Software Engineering and Applications 软件工程与应用, 2014, 3, 86-96
Published Online June 2014 in Hans. http://www.hanspub.org/journal/sea
http://dx.doi.org/10.12677/sea.2014.33011

Hans汉斯

The Design of Dynamic Checking Tool for VXWORKS Systems Concurrent Program

Hao Liang[1], Yunfeng Ai[2], Huairong Shen[3], Yongchao Zhao[4]

[1]Company of Postgraduate Management, The Academy of Equipment, Beijing
[2]College of Engineering & Information Technology, University of Chinese Academy of Sciences, Beijing
[3]Department of Space Equipment, The Academy of Equipment, Beijing
[4]Department of Battle Command and Training, National Defense University, Beijing
Email: mouse1292000@hotmail.com, 85283611@qq.com, shenhuair@tom.com, 314976196@qq.com

Received: Apr. 15th, 2014; revised: May 14th, 2014; accepted: May 22nd, 2014

Abstract

In recent years, with the improving degree of automation of real-time embedded systems, their design complexity continues to increase. Concurrent programming methods were widely used in designing. But due to real-time embedded system interrupts and threads overlap, in the testing and checking process of real-time embedded system, there is always lack of an effective program testing tool. So we designed a testing tool for VXWORKS systems dynamic concurrent programs. We use Labeled Transition Systems as a system of concurrent programming model, and have given formal definition for common concurrency error, and use of partial order reduction algorithm to reduce the state space of the program. Finally, we have realized the testing tool which can detect multi-threaded and multi-interrupt program concurrent errors.

Keywords

Real-Time Embedded Systems, Concurrent Program, Multiple Interrupts, Multithread, Concurrency Errors

VXWORKS系统并发程序动态测试工具设计

梁　昊[1]，艾云峰[2]，沈怀荣[3]，赵永超[4]

[1]装备学院，研究生管理大队，北京
[2]中国科学院大学，工程管理与信息技术学院，北京

[3]装备学院，航天装备系，北京
[4]国防大学，作战与指挥训练教研部，北京
Email: mouse1292000@hotmail.com, 85283611@qq.com, shenhuair@tom.com, 314976196@qq.com

收稿日期：2014年4月15日；修回日期：2014年5月14日；录用日期：2014年5月22日

摘　要

近年来随着实时嵌入式系统自动化程度的不断提升，其设计复杂度不断加大，在设计中大量的使用了并发程序设计方法。但在实时嵌入式系统测试的过程中，由于实时嵌入式系统中中断和线程相互交叠，始终缺乏有效的并发程序测试工具。为此本文设计了针对VXWORKS系统并发程序动态测试工具，提出以标记迁移系统作为并发程序的系统模型，对常见的并发错误给出了形式化定义，使用偏序化简算法缩减程序的状态空间，实现了对多线程、多重中断的并发程序错误检测。

关键词

实时嵌入式系统，并发程序，多重中断，多线程，并发错误

1. 引言

近年来随着控制系统的自动化程度不断提升，对其数据处理能力、通信处理能力、系统集能力都提出的要求也越来越高。因此并发多线程、多重中断程序设计技术在嵌入式实时操作系统中得到了广泛的发展和应用。然而并发程序由于其执行过程中并发交叠的随机性，系统安全性带来了大量的不确定因素。一般情况下，一个由 n 个线程、每个线程有 k 个语句组成的多线程并发程序，多线程之间所有可能交叠总数为 $(nk)!/(k!)^n$ 。同时，操作系统线程调度策略、程序执行环境，如 I/O 状态、处理器状态等，均会直接影响程序的执行序列，最终影响程序执行的结果，即使用同一输入数据执行同一并发程序多次，可能会得到多种不同的结果。

