

一种采用 Lock-Free 同步机制的数据结构的研究

黄姝娟, 容晓峰, 茹 媛, 张 雅

(西安工业大学 计算机科学与工程学院, 陕西 西安 710021)

摘 要: 随着多核芯片在嵌入式实时系统中的广泛应用, 采用传统锁机制对共享数据的访问会引起死锁、优先级翻转以及 CPU 饥饿现象, 进而会影响系统整体性能的发挥, 严重时还会导致不可估量的损失发生. 为此, 针对 Skip-list 的共享数据结构提出了一种 Lock-Free 的同步机制方法, 对该方法的线性一致性进行了证明, 并将其应用于混合关键任务的共享优先级调度队列中. 经过实验, 该方法在吞吐量、核利用率以及平均响应时间方面优于传统的基于锁机制的方法, 有效地提升了多核运行的效率.

关键词: 嵌入式系统; 多核; 无锁机制; 锁机制; 混合关键系统

中图分类号: TP302

文献标识码: A

文章编号: 1000—7180(2016)10—0072—06

A Lock-Free Synchronization Mechanism for Shared Data Structure

HUANG Shu-juan, RONG Xiao-feng, RU Yuan, ZHANG Ya

(School of Computer Science and Engineering, Xi'an Technological University, Xi'an 710021, China)

Abstract: With the multi-core chip wildly used in the embedded real-time system, the traditional lock synchronization may cause deadlock, priority inversion and cpu starvation. It may degrade performance and cause inestimable loss to the real-time system. This paper proposed a lock-free alogrithm for the Skip-list data structure and gave the proof for its linear consistency and used it as a shared priority queue for mixed-critical system. The experimental results show that this method has much better than the lock mechanism on average response time and utilization rate of multi-core processor and have the higher system throughput rate.

Key words: embedeed system; multicore; lock-free mechanism; lock mechanism; mixed-criticality system

1 引言

在嵌入式实时系统中, 共享优先级队列是一种非常重要的数据结构, 经常作为任务调度的就绪队列, 如图 1 所示. 由于共享优先级队列同步机制是影响实时系统响应时间、系统吞吐量、核利用率的重要因素, 而不同处理器核对共享优先级队列的同步访问是其中一个重要环节. 为此, 寻求高效、并行的优先级队列的同步实现机制成为重要的研究课题^[1].

当前共享优先级数据结构可以是循环数组、堆、队列或者循环链表等. 无论哪种数据结构, 其实现同

步的算法大多基于加锁机制, 有些采用粗粒度锁机制, 有些采用细粒度锁机制. 文献[2]指出, 用一个简单的全局变量作为粗粒度锁机制来访问二进制堆, 不但使得并行性效率不高, 而且还要求 put 和 get 操作必须按照顺序执行, 这样才能保证一致性. 文献[3] 为了提高多核并行的效率, 针对多核的共享内存体系结构, 提出了一个基于 Work-Stealing 的数据收集器来实现数据并行操作. Work-Stealing 收集器需要更多的细粒度锁机制来保证 Work-Stealing 的有效执行. 但当收集器数量增加到一定程度时, 也存在死锁现象. 文献[4]提出了基于细粒度锁机制的

收稿日期: 2016—01—07; 修回日期: 2016—02—23

基金项目: 陕西省工业科技攻关项目(2015GY031); 陕西省教育厅基金项目(15JK1342); 西安工业大学校长基金项目(XAG-DXJJ14015)

跳跃表(Skip-list),要求为每个指针加锁.虽然扩展了并发优先级队列的功能,但锁的数目比结点数目还多,随着结点数量增多,空间需求量会增大,且优先级反转现象也难以避免.

