软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(1):132−148 [doi: 10.3724/SP.J.1001.2011.03899]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

网络程序设计中的并发复杂性*

李慧霸[1+], 田　甜[2], 彭宇行[1], 李东升[1], 卢锡城[1]

[1](国防科学技术大学 计算机学院 并行与分布国防科技重点实验室,湖南 长沙 410073)
[2](国防科学技术大学 计算机学院 软件研究所,湖南 长沙 410073)

Concurrency-Related Complexities in Network Programming

LI Hui-Ba[1+], TIAN Tian[2], PENG Yu-Xing[1], LI Dong-Sheng[1], LU Xi-Cheng[1]

[1](National Laboratory for Parallel and Distributed Computing, Computer School, National University of Defense Technology, Changsha 410073, China)
[2](Institute of Software, Computer School, National University of Defense Technology, Changsha 410073, China)
+ Corresponding author: E-mail: lihuiba@163.com

Li HB, Tian T, Peng YX, Li DS, Lu XC. Concurrency-Related complexities in network programming. *Journal of Software*, 2011,22(1):132−148. http://www.jos.org.cn/1000-9825/3899.htm

Abstract: The Internet has become a vital information infrastructure for modern society. However, the concurrent nature of network introduces a wide-range of difficulties in traditional programming methodology in developing high-quality network programs that significantly reduce productivity. The influence of concurrency on the complexity of software development is comparable to the “concurrency crisis” of software brought by multi-core processors, but it receives much less attention here than what it deserves. There is no universal approach to cope with this issue, and there are even disagreements between different approaches. In this paper, the basic concurrency models and their implementations are introduced, and then the paper surveys the inherent complexities of these approaches, comparing their advantages and disadvantages. Finally, this paper offers an opinion on the possibilities for future research on this topic.
Key words: network; concurrency; programming complexity; event-driven; stack-ripping; multi-threaded; future

摘　要: 互联网已成为现代社会的重要信息基础设施,然而网络环境的并发性使得传统程序设计方法在开发高质量的网络程序时遇到了许多困难,严重影响了开发效率.并发问题对网络程序开发复杂度的影响可以类比多核处理器带来的“软件并发危机”,然而其中的并发问题却远远没有得到应有的重视.网络并发问题目前并不存在普适的应对方法,甚至在不同方法之间存在明显的争论.简要介绍了各种基本的并发模型及其常见的实现方法,并在此基础上着重分析了现有方法的内在复杂性,对比各种方法的优势与劣势,最后展望可能的研究和发展方向.
关键词: 网络;并发;程序设计复杂性;事件驱动;stack-ripping;多线程;期货
中图法分类号: TP393　　**文献标识码**: A

---

* 基金项目: 国家自然科学基金(60703072); 国家重点基础研究发展计划(973)(2005CB321801); 湖南省自然科学基金(08JJ3125); 全国高等学校优秀博士学位论文作者专向基金(200953)
收稿时间: 2009-12-10; 修改时间: 2010-04-27; 定稿时间: 2010-06-09
CNKI 网络优先出版: 2010-11-05 11:50, http://www.cnki.net/kcms/detail/11-2560.TP.20101105.1150.000.html

© 中国科学院软件研究所 http://www.jos.org.cn

随着计算技术与网络技术的飞速发展和广泛应用,互联网已成为现代社会的重要信息基础设施[1−3].许多传统软件系统已经具备一定程度的网络功能,越来越多的软件系统在设计之初就开始考虑网络化、分布化的需求.在网络环境下诞生的各种新型应用,如 Web 2.0,SNS,E-Mail,即时通信,BitTorrent 等,更是完全的网络化软件系统.网络已经成为现代软件开发中不可或缺的关注领域,甚至有学者指出,软件不是工作在系统中的,“软件在网络环境下工作”[4].

然而,传统程序设计方法在开发高质量网络程序时遇到了许多困难,其原因在于,网络环境与单机环境之间的差异巨大.概括来说,这些差异主要包括网络环境的分布性、异构性、错误倾向性、异步性、并发性、访问延迟等等.这些特性所带来的问题易于分别处理,但却难以整体解决[5].并发性是造成这种现象的关键因素,因为主流的并发处理方法对软件模块的可组合性具有显著影响[6],使得针对这些问题的独立解决方案难以组合在一起.根据以上分析我们认为,并发问题是网络程序设计中的一个瓶颈问题,具有十分重要的研究意义.并发对网络程序复杂性的影响可以类比多核处理器带来的“软件并发危机”,事实上,这是并发问题在网络中的翻版.多核化与网络化都是软件执行环境的发展趋势,并发问题在这两个趋势中都有着关键的影响作用.然而,研究领域对网络并发问题的关注却远远低于应有的程度.

并发管理在网络程序设计中占据着非常重要的地位,其方法的选择直接关系到整个系统的各个层次.一些分布式系统的开发经历[7]表明,一旦系统采用了不合适的并发模型,开发难度会急剧增大,甚至几乎无法完成.并发问题是公认的困难问题,目前也不存在解决这个问题的普适方法,甚至在不同方法之间存在着明显的争论[8−11].本文简要介绍基本的并发模型以及各种常见的并发实现方法,并在此基础上着重分析现有方法的内在复杂性,对比各种方法的优势与劣势,最后展望可能的研究和发展方向.

1 网络程序设计中的并发问题概述

并发是网络程序的一项基本需求.例如:Web 服务器需要同时向许多客户提供服务;网页浏览器也具有同时打开多个网站的能力;P2P 系统(如 BitTorrent)中的每个节点更是需要同时与许多其他节点协同完成任务.没有并发管理,这些功能都无法实现.除此之外,网络程序的性能往往也对并发提出了要求.现代互联网中越来越多地部署了高带宽、高延迟的物理链路,跨地域的数据中心及其承载的云计算应用往往也由高带宽、高延迟的线路连接,这种高延迟特性需要应用程序更好地开发利用并发机会,以充分隐藏传输延迟,提高应用性能.例如文献[12]透露,Amazon 对其页面生成的时间要求是在峰值负载时 99.9%的请求小于 300ms,而生成一个页面平均需要通过内部网络调用 150 多次底层服务.若这些服务不能并发执行,那么平均每个服务执行时间必须小于2ms.这几乎是一个不可能达到的要求,因为硬盘的平均寻道时间就超出了这个数值,数据中心内的网络延迟也可达到这个量级.因此我们推断,Amazon 一定采用了并发执行技术以优化单个页面的生成延迟.

并发(concurrent)与并行(parallel)是一对孪生兄弟,两者十分相似,但却是不同的两个概念,有明确的区别.加州大学伯克利分校的一篇技术报告[13]给出了并发与并行的概念定义,我们将其引用过来以清晰界定本文的讨论范畴:

定义 1. “并发”是在逻辑层面上的同时工作.

定义 2. “并行”是在物理层面上的同时工作.

根据“并发”的定义,在单处理器上运行的多线程程序、在单线程内通过事件驱动来同时处理多条连接的网络通信程序等都是并发的,而不是并行的.根据“并行”的定义,SISD 形式的超标量计算技术、SIMD 形式的向量计算技术、硬件线程级前瞻技术等都是并行计算技术,而不是并发计算技术.由于大多数并行程序同时也是并发程序,这两个概念的外延存在很大的交集,导致大量的混用和混淆.本文关注分布式程序设计中的并发问题,并不涉及仅与并行计算相关的内容,如多核、事物内存等等.本文也不涉及一些特定环境的专用并发方法,如 MapReduce 等.

并发程序设计的基本模型包括事件驱动模型、多线程模型以及从多线程模型中派生出来的期货(future)[14]模型.无论哪一种模型,其任务调度方法都既可以是协作式的(cooperative),也可以是抢占式的(preemptive).这两

© 中国科学院软件研究所 http://www.jos.org.cn

种任务调度方法的根本区别在于任务调度的时机.协作式任务调度的时机由任务自身确定,当且仅当当前任务主动放弃执行权,调度器才会将执行权交给其他任务;而抢占式任务调度的时机由底层软件(通常是操作系统)确定,一个任务可能在任何时候被唤醒或挂起,因此,不同任务也可能并行执行.表 1 总结了上述 3 种并发模型和两类任务调度方法的各种组合方式,其中,协作式事件驱动模型和抢占式多线程模型是最常用的两种并发任务处理技术(灰色背景).

Table 1 Approaches to concurrent task processing

表 1 并发任务处理方法

Model	Property				
	Scheduler	Cooperative task scheduling		Preemptive task scheduling	
		Name	Typical implementation	Name	Typical implementation
Event	Event dispatcher	Event-Driven	*select*(), *poll*()	Parallel event processing	Libasync-SMP[15]
Thread (including future)	Thread scheduler	Coroutine	Win32 fiber, GNU pth, Capriccio[10]	Multi-Threaded (kernel threads)	Main-Stream OS kernels

* Items with gray backgrounds are the most common combinations.

