心的线程后,反而会带来子线程等待和切换的开销,所以再增加分屏数目或者增加队列长度应当也不能再持续提高加速比,可以近似地认为该算法在参数 $M=1024$,$\log_2(N)=20$,$S_x \cdot S_y=2\times3$ 时取得加速比极大值 2.06.此时,x11perf 的得分约为 935 333.

假设算法中串行执行的部分为 a,因为本实验中可同时并行执行的线程最多为 4 个,根据阿姆达尔定律[40],理想状态下的加速比应当为公式(1)中等号左边的多项式.考虑到实际测试时的各种开销,那么实际测试所得极大值 2.06 应满足:

$$\frac{1}{a+(1-a)/4}>2.06 \tag{1}$$

所以可以推出 $a<0.3139$.所以,根据极限状态下的阿姆达尔定律,本算法的极限加速比将是串行部分比例 a 的倒数,即:

$$\frac{1}{a}>3.18 \tag{2}$$

上述分析可以得出结论:通过不断增加 CPU 的核心数目,该算法加速比的极限值至少可以达到 3.18;从另一个方面说,该算法仍然还有改进空间,例如将主线程 T_0 用于分配任务的串行计算时间以进一步优化压缩或者并行化.

5 未来的工作

5.1 顺序一致性

在本文算法中,如图 20 所示,如果矩形 R 的右下顶点 P_{lr} 超出所属子屏幕的范围,可能导致与其他子屏幕中的矩形 R' 有重叠.由于 R 和 R' 分属不同的子线程绘制,如果由于线程调度的原因,最终导致较早请求绘制的矩形遮挡了较晚请求绘制的矩形,可能会对用户交互产生影响.下一步将会在考虑顺序一致性的基础上对算法进行改进,例如改进矩形划分算法,将同时出现在屏幕上的所有矩形动态划分成互不相交的若干组,以避免不同线程之间的相互干扰.

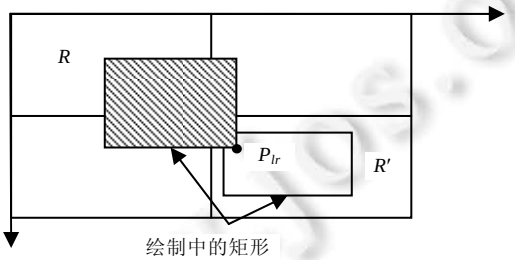


Fig.20 Sequence consistency

图 20 顺序一致性

5.2 子线程间的负载均衡

本文已经研究了主子线程间的负载均衡,可以避免出现单个子线程负担过重,但无法防止出现单个子线程负载较低的情况,这种情况下可能会影响算法的并行加速效果.虽然此时可以通过操作系统调度其他线程运行,但是如果一开始就把任务更加平均地分配给所有的子线程,那么将会有更好的加速效果.但从另外一个角度来讲,更精确的任务分配算法也会带来更大的开销,这需要进一步权衡和研究.

5.3 单桌面多用户

同一个 X 图形服务器,仅允许一个用户(每个用户使用一对鼠标键盘)使用.X 图形服务器的 XInput 子系统

可以在同一个桌面上支持多个相对独立的鼠标键盘,但是目前,测试 XInput 的多个鼠标键盘只能按照主从模式交替使用,还无法做到指针移动和点击等全部功能完全对等地同时使用.另外,XInput 子系统对于应用程序不透明,导致有些应用程序暂时还无法使用 XInput 子系统.

如果能够实现单桌面多用户,即同一个桌面允许多个用户(每个用户使用一对鼠标键盘)同时操作,与通过网络连接起来的多桌面多用户相比,就可以共享更多资源,并降低用户间的通信开销,提高多用户之间互操作和协作的效率,例如共用显存中已经渲染好的 3D 图像、使用共享内存作为高带宽通信等.结合多屏显示技术还可以通过一台终端并行完成多台显控终端的功能,减小显控终端的体积和重量.这对于车载或者机载条件下的协同指挥而言,可能具有潜在的应用价值.

将主事件循环完全线程化,是实现单桌面多用户的技术途径之一,还可以避免因为某个用户的某个耗时操作导致主事件循环的后续操作无法进行,以至整个桌面陷入无响应死锁状态的情况.本文中所研究的多任务队列,将为这些研究工作提供技术支持.

References:

[1] http://www.x.org

[2] Simson G, Weise D, Strassmann S. UNIX-Hater Handbook. IDG Books Worldwide, Inc., 1994.

[3] http://wayland.freedesktop.org

[4] https://en.wikipedia.org/wiki/Direct_Rendering_Infrastructure

[5] https://www.kernel.org/doc/html/latest/gpu/drm-internals.html

[6] http://dri.freedesktop.org/wiki/

[7] https://www.kernel.org/doc/Documentation/fb/framebuffer.txt

[8] Crow FC, Howard MW. A frame buffer system with enhanced functionality. ACM SIGGRAPH Computer Graphics, 1981,15(3): 63−69.

[9] Han K, Min AW, Jeganathan NS, *et al*. A hybrid display frame buffer architecture for energy efficient display subsystems. In: Proc. of the IEEE Int'l Symp. on Low Power Electronics & Design. IEEE, 2013. 347−352.

[10] Fournier A, Fussell D. On the power of the frame buffer. ACM Trans. on Graphics (TOG), 1988,7(2):103−128.

[11] Bando Y, Saito T, Fujita M. Hexagonal storage scheme for interleaved frame buffers and textures. In: Proc. of the ACM SIGGRAPH/Eurographics Symp. on Graphics Hardware. DBLP, 2005. 33−40.