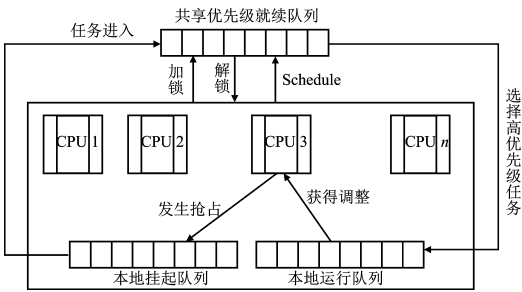


图1 共享优先级队列作为任务调度的就绪队列

上述无论哪种形式的互斥锁都易引起阻塞,进而导致死锁、优先级反转和CPU饥饿现象^[5],从而降低系统的整体性能.为了解决这些问题,研究者建议为共享数据结构使用非阻塞算法.文献[1]指出非阻塞算法优于基于锁机制的竞争算法.非阻塞算法因不包含互斥,不用担心因阻塞而引发的问题.典型的非阻塞算法是 Lock-free. Lock-free 的实现保证了不会出现因并发操作而引起的竞争,且不会因为阻塞引发CPU饥饿.近年来,非阻塞算法因其良好特性一直引起极大关注.文献[6]针对高吞吐量的I/O系统、数据中心、视频服务,提出了一个并行的非阻塞时隙调度算法,将时间分为不同的时隙粒度,以时隙为单位进行调度.文献[7]提出了一种无锁的cuckoo的哈希算法.文献[8]提出了一种无锁的内存分配算法以减少锁机制带来的开销.文献[10]提出了一种无锁机制实现的ELB-trees (Efficient and Lock-free B-tree).文献[11]指出目前大多数的无阻塞算法存在的问题是缺乏精确的语义定义以及对并发对象的正确性做出证明.

本文将文献[4]中提出的 Skip-list 的数据结构进行改进,并利用现代操作系统都可获得的同步原语(Compare And Swap, CAS)实现无锁访问.首先给出了该无锁算法主要操作的语义定义,并对该算法的线性一致性给予证明,最后和只用锁机制实现的并发优先级队列的算法进行对比,结果表明该方法有更好的效果.

2 相关内容介绍

Skip-list(跳表)是一种基于分层的数据结构,如图2所示的一个4层跳表,第二层的链表是每隔1个结点增加一个指向它之后具有2层指针的结

点,第三层链表是每隔3个结点就增加一个指向它之后具有3层指针的结点,第四层链表是每隔7个结点就增加一个指向它之后具有4层指针的结点,如此类推,跳表第 $i(i \geq 2)$ 层的结点都有一个指向每隔 $2^{i-1}-1$ 个结点之后,具有 i 层指针的结点,那么只要搜索不超过 $\lceil \frac{n}{2^{i-1}} \rceil + 2^{i-2}$ 个结点即可,而此时每层只使指针的数目增加了 $\lceil \frac{n}{2^{i-1}} \rceil$ 个.这样的数据结构搜索速度是很快的.

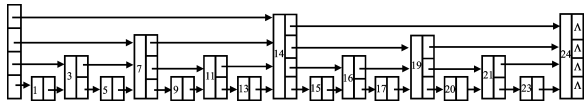


图2 4层 Skip-list 数据结构

要在跳跃表中进行插入和删除操作,需要先搜索要插入的位置.搜索的方法是从首结点的最高级别开始搜索,如果此级别的下一个结点的值小于插入的值,则继续在此级别上遍历下一个结点;否则降低一个级别再搜索,一直到最低级别.例如在图2所示的结构中,要插入值为10的结点,那么搜索的路径如图3中虚线所示.搜索每一个级别时,要记录下该级别最后读取的结点,在之后的插入操作中使其相应级别的后继指针指向新结点.插入后的列表如图4所示,可以看出,为了维护 Skip-list 表的结构保持原来的隔离特征,必须将插入的结点及后续结点的层级做统一调整,这样会加大插入结点的开销,于是在后续改进中不对同一层之间的隔离结点数做严格要求.