关于并发管理的研究已经持续了几十年,至今也没有一种在各方面都令人满意的方法.研发人员往往拥护事件或线程中的一种,并且反对另外一种.在这两派人员之间出现了一场数十年之久的论战.由于当时网络技术远不像今天这样发达,争论的焦点在于操作系统的体系结构上.具体来说,系统结构可以分为面向消息的和面向过程的这两大类,它们分别对应于事件驱动和多线程这两种模型.争论的内容是哪种系统结构更具有优势.Lauer 和 Needham 于 1979 年发表文章[16],论证了这两种结构模型是对偶的,它们能力等价,可以相互转换,也可以达到同样高的效率.他们试图以此结束这场争论,然而研究人员仍旧认为自己的偏好在其他方面更具有优势.

时至今日,争论战场转移到了网络领域.文献[8,9]是两篇针锋相对的论文,前者认为多线程会带来同步问题、死锁问题、组合性问题、调试性问题等等,因而在多数情况下是不好的选择,尤其是对于 GUI、分布式系统和低端服务器.文献[8]同时也认为,线程是不可或缺的,应该在性能敏感的场合使用.然而后者[9]认为,线程在控制流的灵活性等问题上确实具有一定劣势,但这些劣势在实际中很少被用到.关于线程的诸多问题,其实主要源于抢占式任务调度技术而不是线程模型本身,例如,协作式线程在这些方面具有与事件驱动类似的特性.文献[9]进而以线程库 capriccio 证实了这种模型在高并发服务器上甚至可以取得比事件驱动更好的性能.

孰优孰劣仍然没有定论,许多研究人员提出了事件与线程的混合模型[7,17−20].他们认为,有些场合明显适合使用事件,同样,也有些场合应该使用线程来表达控制流程,因而应该允许程序员混合使用这两种模型.并且,这两种模型下的模块应该能够相互调用,协同工作.此外,包装遗留代码也有可能需要混合这两模型.

2 事件驱动并发模型

事件驱动模型在网络程序中有着广泛的应用.根据事件的语义,可以将这种模型划分为两大类:反应式(reactive)与前摄式(proactive).反应式事件的语义是“当可以做某事”,如“当可以发送数据”、“当可以接收数据”等等.在这种模型中,应用程序需要先等待事件通知,然后发出具体的操作指令.操作结果的错误情况立刻可以从函数的返回值中得到.在前摄式事件驱动模型中,应用程序总是先发出操作指令,然后等待相应的完成事件,因而这些事件的语义就是“当某事完成”,如“当发送完成”、“当接收完成”.在这种模型中,操作的错误情况一般作为相应完成事件的参数传递给应用程序.若需发出多个相似的并发操作,则应用程序需要在发出操作时增加一个标识参数,如当前会话的上下文状态.这个标识参数将在相应的完成事件中作为参数传递给事件处理函数,以便于区分这些相似而不同的并发操作.

反应式模型也叫做非阻塞(non-blocking)模型,其典型代表包括 *select*(),*poll*(),*epoll*()等.前摄式模型也称为异步(asynchronous)模型,其典型代表包括 aio,iocp 等.这两种事件模型十分相似,可以相互转换,因而它们具有许多共同的问题.本节将在分析这些问题的基础上,研究相应方法的复杂性.

© 中国科学院软件研究所 http://www.jos.org.cn

2.1 事件驱动模型中的内在问题

事件驱动模型的主要问题可以概括为这样几类:控制流反转、函数分裂、变量重定位、调用栈重构和侵入性原则,其因果关系可以由图 1 描述出来.这些问题是事件驱动模型的固有问题,它们既相互联系又相对独立,每一个都显著增加了网络程序的开发难度.由于这些问题都有一个共同的根源,文献[7]将这些问题统称为 stack-ripping 问题.

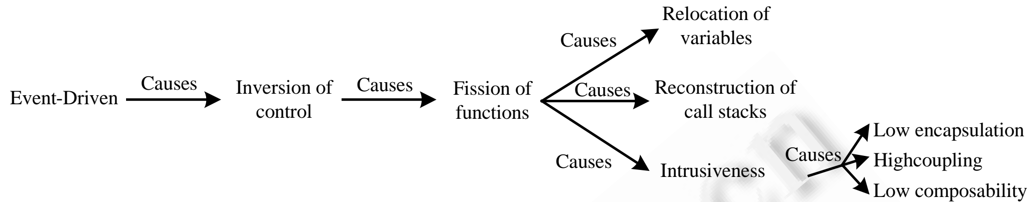
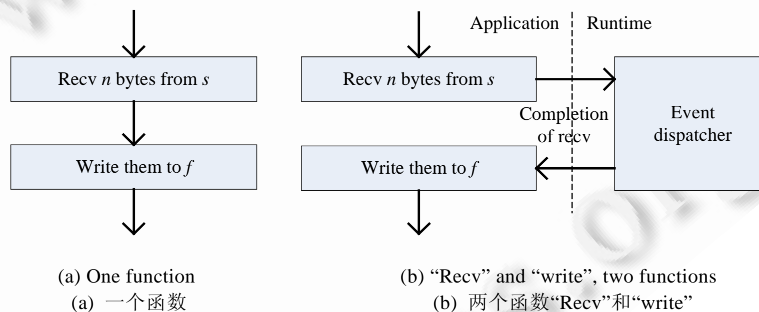


Fig.1 Problems derived from the event-driven model

图 1 事件驱动模型的主要问题推演

在事件驱动模型中,事件及其处理函数的关联匹配关系由事件分派器来维护,这必然要求应用程序在处理完每个事件之后,都必须将 CPU 控制权还给底层的事件分派器,以使其他事件能够继续投递到应用层,因而形成了控制流反转(inversion of control,简称 IoC).与正常的控制流相比,这种反转的控制流与开发人员的直觉差别较大,相对难以理解和掌握.

与控制流翻转相伴的往往是函数分裂.如果一个任务(或子任务)中包含了需要事件来驱动的部分,那么这个任务就需要写成多个函数.即便这个任务本身非常简单,最终形成的代码也可能非常繁冗.例如,假设一个任务是“从 TCP 连接 s 中接收 n 个字节长的数据,并将其写入到文件 f 中”.这个任务如果不用事件驱动模型来写,可能只需要两行代码,如图 2(a)所示;而如果用事件驱动模型来编写,则必须写成至少两个函数,因为当数据接收完毕之后,底层必须通过一个函数入口来回调本任务,以继续其余的工作,因而“写入到文件 f 中”这一步骤需要单独形成一个函数,如图 2(b)所示.



(a) One function
(a) 一个函数

(b) “Recv” and “write”, two functions
(b) 两个函数“Recv”和“write”

Fig.2 Fission of function

图 2 函数分裂

如果我们引入了错误处理,图 2 中的两种程序就都会变得更复杂,然而图 2(b)中的程序复杂度增长得会更快,因为我们不得不引入一个新的函数,在“写入到 f”完成之后来接收事件分派器的回调,以便检查错误状态.如果任务的流程更加复杂,这种函数分裂的风格会使得程序变得散乱而缺少调理、复杂而容易出错.

归纳来说,一个无分支、无循环的顺序过程若有 n 次事件驱动的网络 I/O 操作,那么这个过程就会被这 n 个 I/O 操作分裂为 n+1 个函数.若此过程中存在循环,且有 I/O 操作出现在循环体当中,那么这个循环就必须退化为判断、分支结构,而不能用 for,while 等现成循环结构来表达.

函数分裂会迫使原本在栈中的函数局部变量转移到堆中,造成变量重定位.在事件驱动模型中,许多原本可

© 中国科学院软件研究所 http://www.jos.org.cn

以存在于栈中的局部变量不得不迁移到堆中,以使其生命周期能够跨越多个分裂形函数.例如图 2 中的任务状态 n 和 f,在图 2(a)所示模型中可以声明为函数局部变量,而在图 2(b)所示模型中必须动态分配在堆中,以使它们在“写入到 f”这一步骤中仍然可用.变量重定位问题导致堆内存分配显著增长,使得程序的复杂度和运行时开销都会相应增长,并且还会在一些语言中(如 C/C++)增加内存泄露的隐患.

在调试事件驱动程序的时候,开发人员往往需要手工重新构造函数调用堆栈,造成调用栈重构问题.假设图 2 中的任务在某应用程序中具有一定的通用性,则将其封装为一个模块,可供多个上层任务调用.在调试时,对于图 2(a)所示模型的程序,我们可以通过回溯堆栈的方法了解其调用路径以及路径上的各个函数的局部变量状态;然而对于图 2(b)所示模型的程序,回溯堆栈的方法只能得到事件分派器的状态.此外,图 2(b)所示模型的程序在调试的时候也难以单步跟踪,因为调试器只理解函数调用、返回,对消息发送的语义并不知晓.为了便于调试,采用图 2(b)所示模型的程序往往需要手工记录调用路径,从而增加了无谓的复杂度和运行时开销.

函数分裂具有侵入性原则,因为一个分裂函数会强制分裂它的直接和间接调用者.具体来说,如果一个函数是基于事件驱动模型的,那么调用它的函数一般来说也必须是基于这种模型的.如果一个最底层的功能,比如发送、接收数据,是事件驱动的,通过反复应用侵入性原则,则所有直接或间接使用这个功能的函数、模块都是事件驱动的.

