

基于多线程监控器的运行时验证

陈 韬 ,王明明

(南京航空航天大学 计算机科学与技术学院 江苏 南京 211106)

摘 要: 运行时验证是一种轻量级的新型自动化验证技术。运用了该技术的验证软件由两部分组成:一部分是被监控的目标程序;另一部分是监控器。对于基于形式化语言的运行时验证方法主要思想就是输入表示描述事件和性质的形式化规约语法,目标程序。输出插桩好的新程序。插桩好的新程序在遇到需要监控的切点时,就会执行相应的函数去判断是否满足形式化规约语法。然而传统的单线程运行时验证监控器在目标程序需要监控的规约性质比较多的时候,重新生成的程序可能会因为要验证比较多的规约性质,造成程序的性能变慢。文中利用多核并行技术,对原型工具 Movec 进行优化。通过使用串行程序中多个监控器分配到多线程的方法,Clang 编译器的插桩技术和多核任务分配方法,实现了 Movec 原型工具的优化。并将优化之后的 Movec 与没有改进之前的进行实验数据对比,实验结果表明采用多线程的运行方法具有很好的效果。

关键词: 运行时验证;多线程;源代码插桩;编程语言

中图分类号: TP316.2

文献标识码: A

文章编号: 1673-629X(2019)02-0029-06

doi: 10.3969/j.issn.1673-629X.2019.02.006

Runtime Verification Based on Multi-thread Monitor

CHEN Tao ,WANG Ming-ming

(School of Computer Science and Technology ,Nanjing University of Aeronautics and Astronautics ,
Nanjing 211106 ,China)

Abstract: Runtime verification is a new lightweight automatic verification technique. The verification software used in this technology is composed of two parts: one part is the monitored target program; the other is the monitor. The main idea of the runtime verification method based on formalized language is to input a formal specification syntax that represents events and properties and target program, and output a new program. The new program with inserting pile will execute the corresponding function to judge whether it satisfies the formal specification grammar when it meets the point that needs to be monitored. However, when the traditional single thread runtime verification monitor has many properties that the target program needs to monitor, the regenerated program may slow down the performance of the program because of the more specified nature. In this paper, we optimize the prototype tool Movec by using multi-core parallel technology. Through the way of serial program monitor assigned to multithreading, Clang compiler's piling technology and multi-core task allocation method have realized the optimization of Movec prototype tools. The optimized Movec is compared with the experimental data without the improved tools, which shows that the multithread operation method has a better effect.

Key words: runtime verification; multi-thread; source code piling; programming language

1 概 述

软件运行时的验证技术^[1-3],是对其他验证技术例如模型检测和软件测试的一种补充,是一种轻量级的新型自动化验证技术。由于运行时验证技术是在程序运行过程中进行验证,所以它能保证验证程序的实时性。基于形式化语言的程序运行时验证的基本思想是输入表示描述事件和性质的形式化规约语法,目标

监控 C 程序,输出插桩好监控器的新 C 程序。并采用该方法实现了原型工具 Movec^[4]。该工具的基本框架主要由解析器、监控器、插桩器构成。

其中形式化规约主要由以下几部分组成:

- (1) 形式化规约的头部分,包含了修饰符、名字以及名字所带的参数;
- (2) 规约文件变量的声明、定义,变量可由用户自

收稿日期: 2018-03-21

修回日期: 2018-07-24

网络出版时间: 2018-11-15

基金项目: 江苏省普通高校研究生科研创新计划项目(SJZZ16_0062)

作者简介: 陈 韬(1993-),男,硕士研究生,研究方向为软件运行时验证。

网络出版地址: <http://kns.cnki.net/kcms/detail/61.1450.TP.20181115.1051.100.html>

已操作;

(3) 切点声明,采用了面向方面^[5]编程思想,切点指函数调用、变量的赋值、函数执行、变量设置等;

(4) 事件声明,即匹配的连接点之前或者之后进行额外的操作,也可替换连接点的执行;