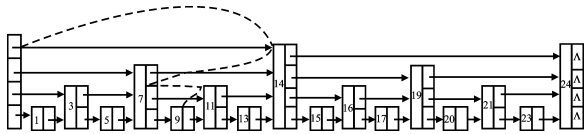


图3 在 Skip-list 中查找插入值为 10 的结点位置

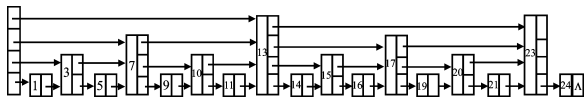


图4 插入值为 10 的结点的 Skip-list

3 Skip-list 无锁算法实现及其线性一致性证明

3.1 Skip-list 的改进以及无锁算法实现

为了将 Skip-list 数据结构应用于混合关键系统的共享就绪队列中,对该数据结构每层含义以及节点结构进行改进,改进后的 Skip-list 具有如下

特征:

(1)在 Skip-list 数据结构中,每个节点就是一个关键任务,该任务的层数就是该任务的最高关键级别.如果一个任务具有 h 个关键级别,那它就是一个 h 层结点,那么对应的 forward 将指向一个包含 h 层的任务.如果一个关键任务具有 k 个关键级别,则该任务一定是一个 k 层结点.

(2)所有关键任务的最高关键级别对应 Skip-list 中结点所拥有的最高层数.

头节点的各个层数代表了系统所对应的关键级别.当系统关键级别为 k 时,调用第 k 层链表中各个任务,如果该层的任务还没有启动,则调用 $k-1$ 层的任务,当 k 层的任务一旦启动则系统关键级别自动升级为 k 级,则调用 k 层的任务.

(3)混合关键任务在跳表中每层的排列顺序是在不同关键级别情况下,基于不同调度算法得出的优先级次序.

根据如上特征,设计出如图 5 所示的混合关键任务的 Skip-list 数据结构图.从图中可以发现,系统在关键级别分别为 2 和 1 时,任务 3 和任务 1 的执行顺序是不同的.在系统关键级别为 2 时,任务 3 先于任务 1 执行,而在系统关键级别为 1 时,任务 1 先于任务 3 执行,这种情况在实际当中是存在的.

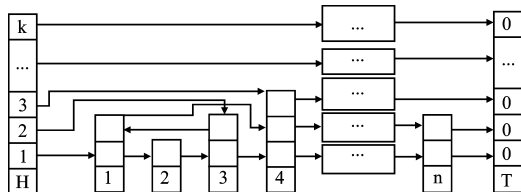


图 5 混合关键系统中改进的 Skip-list 跳表设计

3.2 线性一致性的证明

线性一致性是由 MAURICE P. HERLIHY 和 JEANNETTE M. WING 提出的一种判断并发对象被访问的正确性条件^[12].并发对象是指被并发进程共享的数据对象.线性一致性是一种从抽象数据类型语义来说,对并发对象进行操作的正确性条件.它允许高度的并发性,并且可以使用已知的顺序性领域的知识,对并发对象的操作进行定义和论证.线性一致性假设所有的并发对象的操作都是在其被调用和完成之间的某一点瞬间产生影响,也就是说一个并发对象的操作可以用前置条件和后置条件来定义.如果一个系统每个独立的部分都是线性一致的,那么该系统也是线性一致的.

使用线性一致性的概念来论证并发性的意义在

于以下两点.

(1)证明对象的线性一致性时,可以将不变式的表示和抽象方法等理论运用到并发领域.

(2)在使用线性一致性对象论证计算的正确性时,可以将关于对象的并发计算的断言转化成它们对应的顺序计算的断言.

文献[13]给出了如下证明无锁算法线性一致性的定义和方法.

定义 1 一个共享并发数据的线性化,是指在每次并行执行该数据的操作过程时,都最终对应一个等效的串行执行过程.

定义 2 一个 history 过程是指一系列的 invoke 和 return 事件.将 $\text{inv}(p, \text{op}, \text{in})$ 加入到 history 事件中,是指进程 p 激活了带有 in 输入的操作 op . $\text{ret}(p, \text{op}, \text{out})$ 是指进程 p 返回了带有 out 的输出操作 op .

定义 3 一个 history 是线性化的,是指它能被重新安排到一个顺序的 history 中,且重新排的顺序必须遵从最原始的 history:一个操作的激活必须发生在另一个操作返回之后,是按照串行 history 来结束.