事件驱动模型原本应属于实现细节,然而其侵入性却导致这个细节必须暴露在模块接口之中,造成了模块的低封装性.此外,不同系统的事件机制差别巨大,模块的接口将事件机制暴露出来,导致模块之间通过事件机制形成了高耦合性.大量现有模块可能因为不支持事件模型,或者支持的机制不同而导致难以组合在一起使用,甚至在极端情况下会出现排他性.例如 MFC 是 Win32 平台上 C++程序常用的 GUI 库,ACE 是跨平台的高性能网络通信库,它们的主体模型都是事件驱动的,然而根据我们的经历,在一个应用程序中同时使用这两个库却不是一件容易的事情.

对于任何有解的困难问题,我们可以用抽象、封装、隐藏等手段重用现有解决方案.同时,各种解决方案在不断使用过程中也得到逐步的完善,形成一个积累的过程.然而,事件驱动模型的侵入性原则导致这些手段不再有效,现有方案难以重用,导致开发人员往往需要从头开始“做轮子”.

数据竞争往往被认为是只有多线程模型才具有的问题,如文献[8]认为,事件驱动模型中没有数据竞争,也不需要互斥量等同步机制.然而,根据两者的对偶关系[16],实际上,事件驱动模型中也存在数据竞争问题,这是所有并发程序都面临的公共问题.通常情况下,事件驱动模型中的任务切换时机十分明确,即事件处理函数返回后调度下一个事件,所以数据竞争容易避免.然而,当出现嵌套事件处理时,数据竞争就容易产生.例如,在处理某事件 A 的过程中,可能引发并立刻处理另外一个事件 B,若这两个处理过程共享某些数据,在嵌套处理条件下就可能会产生竞争错误.

在非嵌套处理条件下,数据竞争也有可能出现.例如,某两个并发任务都由若干步骤组成,同一个任务的各个步骤之间由相应的完成事件来关联,即前一个步骤的完成事件引发下一个步骤开始执行.若这两个任务共享变量 x,且其中一个任务在其自身状态中暂存数据 $y=f(x)$,而另外一个任务更改了 x,就有可能造成数据竞争错误.

2.2 解决问题的已有方法

控制流反转与否是一个二值问题,没有中间状态;事件驱动模型必然导致控制流反转.针对控制反转的问题,libasync[21],boost.asio,ACE 等工作通过封装各种常见而琐碎的事务显著提高了易用性,同时也实现了高性能、跨平台的目标.Spring 是一套基于 Java 的企业级应用框架,它利用控制反转的原理实现依赖注入(DI)和面向方面的程序设计(AOP)等技术,使得模块之间的耦合度显著降低,整个应用的灵活性大为提高.然而,控制反转及其所带来的思维方式的改变仍然存在,这是事件驱动模型的一个固有问题.

函数分裂问题可以较好地从语言上解决.例如,JavaScript,Python,Ruby,Lisp,Lua 等语言支持就地函数(in-place function)[22],可以在分裂处就地定义一个新函数,显著降低了程序的复杂度,同时提高了连贯性.对于一些简单的就地函数,如 Python,Ruby,Lisp 还支持λ演算[22],可以用表达式的形式定义就地函数,进一步降低了复杂度.使用这些语言设施虽然仍会将原本的一个函数分裂为多个,但分裂函数在形式上是连贯的,因而比原有方法

© 中国科学院软件研究所 http://www.jos.org.cn

具有更好的可维护性.

然而,许多主流语言,如 C/C++,Java,C#等,仍然对上述语言特性支持甚少.文献[17]在 libasync 的基础上提出了一种 C++语言扩展技术 Tame,使得应用程序在开发时可以使用线程风格的代码,但在编译前通过一个源程序到源程序的预处理自动进行函数分裂工作.Li 在文献[18]中基于 Haskell 语言设计了一套伪线程机制,借助该语言对 monad[23]语法糖的支持,使得线程代码在可以编译时转换成为事件驱动代码.这项工作表明,函数分裂问题几乎可以从语言层面上解决.然而事实上,无论是 Tame 的自动函数分裂还是文献[18]中的 monad 语法糖,在它们作用下的程序代码从形式上来看已经属于多线程模型了.这些复杂技术所带来的好处不一定比直接使用真线程要多.

变量重定位问题可以通过词法定界的闭包(lexically-scoped closure)[22]来解决.在就地函数、λ演算或者自动函数分裂的基础上,支持闭包的编译器(或解释器)可以自动地确定后续函数可访问的局部变量集合,将其"打包",并作为环境传入新建立的函数中.这种方法可以较为系统地解决变量重定位的词法复杂性问题,然而闭包创建、处理、访问等操作在性能方面仍会有一定的影响.

变量的重定位问题给 C/C++等必须手工管理内存的语言带来了复杂性,因为内存的分配与回收不在同一个函数内,甚至回收点也并不确定.Tame 采用了引用计数的闭包,boost.asio 也推荐用户使用基于引用计数的智能指针 boost.shared_ptr.引用计数最大的问题是循环引用问题,其次还有运行时开销较高的问题.另一方面,APR 等 C 库则广泛使用内存池技术,以便在确定的环节(如子任务完成时)统一回收相应池内所有对象.这种技术在保证正确性和高性能的同时,降低了内存的使用效率.

调用栈重构问题可以通过一些辅助工具来解决.文献[24]设计了 eel 工具集,其特点在于引入了 group 的概念来表示相互关联的一组操作.该工具集提供的调试工具正是利用了 group 的概念来实现相关操作的单步跟踪.该文献并没有明确表明 eel 能否利用 group 标识符在运行重建调用栈结构,但这个功能应该不难实现.Eel 工具集的另外一个特点是提供了程序流程的可视化工具和模型检验工具,能够显著简化程序调试工作.Mace[5]是一个 C++语言的扩展包和源代码到源代码的编译器,它对基于事件驱动的网络程序开发提供了强有力的支持.在调试时,Mace 能够跟踪事件之间的因果关系,实际上重构了整个调用堆栈.Mace 甚至能够跨节点记录这些关系信息,在这方面比 eel 更具优势.

侵入性原则必须通过模型转换来阻断.文献[16]证明了事件驱动与多线程是对偶模型,两者能力相当.因此,一定存在一种变换机制可以将两者桥接在一起.Tame[17],文献[18]和 ProtoThreads[19]各自提出了编译时的模型变换方法,将基于多线程模型的代码变换为事件驱动模型.ProtoThreads 通过 C 语言的宏机制使用标准预处理器实现编译时的代码转换功能,其特点是小巧且完全跨平台,但其功能十分有限.Tame 是一种 C++扩展语言,它实现了一个的源代码到源代码编译器,将基于 Tame 的多线程代码先转换为基于标准 C++的事件驱动代码,然后再用标准 C++工具链进行编译、链接.文献[18]针对 Haskell 语言设计了一套伪线程机制,借助该语言对 monad[23]语法糖的支持,无需额外的工具便实现了编译时的代码转换.文献[18]相对于 Tame 的另外一个优势是实现了异常的支持.Tame 简单地禁止用户使用异常;ProtoThreads 针对 C 程序,本身就不支持异常.编译时的模型变换基本原理大致相同,如图 3 所示,以 Tame 为例,变换工作主要包括如下两部分:

(1) 将局部变量提出成为全局的状态结构体,并在堆中动态分配出来;

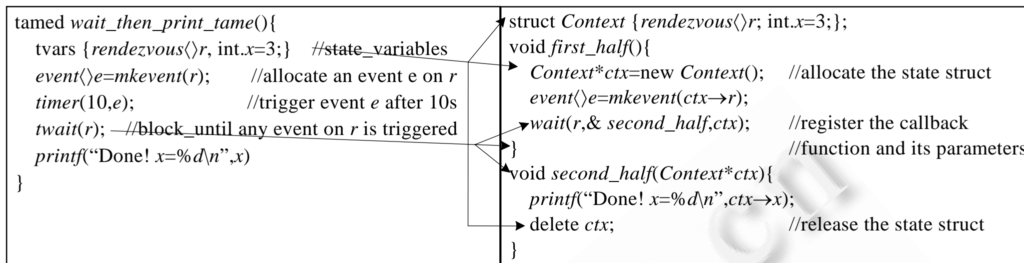
(2) 在等待点将函数"撕裂"成为上下两部分,并将状态结构体作为上下文传递给下部分.

由于事件驱动模型具有更高的灵活性,将其变换为多线程模型虽然理论上可行,但具有更高的难度,并且最终系统在性能上可能还具有劣势,因而鲜有此类研究工作.本文第 4.2 节将会讨论到,事件驱动模型是一种层次更低、更接近于硬件的并发模型,而多线程模型则层次相对较高.将事件驱动的代码转换成为多线程代码在难度上可以(定性地)类比于反编译目标代码到高级语言源程序.在性能方面,由于底层硬件的工作方式是事件驱动方式,因而源程序的执行实际上经历了事件→线程→事件的变换.这种"绕弯"的做法一定会带来某种程度的额外开销,使得程序性能低于"直接"的事件驱动模型.

