4. *t*8:判单双精度.当函数类型为双精度时 *t*8=1;当函数类型为单精度时 *t*8=0.
5. *t*8~*t*13,*t*14~*t*17:返回地址.
6. *t*63:异常总标识.*t*1~*t*6 存在 1,则 *t*63=1,否则 *t*63=0.
7. *exc_num*=*t*0 or (*t*1<<1) or (*t*2<<2) or … or (*t*63<<63);
8. **return** *exc_num*;

运行算法 2 实现异常类型编码,能够确保程序快速准确地判断异常类型并返回预设的异常结果.以双精度为例,经过 ExceptionCoding 算法后的异常编码如下:

| 异常标识 | 编码 |
|---|---|
| EXC_INV_ZERO_EDOM | 0x8000000000000502 |
| EXC_INV_NANZERO_EDOM | 0x8000000000004502 |
| EXC_INV_NANINF_EDOM | 0x8000000000019902 |
| EXC_UNF_ZERO_ERANGE | 0x8000000000000511 |
| EXC_UNF_NINF_ERANGE | 0x8000000000039909 |
| EXC_DNO_DNO_ERANGE | 0x8000000000000551 |
| EXC_DZE_NINF_EDOM | 0x8000000000039904 |
| EXC_DZE_INF_EDOM | 0x8000000000009504 |
| EXC_OVF_INF_ERANGE | 0x8000000000009509 |
| EXC_ISIEEE | 0xffffffffffffffff |

其中,EXC_ISIEEE 为特殊标识,当异常类型被设置为该标识时则直接返回,不对异常进行处理;EXC_INV_ZERO_EDOM 表示 INV 异常,返回值为 0,且错误码为 EDOM(源自于函式的参数超出范围);EXC_DNO_DNO_ERANGE 表示 DNO 异常,返回值为非规格化数,且错误码为 ERANGE(源自于函式的结果超出范围).

### 2.3 输入参数检测

程序员在编程过程中,总是期望所有输入都不会引起程序的异常情况.但不幸的是:即使将输入参数小心地控制在函数定义域范围内,也可能触发一些异常(如 DNO 异常);幸运的是,该异常一般不会引起严重的后果.而当浮点数的特殊数参与浮点运算时,必然会引发浮点算术异常,从而使得函数运算过程中断,降低浮点函数的可靠性及性能.因此,在进入函数正常运算阶段前需要对函数的输入参数做特殊数检查:一方面将错误码为 EDOM 的异常解决在函数核心运算之前,另一方面确保函数核心运算的连贯性.

虽然不同的浮点函数在核心运算上有所不同,但却可以在函数的开始阶段对数据进行预判断,以区分输入参数的类型.本文利用比较指令对输入参数的判断进行统一的规范化操作,实现了输入参数检测子程序 PartitionInputParameter.该程序的实现采用的是汇编语言,为了方便描述,转化成相应的 C 程序呈现.

**算法 3**. PartitionInputParameter:实现输入参数的预判断,确保引起 EDOM 类异常的输入进入相应的异常处理子程序,其他则进入核心运算子程序.

Inputes:*V*,输入参数向量(*p*1,*p*2,*p*3,*p*4);

*a*,*F* 定义域下界;

*b*,*F* 定义域上界;

*num*,*F* 的参数个数 1~3;

MAX,无穷数;

*VCPYF*($1,$2):将寄存器$1 中的浮点数扩展为向量并存入寄存器$2;*VINSF*($1,$2,$3):将$1 中的浮点数插入到向量寄存器$2 中的第$3 个位置;*CheckINFNAN*(*inf*,*V*):检测向量 *V* 中是否存在无穷数或非数;*CheckInterval*(*a*,*b*,*V*):检测向量 *V* 中的变量是否在定义域区间内.