定义 4 并行实现一种数据结构是线性一致性的,是指如果它创建的每个 history 都是线性化的.

对 Skip-list 采用 Lock-Free 算法实现,主要是插入和删除两种操作.为了证明其线性一致性,现将执行插入和删除的伪代码给出:

```
maxlevel: integer // 最大安全关键级别;
prior: integer // 优先级;
mark: Boolean // 删除标记;
value: all kinds of data // 赋值数据;
Head, tail: Skip_List; // 头、尾指针;
prv, last, Node_head, Node1, Node2, NewNode, saved-
Nodes[maxlevel]: // 指向 Node 的指针;
```

执行插入的伪代码如下:

```
Boolean Skip_List_Insert(level, prior, value)
{ // 获得时间值
S1 <- TraverseTimeStamps(); >
// 创建新节点并赋值
S2 newNode = CreateNode(level, key, value);
S3 newNode.value = value;
S4 While (true) {
// 如果加入节点已经被标记删除,则删除
If (IS_MARKED(newNode, mark))
{ newNode = HelpDelete(newNode, i);
// 获得需要删除的结点
RELEASE_NOTE(newNode);
// 释放该结点 Return false ; Break; }
else { // 保存该节点各级别头指针
S5 for (i = maxlevel - 1; i >= 0; i--)
```

```

if(i≤level) savedNode[i]=COPY_NODE(head);
//插入到每层节点处
S6 for(i=0;i<level-1;i++){
//找出插入点之后的结点 node2
S7 node2=find_last(&.head,i,prior);
S8 while(IS_MARKED(node2.mark))
{ newNode=HelpDelete(node2,i);
//获得需要删除的结点
RELEASE_NODE(newNode);
//释放该结点
node2=find_last(&.head,i,prior);
//再次查找 node2 结点}
//找出插入点之前的节点 node1
S9 node1=find_prv(&.head,i,prior);
//如果该节点被标记为删除节点
S10 while(IS_MARKED(node1.mark))
{ newNode=HelpDelete(node1,i);
RELEASE_NODE(newNode);
node1=find_prv(&.head,i,prior);}
//将新节点插入
S11 newNode.next[i]=node2;
If(CAS(&.node1.next[i],node2,newNode)
//如果找到该结点为 node2 结点
Break;}} RELEASE_NODE(node1);
RELEASE_NODE(node2);
RELEASE_NODE(newNode);
Break;}
S12 if(IS_MARKED(newNode.mark))
{ newNode=HelpDelete(newNode,i);
RELEASE_NODE(newNode);
Return false ;Return true;}}

```

执行删除操作的伪代码如下:

```

Pointer to Node HelpDelete(node:point to Node, level;integer ){ //标记该节点
S13 for(i=0;i<node.level;i++){
{do{node2=node.next[i];
If(not node2) break;}
while( IS_MARKED(node2) ||
CAS(&.node.next[i],node,GET_MARKED(node2)));
//针对每一个级别找到前一个节点
S14 for(i=0;i<level;i++){
{node2=find_prv(&.head,i,node.prior);
prev[i]=node2;
RELEASE_NODE(node2);}
S15 //查找并删除、链接
While(true)
{If node.next[level]=1 break;
Last=find(&.head,i,node.prior);
RELEASE_NODE(last);
if(CAS (&.prev[i].next[level],node,GET_UN-
MARKED(node.next[level]))
{node.next[level]=1;
break;}
If node.next[level]=1 break;}}
RELEASE_NODE(node);
return prev;}
//CAS 原子操作的实现

```

```

Boolean CAS(address:pointer to word, oldvalue;word,
newvalue;word){
S16 Atomic do
if * address=oldvalue
{ * address=newvalue;return true;}
else return false;}

```

定理 1 对 Skip-list 的 Lock-Free 算法实现是具有线性一致性的。

证明上述对 Skip-list 的 Lock-Free 算法主要体现了两种类型的并行操作:Insert 和 HelpDelete. 下面通过并行线程来论证其线性一致性。