除模型变换法之外,文献[7]还提出了一种基于运行时库的双向桥接方法.这种桥接机制能够较为完善地阻

© 中国科学院软件研究所 http://www.jos.org.cn

断事件驱动模型的侵入性,从而避免了进一步的各种问题.由于用户程序无需变换,异常可以很好地在线程代码内传播.然而文献[7]的方法并没有处理异常,因此异常无法跨越线程边界.在引入模型变换机制的同时,程序设计系统实际上也就是引入了混合并发模型.上述这些工作都允许开发人员同时使用时间和线程这两种并发实现方法.混合并发模型将在第 4.1 节中加以详细阐述.



(a) A Tame-baseded program in thread-style (b) Translated program in event-style (pseudocode)
(a) 基于 Tame 的线程风格代码 (b) 翻译后的事件风格代码(伪代码)

Fig.3 Model translation: Threads→Events
图 3 模型变换:线程→事件

数据竞争问题需要通过锁等同步机制来解决,然而现有事件驱动引擎很少提供这种机制,开发人员很少能意识到这其实是并发任务之间的同步问题,而往往通过一些 ad hoc 的方法来解决.例如:NesC[25]是一种传感器网络开发语言,它针对其特定应用环境实现了一种编译时自动竞争检测的机制;Libasync-SMP[15]允许给事件设置颜色属性集,其调度器保证有共同颜色的事件不会同时执行.

2.3 针对性优化

由于事件驱动模型会将处理过程分裂为小函数,并且这些函数是通过指针间接调用的,这些间接调用将一个完整的流程割裂开,导致许多优化技术——尤其是静态优化技术——难以应用.针对这个问题,文献[26]采用了基于 profile 信息的方法进行了优化,使得程序性能得到显著提高.这项工作不仅使用于网络程序,也对事件驱动的 GUI 程序具有优化效果.Ensemble[27]也针对事件投递的轨迹提出了内联等优化措施.这两者的主要区别在于, Ensemble 需要开发人员手工标注概率信息,而前者则通过 profile 的方法获取.此外,前者在设计上具有更好的通用性.

由于事件驱动模型存在变量重定位的问题,使得使用这种模型的程序大量使用动态分配的小内存片段,从而造成处理器数据 cache 的效率降低.文献[28]针对这个问题提出一个全新的内存管理器和一个精心裁剪的任务调度器,使得程序的工作数据集可以直接映射到数据 cache 上,大幅降低了数据 cache 的失效率.

Libasync-SMP[15]引入了并行事件处理的技术,它允许程序员给每个事件赋予一个颜色,只有不同颜色的事件才会并行执行.事件在默认情况下具有相同的颜色,因此,程序员可以逐渐地增加事件处理的并行性. Libasync-SMP 的一个突出特点是后向兼容性好:现有基于 libasync 的代码在导入 libasync-SMP 后仍然是合法的程序(非并行);程序员可以根据事件的热度有选择地将代码并行化,这样可以投入少量并获得显著性能的提高.根据事件——线程的对偶论,事件的颜色实际上就相当于多线程中的互斥锁;事件分派时的颜色检查就相当于线程调度时保证锁的互斥关系.

3 多线程并发模型

根据调度时机,线程可以分为抢占式(preemptive)和非抢占式(non-preemptive)两大类.抢占式线程机制多为操作系统内核提供,由内核调度和管理,其调度时机由调度器来确定,不受具体任务的直接控制(需要通过各种同步原语来影响).非抢占式线程也称为协作式(collaborative)线程或者协程(coroutine),多为应用层软件库实现,

© 中国科学院软件研究所 http://www.jos.org.cn

一般不被内核知晓.常见的协作式线程实现包括 Windows NT 中的 fiber,GNU pth,ucontext,capriccio[10]等等.很多程序设计语言也实现了内置的协作式多线程机制,如 Erlang,CML,Oz,Lua,Go 等.协作式线程的调度时机由当前任务自行确定,不受调度器的控制.

由于调度时机的差异,这两种线程在实际应用中具有显著的区别.抢占式线程的调度时机由调度器决定,因此,CPU 控制权可能在一个“不应该”的时候被强行切换到另外一个任务,导致数据竞争.协作式线程则当且仅当任务主动放弃执行的时候才会将 CPU 控制权切换到另外一个任务,因而能够有效减少数据竞争问题以及它带来的锁问题.也同样是由于这个差异,只有抢占式线程才能充分利用处理器的线程级并行能力.

许多现代程序设计语言都对线程有直接的支持.例如,Java,C#都定义了并发内存模型,提供了创建、操作抢占式线程的 API,同时都在语法上提供了便利的同步机制.Erlang,Lua,Oz,Go 等语言内置了对协作式线程的支持(Erlang 称其为进程,因为 Erlang 不允许并发任务间共享数据),并且优化了调用堆栈,使得空间开销显著降低.Python,Ruby 等语言从不同程度上支持协作式线程,如迭代器(iterator)或者产生器(generator)等.然而,极少有语言同时支持这两种线程.我们认为其原因在于,同时支持两种线程会导致语言复杂性大增,而且这两种线程各自的特性与垃圾收集机制难以同时存在于一种语言之中.作为对比,C/C++在语言层面不直接支持任何线程机制,却可以通过各种扩展库来使用它们,甚至可以同时使用.然而,C++0*x* 将纳入对抢占式线程和可选垃圾收集的支持,这可能会影响协作式线程在其中的应用.

3.1 多线程模型中的内在问题

关于抢占式线程的研究进展集中于事务内存、自动并行化或者自动同步等方向,然而,由于分布式应用很多时候并不需要利用计算资源的线程级并行性,或者计算密集型任务可以被很好地与网络通信任务隔离开,协作式线程在文献[7]中被认为是“甜区”.因为它既避免了大多数的数据竞争、死锁等问题,也没有事件驱动模型的 stack-ripping 问题.此外,文献文献[9,10]还证明了这种方式可以达到很高的 I/O 性能.多线程模型的主要问题包括灵活性较弱、时空开销较高、普适性不强等.由于线程间的数据竞争、死锁等问题在并行计算领域已有许多研究,本节不再对此作过多分析.

许多研究人员认为多线程模型灵活性较弱,某些精巧的控制流程难以直观地表达出来.如:文献[5]认为,多线程模型不适合表达类似于 TCP 的状态机的控制流;文献[9]认为,多线程模型不适合表达“广播”、“订阅/发布”类型的控制流.然而文献[9]同时也认为,这些精巧的控制流程在实际应用中很少遇到,因此,多线程的表达力多数情况下已经足够.

线程调度器往往位于操作系统内核,应用程序的任何线程操作都需要通过系统调用,因而在高并发或线程间协作复杂的任务中时间开销较高.此外,由于普适性要求,其调度算法往往较为复杂,也影响了性能.空间开销通常是制约并发规模的主要因素,它包括两个方面:物理内存空间开销和虚拟地址空间开销.由于任务状态变量位于函数堆栈,往往在函数入口统一分配,在函数出口统一释放.这样,在一层一层的堆栈当中,会有相当一部分空间由已经不再使用的状态空间占用,造成物理存储空间的浪费.此外,线程在创建的时候,系统无法预知应用会需要多少堆栈空间,往往会分配一块足够大的连续内存区域,这是造成多线程模型的空间开销较高的主要原因.预分配的连续堆栈可以造成虚拟地址空间的浪费.例如,Win32 系统默认给每个线程(无论抢占式还是协作式)分配 1MB 的堆栈空间,而用户态程序最多可以使用约 2GB 的虚拟地址空间,这种情况下应用的并发上限大约只有 2 000.预分配的连续堆栈也可以造成实际物理内存的浪费,因为程序对堆栈的使用量随时间的推进会有涨有落,上涨时操作系统会及时分配物理页面供应用使用,而跌落时操作系统并不能随之回收不再使用的物理页面,从而造成物理存储空间的浪费.

多线程模型在各种系统上的支持程度和方法具有较大差异,线程代码在它们之间移植比较困难,导致多线程模型的普适性不足.例如,Windows 系统同时支持抢占式和协作式两种线程,而标准 Linux 只支持抢占式线程,甚至在 NPTL(Native POSIX Thread Library)出现之前是用进程来模拟线程的.在一些资源受限的系统中,如面向传感器网络中的 TinyOS,线程往往是不被支持的.此外,这些系统的线程接口函数(如果有的话)从语法到语义上也具有显著区别.

© 中国科学院软件研究所 http://www.jos.org.cn

关于协作式线程存在一个常见的误区,即认为这种开发方法不会引起数据竞争问题,也不需要互斥量等同步机制[9].虽然只有当前任务主动释放CPU控制权,其他任务才能继执行,协作式多线程仍然存在数据竞争的危险.当一段程序中存在函数调用,被调用的函数可能会引起任务切换,可能导致共享状态在调用前后不一致时,就需要互斥量来保证正确性.除此之外,各种同步原语还可以用来完成任务间的协同工作.例如,当应用逻辑本身要求任务 A 在任务 B 和任务 C 完成之后才能进行时,就需要一种适当的同步原语来协调各个任务之间的运行.这个误区不仅仅存在于协作式多线程,事实上,这是一个关于协作式任务管理的误区.它的另外一个典型表现就是认为事件驱动模型中没有数据竞争,也不需要互斥量等同步机制[8].如前例所述,应用逻辑本身的要求与实现模型无关,无论采用多线程还是事件驱动模型,任务间的同步都是必须的,区别只是在于实现方式有所不同.