1. *VCPYF*(*p*1,*V*); //*V*=(*p*1,*p*1,*p*1,*p*1)
2. **if** *num*=2 **then**
3. *VINSF*(*p*2,*V*,1); //*V*=(*p*1,*p*2,*p*1,*p*1)
4. **else if** *num*=3 **then**
5. *VINSF*(*p*2,*V*,1);*VINSF*(*p*3,*V*,2); //*V*=(*p*1,*p*2,*p*3,*p*1)

```
    end if
  end if
  if CheckINFNAN(MAX,V)=TRUE then
    return INV;
else if CheckInterval(a,b,V)=TRUE then
      retuen INV;
    else
      return;
    end if
end if
```

PartitionInputParameter 算法的贡献并不仅仅是实现了输入参数的检测这样一个功能,而是在实现的过程中结合 SIMD 编程思想,确保了算法的高效性,同时也实现了多参数函数的统一处理.程序中对 SIMD 的运用主要体现在向量 $V$ 的构建.研究中发现:浮点函数的输入参数个数从 1~3 不等,输入参数的不确定性,增加了标量程序的工作量,如果使用标量运算,对于 3 个输入的函数而言,同样的判断则需要重复进行 3 次;而通过向量实现,则可以同时进行判断.通过实验分析可知:与标量实现相比,向量实现平均能有 10%左右的性能提升.

### 2.4 特定代码检测

异常总是可能伴随着浮点函数运算存在,如随时都可能出现的溢出异常.如果在浮点运算过程中时时检测异常,必将严重影响运算效率.本文结合浮点函数实现过程中的特点,在浮点运算过程中并不对所有异常进行检测,只对由固定指令触发的 DZE 异常及 INF 异常进行检测,其余都交给运算结束后的输出结果检测子程序处理.本文通过对 FDIVS(单精度除)和 FDIVD(双精度除)指令的监控以处理可能出现的 DZE 异常,以及对 FCVTDL(双精度浮点转换为长字)和 FCVTLW(长字整数转换为字整数)指令的监控以处理可能出现的 INF 异常.

当前,对 INF 异常检测的研究尤为深入,大致可分为两类.

- 一类是动态检测技术.Brumley 等人[18]通过对可能引起 INF 的每一个整数操作进行检测实现的 C 程序检测工具 RICH,该工具最大的优势在于对程序性能干扰的控制,平均只有 5%的性能下降;类似的方法还有 Chinchani 等人[23]的研究.Dietz 等人[24]针对 C/C++语言并从软件工程的角度实现的检测工具 IOC,并发现了众多存在的 INF,该工具并没有关注于对程序性能的影响;而 Chen[25]实现的检测方法最大的缺陷就是造成程序性能下降了 50 倍;
- 另一类是静态检测技术.Molnar[26]利用符号执行实现了检测工具 SmartFuzz;在此基础上,Wang[27]结合污点分析实现了检测工具 IntScope.

上述研究都是针对面向对象语言实现,而浮点函数由汇编语言实现,加上 DZE 异常及 INF 异常出现位置固定且可能触发的次数有限,造成现有的检测方法在当前环境下无法发挥该有的优势.虽然相比较于其他学者的研究,本文的方法显得过于简单,但对于汇编实现的浮点函数,这样的方法确实是最准确有效的,能够在准确检测的同时最大限度地降低异常检测对函数整体性能的影响.通过统计发现:这两类指令在浮点函数中运用的比例都相对较小,对于 FDIVS/FDIVD 指令,只在 152 个函数中的 28 个函数中出现,并且在平均有上千行汇编代码的函数中最多出现次数也只有 5 次;对于 FCVTDL/FCVTLW 指令,只在 15 个函数中出现,最多出现次数也只有 6 次.

浮点函数中,DZE 异常和 INF 异常的可能触发条件可以限定为上述 4 种指令的执行,而可能触发这两类异常的情况较少又决定了本节提出检测方法的有效性及高效性.具体而言,通过对指令 FDIVS/FDIVD 的搜索,实现核心运算中对 DZE 异常的检测;通过对指令 FCVTDL/FCVTLW 的搜索,实现核心运算中对 INF 异常的检测.

### 2.5 输出异常检测

为了保证核心运算的完整性,在浮点函数实现过程中并没有对 FPF 异常(DNO 异常可以看作是 FPF 异常中

的下溢)进行检测,而是将这部分工作统一放到了运算结束后.FPF 异常的触发,可以转化为有限数运算结果为非有限数的情况.通过对运算结果的检测衡量函数是否出现异常,这样做基于第 2.1 节论证的两个结论.具体的输出异常检测算法如下.

**算法 5**. PartitionOutputParameter:实现运算结果的 FPF 异常检测.

Inputs:*R*,核心运算结果;

*a*,*F* 定义域下界;

*b*,*F* 定义域上界;

*num*,*F* 的参数个数 1~3;

MAX,无穷数;

MIN,最小规格化数;

*Checkoverflow*(MAX,*V*):检测 *R* 是否出现上溢;*Checkunderflow*(MIN,*R*):检测 *R* 是否出现下溢.