目前国内外针对并发程序的测试工具主要分为：静态分析工具、动态测试工具。

静态分析工具的设计思想是不编译运行程序，而对程序中多线程访问共享资源时的锁集合进行静态检查，如果与不同线程访问同一共享对象相关的两个锁集合的交集为空，则发出警告，提示可能出现数据竞争。如国防科技大学设计的 MIDAC[1]检测工具。MIDAC 采用了函数摘要的技术来缩减静态分析过程中需要遍历的状态空间。文档[2] [3]所采用的类线程测试方法为多重中断的测试提出了新的思路，其主要原理是将中断程序改写为“语义”等价的多线程程序，然后对多线程程序进行静态或者动态测试。

动态验证工具的基本设计思想是对在给定测试数据驱动下执行的程序进行验证，它避免了可执行程序代码(包括编译、运行时库等)与模型之间的不一致问题。其中典型的工具有：

VeriSoft[4]面向可执行程序，能够验证由 C、C++、Tcl 等任何程序编写的可执行代码，它还支持对包含多个进程的并发程序的验证。VeriSoft 采用了无状态的动态模型检验方法。

加州伯克利大学开发了可扩展的主动测试工具框架 CalFuzzer[5]，其可对 Java 程序的数据竞争、原子性违背、死锁进行测试；并可集成不同的动态分析工具，已集成动态分析工具有：Lockset、Atomizer 和 ATOMFUZZER、iGoodlock。已设计了用于数据竞争、原子性违背、死锁检测工具；亦可集成用户定制的线程调度策略。

2009 年美国犹他州大学研发的动态验证框架原型 Inspect[6]，可发现多线程 C 程序中的“并行错误”，并保证在给定测试数据下没有数据竞争、原子性违背这两种并发错误。

航天二院 706 提出的多重中断程序测试框架[7]亦是一种动态测试工具，它的主要设计思想是对中断进行两两对比测试，通过不断提高两个中断的触发频率来加速并发错误的出现。但是该方是只能针对两个中断进行独立测试，且需要人为干预，很难保证测试的覆盖率。

本在参考了国内外相关技术的基础上，设计了面向 VXWORKS 实时操作系统并发程序的动态测试工具，该工具可以测试包括多线程、多重中断在内的并发程序，错误检测范围涵盖了数据竞争、死锁、原子性违背这三种常见的并发错误。本文首先介绍了动态测试的总体构架；之后详细叙述了程序分析器、程序插装器、控制执行器、中断发生器这四个主要组成部分的详细设计方案；最后通过实验数据说明了该测试工具在功能完备的前提下拥有较好的时效性和可靠性。

2. 测试工具总体设计概述

2.1. 动态测试工具设计框架

多重中断程序动态测试工具构架主要包含 4 个部分：程序分析器、程序插装器、控制执行器、中断发生器。其总体设计构架如图 1 所示。

2.2. 动态测试工具执行流程

第一步对源程序使用程序分析器，得到并发程序的状态空间模型；第二步对分析后的程序使用程序插装器对程序中的共享对象、中断服务函数、线程函数进行标记、插装；第三步对插装后的程序进行编译，因此在编译的过程中，必须配合为本测试工具所设计的相应软硬件平台的 C 语言插装库文件，这样就可以到的可被控制执行的目标程序；最后将目标程序烧写入目标机，建立目标机和控制执行器的通信