3.2 解决问题的已有方法

灵活性问题是典型的抽象惩罚问题.线程模型具有较高的抽象程度,在简化一些情况(常见情况)的同时,必然会导致另外一些情况(非常见情况)的不适.事实上,文献[9]研究了 Flash Web Server 以及一些运行在 Ninja, SEDA和TinyOS之上的应用程序源代码,发现其中所有的控制流程都适合用多线程模型来表达.此外,也有许多研究认为,多线程和事件驱动又各自有不同的适用场合,应该支持它们混合使用.这种混合模型将在第 4.1 节详细加以讨论.

时间开销较大的问题主要是由于操作系统内核调度引起.文献[7,10]都实现了用户态的线程调度,可以显著降低状态切换的开销.同时,它们也优化(简化)了调度算法,使得高并发条件下,线程调度性能显著有所提高.实际上,文献[9]证明了用户态的协作式线程即使在高并发条件下也能够实现很高的性能.然而我们认为,线程模型的抽象层一定会引入比事件模型更高的开销,这一点将在第4.2节中详细加以讨论.

空间开销较大的问题源于函数对栈的使用.许多语言,如 Java,C#等,已经不将数组等大对象分配在栈空间,缓解了这个问题.此外,文献[9]提出了一种基于编译器的自动堆栈增长方法,能够自动适应线程执行时对堆栈空间需求的变化.然而文献[16]指出,这种自动堆栈增长方法在实际中并不稳定.Shared-Stack[29]针对资源十分受限的传感器网络环境,实现了一种以时间换取空间的共享堆栈方法.在 Google 最新推出的程序设计语言 Go 中,函数堆栈已经不再是连续空间,而是一种链表结构,较为彻底地解决了堆栈的空间效率问题.

普适性不足的问题是一个标准化问题,甚至可能是一种商业策略问题.针对这个问题的解决方法只有封装.如 pthread,boost.thread,Capriccio,GNU pth 等都是跨平台的线程库,既包括内核态调度的抢占式线程,也包括用户态调度的协作式线程.

数据竞争问题的常规解决办法是在各种同步原语的保护下访问共享数据,此外还可以使用事务内存,或者富有技巧性的 lock-free 数据结构.事务内存是近些年的研究热点,它允许多个线程以事务的方式并发访问共享数据,而不必担心破坏数据的一致性.然而,这些工作都是针对抢占式线程(并行计算)展开研究的,而协作式线程具有自身不同的特点:较为固定的调度时机,因而直接使用现有方法并不能取得最好的效果.

此外,这些方法实际上不仅仅是避免数据竞争的措施,它们也是线程间相互协作的原语,即便任务间没有共享数据,不存在竞争的顾虑,线程间的协作也需要这些原语.

3.3 针对性优化

存在这样一类流式多线程应用:每个线程都执行“输入→计算→输出”的无限循环,线程间依靠输入和输出进行耦合,如图 4 所示.许多重要系统都是由这种结构构成的,例如,流处理器、网络协议栈、DSP 以及其他媒体处理系统、UNIX 管道、图形渲染流水线等.然而,这种模块划分与组合的方式却阻碍了传统的优化技术的应用,使得模块组合开销过大,相对于轻量级的计算模块,这种开销甚至无法接受.文献[30]针对这个问题提出了一种基于 CPS(continuation-passing style)的优化算法,使得最终结果具有类似于代码内联或者β-消解的效果.

© 中国科学院软件研究所 http://www.jos.org.cn

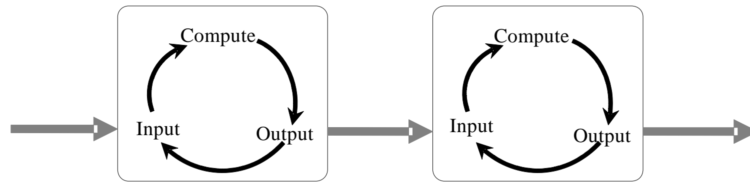


Fig.4 A multi-threaded streaming application[30]

图 4 流式多线程应用[30]

3.4 派生并发模型:期货

Baker 和 Hewitt 于 1977 年在文献[31]中提出了期货的概念,用于解决垃圾收集中的问题.Halstead 于 1985 年首次将期货引入到 Lisp 的方言 Scheme 当中[14],以解决分布式并行计算问题.他这样介绍期货:“(future *X*)会立刻返回一个关于表达式 *X* 的期货,并且开始并发地对 *X* 进行求值.当求值完成,所得结果会替代前面产生的期货.期货在产生之初处于未确定状态(undetermined),当它的值计算出来后就变成确定状态(determined).一个需要知道未确定期货实际值的操作(如加法)将被挂起,直到它变成确定状态.然而许多其他操作,如赋值、参数传递等并不需要知道操作数的实际值,这些操作将透明地处理未确定期货”.

使用期货的程序在很大程度上不需要进行并发管理,甚至不需要知道它们使用的究竟是不是期货,或者期货是否已经确定.并发管理的各种相关事务基本上集中于产生期货的程序,它必须确定任务是否应该并可以被并发执行;它必须遵照多线程或者事件驱动的既定方式来完成任务;并且它在完成任务之后使相应的期货变成确定状态.经过多年发展,期货在实际中的应用仍然较少,其主要原因在于,主流程序设计语言缺乏对期货的支持.目前,这种情况正在得到改观:Java 从 5.0 版本开始引入期货的概念,下一代 C++标准当中也已经纳入对线程和期货的支持.因此,期货的应用有望在未来几年呈现显著的增加.

期货模型的一个重要特性是:并发管理的任务集中于产生期货的代码,而使用期货的代码几乎不需要明确地管理并发,甚至可以不知晓所用的是不是期货.在当前基于多核的并行计算环境中,期货受到了越来越高的重视.然而,期货设施的产生源于分布式计算[14],它既是一种管理并行性的方法,也是一种管理并发性的方法.

期货在不同语言中实现的方法并不一样.在动态类型语言中,期货是一个无具体类型的占位符(placeholder),可以像普通变量一样使用;然而任何对它的“读”操作都将隐含地阻塞线程,以待它变成确定状态.而在静态类型语言中,期货是一个有类型的占位符,其类型往往并不是第 1 类的(first class).如在 Java 语言中,期货被定义为一个模板接口 *java.util.concurrent.Future*⟨*T*⟩,使用期货的代码必须明确地通过期货接口的 *get*()方法等待并获取期货的实际值.在这种情况下,期货实际上并不是一个语言设施,而是一种设计模式.受语言自身特性约束,这种“期货模式”在使用便利性方面明显不如语言直接支持的第 1 类期货,在性能优化余地方面也比较有限.事实上,性能优化正是期货模型主要的研究问题.

Leapfrogging[32]认为,在最小化人工参与的条件下实现高效的期货运行时系统是最主要的挑战性问题.例如,用户程序可能产生了许多细粒度的期货执行任务,如果简单地给每个期货任务创建一个线程,或者使用线程池,那么最终性能反而可能下降,因为线程的创建、调度等开销可能吞噬掉了并行计算带来的好处.为了将线程数量维持在合理水平,研究人员主要采用了两种技术途径:期货内联和被动执行.期货内联法的基本思路是,当条件不允许的时候,在当前线程对期货表达式求值;被动执行法的基本思路是,把每个期货求值任务表达为一个被动对象,放入工作队列,由工作线程在适当的时机取出执行.然而,这两大类方法又都有各自进一步的问题,如图 5 中推演所示.

期货内联法中最重要的工作是 load-based inlining[33],其基本思路是,当系统负载饱和时(如没有空闲的处理器)将期货求值内联,即转换成为普通求值过程.这种转换过程是不可撤消的,因此,某些处理器有可能被赋予了大任务,而其他处理器却处于空闲状态,从而导致负载不均衡问题的产生.惰性任务创建(lazy task creation,简称 LTC)[34]针对复杂不均衡问题,基于延续窃取技术(continuation stealing)设计了一种可撤消的期货内联方法.LTC

默认在当前线程内对期货求值(默认内联),若执行过程中任何时候系统出现空闲处理器,则该处理器将偷窃调用函数的延续,实现内联撤消.这种方法在取得高性能的同时,将调用堆栈的结构变为链表形式,以便于延续窃取.这样的非标准堆栈具有兼容性问题和复杂性问题.

被动执行法中最重要的工作是 WorkCrews[35],它将期货任务封装为被动的任务对象,由工作线程挑选任务对象并执行.这样,工作线程的数目就可以固定为系统处理器的数量,避免无谓的浪费.同时,创建期货的开销也足够低,基本不需要再结合内联技术.然而,这种方法最大的问题在于空等:如果一个任务包含阻塞操作,如等待另外一个期货,那么执行该任务的处理器在阻塞期间不能处理其他任务,造成资源将被浪费.Leapfrogging[32]是一种优化的被动执行方法,可以避免这种空等上的浪费.这种方法的最大优点在于不需要使用延续(非标准堆栈),因而具有实现简单且兼容性、移植性好的特点,然而其多个任务队列的维护开销使得性能比 LTC 较低.