(5) 性质,用形式化语言描述的规约性质;

(6) 性质触发器,当满足或者违反性质时需要进行什么样的操作。

该原型工具首先对形式化规约文件进行解析,然后构造出监控器。该工具支持三种逻辑形式的性质描述:自动机^[6-7]、正则表达式^[8-9]和线性时序逻辑^[10]。监控器的监控算法是通过自动机来进行操作的。最后对源代码进行插桩的插桩器,主要结合形式化规约文件和 Clang 编译器,对 C 程序的源代码中对应的规约文件切点进行插桩,最后生成包含监控器的新 C 程序。文中在该原型工具的基础之上引入多线程技术,实现该原型工具的优化。主要工作如下:提出多监控器运行时验证并行化原理;使用插桩技术对多线程监控器进行划分;使用多线程交互的方法进行交互;实现了原型工具的优化,并进行了对比实验。

2 多监控器运行时验证并行化原理

本节是基于形式化语言的程序运行时验证的优化,将多线程的技术应用到原型工具 Movec,提高工具的性能。原型工具对于多监控器的处理是通过将监控

器的监控事件和响应操作直接插桩到源代码,目标程序运行到触发事件时,调用响应操作,操作结束之后继续执行目标程序。假设目标程序运行时间为 $main_time$, 监控器 1, 2, ..., n 运行的时间是 $m_time[1], m_time[2], \dots, m_time[n]$ 。那么插桩之后新的程序运行的时间为 sum_time :

$$sum_time = \sum_{i=0}^{i=n} m_time[i] + main_time \quad (1)$$

为了实现多线程程序的并行化,本节提出了运行时验证多监控器下的并行方法。具体过程如下:第一步,将监控器的响应操作与目标程序分离。将原来插桩到源代码的监控器验证函数分离到创建的线程中。第二步,将目标程序要验证的事件信息进行记录。将监控器需要验证的事件信息存储到指定的数据结构。第三步,进行事件序列的任务划分。解决多线程负载均衡问题,防止出现一个处理器一直处于运行状态,其他处理器空闲的情况。第四步,将各个监控器与线程进行绑定。通过设置线程亲和性,将监控器和具体的逻辑线程 ID 进行绑定,防止出现操作系统随机调用逻辑线程的情况。第五步,多线程监控器程序和目标程序并行运行。

采用以上并行方法并结合 Clang 编译器^[11]对原型工具 Movec 进行改进。改进之后的 Movec 对目标程序处理之后生成新程序的运行过程由单线程的串行执行变成了并行执行,如图 1 所示。