1. **if** *Checkoverflow*(MAX,*R*)=TRUE **then**
2. **return** FPF;
3. **else if** *Checkunderflow*(MIN,*R*)=TRUE **then**
4. **retuen** FPF;
5. **else**
6. **return**;
7. **end if**
8. **end if**

### 2.6 异常返回

对于最终的异常返回,本文针对不同的异常类型预先设定了异常返回值,通过查表的形式确定返回值 *E*.虽然查表法占用了一些存储空间,但却提高了异常处理的速度.总体而言,该方法牺牲空间,获得了时间.算法主要依据异常类型编码规则实现,通过对编码后各浮点位的判断确定异常返回值,其具体算法如下.

**算法 6**. Exception:根据异常编码确定异常处理的操作及其返回值.

Inputs:*exc_num*,异常类型对应的编码;

*R*,核心运算结果;

*fp_control*:根据错误码设置浮点控制寄存器参数.

1. **if** *exc_num*=−1 **then return**;
2. t6 = *exc_num* & 0x40;
3. **if** *t*6!=0 **then**
4. *t*8=(*unsigned long*)((*exc_num*^0x40)>>0x8) & 0x1;
5. *retv*=*retval_table*[(4+((*R*>>0x3f) & 0x1)*4)+*t*8];
6. **else**
7. *t*7=(*unsigned long*)(*exc_num*>>0x8) & 0x3f;
8. *retv*=*retval_table*[*t*7];
9. **end if**
10. *t*14=*exc_num* & 0x3e;
11. **if** *t*14!=0 **then**
12. *fp_control*();
13. **end if**
14. **return** retv;

## 3 实验分析

本文将实现的异常处理方法应用于 Mlib 函数库[28]的浮点函数中(该数学库应用于 2011 年 9 月发布并安装在济南超算中心的神威蓝光超级计算机中).本实验在与神威蓝光同类型的某高性能计算机中的任意运算节点进行,该节点包括主核和从核两个运算核心,主核运行完整的操作系统,从核为 64 位 RISC 结构多用处理单元.其中,主核处理器主频为 1 000MHz,内存主频 400MHz,内存容量 1.8GB;从核处理器主频为 1200MHz,内存主频为 800MHz,内存容量 1.8GB.

Mlib 函数库由基础函数库及 SIMD 扩展函数库,共计 304 个浮点函数组成.其中,SIMD 扩展函数库由基础函数库通过 SIMD 指令扩展实现,与基础数学库具有相同的函数分类,包括三角函数、反三角函数、双曲函数、指数对数函数、伽马函数、取整取余函数、数值函数、误差函数、贝塞尔函数及其他函数等初等函数.

对应用本文方法前后的 Mlib 函数库进行了测试,测试结果表明:在应用本文方法前 Mlib 中有 90.30%的函数会因为浮点异常而出现中断,而应用后这一比例被降到了 0%.即:本文方法能够有效处理可能出现的浮点异常,且保证浮点函数不会因为浮点异常而出现中断.

在下面的测试中,本文对部分典型函数的测试结果进行了分析,包括 sqrt,atan,pow,sin,fp_class 和 atan2 函数,这些函数分别代表了不同的函数类型,且包含了各类的异常类型.

### 3.1 函数中的异常处理代码

对于异常处理最重要的衡量指标之一就是在其应用中所占的代码百分比.为了实现这一指标,本文对函数中计算代码和异常处理代码的条数进行了统计,结果如图 4 所示.

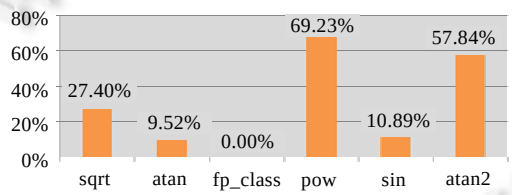


Fig.4 Exception handling code occupations among kernel functions

图 4 典型函数中的异常处理代码比例

图 4 中的结果是部分典型函数的统计结果.一般程序中的异常处理代码是整个代码量的 2/3 左右[16],但从测试结果看,却只有 pow 和 atan2 两个函数符合这个要求,fp_class 函数的异常处理代码甚至为 0%,浮点函数在数学上的一些特性决定了上述测试结果.如 fp_class 函数属于数值函数,其功能是检查输入参数属于哪类浮点数,这样的函数功能决定了该函数没有任何的计算,没有任何错误码,同样也不会有任何的异常处理代码; atan 函数、sin 函数在实现中需要处理的异常只有 INV 异常(由采用的算法决定),只需要很短的代码就能实现,所以其异常处理百分比都集中在 10%左右,对于类似的三角函数(如 cos,tan 等)也具有同样的特点.