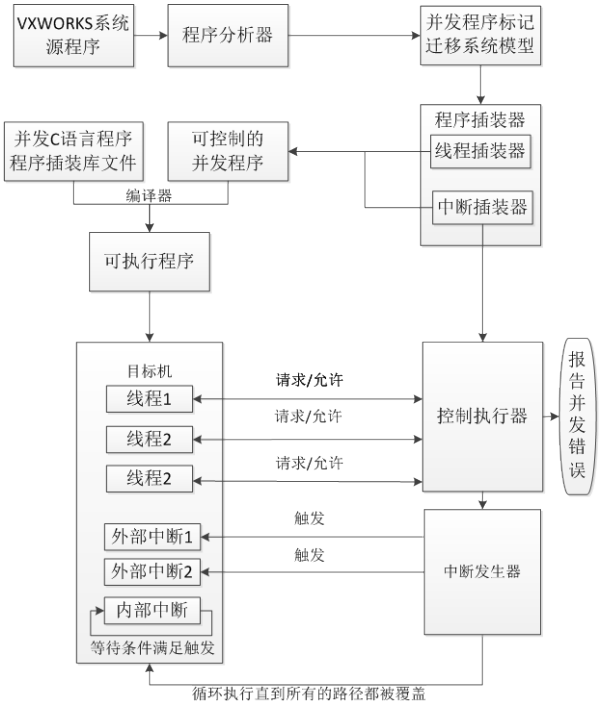

Figure 1. Framework of concurrent program testing tool
图 1. 并发程序动态测试工具设计框架图

链路，启动中断发生器，目标机就进入了实时动态执行阶段，控制执行器根据目标机反馈执行状态和动态测试算法实时地控制中断发生器产生特定的中断，直到所有的并发程序路径被搜索完毕后测试结束，控制执行报告可能存在的并发错误。

3. 动态测试工具详细设计方案

本小节将详细的介绍程序分析器、程序插装器、中断发生器、控制执行器的详细设计方案。

3.1. 程序分析器设计方案

程序分析器的主要作用是构建并发程序的系统模型。程序分析器采用了 CIL(C Intermediate Language) [8]开源工具对被测是程序进行分析，构型被测试程序的标记迁移系统模型。其主要工作可以分为：第一，对并发程序进行分析，标记出程序中的共享对象，得到的共享对象及相关操作，在程序中插入临时变量记录共享对象读写前后的取值，从而抽象出程序状态空间；第二，将程序中的复杂语句转换为：if、else、goto、while、赋值等 5 种简单语句，抽象出程序迁移空间。由此我们可以获得并发程序的标记迁移系统模型。

标记迁移系统(Labeled Transition Systems, LTS)[9]，其模型的具体描述如下：

定义 2.1 LTS(标记迁移系统)是一个四元组 $M=(S,init,T,R)$, 其中 S 表示并发程序的状态集合； $init$ 表示初始状态(也可表示为 s_0)，并且有 $init \in S$ ； T 表示所有迁移集合，并且有 $T \subseteq S \times S$ ； R 表示迁移关系集合，并且有 $R=T \times T$ 。

3.2. 程序插装器设计方案

程序插装器主要完成对共享对象相关操作的插装，对中断服务函数的插装。这里我们使用了 CIL 工具的 API[8]，针对共享对象、线程函数、中断服务函数进行了特殊处理。

共享对象插装主要是标记共享对象，识别对共享对象的相关操作(表 1)。

对中断服务函数的插装主要作用是识别中断服务函数以及相关的中断源；确定中断的性质和中断优先级；对开关中断的函数进行识别，向控制执行发送开关中断的操作(表 2)。

Table 1. Instruments for shared object
表 1. 共享对象插装功能示意表

插装前	插装后	功能说明
//X 为共享对象 X<<Y	Instrument_obj_write(&X)	向控制执行器发送写共享对象操作申请
//Y 为共享变量 X<<Y	Instrument_obj_read(&Y)	向控制执行器发送读共享对象操作申请
//分配共享内存 malloc	malloc()Instrument _obj_reg ()	向控制执行器注册所分配的共享内存区域

Table 2. Instruments for interrupt functions
表 2. 中断服务函数插装示意表

插装前	插装后	功能说明
interrupt	Interrupt Instrument_interrupt_start()… Instrument_interrupt_end() Return;	通知控制执行器中断服务函数开始执行和中断服务函数执行结束
intConnection	Instrument_intConnection	注册中断服务函数与中断向量的关联信息
intLock	Instrument_ intLock	向控制执行器发送禁用制定的中断
intUnlock	Instrument_ intUnlock	向控制执行器发送使能制定的中断

对线程的插装主要作用是识别线程，标记所有线程的相关操作并通知控制执行器(表 3)。

3.3. 中断发生器设计方案

通过分析典型的被测程序，考虑测试工具的通用性，中断信号发生器的包含了大量的常用硬件接口，其硬件设计构架图如图 2 所示。

为了保证中断信号产生的实时性，中断信号发生器采用了 HSSI 高速串行接口与控制执行器进行通信，该接口比常规串行接口拥有更高的速率，比 LAN 接口拥有更短的延时时间。