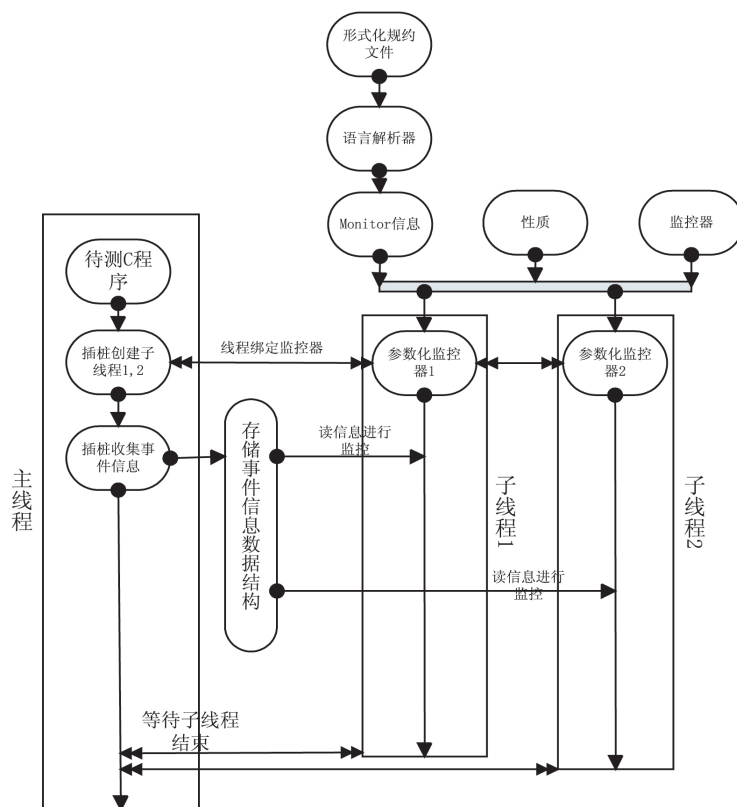


图 1 多监控器工具框架和对应的程序状态

3 运行时验证的多线程运行

3.1 多线程监控器的任务划分

在监控器分配到各个线程并行运行的条件下,需要采用合理的方法去解决线程之间的负载平衡问题。首先介绍一下负载平衡问题对多线程程序的重要性。在多核^[12-14]的并行机上,由于在多线程程序中,整个程序的运行时间往往取决于运行时间最长的线程。如果操作系统在一个处理器分配的任务数量过多,而其

他处理器空闲,这就会导致多线程程序执行性能的严重下降。

对于原型工具 Movec,监控器直接通过插桩的方式直接插桩到待检测程序,文中采用多线程技术将监控器和待检测程序分离,并将多监控器分配到多处理器上实现并行运行。但是在并行机上处理器个数有限的情况下,有以下两种情况,如图2所示。

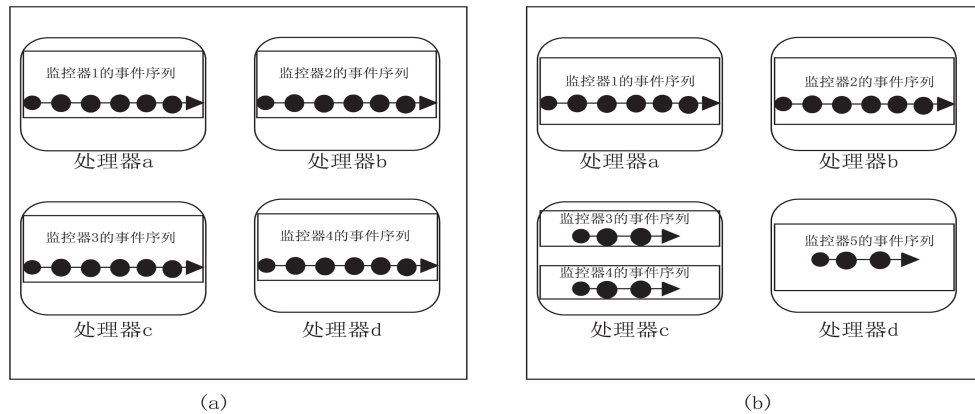


图2 多监控器事件序列分配

情况(a): 处理器个数大于等于监控器个数,这时为每个监控器分配一个独立的线程进行处理。由于监控器进行程序处理时,对程序中事件序列是按顺序验证的,所以分配的线程中验证的事件数量会直接影响整个程序的运行时间。情况(b): 处理器个数小于监控器个数,这时一个处理器就需要对多个监控器要监控的事件序列进行处理,这就需要进行负载平衡处理。而且监控器生成之后,对于事件序列的数量并不清楚,这就进一步加大了负载平衡的难度。对此基于 Clang 编译器对程序进行代码插桩时,通过遍历抽象语法树对事件序列的数量进行统计,然后进行监控器的线程绑定。这样就可以将两个较大数量的事件序列分开,将数量较少的事件序列合并到一个处理器,提高线程并行运行的性能。结合任务分配技术和式1对插桩之后的程序运行时间进行分析,使用图2的任务分配情况。首先由于监控器被分配到各个处理器上并行运行,监控器所有时间由原来的 $\sum_{i=0}^{i=n} m_time[i]$ 变成了 $\text{Max}\{corea_time, coreb_time, corec_time, cored_time\}$, 然后分两种情况:

情况一: 目标程序运行时间 $\text{main_time} \leq \text{Max}\{corea_time, coreb_time, corec_time, cored_time\}$, 总运行时间为:

$$\text{sum_time} = \text{Max}\{corea_time, coreb_time, corec_time, cored_time\} \quad (2)$$

情况二: 目标程序运行时间 $\text{main_time} > \text{Max}\{corea_time, coreb_time, corec_time, cored_time\}$, 总运行时

间为:

$$\text{sun_time} = \text{main_time} \quad (3)$$

3.2 多线程监控器与待检测程序的线程交互

在完成了多线程的任务分配问题之后,多线程还涉及到的问题就是资源的共享问题以及线程间通信。下面将对以上两个问题进行具体的分析和解决。

线程资源共享问题指的是多个线程对要访问的资源只保存一份,多个线程对该资源进行访问,是一种并发进行的同步机制。文中流程如下:首先目标程序将程序中的事件序列存储在一个链表之中,该链表就是一个共享的资源;然后多个监控器从链表之中读取相应的数据;最后每个监控器并行运行,处理各自对应的事件序列。从上可知这是一个单生产者多消费者的多线程问题。其中目标程序为生产者线程进行数据收集并存储到链表(即数据生产),监控器则是作为消费者线程进行数据处理(即数据消费)。事件序列数据通过线程间共享内存块进行存储。其中线程间共享的内存块主要由以下几种:堆、全局变量、静态变量、文件、栈、寄存器等。文中采用的是堆内存进行共享。但是单生产者多消费者模型会带来这么一个问题,如果生产者存放数据的速度远远大于消费者消耗数据的速度,就可能引起消费者处理数据之前生产者多次重写了之前的结果,反之,消费者线程消耗数据的速度远远大于生产者生产数据的速度,就会造成消费者提取还没有存放的数据,或者是造成同时读写的竞争问题。

文中通过设计自旋锁和链表解决了线程共享问

题。下面的消费者线程对应的是监控器所在的线程,生产者线程对应的是目标程序线程。主要思想是:让消费者线程访问的节点不要超过生产者存放的节点。生产者程序初始化链表时,将 producer_head 和 producer_tail 同时指向头节点并将监控器的头节点也指向链表的头,新的节点是在 producer_tail 节点之后进行插入,插入成功之后 producer_tail 节点后移,并且在链表的最后插入一个空节点,保证最后一个数据节点被消费者访问。消费者线程则是通过消费者对应的头节点进行链表访问。因为访问只是读取数据不涉及资源竞争,所以可以并行运行。但是为了防止消费者程序读取还没有插入好的新节点,设置了一个自旋锁。通过一个 while 语句的循环,循环的判断条件是消费者的 head 节点是否等于生产者的 tail 节点,如果相同就需要将消费者线程阻塞,从而防止消费者访问没存储的数据。这个自旋锁有效保证了多个消费者线程在访问共享链表时,只会访问生产线程 tail 节点之前的链表节点,保证了消费者并行的正确性。

由于消费线程在访问共享资源时涉及到了阻塞的情况,这就需要对它们进行唤醒操作。文中使用信号量的方式来唤醒,这就是线程间通信问题。线程间通信指的是主线程和子线程、子线程与子线程进行相互之间的通信。线程间必须有个特定的信息传递渠道,文中使用共享的链表作为信息传输渠道,并采用主线程去唤醒阻塞线程的方法。具体步骤如下:首先存储事件信息;然后唤醒所有子线程,由于消费者线程中唤醒之后,线程会从上上次阻塞的位置之后开始运行,也就是 wait() 方法后开始,所以线程会再次进行自旋锁的判断,从而保证了消费者线程访问数据的安全性;最后如果自旋锁的条件不满足,消费者线程就可以进行数据的读取和访问,否则消费者线程会被再次阻塞。

