服务体执行流模型中消息通信的时间可预测性分析①

邹晓婷, 陈香兰, 李 曦

(中国科学技术大学 计算机科学技术学院, 合肥 230027)

摘 要: Minicore 是基于服务体执行流模型的新型微内核, 它有效的将操作系统中的存储模型和运行模型相分离. 微内核的高度模块化的设计使 Minicore 对服务体(Minicore 的基本单元)间的消息通信的依赖度极高. 于是对于 Minicore 操作系统的时间可预测性分析也无可避免的依赖于通信模块的时间可预测性. 本文的工作即是通过计算 Minicore 通信模块的 WCET, 分析消息通信的时间可预测性, 为未来实现时间可预测的通信机制并分析 Minicore 的时间可预测性提供基础. 对通信模块的 WCET 分析计算采用静态 WCET 分析中的基于路径的算法, 应用到 Minicore 系统的通信模块, 包括四个阶段: 提取目标代码片段, 程序控制流分析, 处理器特征分析和 WCET 计算. 基于 WCET 计算结果本文定义配置相关的时间可预测性(CIPr)作为评估消息通信时间可预测性的指标.

关键词: 实时系统; 可预测性; 服务体执行流模型; 消息通信; WCET

Time Predictability Analysis for Communication Modules in Servant / Exe-Flow Model

ZOU Xiao-Ting, CHEN Xiang-Lan, LI Xi

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)

Abstract: Minicore is a new microkernel based on Servant / Exe-Flow Model, in which the execution model and the storage model are independent of each other. The highly modularity design of microkernel made Minicore rely heavily on the inter-servant communication module, because a servant is the basic unit of the operating system. As a result, in order to analyse its time predictability, it is inevitable to conduct a time predictability analysis for Communication Modules in Servant / Exe-Flow Model. In the thesis, we estimate the WCET of Minicore's communication modules and analyze its time predictability with the definition of CIPr (Configuration-Induced Timing Predictability), providing data to support its future mechanism of timing predictability, and foundation for future work of time predictability analysis for Minicore. Static WCET analysis and a path-based algorithm have been adopted in our work. Applying to the communication modules in Minicore, the method includes four phrases: extract code snippets for each communication mode, analyse communication control flow, analyse processor features, and calculate WCET finally.

Key words: real-time system; time predictability; Servant / Exe-Flow Model; communication by message; WCET

服务体执行流模型(Servant / Exe-Flow Model , SEFM)[1]有效的将操作系统中的存储模型和运行模型相分离, 从根本上提高了操作系统的可扩展性、效率以及对分布式计算的支持. 基于 SEFM 实现的操作系统 Minicore 中, 服务体是实现一系列功能的数据和代码的集合, 是操作系统的基本组成单位, 而执行流是系统计算能力的抽象, 刻画了 CPU 的执行能力[2]. 由此, 执行流和服务体生命周期的相互独立性使他们所刻画的存储模型和运行模型解耦和. 作为任务运行平台的操作系统, 执行流通过消息推动在服务体之间流动, 请求特定的服务, 完成任务. 其中, 服务体将其功能函数组织为端口, 端口的实例是一个小端口, 一个端口可以包含一个或者多个小端口. 当需要请求某个功能时, 程序指定服务名字, 向相应端口发送消息请求

① 基金项目:国家"核心电子器件、高端通用芯片及基础软件产品"重大专项(2012ZX01034001-001);国家自然科学基金(61379040,61272131)
收稿时间:2015-03-31;收到修改稿时间:2015-05-15

该服务.

Minicore 内核主要进行进程调度、中断管理、地址空间管理和服务体间通信, 而其他大部分的功能都由服务体来实现. 这样高度模块化的设计使 Minicore 对服务体间的消息通信的依赖度极高, 因此消息推动的性能直接影响整个操作系统的性能.