3.4. 控制执行器设计方案

动态测试的过程中，函数空间注册、状态空间化简、并发错误检测这三个任务主要是由控制执行器

Table 3. Instruments for threads
表 3. 线程函数插装示意表

插装前	插装后	功能说明
create	Instrument_create()	向控制执行器注册线程，并发送线程创建请求，允许后执行 create
join	Instrument_join()	向控制执行器发送请求，允许后执行 join
exit	Instrument_exit()	向控制执行器发送请求，允许后执行 exit
lock	Instrument_lock()	向控制执行器发送请求，允许后执行 lock
Unlock	Instrument_unlock()	向控制执行器发送请求，允许后执行 unlock
thread	Instrument_Thread_begin() Thread Instrument_Thread_end()	向控制执行器发送请求，允许后执行 thread_being 并执行 thread 本身的代码，执行结束后，通知控制执行器
Main (VXWORKS 系统为 usrAppInit)	Global_object_registration() Initerrupt_ registration () Instrument_main_start() Main Instrument_main_end()	向控制执行器注册所有的共享对象和中断服务函数，并通知控制执行器 main 函数开始执行和 main 函数执行结束

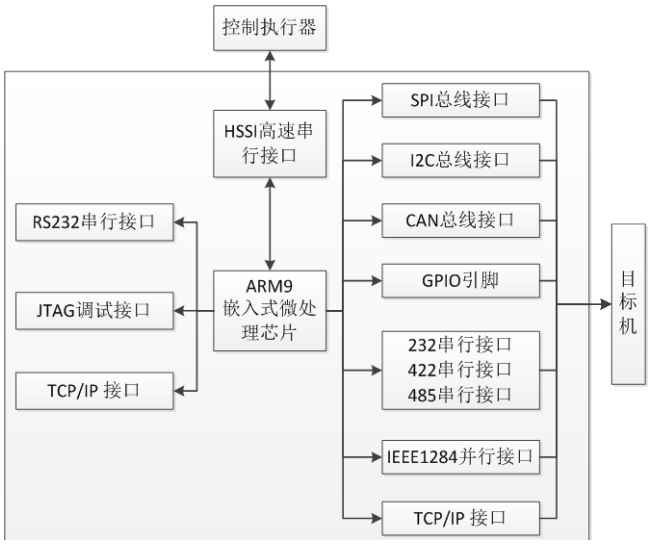

Figure 2. Hardware design for interrupts generator
图 2. 中断发生器硬件设计方案

完成的。

3.4.1. 函数空间注册

由于测试工具要处理包含普通函数、线程函数、中断服务函数的交叠路径，因此本文对被测试程序的函数空间做出了重新的映射。

本文使用 *FID* 来代表程序的函数空间， *fid* 即为函数的唯一标识，函数空间包括：线程集合 *Tid* ；中断服务函数集合 *Iid* ；以及主线程 main 函数。

假设被测试程序包含 α 个函数(1 个 main 函数即主线程， $n-1$ 个线程函数， m 个中断服务函数， $\alpha=n+m$)， $Fid=[1,2,\cdots\alpha]$。本文规定 *Tid* 的优先级为 $[0-1]$，main 函数的中断优先级为 n， *Iid* 的优先级为 $[n+1,\alpha]$，这样所有中断的优先级都高于线程及函数的优先级，主线程的优先级高于其他线程的优先级。由此我可以进一步得到并发程序的扩展模型 $M_{fid}=\left(S_{fid}^{P},init_{fid}^{P},T_{fid}^{P},R_{fid}^{P},IFlage_{fid}\right)$ 其中 *IFlage* 表示允许中断发生的标识， *P* 表示函数的优先级。

3.4.2. 状态空间化简算法

本文在多线程程序动态偏序化简[10]-[12](Dynamic Partial-Order Reduction，以下简称 DPOR)算法的基础上，设计了可以处理多线程程序、多重中断程序的 DPOR 算法。DPOR 算法的关键是剔除状态 *S* 上所有执行迁移中存在的独立关系。