[12] Shim H, Chang N, Pedram M. A compressed frame buffer to reduce display power consumption in mobile systems. In: Proc. of the 2004 Asia and South Pacific Design Automation Conf. 2004. 818−823.

[13] Fowler J, McGowen A. Design and implementation of a supercomputer frame buffer system. In: Proc. of the '88 ACM/IEEE Conf. on Supercomputing. 1988. 140−147.

[14] Boettcher M, Alhashimi BM, Eyole M. Advanced SIMD: Extending the reach of contemporary SIMD architectures. In: Proc. of the Conf. on Design. European Design and Automation Association, 2014. 1−4.

[15] Weiss M. Strip mining on SIMD architectures. In: Proc. of the 5th Int'l Conf. on Supercomputing. 1991. 234−243.

[16] Olson KM. Efficient tree codes on SIMD computer architectures. Computer Physics Communications, 1996,98(3):267−287.

[17] Fatoohi R, Field M. Performance analysis of four SIMD machines. In: Proc. of the 7th Int'l Conf. on Supercomputing. 1993. 117−126.

[18] Narayanan PJ. Processor autonomy on SIMD architectures. In: Proc. of the 7th Int'l Conf. on Supercomputing. 1993. 127−136.

[19] Chen Z, Nicolau A. Veidenbaum AV. SIMD-based soft error detection. In: Proc. of the ACM Int'l Conf. on Computing Frontiers. 2016. 45−54.

[20] Peleg A, Wilkie S, Weiser U. Intel MMX for multimedia PCs. Communications of the ACM, 1997,40(1):24−38.

[21] Krishnaprasad S. SIMD programming illustrated using Intel's MMX instruction set. Journal of Computing Sciences in Colleges, 2004,19(3):268−277.

[22] Ma WC, Yang CL. Using Intel streaming SIMD extensions for 3D geometry processing. In: Proc. of the Advances in Multimedia Information Processing. 2002. 1080−1087.

[23] Hensley J. AMD CTM overview. In: Proc. of the ACM SIGGRAPH. 2007. 7.

[24] Mache C. Multicore CPUs for the masses. Queue, 2005,3(7):64.

[25] Jang M, Kim K, Kim K. The performance analysis of ARM NEON technology for mobile platforms. In: Proc. of the 2011 ACM Symp. on Research in Applied Computation. 2011. 2685−2694.

[26] Pohl A, Cosenza B, Juurlink B. Control flow vectorization for arm neon. In: Proc. of the 21st Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES 2018). 2018. 66−75.

[27] http://www.arm.com/zh/products/processors/technologies/neon.php

[28] Siamashka S. New xf86-video-sunxifb DDX driver for Xorg. 2013. http://ssvb.github.io/2013/02/01/new-xf86-video-sunxifb-ddx-driver.html

[29] Siamashka S. Xorg drivers, software rendering for 2D graphics and Cairo 1.12 performance. 2012. http://ssvb.github.io/2012/05/04/xorg-drivers-and-software-rendering.html

[30] Olukotun K, Nayfeh BA, Hammond L. The case for a single-chip multiprocessor. In: Proc. of the ACM SIGOPS Operating Systems Review. 1996. 2−11.

[31] Gupta R, Epstein M, Whelan M. The design of a RISC based multiprocessor chip. In: Proc. of the '90 ACM/IEEE Conf. on Supercomputing. 1990. 920−929.

[32] Keckler SW. Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor. In: Proc. of the 25th Annual Int'l Symp. on Computer Architecture. 1998. 306−317.

[33] Wong CS, Tan IKT, Kumari RD. Fairness and interactive performance of *O*(1) and CFS Linux kernel schedulers. In: Proc. of the Int'l Symp. on Information Technology. 2008. 1−8.

[34] Packard K. Xfree86: Use threaded input mechanism. 2016. https://lists.x.org/archives/xorg-devel/2016-May/049671.html

[35] http://news.163.com/15/0324/09/ALFAVSSC00014U9R.html

[36] http://mb.zol.com.cn/537/5376644.html

[37] Cao XB, Chen XL. Operating System Principle and Design. Beijing: Machinery Industry Press, 2009 (in Chinese).

[38] Mattson TG, Sanders BA, Massingill BL, Wrote; Zhang YQ, Jia HP, Yuan L, Trans. Patterns for Parallel Programming. Beijing: Machinery Industry Press, 2015 (in Chinese).

[39] Ren Y, Liu L, Zhang Q, Wu QB, Guan JB, Kong JZ, Dai HD, Shao LS. Shared-memory optimizations for inter-virtual-machine communication. ACM Computing Surveys, 2016,48(4):1−42.

[40] Hennessy JL, Patterson DA, Wrote; Jia HF, Trans. Computer Architecture: A Quantitative Approach. 5th ed., Beijing: Machinery Industry Press, 2012 (in Chinese).

附中文参考文献:

[37] 曹先彬,陈香兰.操作系统原理与设计.北京:机械工业出版社,2009.

[38] Mattson TG, Sanders BA, Massingill BL,著;张云泉,贾海鹏,袁良,译.并行编程模式.北京:机械工业出版社,2015.

[40] Hennessy JL, Patterson DA,著;贾洪峰,译.计算机体系结构量化研究方法.第 5 版,北京:机械工业出版社,2012.

高珑(1978－),男,博士,副研究员,主要研究领域为嵌入式操作系统.

杨沙洲(1975－),男,博士,副研究员,主要研究领域为操作系统.

戴华东(1975－),男,博士,研究员,博士生导师,CCF 专业会员,主要研究领域为操作系统,人工智能,体系结构.

丁滟(1977－),女,博士,副研究员,CCF 高级会员,主要研究领域为操作系统,系统安全,可信云计算.