Minicore 的目标应用场景在于航天航空领域, 其中许多的功能要求在确定的时间内执行计算或处理事件并对外部异步事件做出响应. 它的准确性不仅依赖于计算结果的逻辑正确性而且与结果的完成时间有关. 以强实时系统为目标, Minicore 的时间可预测性是它需要实现的一个重要特征.

可预测的实时计算的目的是允许对单个任务以及系统全局的时间属性的评估. 可预测性对体系结构的需求包括有界的指令执行时间和访存时间以及有界的进程间通信开销. 当体系结构满足上述特征, 其上的系统软件即操作系统便能保证有界的调度和同步开销、有界的操作系统原语以及有界的代码执行时间. 这样的操作系统才能支持应用程序的时间可预测性.

可预测系统需要通过精心的设计和实现以达到对时间的要求[3]. 这就意味着需要对操作系统代码进行离线分析, 通过分析证明操作系统是否满足可预测性的时间需求. 静态分析操作系统的时间可预测性是一个很有挑战性的问题, 因为每一个对时间属性的要求都会自上而下的传播到系统的各个模块. 于是对一个操作系统时间可预测性的分析必须涵盖操作系统的所有模块, 包括处理器, 指令集架构, 编程语言, 编译支持, 运行时系统, 调度以及通信等. 作为系统中至关重要的通信模块, 本文分析了其时间可预测性, 从而为未来分析 Minicore 的时间可预测性提供基础.

虽然时间可预测的通信机制是时间可预测操作系统的必要条件, 但其他模块的时间可预测性也至关重要. 实验室课题组的下一步工作是要分析 Minicore 中其他模块的时间可预测性, 以支撑 Minicore 操作系统的时间可预测性.

分析通信模块的时间可预测性的策略之一是, 分析并计算通信模块程序的最差情况下的执行时间 (Worst-Case Execution Time , WCET)[4]. 事先获取系统中通信模块的最差情况下的执行时间对实时系统的时序分析具有特别重要的意义. 事实上, Minicore 中的所有任务都需要消息通信, 而事先得知消息通信的 WCET 是获得任务的 WCET 的必要条件. 而任务的 WCET 既是进行调度及可调度性检测的前提, 又是系统设计中软硬件界限划分的一个依据, 同时还是确定周期性任务是否满足其性能目标,从而发现系统性能瓶颈的基础.

本文的剩余部分的组织如下: 第一部分介绍现有的微内核的进程间通信机制及其时间可预测性, 第二部分计算消息通信代码的最坏执行时间, 基于此第三部分分析 Minicore 消息通信模块的时间可预测性, 从而为分析 Minicore 的时间可预测性提供理论和数据基础, 文章最后进行总结.

1 微内核操作系统的通信方式

本章节介绍微内核的发展历史, 具体来说是第一代微内核代表 Mach 和第二代微内核代表 L4, 着重阐述其进程间通信机制以及分析通信的时间可预测性.

1.1 Mach

卡内基梅隆大学研发的 Mach[5]是第一代微内核操作系统的代表. 其进程间通信, 尤其是客户端和服务端之间的通信对于操作系统的性能至关重要. 由于 Mach 属于微内核结构, 内核自身提供的功能相当有限, 任务必须与其他提供服务的系统模块诸如文件系统和网络服务等进行通信. Mach 中的通信通道被称作端口[6]. 端口是一个拥有消息队列的单向通道, 存在于内核. 消息在消息队列中以 FIFO 的模式进行组织. 每一个端口可以有多个发送者但只能有一个接收者. Mach 消息模块传递消息需要两次数据拷贝[7]: ①往端口发消息时, 微内核将消息数据从发送进程的地址空间拷贝到内核缓存中; ②接收进程将消息数据从内核缓存拷贝到自己的地址空间. 频繁的消息拷贝是 Mach 性能较差的主要原因. 为此开发者们在后续的开发中提出了诸如共享内存, 写时复制等策略进行优化, 但效果不佳.