定义 3.1 对于一个全局状态 $S_i=\left(s_i^1,s_i^2,\cdots s_i^n\right)$，当且仅当其上的迁移 t 在全局状态 S_i 上的一个局部状态 s_i^x 上是可执行的，此时不属于 S_i 的局部状态有 $s_i^x=s_j^x$，则迁移 t 在全局状态上才是可执行的，记作 $t\in S_i.enable$ 。

定义 3.2 $R\in T\times T$ 是独立关系，当且仅当如果对于每一个 $<t_1,t_2>\in T$，如果他们是相互独立的，则必须满足下面两个关系：

1) 若 t_1 在状态 s 是可执行的，且有 $s\xrightarrow{t_1}s'$，当且仅当 t_2 在状态 s' 是可执行的，则 t_2 在状态 s 是可执行的。

2) 如果 t_1,t_2 在状态 s 是可执行的，则存在一个唯一的状态 s'，有 $s\xrightarrow{t_1t_2}s'$ 并且 $s\xrightarrow{t_2t_1}s'$。

定义 3.3 当且仅当状态 s 上所有非空迁移序列 $s_1\xrightarrow{t}s_2\xrightarrow{t_2}s_3\cdots.\xrightarrow{t_{n-1}}s_n\xrightarrow{t_n}s_{n+1}$，满足迁移 $t_i\notin T_p$，$1\le i\le n$，t_n 与集合 T_p 中的所有迁移是相互独立的，则在状态 s 上可执行的迁移集合 $T_p\in T$ 是一个 persistent 集合。使用 persistent 集合的选择性搜索可以保证测试工具在动态测试过程中的完备性[11]。

动态偏序化简的主要思想就是计算各个状态的 Persistent 集合。

图 3 的算法设计中，我们使用了递归的调用方式，采用基于深度优先的搜索方式来对程序的状态空间进行搜索。算法中使用集 $fid(s)$ 来表示动态执行过程中状态 s 所处的当前函数， $fid(t)$ 表示该迁移所在的函数表示。 *S* 为全局对象的搜索堆栈， *s.enable* 为每一个状态 *s* 的可执行迁移集合， *s.backtrack* 为回溯集合表示需要搜索的状态 *s* 的中断集合。 *s.done* 表示被检测过的状态 *s* 的中断集合。

由于控制系统程序往往主线程中会包含无线循环以维持系统的不断工作。因此本文在算法中加入了一个 hash 表 *H* 用来记录检索过的状态，从而达到使控制执行器不反复循环检索已经检索过的状态。

本算法引入了可见操作依赖关系图的处理机制，可见关系依赖图记作 $G=<V,E>$ 为模型 *M* 中的有向图，它包含了遍历状态空间中所有可见操作之间的发生前关系[10]。对于 *G* 中的每一个节点 $v\in V$ 是一个可见操作，即 $\forall v\in V:\exists t\in T:t_g=v$。对于在动态搜索中执行的每一个迁移序列 $s_1\xrightarrow{t}s_2\xrightarrow{t'}s_3$，算法都会在图中加入一个方向边 (t_g,t'_g)，这样在除程序的第一次动态执行以外，之后的执行过程会首先搜索可见关系依赖图并执行回溯，进一步提高了状态空间搜索效率。