4 多核监控器的运行时验证工具实现

4.1 数据结构设计

共享资源链表结构如图 3 所示。

其中 action 为目标动作,即监控器的响应操作;trace_node 类型为链表的节点信息;通过 action_id 判断当前目标动作是属于哪一个监控器;action 联合类型可以达到特定的目标动作对应特定的类型存储,其中的特定存储类型由目标动作函数决定,如 action_1 类型, pfunc 为函数指针, join_poin 类型记录了切点的上下文,其余是函数指针剩下的参数。

4.2 插桩实现

文中的工具采用的是面向方面编程的思想。其中面向方面的程序包括两部分,一是实现软件系统核心关注点的基础代码,二是软件系统横切关注点的代码。

面向方面编程由接入点模型、切入点、通知和方面组成。接入点指的是程序结构中特定位置或运行过程中的特定时刻;切点指的是符合条件的连接点集合;通知指在切入点描述的位置所做的响应动作;方面则是由若干个关注点集合构成,其中的关注点是由切入点和通知共同组成。

```
typedef struct {
    void( * pfunc) ( struct join_point * tjp ,size_t size ,void *
address) ;
    struct join_point tjp;
    size_t size;
    void * address;
} action_1;
typedef union {
    action_1 a1; action_2 a2;
    action_3 a3; action_4 a4;
} action;
struct trace_node{
    int property_id;
    int action_id;
    action act;
    struct trace_node * next;
};
```

图 3 共享资源链表结构

主要思想是第一步:进行切点的匹配,重写切点操作,记录响应操作,插桩新的切点操作函数;第二步:源代码插桩,创建新的线程,读取记录的信息并执行响应操作。本节详细介绍这两步的具体实现方法。

第一步实现。在待监控目标程序中,切入点可能是函数调用、函数指针调用、函数的执行、变量的赋值、变量的取值、函数执行流中返回值变量命名等。为了将切点进行重写以及对响应操作进行记录,通过遍历抽象语法树,实现切点的具体插桩动作。表 1 为不同类型的切点对应的抽象语法树类型。

表 1 切点类型对应的抽象语法结构

切点类型	抽象语法树的类型
函数定义体内	FunctionDecl
类型定义体内	TagDecl
在文件内	TranslationUnitDecl
函数调用	CallExpr
函数指针调用	CallExpr
函数执行	FunctionDecl
变量赋值	VarDecl、BinaryOperator 等
变量取值	BinaryOperator、DeclRefExpr 等

下面以函数调用切点为例介绍插桩的具体方法。由于 Clang 编译器访问抽象语法树中的 CallExpr 类型

节点是通过 RecursiveASTVisitor 的 VisitCallExpr 方法, 所以通过重写 VisitCallExpr 方法完成插桩。具体算法如图 4 所示。

切入点函数调用插桩算法
输入: 抽象语法树中的 CallExpr 类型节点 输出: 插桩函数后的新程序
<pre> 1: for(遍历所有的监控器) { 2: for(遍历监控器的里切点) { 3: if(函数调用切入点匹配成功) { 4: 获取切入点调用函数声明; 5: 获取切入点调用函数返回值类型; 6: 获取存储语法树节点代码片段的存储器(存储新函数的定义); 7: 将原来的切入点函数调用替换成新函数调用; 8: 构造新函数{ 9: 新函数命名, 函数参数构造; 10: 函数体内的初始化(记录切点函数的上下文信息, 参数和参数类型); 11: 获取监控器中的响应操作条件; 12: 记录响应操作的函数名和函数参数到图 3 的共享资源链表; 13: 调用原来的切入点函数; 14: } 15: } 16: } 17: }</pre>