(1)第一种类假设存在两个进程 p,q 同时分别执行这两类操作. 插入操作主要存在两种结果:true 和 false; 删除操作存在三种可能性:marked、unmarked 和 deleted,按照顺序执行过程主要分为如下执行的 12 种 history 情况:

- ① $H = \{inv(q, Delete, Node2), ret(q, marked), inv(p, insert, Node1), ret(p, false)\}$
- ② $H = \{inv(q, Delete, Node2), ret(q, unmarked), inv(p, insert, Node1), ret(p, false)\}$
- ③ $H = \{inv(q, Delete, Node2), ret(q, deleted), inv(p, insert, Node1), ret(p, false)\}$
- ④ $H = \{inv(q, Delete, Node2), ret(q, marked), inv(p, insert, Node1), ret(p, true)\}$
- ⑤ $H = \{inv(q, Delete, Node2), ret(q, unmarked), inv(p, insert, Node1), ret(p, true)\}$
- ⑥ $H = \{inv(q, Delete, Node2), ret(q, deleted), inv(p, insert, Node1), ret(p, true)\}$
- ⑦ $H = \{inv(p, Insert, Node1), ret(p, false), inv(q, Delete, Node2), ret(q, unmarked)\}$
- ⑧ $H = \{inv(p, Insert, Node1), ret(p, false), inv(q, Delete, Node2), ret(q, marked)\}$
- ⑨ $H = \{inv(p, Insert, Node1), ret(p, false), inv(q, Delete, Node2), ret(q, deleted)\}$
- ⑩ $H = \{inv(p, Insert, Node1), ret(p, true), inv(q, Delete, Node2), ret(q, unmarked)\}$
- ⑪ $H = \{inv(p, Insert, Node1), ret(p, true), inv(q, Delete, Node2), ret(q, marked)\}$
- ⑫ $H = \{inv(p, Insert, Node1), ret(p, true), inv(q, Delete, Node2), ret(q, deleted)\}$

而对于并行执行的过程主要有如下情况。

如果 $node1 \neq node2$, 且 $node1$ 与 $node2$ 相互独立,没有任何关系,则以上 12 种情况分别可以代表 p,q 两个进程之间执行的先后顺序和结果. 且是一一对应. 即使 pq 两个进程同时结束结果也都在 12

种情况之内.

如果 node1≠node2 且 node1 是在 node2 之前插入的节点,如果 q 先执行且标记节点还未删除,那么 node2 就被标记, p 进程在 S8 处会进行删除 node2,然后再插入 node1. 则对应于①④的顺序过程. 如果 q 已经将节点删除,那么 p 就可直接进行 S11 插入 node1. 对应于③⑥. 如果 q 还没有将节点进行标记, p 已经插入节点 node1,无论其返回 true 还是 false, q 都会通过 S15 可以将 node2 删除. 对应串行顺序为⑩⑦⑤②.

如果 node1=node2,如果 p 执行过程中如果在 S4 之前, q 已经标记了该节点那么 p 就会删除该节点而返回 false. p 执行过程中如果在 S4 之后, q 才标记了该节点,那么 p 就会在 S12 处删除该节点返回也是 false. 这两种情况结果一致. 对应①②. 如果 q 在 p 之后完成,如果 p 插入成功, q 就会删除该节点. 如果 p 插入不成功,则 q 就会无法找到该节点,通过 S13 中的判断语句跳出循环. 对应于⑦.

(2)第二种类型假设所有线程同时执行插入/删除操作. 如果是插入/删除同一个节点那么由于 CAS 是一种原子类操作,只有其中一个执行成功,其他的进程则会进入失败状态. 因此结果也只有一个成功是串行的. 如果是插入/删除不同节点,不同级别是互不干扰的,结果都可以保证线性安全. 如果是相同级别的, CAS 中 S16 对后面的链接节点先进行判断,然后才修改值. 如果后面的值被修改了,则会执行失败,会重新再利用 S4 循环重新执行,因此也是顺序的过程.