在实时环境中, 多个消息可能在队列中排队等待, 并且消息的接收者的执行依赖于优先级调度算法. 而 Mach 中没有一个机制能保证消息被及时的接收处理. 于是 Mach 的通信模块不具有时间可预测性.

除了 Mach, 典型的第一代微内核操作系统还有 L3[8], Minix[9]等. 第一代操作系统设计时关注更多的是一种区别于宏内核的机制, 没有关注性能的问题, 更何况实时性以及可预测性. 所以第一代微内核的通信模块是不具有时间可预测性的.

1.2 L4

分析了微内核性能低下的原因, 即强隔离带来的大量进程间通信以及由此带来的大量在内核中进行的验证等开销, L4[10]应运而生.

L4 中的进程间通信是同步的, 没有消息缓存, 没有两次拷贝, 仅仅简单的传递消息. 进程间通信的一个基本问题是数据需要跨越不同的地址空间. 大多数操作系统的解决办法是通过内核使用两次数据拷贝. 这两次数据拷贝本身耗时很大, 如果考虑 TLB 和 Cache miss 耗时会更大. 但两个地址空间直接共享逻辑地址空间会带来一系列的安全问题. L4 巧妙地通过暂时地址映射[11]将数据由发送进程地址空间直接转移到接收进程地址空间, 这大大提高了 IPC 性能. 具体做法是: 内核把数据的目的地址暂时映射到一个位于内核空间的通讯窗口, 然后内核把数据从发送进程地址空间拷贝到该通讯窗口, 至此消息数据有且仅有接收者可以访问.

通信性能的改善使 L4 具有支持实时性和可预测性的潜力. 在现有的 L4 项目中, Fiasco 作为最早出现并获得最多关注的一个项目, 使用类似于一般实时系统的资源预留机制 (Resource Reservation)[12], 达到硬实时的效果, 也具备了时间可预测性.

2 消息通信的WCET

Minicore 中唯一的消息发送原语是 sendmsg. sendmsg 首先根据系统中服务体和端口的组织方式由程序指定的服务名字找到对应的端口, 再指定一种消息发送方式请求服务. 系统支持三种消息发送方式[13].

同步分离. 同步分离方式不需要应答消息. 以该模式发送消息时, 执行流立即切换到目标服务体的地址空间执行服务. 当服务完成, 执行流继而切换到另一个事先指定的端口或者返回给核心服务体进行调度. 后者的情况表明服务所属的事务已经执行完毕. 无论何种情况, 执行流都不会返回原来的小端口, 于是源小端口在发送消息后即被系统回收.

异步方式. 程序发送异步消息之后继续执行, 而消息被异步地处理. 异步消息为目标端口新建一个小端口作为一个事务, 该事务的调度运行交给操作系统, 与源端口无关.

同步连续. 程序发送同步消息之后阻塞, 等待应答, 而执行流即被引导到目标端口新建的小端口执行. 当服务完成, 应答消息引导执行流回到源小端口的断点处继续执行. 当执行流跨越地址空间时, 寄存器上下文以及栈空间被保存在小端口; 执行流返回时从小端口恢复. 该过程类似进程线程模型中跨地址空间的函数调用.

根据论文[14]中的定义, 对 WCET 的分析需要满足: ①计算得出的执行时间上限是安全的, 即程序无论在何种情况下的执行时间都不能超过这个上限; ②计算得出的执行时间上限是紧致(tightness)的, 即必须是可接受的高估值.

计算程序的WCET的方法分为有静态分析方法和动态分析方法. 其中静态分析方法并不要求代码在实际的硬件平台或者模拟器上运行, 而是基于代码本身, 分析代码的所有可能执行路径, 并结合底层硬件平台的建模, 获得代码的执行时间上限. 本文的分析计算采用静态 WCET 分析, 应用到 Minicore 系统的通信模块, 包括四个阶段, 如图 1 所示: 提取目标代码片段, 程序控制流分析, 处理器特征分析和 WCET 计算.