图 4 切入点函数调用插桩算法

通过这样一个插桩算法可以对函数调用的切点进行处理。对于其他类型的切点, 文中采用类似的插桩方法分别对 RecursiveASTVisitor 的 VisitFunctionDecl 方法、VisitUnaryOperator 方法、VisitCompoundAssignOperator 方法、VisitVarDecl 方法、VisitBinaryOperator 方法、VisitDeclRefExpr 方法进行重写, 完成了切点信息记录、响应函数与目标程序的分离。

第二步实现。要在目标程序中加入线程创建, 线程相关的变量定义声明以及线程创建调用的执行函数。其中线程创建和相关变量定义声明通过 VisitFunctionDecl 方法。

对创建线程执行的函数, 将调用线程调用函数的定义插桩到 main 函数声明的位置之前, 函数定义的内部由以下几部分组成:

- (1) 3.2 节中介绍的自旋锁;
- (2) 根据 3.1 节中的任务划分方式进行监控器事件序列的划分;
- (3) 通过读取共享资源中节点的 action_id 来判断是否进行监控函数的调用, 如果是指定的 action_id 则监控函数执行, 否则节点后移;
- (4) 进行访问节点指针和空指针比较, 如果为空, 结束线程。

5 实验分析

文中将多线程技术应用到多监控器与单目标程序的运行时监控模式中, 从而将原本串行运行的程序转化成并行运行的多线程程序。实验对象是单目标程序与 6 个性质的监控器。6 个监控器的事件序列个数分别是: 40003、40003、40003、4000、40003、20002。监控的性质分别是用正则表达式规约的函数执行顺序 (ere: f1+f2+f1f2f3)、用正则表达式规约的内存申请释放 (ere: (malloc free) * malloc end); 六个性质都是用正则表达式进行规约, 其中 5 个性质是用来规约函数调用的顺序, 第 6 个性质是用来判断是否存在目标程序、是否存在内存没有释放。切点分别是 malloc 函数调用、free 函数调用、main 函数执行之后以及关于被规约执行顺序的函数调用。实验结果如图 5 所示。图中记录的时间值都是进行 10 次统计后取平均值的方式获取的。

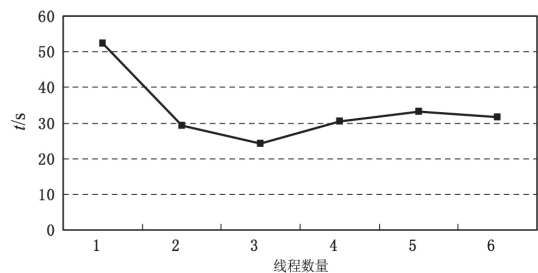


图 5 优化实验数据

图中横坐标分别表示使用了几个线程去验证 6 个事件序列。如果使用一个线程去运行所有事件序列的监控函数就是一种串行结构, 所以使用的时间最长。但是随着线程个数的增加, 时间并不是一个递减的趋势, 结果和预期的不同。这是因为该实验的硬件平台是四个处理器, 但是支持超线程模式, 一个处理器可以同时运行两个线程。当超过 4 个线程时, 就会出现两个线程竞争同一个处理器资源的情况。同一时刻一个处理器也只是在运行两个线程中的一个。所以当同时运行 4 个线程时, 才是真正意义上的并行运算。从图 5 可知, 在 3 个线程运行监控器的事件序列与一个主线程的情况下, 运行的时间最短。在最优情况下, 对于该实验的数据进行分析。插桩后程序运行的总时间由原来的 51.51 s 变为 23.53 s。程序的性能提升了 45%。通过将工具优化之前的串行执行时间数据和优化之后的并行运行时间数据进行对比, 得出采用了多线程技术后的工具性能提升明显。优化之后的原型工具 Movec 在进行多个监控器监控时, 插桩之后的程序性能的提升效果显著。

6 结束语

本文利用多核并行技术^[15-17], 对原型工具 Movec

进行优化。通过将串行程序的监控器分配到多线程的方式,Clang 编译器的插桩技术和多核任务分配方法,实现了 Movec 原型工具的优化。首先介绍了多监控器与单目标程序情况下,将插桩之后程序并行化的原理。然后主要介绍了多线程程序中负载均衡处理方法和线程间线程通信的方式。