由此看来,并行执行该算法过程中每种情况都可以对应于串行情况中的执行顺序,结果是安全的. 因此,该算法是线性一致性的.

4 仿真实验及结果分析

4.1 仿真实验平台

测试环境是在一个 Intel (R)Core(TM)2 Quad Q8400 多核平台上. 该平台是一个 32 位包含 4 个 2.66 GHz 的核,每个核有 128 kB 的 Cache 和一个共享的 4 MB L2 级 Cache. 系统配置有 4 GB 芯片的主存.

4.2 仿真结果分析

将本文所提出的 Skip-list 的 Lock-Free 同步方法和 Linux 系统中基于传统锁机制的位图结构以及基于文献[4]的细粒度锁机制的 Skip-list 进行仿真实验比较. 其中位图的纵向表示为关键级别,横向表

示所要调度的任务. 随机产生 1 000 个任务集,每个任务集中产生{10~50}个不等的混合关键任务,设定系统最大关键级别为 10,每个任务的关键级别在{1~10}的范围内随机取整,任务在不同关键级别下的调度顺序保持一致. 在整个仿真实验过程中,为了描述算法之间的性能差异,采用多次模拟求平均值的方法按照在某段时间内的吞吐率、核总利用率以及平均响应时间三个方面进行性能分析,如图 6 和表 1 所示.

从图 6 和表 1 中可以看出,在三者当中,无论是四个核的吞吐率,还是核利用率以及对任务的平均响应时间,采用 Lock-Free 的 Skip-list 数据结构有较好的效果.

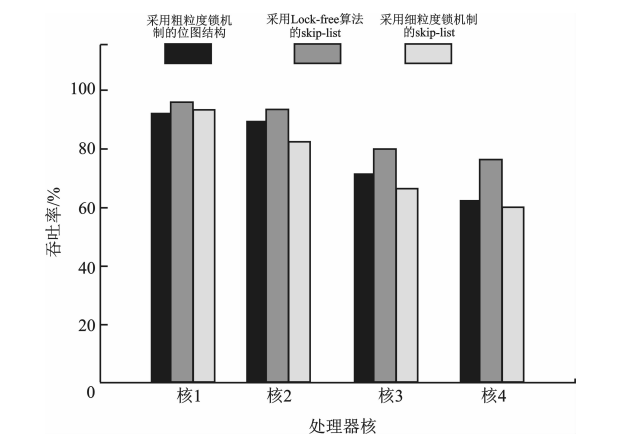


图 6 三种同步机制下不同核上的吞吐率

表 1 三种机制的核总利用率和响应时间值对比表

核	采用 Lock-Free 的 Skip-list 数据结构		采用粗粒度锁机制的位图结构		采用细粒度锁机制的 Skip-list 数据结构	
	核总利用率	平均响应时间/ms	核总利用率	平均响应时间/ms	核总利用率	平均响应时间/ms
1	0.921	1.215	0.891	1.871	0.910	1.961
2	0.937	1.121	0.910	1.543	0.916	1.832
3	0.903	1.0344	0.893	1.731	0.899	1.829
4	0.913	1.333	0.845	1.962	0.892	2.084

5 结束语

本文分析了基于传统锁机制的共享优先级队列面临的问题,给出了 Skip-list 数据结构 Lock-Free 算法的实现方法以及线性一致性方面的证明,最后与采用粗粒度锁机制的位图数据结构和采用细粒度的 Skip-list 结构进行了仿真实验比较. 结果表明,采用 Lock-Free 算法的 Skip-list 方法在平均响应时间和核利用率方面都较优.