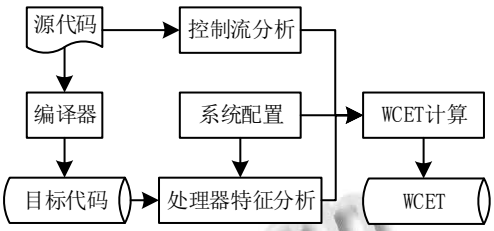

图 1　静态 WCET 分析流程

其中程序控制流分析的目的在于提取程序的动态行为, 分析内容包括函数调用、循环次数、判断语句之间知否有依赖关系等. 程序控制流分析的方式包括手工标注以及自动分析. 本文采用自动分析的方式, 根据程序语法和语义自动获取程序的流信息. 而处理器特征分析是指根据处理器的特征对程序的目标代码进行分析, 以获得每一条指令、每一个基本块实际的时间行为. 对于不带 cache、没有流水线的简单 CISC 指令, 此分析就非常简单. 一个代码段的执行时间就是其所对应的每条指令执行时间的累加. 但对于带有高速缓存和流水线等的现代处理器, 代码段执行时间的估算就复杂得多.

2.1 提取目标代码片段

静态 WCET 分析中基于路径的算法, 是以控制流图模型为基础, 通过计算控制流图中可能存在的路径

的执行时间来估计代码片段的 WCET. 由于三种通信方式使用的是同一接口 sendmsg, 在分析它们的 WCET 之前需要将三种通信方式的代码相分离, 即产生三条执行路径, 分别为 sendmsg_CONT, sendmsg_SPLIT 和 sendmsg_ASYN. 以 sendmsg_CONT 为例, 其伪代码见代码 1.

代码 1 同步连续消息发送过程伪代码

```
Uint32 sendmsg_CONT(Msg* p_msg)
{
    准备工作，包括查找目标端口并创建目标小端口等;

    //将msg压入dest_mport的栈;
    my_memcpy(p_dest_mport->context.esp-sizeof(Msg),
            p_msg, sizeof(Msg));
    p_dest_mport->context.esp-=sizeof(Msg);
    p_dest_mport->context.esp -= sizeof(Uint32);
    //设置小端口状态
    p_dest_mport->state = ready;
    p_current->state = sendout;

    //上下文切换
    switch_to(p_current,p_dest_mport);

    //返回时断点，获得服务计算结果
    asm volatile(
        "movl %%eax, %0\n\t"
        : "=r"(result)
        :\
    );
    return result;
}
```

同步连续消息需要应答, 于是其消息应答伪代码如代码 2 所示.

代码 2 同步连续消息应答过程伪代码

```
void returnmsg_CONT(int result, Msg* p_msg)
{
    //获取当前小端口的ID
    Miniport* p_current = current;
    //找到请求该服务的源小端口，即要返回的小端口
    Miniport* p_dest_mport =
      get_miniport_from_id(p_msg->source_mport_id);

    //释放服务小端口
    miniport_put(p_current);
    p_dest_mport->state = ready;
    //上下文恢复
    switch_return(p_current, p_dest_mport,result);
    return;
}
```

一个完整的同步连续消息过程的 WCET 为以上两者的 WCET 之和.

2.2 程序控制流分析

程序的控制流依赖于输入数据. 然而一般情况下, 最坏执行情况的输入是不确定的, 于是可以通过对程序代码的分析, 提取出程序包含的基本块以及基本块之间的控制关系, 为每一个目标程序片段维护一个控制流图(CGF)[15]. 对包含循环的程序还要进一步确定循环的上界, 排除多个分支之间可能存在的不可行路径等. 控制流图保存了所有可能的执行路径.