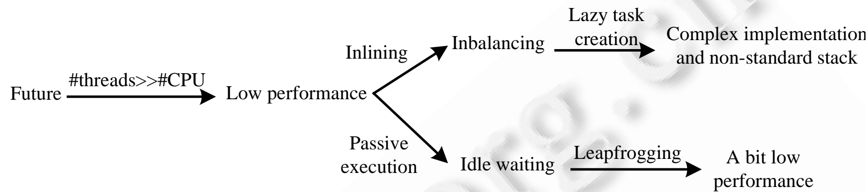


Fig.5 Performance problems in the future model

图 5 期货模型的性能问题推演

期货作为一种语言设施被提出以来,已有 20 多年的历史.然而无论是关于期货的技术研究还是应用推广,进展都比较缓慢.我们认为其原因有两点:

(1) 并发程序设计直到近年才成为研究和应用的热点;

(2) 期货的实现需要语言层面的支持,而主流语言直到近年才刚刚开始在标准库的层面引入期货概念,语言层面的支持还尚未开展.

Java 语言从 5.0 版本开始在其标准库中定义了期货接口,缺乏更全面的支持.ProActive[36]是一个基于 Java 的分布式应用中间件,它内部实现了一套自有的期货机制,以方便开发节点间通信的并发性.ProActive 的期货机制通过动态子类化和动态代码生成的技术使得期货可以透明地、自动地引入到应用当中,显著降低了程序复杂性.然而,ProActive 对期货支持仅仅局限于库的层面,仍然属于“二等公民”,对于不能子类化的对象不能良好处理,如标准字符串 String 的对象.

Zhang 在文献[37–40]中对基于 Java 的期货机制进行了深入的研究,提出了基于标注的惰性期货技术.这项技术允许应用开发人员先写出串行版本的应用代码,然后逐步在代码中添加期货标注“@future”就可以将程序并行化.除了简洁以外,这项技术的另一个突出特点是能够保持串行版本的异常语义,允许程序员在完成并行化工作时不必修改程序的异常处理结构.文献[41]针对 Java 语言提出了安全期货的概念,通过自动对象版本记录和任务重执行来自动地获得期货语义的透明性(无副作用).这项工作对于Zhang的各项工作起到补充作用.程序设计语言 Oz 将期货的应用推广到了极致:Oz 程序中任何逻辑变量在概念上其实都是期货,任何线程若读取未绑定的逻辑变量都将被阻塞,直到其他线程将此变量绑定.

细粒度高性能的期货程序需要程序设计语言以及运行时库的直接支持,否则,大部分优化技术都难以应用.例如,当应用程序创建一个线程的时候,编译器/运行时难以区分这个线程将进行期货求值还是做其他什么工作,也就难以应用针对期货的优化技术,如惰性线程创建.此外,若有语言层面的支持,期货在使用上也会更加简洁、方便.期货并发模型的另外一方面不足是,它只能将并发管理中的具体问题集中到产生期货的程序中去,而并不能减少或者解决任何这些问题.因而,期货的作用从某些方面讲是一种任务划分.

4 混合并发模型

事件和线程是网络程序设计的主流并发方法,其中,基于事件驱动的异步模型被更多的研发人员所偏好,也

© 中国科学院软件研究所 http://www.jos.org.cn

有许多研究人员认为两者的混合模型更具有优势.这一节中给出我们关于这两种模型的比较性观点,并初步探讨它们在不同场合的适用性.

4.1 事件+线程的混合模型

事件驱动模型与多线程模型具有不同的适用性,相互不能取代,因此,混合模型具有明显的优势:在适合事件驱动的场合使用事件驱动模型,在适合多线程的场合使用多线程模型.混合模型的实现方法大致可以分为 4 类,见表 2.

Table 2 Hierarchy and the occasions of various hybrid model implementations

表 2 混合模型的实现层次与时机

Hierarchy	Occasion	
	Compile time (static)	Runtime (dynamic)
Library	Ref.[16], ProtoThreads[19]	Ref.[5]
Language	Tame[17], Tasks[20]	Erlang

文献[18]利用 Haskell 语言的 monad[23]语法糖机制,完全以库的形式为 Haskell 提供了线程支持,并且实现了异常传递机制.Monad 语法糖实际上在编译时会将线程形式的代码转换成为事件驱动形式的代码.Protothreads[19]则是针对传感器网络环境,利用 C 语言的预处理宏机制进行代码模型转换.Protothreads 的特色是十分轻量级,其实现代码仅有 10 行,每条线程只需要 2 字节的额外内存空间.然而,Protothreads 的简单也可以理解为简陋:在任何阻塞等待之后,函数的所有局部变量将不再有效.其原因在于,阻塞等待经过宏替换之后,实际上已经导致函数返回;逻辑上等待后的继续执行只是一个假象:其实是另外一次函数调用,并且直接跳转到预定位置.所以,Protothreads 建议使用专门的状态对象,或者静态变量.

另外一类静态编译的混合模型是 Tame[17]和 Tasks[20],它们分别基于 C++语言和 Java 语言.它们都引入了新的关键字、语法形式和闭包设施,使得程序能够很直观地实现多线程的阻塞等待执行效果.它们也都提供了预编译器,将源代码翻译为标准的 C++或者 Java 语言,其中,Tame 翻译后的 C++代码需要使用 libasync.需要指出的是,Tame 并没有实现对异常机制的支持,而 Tasks 做到了这一点.

这种静态的混合模型方法有一个共同的问题,就是不能很好地与自有体系之外的程序在运行时相互调用.例如,基于 Tame 或者 Tasks 的线程形式代码都在静态条件下编译成为了事件驱动模型,因而其运行时环境缺乏线程的概念和设施,也就难以直接与现有的线程模型代码在运行时相互调用.基于运行时技术的混合模型从原理上将能够较好地解决这个问题.文献[7]提出了一种基于运行时库的双向转换方法,然而,它并没有考虑异常传播问题,且其表述过于复杂,我们将其算法简化复述于表 3.Erlang 是一种面向并发的语言,同时支持多线程和事件驱动模型.Erlang 线程之间不共享任何变量,因此,它们实际上被称为进程.进程之间的协作通过消息(事件)来完成.Erlang 消息分派机制的最大特色是在语言层面上支持了模式匹配,可自动根据消息的结构模式查找匹配的消息处理器.

Table 3 Simplify and retell the hybrid model proposed in Ref.[7]

表 3 简化复述文献[7]提出的混合模型方法

Thread calling event:	1. Send the event to the target module; listen to the feedback event; suspend the current (calling) thread. 2. When the feedback event is received, deliver the result and active the calling thread.
Event calling thread:	1. Spawn a new thread to execute the called function. 2. Wrap the returned value into a message, and send it to the calling module.

混合模型在整体上仍然是一种新兴的并发方法,目前还缺乏广泛的支持,其有效性也需要在各种实际应用中进一步加以检验.

© 中国科学院软件研究所 http://www.jos.org.cn

4.2 事件vs.线程

网络通信在根本上是一种 I/O 行为,而从计算机系统的硬件层面上看,各种 I/O 设备,尤其是高速 I/O 设备,基本上都以异步方式运行,并通过中断等方式通知中央处理器,这在本质上属于事件驱动模型.因而我们认为,同属于事件驱动模型的上层软件与底层硬件具有更相近的运行方式,在同等优化程度的条件下,也应该比多线程方式具有更好的硬件资源使用效率.这个结论可以在硬件资源十分受限的无线传感器网络领域得到印证:传感器节点中广泛使用的 tinyos[42]甚至不支持线程(本身不支持,但有许多线程扩展库).另外,这种更底层的并发模型在一定程度上具有更好的灵活性.然而有研究认为,这种灵活性在实践中很少用到[9].

线程并不能直接映射到硬件的 I/O 模型上,因而必须在某个时刻进行模型转换.这必然会带来一定的额外开销,即所谓“抽象惩罚”.所以,在同等优化程度的条件下,多线程模型的硬件资源使用效率低于事件驱动模型.然而,线程(包括协作式线程、抢占式线程以及进程)具有更高的抽象层次,一般情况下更接近人类思维,具有更高的开发效率.这是一个经验性结论,存在例外情况,如 GUI 相关的程序设计虽然可以用多线程方式,但往往更适合使用事件驱动模型.表 4 总结了这两种 I/O 模型在各个软件层次上的变换关系,从中可以看出,事件驱动模型与硬件完全对应,而其他几种模型都在不同时机进行了转换.这种转换发生在不同的软件层次,我们推断,越接近应用程序的变换工作,越可能产生更高质量的变换结果,因为更高层次具有更多的应用相关信息.除了性能以外,“更高质量”还可以包括更高的可靠性(由于类型安全)、更高的灵活性(由于语法糖)等方面.