通过在图 2 中第 8、14、15 行中，加入对函数优先级的比较，如果函数 $fid(s)$ 的优先级高于函数 $fid(t)$

```
01 :Initially: S.push(s_0); H is empty; G is empty
02 :SDPOR(S,H,G){
03 :if (DETECTAtomicity (s)) exit(S)
04 :let  s = S.stop;
05 :if(s ∈ H){
06 :   let  U = {v|∃t ∈ s.enable, v is reachable in G from the node t_g}
07 :      for each  t ∈ U{
08 :         if (fid(s).PRI < fid(t).PRI),UPDATEBACKTRACKSETS(S,t);
09:          return;
10 :      }
12 :}
13 :Add  s  into  H;
14 :for each  (t ∈ s.enable && fid(s).PRI < fid(t).PRI ),  UPDATEBACKTRACESET(S,t);
15 :if(∃τ ∈ Fid: ∃t ∈ {s.enable: fid(t) = τ and fid(s).PRI < fid(t).PRI}){
16 :   s.backtrack← {τ};
17 :   s.done← ∅;
18 :   while(∃q ∈ s.backtrack\s.done){
19 :      s.done ← s.done ∪ {q};
20 :      s.backtrack← s.backtrack\{q};
21 :      let t∈ s.enable such that fid(t) = q;
25 :      if((fid(t) ∈ Iid And fid(t).IFLAG = enable)or( fid(t ∈ Tid)) ){
26 :         let  s'  be a state such that  s -t-> s';
27 :         S.push(s') ;
28 :         if(∃x ∈ S s.t. x -t_x-> s -t-> s', add a directed edge(x_g,t_g)to  G
29 :         SDPOR(S,H,G) ;
30 :      }
31 :    S.pop();
32 :   }
33 :   }
34 :}
```

Figure 3. The DPOR algorithm
图 3. DPOR 算法描述

的优先级，则该路径不可行，也就不再求取状态 s 上迁移 t 的回溯点。

算法的 18~30 行为控制执行部分，与传统的 DPOR 算法不同的之处在于算法第 25 行包含了对函数性质以及中断允许标志位的判断，如果函数为线程函数则迁移可行；如果函数为中断服务函数则必须要在使能中断的情况下，迁移才可行。

函数 UPDATEBACKTRACESETS(S , s)作用是动态计算回溯集合，与经典的 DPOR 算法相同，其具体描述如图 4 所示。

关于 DPOR 的算法的详细理论与数学证明请参考文献[10]。

3.4.3. 并发错误检测算法

函数 BUGDETECT 的作用是在状态 s 处检测是否存在原子性违背和数据竞争错误。算法的描述如图 5 所示。

如图 5 算法的第 2、3 行为对死锁算法的检测，算法的第 4 行至第 16 行是目标程序的原子性违背错误的检测方法，主要是检测多重中断程序中多变量读写时是否存在原子性违背错误；第 15、16 行主要是检测程序中不同优先级下，高优先级中断打断低优先级中断时，程序是否存在数据竞争的情况。

3.4.4. 控制执行机制

前文中我们已经介绍了针对多重中的 DPOR 偏序化简算法，如图 3 算法中第 26 行，此时控制执行

```
01 :UPDATEBACKTRACESETS(S,t){
02 :   let T be the sequence of transitions associated with S;
03 :   let t_d be the lasted transition in T that is dependent and may be co_enable with t;
04 :   if(t_d==null) return;
05 :   led s_d be the state in S from which t_d is executed;
06 :   let E be {q ∈ s_d.enable | fid(t) = fid(q), or q in T, t_d →π q and there is q →π t_d}
07 :   if (E! = ∅)
08 :      choose any q in E, add tid(q) to s_d.baktrack;
09 :   else
10 :   s_d.baktrack ← s_d.baktrack ∪ {tid(q)|s_d.enabled}
11 :}
```

Figure 4. UPDATEBACKTRACESETS function
图 4. UPDATEBACKTRACESETS 算法描述

```
01 : BUGDETECT(s){
02:   if(s.enable = ∅ =)
03:      return a deadlock;
04 :   if (∃A : which is a atomic blook such that state s is a state in a atomic block A){
05 :      let S_Atom be the sequence of state in atomic block A;
06 :      let T_Atom be the sequence of transition with S_Atom;
07 :      let OPW be the write operations in T_Atom;
08 :      let OPR be the read operations in T_Atom;
09 :      for (each t ∈ T_Atom){
10 :         if(∃t':fid(t').PRI > fid(t).PRI and t' is co_enabled at the same state with t){
11 :            if (op(t') = write and op(t') ∪ OPR ≠ ∅) return a atom write mistake;
12 :            if (op(t') = read and op(t') ∪ OPW ≠ ∅) return a atom read mistake;
13 :            if (op(t') = write and op(t') ∪ OPW ≠ ∅) return a atom Dual write mistake;
14 :         }
15 :      }
16 :   }
17 :   if(∃t':t' ∈ s.enable and fid(t').PRI > fid(s).PRI > 0){
18 :      if(op(t') = write) return maybe a Witten out of sync;
19 :      }
20 :   }
```