由于消息发送模块代码涉及服务查找和新建小端口等对象管理操作, 代码结构复杂, 实验中使用一款由新加坡国立大学研发的开源软件 Chronos[16]进行分析计算. Chronos 通过分析源码, 得出消息发送过程调用了 17 个相关函数, 一共划分为 72 个基本块, 包含 297 条 MIPS 指令. 其中 sendmsg 函数标号 15, 控制流图以及目标代码如下图 2 所示.

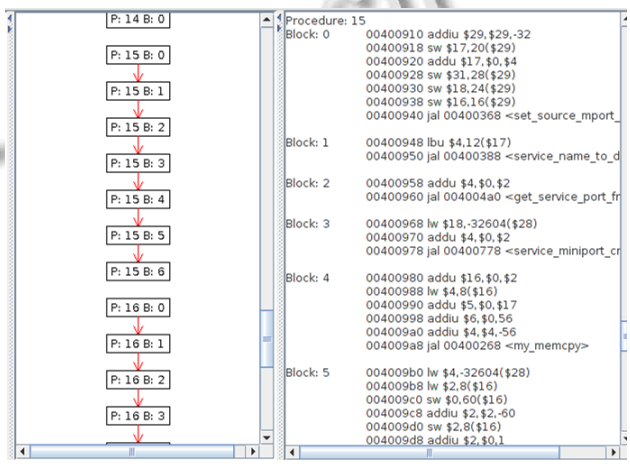

图 2 sendmsg 函数控制流图和目标代码

由于 sendmsg 函数代码没有条件判断或者循环语句, 各自组成基本块, 于是控制流图比较简单. 由于目标代码片段控制流关系的简单性, 通信模块不存在不可行路径.

2.3 处理器特征分析

程序控制流分析之后, 需要针对具体的目标执行环境, 确定控制流分析所解构出来的基本块的语句或指令序列的执行时间. 处理器特征分析必须基于特定的目标硬件体系结构, 以目标硬件指令执行时间模型为依据, 以基本块语句或指令序列为输入, 计算或估计基本块的执行时间. 处理器分析的主要考虑的内容是缓存、流水线和分支预测等. 本文使用 Chronos 中默认的处理器配置, 如下图 3 所示: 指令乱序执行, 指令预取队列大小为 4, 指令缓存块大小 8B, 开启分支预测, BHT(分支历史表)大小为 16.

2.4 WCET 计算

采用基于路径的 WCET 计算方法, 通过遍历程序控制流图上所有可能的执行路径, 计算各路径的执行时间, 从中找出执行时间最长的路径, 该路径的执行

时间作为程序的 WCET 估计值. Chronos 计算出程序控制流图中各基本块结点的执行时间(估计值)并要求人为确定循环的上界次数, 在此基础上执行基于路径的 WCET 计算算法. 根据论文[16], Chronos 计算程序 WCET 的目标函数是:

图 3　处理器参数配置

$$max \sum_{B \in B} N_B * c_B$$

其中 B 表示基本块的集合, N_B 表示基本块 B 被执行的次数, 而 c_B 表示基本块 B 估计的最坏执行时间. 根据前面的程序控制流分析以及对处理器特征的假设, Chronos 计算得出三种通信方式(其中同步连续包括消息应答)的最坏执行时间, 如下表 1 所示. 其中目标代码指令是编译之后的 MIPS 目标代码指令, WCET 的单位是机器周期数.

表 1　WCET 计算结果

程序名称	目标代码指令数	基本块数	WCET
sendmsg_CONT	319	75	47890
sendmsg_SPLIT	307	67	46337
sendmsg_ASYN	298	68	47370

计算结果显示, 三种通信方式的 WCET 之间差距不大, 这是因为查找服务端口和新建小端口占据了消息发送过程很大一部分, 而三种通信方式都需要做这两个动作. 另外, 同步连续消息的 WCET 比同步分离和异步略大, 多出的部分是消息应答, 以及由此产生的第二次的上下文切换的开销. 然而这里仅仅涉及通信部分, 在实际系统中, 同步分离消息附带有服务结束后消息转移的开销, 而异步消息附带有调度开销, 这些都是本文不作考虑的部分.