Table 4 Translations of synchronous and asynchronous model in the software hierarchy

表 4 同步模型与异步模型在不同软件层次上的转换

Hierarchy	Model				
	Event-Driven	Multi-Threaded		Hybrid	
		Preemptive	Cooperative	Runtime	Compile time
Application	*A*	*S*	*S*	*S*, *A*	*S*, *A*
Compiler or interpreter	*A*	*S*	*S*	*S*, *A*	*S/A*, *A*
Library	*A*	*S*	*S/A*	*S/A*, *A*	*A*
OS kernel	*A*	*S/A*	*A*	*A*	*A*
I/O hardware	*A*	*A*	*A*	*A*	*A*

“*A*” denotes an asynchronous model; “*S*” denotes a synchronous model;
“*S/A*” denotes a translation from an asynchronous model to a synchronous model.

4.3 适用性分析

虽然事件驱动模型在同等优化程度的条件下可以具有更高的效率,但在实际使用当中,这可能并不成为一个主导优势,此时,我们更应该考察哪种模型会带来更高的开发效率以及更高的代码质量.在进一步讨论之前,我们需要引入等待点的概念,它是任务中可能发生等待的环节,在等待过程中,系统可能执行其他任务.例如,每个网络 I/O 操作都是一个等待点.等待点在多线程模型中对应于阻塞操作,在事件驱动模型中对应于事件处理函数的返回.

我们认为,如果一个任务包含的等待点越多,流程越长,则越可能适合使用线程模型;反之,则越可能适合使用事件模型.例如,图 6(a)展示了一个 HTTP 服务过程的框架,这段代码只能处理 GET,POST 和 PUT 请求.框架中包含 *recv_header*(),*recv_body*(),*read_file*(),*send_response*()共 4 个等待点,分别对应于图 6(b)中的 4 个“▬”符号.如果我们以事件驱动模型来实现这个服务框架,如图 6(c)所示,那么每个等待点以及分支合并处都会将框架切分成两个单独的函数,最终形成 6 个函数,分别对应于图 6(b)中的①~⑥部分.函数的增多不但会带来代码的冗余,而且会影响程序流程的表达,降低程序的可读性、可调试性、可维护性.等待点越多,任务流程越杂,这些问题就会越突出,也就越难以应用事件驱动模型.

例如,一个 HTTP 请求任务可以由这样几步完成:构造请求内容→发送请求→接收响应头部→根据头部信息接收响应内容.其中,每次发送和接收都需要循环重复,导致实际任务执行流程中可能包含许多等待点.这种情况下使用线程模型,就能够明显地简化程序设计.然而在有些情况下,用事件驱动模型可以写出更简洁的程序.例如,我们将一组域名并发解析成为 IP 地址,由于域名数量可能很多,于是,我们限定了最大并发量,以免被服

© 中国科学院软件研究所 http://www.jos.org.cn

务器误认为是攻击行为.由于域名解析只需要调用一个函数即可实现,流程简单,事件驱动模型此时不会受到太大影响;反而多线程模型需要处理线程间的协作(不超过最大并发量),此时并不具备优势.此外,硬件资源受限的环境或者性能敏感的应用则更需要事件驱动模型.

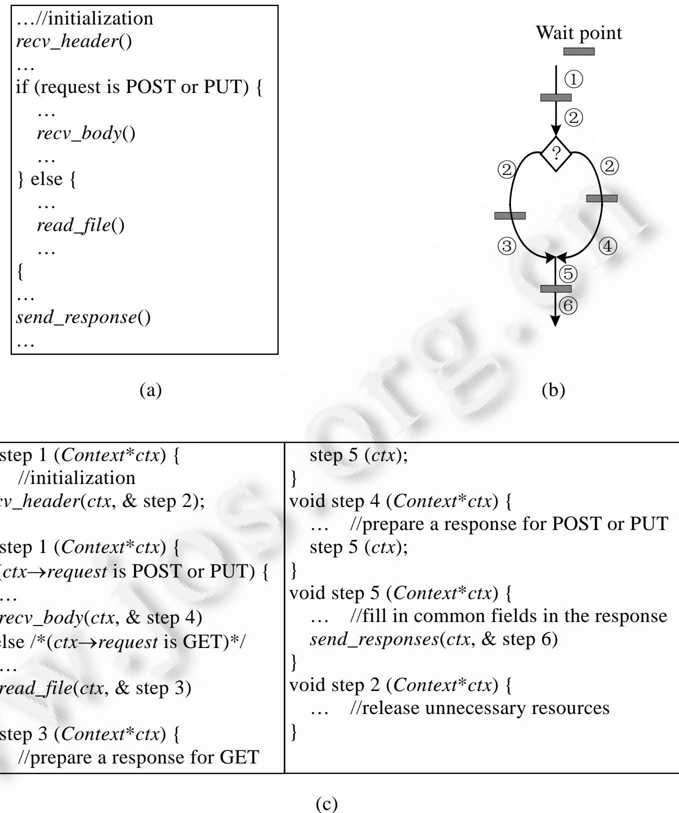


Fig.6 Wait points and their influences on the control flow

图 6 等待点及其对流程的影响

5 总结与展望

随着时间的推移,软件系统越来越呈现出网络化的趋势,AJAX,Mashup,Flex,SliverLight 等新一代富互联网应用(rich internet application,简称 RIA)技术的出现更助长了这种趋势,甚至有许多人认定 Web 将成为下一代系统平台的主角.网络程序设计将成为应用设计和开发的主体任务,而不再是附加功能.因此,网络程序设计技术在今后很长时间内都将是艰巨而重要的研究内容,而并发技术则是问题的核心所在.

并发问题是一个本质复杂的问题,没有一种方法能够将并发问题彻底化简.因此我们认为,使已有的解决方法能够沉淀并逐步提高抽象层次是一条重要途径.而现有的主流方法,如事件驱动、抢占式线程等,都在这方面有所欠缺.此外,另一条合理的途径便是并发的自动化.应用中的并发需求可以粗略地划分为两类:一类是源于应用功能上的需求,如服务器为多个客户提供并发服务;还有一类是源于性能需求.例如,许多下载工具可以并发下载一份文档的不同部分以充分利用网络带宽,缩短下载时间.这种源于性能的并发需求有望在一定程度上实现自动化,类似于高性能计算、数据库等领域中的自动并行化技术.截止到目前,只有极个别的工作[43,44]对此进行了初步研究,因此,这个方向仍有待更深入的开拓和探索.

© 中国科学院软件研究所 http://www.jos.org.cn

References:

[1] Lazowska ED, Patterson DA. Distributed computing. Science, 2005,308(6).

[2] Hoffman D, Novak T, Venkatesh A. Has the Internet become indispensable? Communications of the ACM, 2004,47(7):37−42. [doi: 10.1145/1005817.1005818]

[3] CNNIC. The 23rd Report on the Development of Internet in China. 2009 (in Chinese). http://www.cnnic.net.cn

[4] Li DY. Software engineering in the network era. Communications of CCF, 2009,1:7−12 (in Chinese with English abstract).

[5] Killian CE, Anderson JW, Braud R, Jhala R, Vahdat AM. Mace: Language support for building distributed systems. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2007). New York: ACM, 2007. 179−188. http://portal.acm.org/citation.cfm?id=1250734.1250755

[6] Li HB, Peng YX, Lu XC. The composability problem of events and threads in distributed systems. In: Proc. of the 2010 Int'l Conf. on Education Technology and Computer (ICETC 2010). Shanghai, 2010. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5529673

[7] Adya A, Howell J, Theimer M, Bolosky WJ, Douceur JR. Cooperative task management without manual stack management. In: Proc. of the General Track of the Annual Conf. on USENIX Annual Technical Conf. (ATEC 2002). Berkeley: USENIX Association, 2002. 289−302. http://www.usenix.org/events/usenix02/adyahowell.html

[8] Ousterhout JK. Why threads are a bad idea (for most purposes). In: Proc. of the Presentation Given at the '96 Usenix Annual Technical Conf. Berkeley: USENIX Association, 1996. http://home.pacbell.net/ouster/threads.pdf

[9] von Behren R, Condit J, Brewer E. Why events are a bad idea (for high-concurrency servers). In: Proc. of the 9th Conf. on Hot Topics in Operating Systems (HOTOS 2003). Berkeley: USENIX Association, 2003. http://www.usenix.org/events/hotos03/tech/vonbehren.html

[10] von Behren R, Condit J, Zhou F, Necula GC, Brewer E. Capriccio: Scalable threads for Internet services. In: Proc. of the 9th ACM Symp. on Operating systems Principles (SOSP 2003). New York: ACM, 2003. 268−281. http://portal.acm.org/citation.cfm?id=945471 [doi: 10.1145/1165389.945471]

[11] Lee EA. The problem with threads. Technical Report, No.UCB/EECS-2006-1, 2006. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html

[12] DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon's highly available key-value store. In: Proc. of the 11th ACM Symp. on Operating Systems Principles (SOSP 2007). Washington: ACM, 2007. 205−220. http://portal.acm.org/citation.cfm?id=1294261.1294281

[13] Lee EA. Threaded composite: A mechanism for building concurrent and parallel Ptolemy II models. Technical Report, No.UCB/EECS-2008-151. 2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-151.html

[14] Jr. Halstead RH. Multilisp: A language for concurrent symbolic computation. ACM Trans. on Program Language System, 1985,7(4):501−538. [doi: 10.1145/4472.4478]

[15] Zeldovich N, Yip A, Dabek F, Morris R, Mazieres D, Kaashoek F. Multiprocessor support for event-driven programs. In: Proc. of the 2003 USENIX Annual Technical Conf. (USENIX 2003). San Antonio, 2003. http://www.usenix.org/events/usenix03/tech/zeldovich.html

[16] Lauer HC, Needham RM. On the duality of operating system structures. SIGOPS Operating System Review, 1979,13(2):3−19.