Figure 5. Concurrent error detecting algorithm
图 5. 并发错误检测算法

器将控制目标机中的被测试程序执行 $fid(t)=q$ 的函数)，该函数可能是中断也可能是线程。

若 $fid(t)\in Tid$，控制执行器阻塞除线程函数 $fid(t)$ 之外的所有的线程并关闭所有的内部中断，则线程函数 $fid(t)$ 可由 VXWORKS 系统调度为执行状态。

若 $fid(t)\in Iid$，控制执行器通知中断发生器触发中断服务函数 $fid(t)$，并关闭所有的内部中断。

图 6 中所描述的共享变量读写流程图也就是表 1 中插装函数 Instrument_obj_write (&X)以及 Instrument_obj_read (&X)所完成的功能。

4. 实验平台和实验结果

4.1. 实验平台

实验平台的具体硬件配置见表 4。

被测试程序方面，我们在 ARM 平台上选用了通讯加密编码程序，在 PPC 平台上选用了无人飞行器控制程序。为了通过实验验证控制系统并发程序错误检测算法的执行效率，我们对目标程序的线程、中

断数量进行了修改。具体的实验结果如表 5 所示。

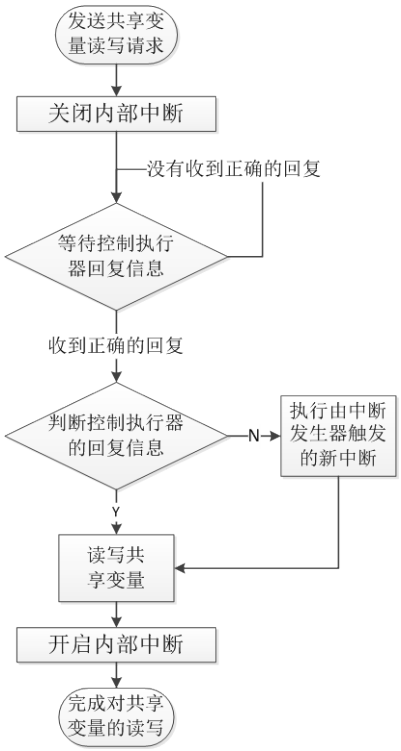

Figure 6. The implementation for controll zing excutions
图 6. 控制执行机制

Table 4. The software and hardware for testing tool
表 4. 测试软硬件平台

测试平台	处理器	RAM	操作系统
控制执行器	Intel I7 4770	8 GB	Windows 7
中断发生器	S3C6410	512 MB	Linux
ARM 平台	Exynos 4412	2 GB	VxWorks 5.5
PPC 平台	MPC8247	1 GB	VxWorks 5.5

Table 5. The 1st experimental result
表 5. 实验结果 1

测试用例	线程数/中断数	不使用 DPOR 算法		使用 DPOR 算法	
		执行迁移数	执行时间	执行迁移数	执行时间
ARM1	2/2	1193 k	472 s	236 k	89 s
ARM2	3/2	2764 k	993 s	501 k	294 s
ARM3	4/3	8769 k	/	1152 k	9578 s
PPC1	2/2	241 k	4156 s	45 k	93 s
PPC2	4/4	/	/	82 k	860 s
PPC3	6/5	/	/	157 k	7985 s

Table 6. The 2nd experimental result
表 6. 实验结果 2

测试用例	线程/中断数	Verisoft 类线程测试			实时嵌入式系统并发程序测试工具		
		迁移数	执行时间	错误数	迁移数	执行时间	错误数
ARM1	2/2	685	267 s	0	236 k	89 s	0
ARM2	3/2	1070	986 s	1	501 k	294 s	1
ARM3	4/3	2970	2898 s	3	1152 k	578 s	2
PPC1	2/2	143	310 s	1	45 k	93 s	1
PPC2	4/4	347	2987 s	2	82 k	860 s	2
PPC3	6/5	465	29,981 s	5	157 k	7985 s	4