最后,通过具体的插桩算法实现了原型工具针对多监控器与单目标程序的并行化,实现了插桩之后生成新程序的并行化运行方法。并将优化之后的 Movec 与没有改进之前的工具进行实验数据对比,实验结果表明采用并行化的运行方法具有很好的效果。今后进一步的工作包括下面几个方面:

(1) 结合静态分析技术,减少 C 程序中不必要的一些插桩。

(2) 在原有的数据结构存储结构上进行优化,提高数据查找的速度。

(3) 将源代码的插桩过程分配到不同的处理器核心上,提高 Movec 工具编译运行的性能。

参考文献:

- [1] CHEN Zhe ,WANG Zhemin ,ZHU Yunlong ,et al.Parametric runtime verification of c programs[C]//International conference on tools and algorithms for the construction and analysis of systems.Berlin: Springer ,2016: 299-315.
- [2] MEREDITH P O N ,JIN D ,GRIFFITH D ,et al.An overview of the MOP runtime verification framework [J].International Journal on Software Tools for Technology Transfer ,2012 ,14 (3) : 249-289.
- [3] LEUCKER M ,SCHALLHART C.A brief account of runtime verification [J].The Journal of Logic and Algebraic Programming ,2009 ,78(5) : 293-303.
- [4] 王哲民 ,陈 哲 ,朱云龙 ,等.参数化运行时验证研究和工具实现 [J].小型微型计算机系统 ,2016 ,37(12) : 2667-2672.
- [5] LATTNER C ,ADVE V.LLVM: a compilation framework for lifelong program analysis & transformation [C]//International symposium on code generation and optimization. [s.l.]: IEEE ,2004: 75-86.
- [6] HOPCROFT J E ,ULLMAN J D.Introduction to automata theory ,languages ,and computation [M].2nd ed. [s.l.]: [s.n.] 2001.
- [7] SHIELDS M W.An introduction to automata theory [M].[s.l.]: Blackwell Scientific Publications Ltd. ,1987.
- [8] ZHANG X ,LEUCKER M ,DONG W.Runtime verification with predictive semantics [M]//NASA formal methods.Berlin: Springer ,2012: 418-432.
- [9] BAUER A ,LEUCKER M ,SCHALLHART C.Runtime verification for LTL and TLTL [J].ACM Transactions on Software Engineering and Methodology ,2011 ,20(4) : 14.
- [10] LEVINE J.Flex & Bison: text processing tools [M]. [s.l.]: [s.n.] 2009.
- [11] MEREDITH P O ,JIN D ,CHEN F ,et al.Efficient monitoring of parametric context-free patterns [J].Automated Software Engineering ,2010 ,17(2) : 149-180.
- [12] LUK C K ,HONG S ,KIM H.Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping [C]//Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture. New York ,NY ,USA: ACM ,2009: 45-55.
- [13] GIACOMONI J ,MOSELEY T ,VACHHARAJANI M.Fast-forward for efficient pipeline parallelism a cache-optimized concurrent lock-free queue [C]//Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming.Salt Lake City ,UT ,USA: ACM ,2008: 43-52.
- [14] NAVABPOUR S ,BONAKDARPOUR B ,FISCHMEISTER S.Time-triggered runtime verification of component-based multi-core systems [M].Waterloo: University of Waterloo , 2015.
- [15] 张林波.并行计算导论 [M].北京: 清华大学出版社 ,2006.
- [16] 王 锋.面向千万亿次 CPU-GPU 异构系统的编程模型与性能优化关键技术研究 [D].长沙: 国防科技大学 ,2013.
- [17] 张 剑 ,胡 军 ,郭丽娟 ,等.多核处理器架构下面向监控的软件运行时验证方法研究 [J].小型微型计算机系统 , 2012 ,33(1) : 102-109.