[17] Krohn M, Kohler E, Kaashoek MF. Events can make sense. In: Proc. of the General Track of the Annual Conf. on USENIX Annual Technical Conf. (ATEC 2007). 2007. http://www.usenix.org/event/usenix07/tech/krohn.html

[18] Li P, Zdancewic S. Combining events and threads for scalable network services. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2007). New York: ACM, 2007. 189−199. http://portal.acm.org/citation.cfm?id=1273442.1250756 [doi: 10.1145/1273442.1250756]

[19] Dunkels A, Schmidt O, Voigt T, Ali M. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In: Proc. of the 4th Int'l Conf. on Embedded Networked Sensor Systems (SenSys 2006). 2006. 29−42. http://portal.acm.org/citation.cfm?id=1182807.1182811 [doi: 10.1145/1182807.1182811]

© 中国科学院软件研究所 http://www.jos.org.cn

[20] Fischer J, Majumdar R, Millstein T. Tasks: Language support for event-driven programming. In: Proc. of the 2007 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2007). New York: ACM, 2007. 134−143. http://portal.acm.org/citation.cfm?id=1244403 [doi: 10.1145/1244381.1244403]

[21] Dabek F, Zeldovich N, Kaashoek F, Maziμeres D, Morris R. Event-Driven programming for robust software. In: Proc. of the 10th Workshop on ACM SIGOPS European Workshop. New York: ACM, 2002. 186−189. http://portal.acm.org/citation.cfm?id=1133373.1133410 [doi: 10.1145/1133373.1133410]

[22] Abelson H, Sussman G. Structure and Interpretation of Computer Programs. Cambridge: Massachusetts Institute of Technology Press, 1984.

[23] Wadler P. Monads for functional programming. In: Proc. of the 1st Int'l Spring School on Advanced Functional Programming Techniques-Tutorial Text. 1995. 24−52. http://www.springerlink.com/content/715264614rh13340/

[24] Cunningham R, Kohler E. Making events less slippery with eel. In: Proc. of the 10th Conf. on Hot Topics in Operating Systems (HOTOS 2005). Berkeley: USENIX Association, 2005. 3. http://www.usenix.org/events/hotos05/final_papers/cunningham.html

[25] Gay D, Levis P, von Behren R, Welsh M, Brewer E, Culler D. The NesC language: A holistic approach to networked embedded systems. In: Proc. of the Programming Language Design and Implementation (PLDI 2003). ACM, 2003. http://portal.acm.org/citation.cfm?id=781131.781133 [doi: 10.1145/780822.781133]

[26] Rajagopalan M, Debray SK, Hiltunen MA, Schlichting RD. Profile-Directed optimization of event-based programs. In: Proc. of the 2002 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2002). Berlin: ACM, 2002. 106−116. http://portal.acm.org/citation.cfm?id=512543 [doi: 10.1145/543552.512543]

[27] Hayden MG. The ensemble system [Ph.D. Thesis]. New York: Graduate School, Cornell University, 1998.

[28] Bhatia S, Consel C, Lawall J. Memory-Manager/Scheduler co-design: Optimizing event-driven servers to improve cache behavior. In: Proc. of the 5th Int'l Symp. on Memory Management. 2006. 104−114. http://portal.acm.org/citation.cfm?id=1133971 [doi: 10.1145/1133956.1133971]

[29] Gu BC, Kim YT, Heo JY, Cho YK. Shared-Stack cooperative threads. In: Proc. of the 2007 ACM Symp. on Applied Computing (SAC 2007). 2007. http://portal.acm.org/citation.cfm?id=1244002.1244258 [doi: 10.1145/1244002.1244258]

[30] Shivers O, Might M. Continuations and transducer composition. In: Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2006). Ottawa, 2006. 295−307. http://portal.acm.org/citation.cfm?id=1133981.1134016 [doi: 10.1145/1133981.1134016]

[31] Jr. Baker HG, Hewitt C. The incremental garbage collection of processes. In: Proc. of the '77 Symp. on Artificial Intelligence and Programming Languages. ACM Press, 1977. 55−59. http://portal.acm.org/citation.cfm?id=800228.806932 [doi: 10.1145/800228.806932]

[32] Wagner DB, Calder BG. Leapfrogging: A portable technique for implementing efficient futures. In: Proc. of the ACM Symp. on Principles and Practice of Parallel Programming. 1993. 208−217. http://portal.acm.org/citation.cfm?id=155332.155354 [doi: 10.1145/155332.155354]

[33] Kranz DA, Jr. Halstead RH, Mohr E. Mul-T: A high-performance parallel lisp. In: Proc. of the ACM SIGPLAN'89 Conf. on Programming Language Design and Implementation (PLDI). ACM Press, 1989. 81−90. http://portal.acm.org/citation.cfm?id=74818.74825 [doi: 10.1145/74818.74825]

[34] Mohr E, Kranz DA, Jr. Halstead RH. Lazy task creation: A technique for increasing the granularity of parallel programs. IEEE Trans. on Parallel and Distributed Systems, 1991,2(3):264−280. [doi: 10.1145/91556.91631]

[35] Van Devoorde MT, Roberts ES. WorkCrews: An abstraction for controlling parallelism. Int'l Journal of Parallel Programming, 1988,17(4):347−366. [doi: 10.1007/BF01407910]

[36] Caromel D, Delbe C, di Costanzo A, Leyton M. ProActive: An integrated platform for programming and running applications on grids and P2P systems. Computational Methods in Science And Technology, 2006,12(1):69−77.

[37] Zhang LL, Krintz C, Soman S. Efficient support of fine-grained futures in Java. In: Proc. of the Int'l Conf. on Parallel and Distributed Computing Systems (PDCS 2006). 2006. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.966&rep=rep1&type=pdf

© 中国科学院软件研究所 http://www.jos.org.cn

[38] Zhang LL, Krintz C, Nagpurkar P. Language and virtual machine support for efficient fine-grained futures in Java. In: Proc. of the 16th Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT 2007). IEEE, 2007. http://portal.acm.org/citation.cfm?id=1299043 [doi: 10.1109/PACT.2007.45]

[39] Zhang LL, Krintz C, Nagpurkar P. Supporting exception handling for futures in Java. In: Proc. of the Principles and Practice of Programming in Java (PPPJ 2007). 2007. http://portal.acm.org/citation.cfm?id=1294349

[40] Zhang LL. Exploiting adaptation in a Java virtual machine to enable both programmer productivity and performance for heterogeneous devices [Ph.D. Thesis]. Santa Barbara: Graduate School, University of California Santa Barbara, 2008.

[41] Welc A, Jagannathan S, Hosking A. Safe futures for Java. In: Proc. of the 20th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005). 2005. http://portal.acm.org/citation.cfm?id=1094845

[42] Hill J, Szewczyk R, Woo A, Hollar S, Culler DE, Pister KSJ. System architecture directions for networked sensors. In: Proc. of the 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Boston, 2000. 93−104. http://portal.acm.org/citation.cfm?id=356989.356998 [doi: 10.1145/356989.356998]

[43] Oancea C, Selby JWA, Giesbrecht MW, Watt SM. Distributed models of thread-level speculation. In: Proc. of the 2005 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2005). 2005. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.7774&rep=rep1&type=pdf

[44] Li HB, Liu SY, Peng YX, Li DS, Zhou HJ, Lu XC. Superscalar communication: A runtime optimization for distributed applications. Science China (Information Sciences), 2010,53:1931−1946. [doi: 10.1007/s11432-010-4051-4]

附中文参考文献:

[3] 中国互联网络信息中心.第 23 次中国互联网络发展状况报告.2009.http://www.cnnic.net.cn

[4] 李德毅.网络时代的软件工程.中国计算机学会通讯,2009,1:7−12.

李慧霸(1980－),男,陕西西安人,博士生,主要研究领域为分布式计算,流媒体技术.

李东升(1978－),男,博士,副研究员,CCF 会员,主要研究领域为分布式计算.

田甜(1982－),女,助教,主要研究领域为分布式计算.

卢锡城(1946－),男,教授,博士生导师,主要研究领域为计算机网络,分布式计算.

彭宇行(1963－),男,博士,研究员,博士生导师,主要研究领域为分布式计算,流媒体技术.

© 中国科学院软件研究所 http://www.jos.org.cn