参考文献:

- [1] Sundell Håkan, Tsigas Philippos. Fast and lock-free concurrent priority queues for multi-thread systems [J]. Journal of Parallel and Distributed Computing, 2005, 65(1): 609-627.
- [2] Kristijan Dragic, Daniel Bauer. A survey of concurrent priority queue algorithms [C] // Proceedings of the International Symposium on Parallel and Distributed Processing (ISPDP), Switzerland: IEEE, 2008: 1-6.
- [3] Aleksandar Prokopec, Dmitry Petrashko, Martin Odersky. Efficient lock-free work-stealing Iterators for data-parallel collections [C] // Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Lausanne, Switzerland: IEEE, 2015: 248-252.
- [4] Nir Shavit, Itay Lotan. Skiplist-based concurrent priority queues [C] // Proceedings of the 14th International Parallel and Distributed Processing Symposium, Washington, DC, USA: IEEE, 2000: 263-268.
- [5] Sirajo Abdullahi Bakura, Aminu Mohammed. Lock-free hybrid concurrency control strategy for mobile environment [J]. Proceedings of the 6th International Conference on Adaptive Science & Technology (ICAST), 2014, 66(5): 1-5.
- [6] Pooja Aggarwal, Smruti R Sarangi. Lock-free and wait-free slot scheduling algorithms [C] // Proceedings of the 27th International Parallel and Distributed Processing Symposium (IPDPS), Washington, DC, USA: ACM, 2013: 961-972.
- [7] Nhan Nguyen, Philippos Tsigas. Lock-Free cuckoo hashing [C] // Proceedings of the 34th International Conference on Distributed Computing Systems (ICDCS), Madrid, Spain: IEEE, 2014: 627-636.
- [8] Youngjoong Cho, Dongwoo Lee, Hyungkook Jun, et al. Lock-free memory allocator without garbage collection on multicore embedded devices [C] // Proceedings of the International Conference on Consumer Electronics (ICCE), Istanbul, Turkey: IEEE, 2014: 428-429.
- [9] Deli Zhang, Damian Dechev. A lock-free priority queue design based on multi-dimensional linked lists [J]. Journal of IEEE Transactions on Parallel and Distributed Systems, 2015, 27(3): 1-14.
- [10] Bonnicksen L F, Karlsson S, Probst C W. ELB-trees an efficient and lock-free B-tree derivative [C] // Proceedings of the 6th IEEE International Workshop on Multi-/Many-core Computing Systems (MuCoCoS) [s.l.], IEEE, 2013: 1-10.
- [11] Gerhard Schellhorn, Simon Baumler. Formal verification of lock-free algorithms [C] // Proceedings of the 9th International Conference on Application of Concurrency to System Design (ACSD), Washington, DC, USA: ACM, 2009: 13-18.
- [12] Herlihy M P, Wing J M. Linerizability: A correctness condition for concurrent objects [J]. Journal of ACM Transactions on Programming Languages and Systems, 1990, 12(1): 463-492.
- [13] Amos Israeli, Lihu Rappoport. Efficient wait-free implementation of a concurrent priority queue [C] // Proceedings of the 7th International Workshop Distributed Algorithms (WDAG), Lausanne, Switzerland: IEEE, 1993: 1-17.

作者简介:

黄姝娟 女, (1975-), 博士, 讲师. 研究方向为嵌入式与分布式计算. E-mail: 349242386@qq.com.

(上接第 71 页)

- [3] Schreier R. An empirical study of high-order single-bit delta-sigma modulators [J]. Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on, 1993, 40(8): 461-466.
- [4] 李宏星, 冯晖, 林争辉. 高阶 Sigma-delta 调制器传递函数设计方法 [J]. 微电子学与计算机, 2004, 21(6): 147-1503.
- [5] Norsworthy S R, Schreier R, Temes G C. Delta-sigma data converters: theory, design, and simulation [M]. New York, IEEE press, 1997.
- [6] Yu J. Design and analysis of fixed and adaptive sigma-delta modulators [D]. King's College London: University of London, 1992.
- [7] van Roermund A, Hoekstra J. Design philosophy for nanoelectronic systems, from sets to neural nets [J]. International journal of circuit theory and applications, 2000, 28(6): 563-584.
- [8] 曹桂平. 高精度 Sigma-Delta 调制器研究及 ASIC 实现 [D]. 合肥: 中国科学技术大学, 2012.

作者简介:

刘富锐 男, (1992-), 硕士生, 研究方向为传感器技术与信号处理. E-mail: royle035@163.com.