4.2. 实验结果

在以上的表格中，我们用“/”表示程序无法在 48 小时(172,800 秒)内测试完成。从图中可以看出，在被测程序包含中断数、搜索迁移数、检测时间等三个比较指标上，在使用了偏序化简算法以后，执行时间大大缩短，提高了检测效率。

表 6 我们分别使用了类线程测试的方法[2] [3]与我们的测试工具进行对比。由于中断服务函数和线程函数在执行方面存在着本质上的区别，因此在测试过程中类线程测试工具产生了误报的情况，另外由于 verisoft 采用了无状态的动态偏序化简算法，因此执行效率较低，而且本测试工具还有对中断服务函数和线程函数优先级的判断，可以进一步的缩减被测试程序并发模型的状态空间，所以在执行效率上有了进一步的提升。

5. 结束语

本文根据实时嵌入式系统中并发 C 语言程序的相关特点，使用 LTS 模型对并发程序进行建模，在 DPOR 算法的基础上，对中断服务函数进行了功能扩展，实现了对包含多线程、多重中断状态空间的化简。本文根据常见的三种并发错误的经典定义，给出了在标记迁移系统中的形式化描述和具体的检测算法，最终实现了对于包含多线程、多重中断的 C 语言并发错误检测。

参考文献 (References)

[1] 吴学光, 文艳军, 王戟, 傅秀涛, 綦艳霞, 顾斌 (2011) 多重中断 C 程序中数据竞争及原子性违背检测. *计算机科学与探索*, **12**, 1086-1093.

[2] Regehr, J. and Cooprider, N. (2007) Interrupt verification via thread verification. *Electronic Notes in Theoretical Computer Science*, **174**, 139-150.

[3] Hofer, W., Lohmann, D., Scheler, F. and Schroder-Preikschat, W. (2009) Sleepy sloth: Threads as interrupts as threads. 30*th IEEE Real-Time Systems Symposium*, Vienna, 29 November-2 December 2011, 67-77.

[4] Dingel, J. (2003) Computer-assisted assume/guarantee reasoning with VeriSoft. *Proceedings of the* 25*th International Conference on Software Engineering* (ICSE'03), 138-148.

[5] Joshi, P., Naik, M., Park, C.-S. and Sen, K. (2009) CalFuzzer: An extensible active testing framework for concurrent programs. http://srl.cs.berkeley.edu/~ksen/calfuzzer/

[6] Yang, Y., Chen, X.F., Gopalakrishnan, G. and Kirby, R.M. (2009) Inspect: A runtime model checker for multithreaded C programs. School of Computing, University of Utah Salt Lake City, Technical Report UT 84112.

[7] 傅修峰, 陈丽容 (2012) 多重中断程序测试框架. *计算机工程与设计*, **2**, 617-623.

[8] Anderson, Z. (2013) A CIL tutorial-using CIL for language extensions and program analysis. Systems Group Department of Computer Science, ETH Zürich.

[9] Chaki, S., Clarke, E.M., Ouaknine, J., et al. (2004) Automated, compositional and iterative deadlock detection. *Proceedings of the 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, IEEE Press, 201-210.

[10] Flanagan, C. and Godefroid, P. (2005) Dynamic partial-order reduction for model checking software. *Proceedings of POPL* 2005, Long Beach, ACM Press, 110-121.

[11] Godefroid, P. (1996) Partial-order methods for the verification of concurrent systems—An approach to the state-explosion problem. Springer-Verlag New York, Inc., Secaucus.

[12] Valmari, A. (1991) Stubborn sets for reduced state space generation. In: *Lecture Notes in Computer Science*, Vol. 483, Advances in Petri Nets 1990, Springer Press, Berlin, 491-515.