3　系统通信模块的可预测性分析

对于实现 Minicore 的时间可预测性, 通信模块的策略是实现带时间的消息, 消息中设置了消息送达时间, 服务时间, 响应时间等时间限制. 而本文的工作, 即通过对通信模块的 WCET 的计算, 为带时间的消息的时间设置提供数据支撑, 并为消息中的时间设置的可行性判断提供了基础.

根据论文[4], 定义 Q 和 I 分别为系统所有硬件状态的集合和程序所有可能输入的集合, 以及硬件状态的不确定性 $Q \in Q$ 和程序输入的不确定性 $I \in I$. Q 和 I 的所有可能组合表示了系统的不确定性. 那么程序的时间可预测性可以定义为:

$$\Pr_p(Q, I) := \min_{q_1, q_2 \in Q} \min_{i_1, i_2 \in I} \frac{T_p(q_1, i_1)}{T_p(q_2, i_2)}$$

其中 T_p 表示预测的执行时间, 比值 Pr_p 的取值范围为[0,1], 等于 1 时表示程序的时间可预测性最好. 论文[4]还定义了如下的输入相关的时间可预测性(Input-Induced Timing Predictability, IIPr).

$$IIP\ \mathrm{r}_p(Q, I) := \min_{q \in Q} \min_{i_1, i_2 \in I} \frac{T_p(q, i_1)}{T_p(q, i_2)}$$

本文的目的是分析 Minicore 操作系统的消息通信代码的时间可预测性, 诸如服务体个数, 每个服务体拥有的端口个数上限等系统配置也是消息通信代码的输入. 简化以上定义, 本文定义 Minicore 中配置相关的时间可预测性(Configuration-Induced Timing Predictability, CIPr).

$$CIP\ \mathrm{r}_p(C) := \min_{c_1, c_2 \in C} \frac{WCET_p(c_1)}{WCET_p(c_2)}$$

其中 C 表示系统所有可能的配置, $WCET_p$ 计算系统在某个配置下程序 p 的最坏执行时间. 同样地, $CIPr_p$ 的取值范围为[0,1], 等于 1 或接近 1 时表示程序是绝对可预测的, 即与系统的配置无关.

微内核 Minicore 中, 在发消息请求服务时, 系统需要根据当前任务指定的服务名字找到相应的服务端口并为其分配一个小端口, 于是与消息通信模块相关的系统配置项包括表 2 中内容, 表中给出了对配置项的描述以及可选的配置方案.

表 2　与消息通信模块相关的系统配置项

相关配置	描述	配置方案
SERVANT_NUM	系统中服务体的数量上限	16, 8, 4
PORT_NUM	每个服务体允许拥有的端口数量	32, 16, 8
MPORT_NUM	每个端口允许拥有的小端口数量	64, 32, 16
STABLE_SIZE	全局服务表大小	256, 128, 64

根据表中的配置方案组合对 Minicore 通信模块代码的 WCET 进行计算, 使用最大值和最小值计算配置相关的时间可预测性, 结果见表 3. 表中 *CIPr* 的值不太理想, 说明通信模块代码的可预测性与系统配置相关性大. 原因存在于系统在查找服务端口和新建小端口时需要对系统中对象的组织结构进行查询、插入和删除等操作, 而这些操作的时间复杂度取决于表 2 中的系统配置值.

表 3 配置相关的时间可预测性计算

程序名称	最小 WCET	最大 WCET	CIPr
sendmsg_CONT	47890	70404	0.680217
sendmsg_SPLIT	46337	68334	0.678096
sendmsg_ASYN	47370	69945	0.677246

此外, 对系统通信模块的 WCET 分析数据是确定任务是否能满足其性能目标的基础. 但由第二节的数据并不能得出一下结论: 更少的使用同步连续消息就能提高系统实时性. 相反地, 同步连续消息是 Minicore 通信的优势. 原因是, 在分析任务或者通信代码的 WCET 时, 本文并不考虑系统的并发和强占, 而是将他们委托给操作系统. 然而为了减少系统的响应时间, Minicore 引入了引流的概念, 即不通过调度器的上下文切换. 在任务发送一个同步连续消息的时候, 执行流从当前上下文切换到目标小端口, 请求服务. 服务完成, 执行流返回也不经过调度器. 由此不难看出, Minicore 中的同步消息是不会阻塞的, 是同步的. 这样的消息通信方式对于实现系统的实时性以及时间可预测性具有很大的贡献.

4 总结

操作系统的时间可预测性是运行在其上的用户程序时间可预测的保障. 为了将服务体执行流模型更广泛的应用到国防、航空、航天、自动控制等对系统实时性要求较高的领域, Minicore 必须具有时间可预测性. 本文分析计算了 Minicore 通信模块代码的 WCET, 并基于此对通信模块的时间可预测性进行了分析, 为今后实现带时间限制的消息提供了数据支撑, 并为进一步分析和实现 Minicore 操作系统的时间可预测性提供基础.

参考文献

1 Hong L, Xiang-lan C, Ming-qiao W, et al. Design of a servent based operating system. Journal of Computer Research and Development, 2005, 42(7): 1272–1276.
2 Wu M, Chen X, Zhang Y, et al. A new operating system construction model based on servant and executive flow. Journal of University of Science and Technology of China, 2006, 2: 019.
3 Axer P, Ernst R, Falk H, et al. Building timing predictable embedded systems. ACM Trans. on Embedded Computing Systems (TECS), 2014, 13(4): 82.
4 Wilhelm R, Engblom J, Ermedahl A, et al. The worst-case execution-time problem--overview of methods and survey of tools. ACM Trans. on Embedded Computing Systems (TECS), 2008, 7(3): 36.
5 Accetta M, Baron R, Bolosky W, et al. Mach: A new kernel foundation for UNIX development. 1986.
6 Lepreau J, Hibler M, Ford B, et al. In-Kernel Servers on Mach 3.0: Implementation and Performance. USENIX MACH Symposium. 1993: 39–56.
7 Kitayama T, Tokuda H, Nakajima T. RT-IPC: An IPC Extension for Real-Time Mach. USENIX Microkernels and Other Kernel Architectures Symposium. 1993: 91–104.
8 Hohmuth M, Rudolph S. Steps towards porting a Unix single server to the L3 microkernel. 1996.
9 Herder JN, Bos H, Gras B, et al. MINIX 3: A highly reliable, self-repairing operating system. ACM SIGOPS Operating Systems Review, 2006, 40(3): 80–89.
10 Elphinstone K, Heiser G. From L3 to seL4 what have we learnt in 20 years of L4 microkernels. Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 133–150.
11 Klein G, Andronick J, Elphinstone K, et al. seL4: formal verification of an operating-system kernel. Communications of the ACM, 2010, 53(6): 107–115.
12 Liedtke J. Toward real microkernels. Communications of the ACM, 1996, 39(9): 70–77.
13 Gong Y, Tang L, Shi Z, et al. Binary Compatible Linux Running Environment on Minicore3. 0 Operating System. Journal of Chinese Computer Systems, 2008, 6: 003.
14 Puschner P, Burns A. Guest editorial: A review of worst-case execution-time analysis. Real-Time Systems, 2000, 18(2): 115–128.
15 Bernat G, Colin A, Petters S M. WCET analysis of probabilistic hard real-time systems. Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE. IEEE, 2002: 279–288.
16 Li X, Liang Y, Mitra T, et al. Chronos: A timing analyzer for embedded software. Science of Computer Programming, 2007, 69(1): 56–67.