### 3.2 异常动作

区别于对异常代码量的统计,了解异常发生后带来的伴随动作的合理性,是衡量异常处理方法是否有效的又一重要指标.浮点函数在实现过程中,需要遵守函数本身的一些数学上的定义,发生异常该如何处理、返回值是什么,都可能和最初异常处理方法中设定的有出入.为了准确获取函数真实的异常动作,对代码量超过 20 万行运行于主从核的 608 个浮点函数进行了测试,虽然由于测试集数据量庞大的原因,并没有能够测到每个函数可能出现的所有异常处理,但依然反映出了一些性质,并将测试结果总结在表 1 中.

**Table 1** Exception categories and their details

**表 1** 异常行为及其描述

| 分类 | 描述 |
|---|---|
| 空 | 不出现异常,不作任何反应 |
| 标记 | 调用浮点控制寄存器,设置其值,主要用于INF异常 |
| 补指 | 当*dnp*参与运算且不被当作零处理时,通过强加一个大的指数,保证计算过程的可靠性 |
| 0 | 将*dnp*当作0处理 |
| 直接返回 | 将输入直接输出 |
| 继续 | 计算过程中出现异常,不作处理程序继续执行,一般在FPF异常中出现 |
| 返回 | 调用异常返回,返回预设的结果 |

需要注意的是:对于一个在执行过程中的浮点函数可能出现多个异常行为,如某个函数在计算中将 *dnp* 当作 0 后继续计算,且在某处发生了溢出,在最后检测计算结果时调用了异常返回.这样的一个计算过程就触发了 3 种异常行为:0、继续和返回.因此,这样的一个异常处理过程将会被分别记录到这 3 种异常行为中.对于典型函数的统计结果,如图 5 所示.

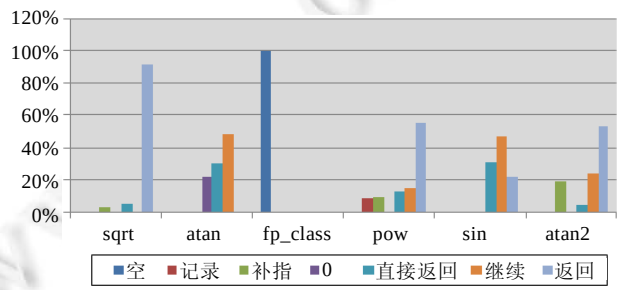


Fig.5 Some kernel functions' handling actions

图 5 典型函数异常行为

空只在函数 fp_class 中出现,因为该函数不存在任何的异常,类似的数值类函数也都有同样的特点.对于 sqrt 函数,因为该函数主要处理的异常为参数小于 0 时的 INV 异常,其 90%左右的异常行为都是返回.对于 atan2 函数,由于以 *x*/*y* 形式作为输入参数,与 atan 函数相比,就需要处理可能出现的下溢,表现的异常行为也有所不同,atan2 函数明显需要更多的返回.从图 5 中可以看出:对于 *dnp* 的两种异常行为不会同时出现,在一个函数中要么将 *dnp* 当作 0,要么进行补指,当然也可以直接返回.

### 3.3 异常触发的可能性

各种异常是否存在一定的关系?在浮点函数中,每一类异常触发的可能性又有多大?这是在本节测试中需要回答的问题.在测试中,将异常类型分为 4 大类 9 小类,大类分为 DZE 异常、INF 异常、FPF 异常和 INV 异常,小类按异常类型_返回值_错误码的形式分为 INV_ZERO_EDOM,INV_NAN_EDOM,INV_INF_EDOM,UNF_ZERO_ERANGE,OVF_NINF_ERANGE,OVF_INF_ERANGE,INF_NaN_ERANGE,DZE_NINF_EDOM 和 DZE_INF_EDOM.

图 6 中的测试结果反映了各种异常行为被触发的可能性,例如:INF_ERANGE 的 10%表明,在 Mlib 中只有 10%的函数可能发生 INF 异常;与之对应的 INF_NaN 的 100%表明,触发 INF 异常后必然返回 *NaN*.

从 4 个大类异常的角度对浮点函数进行测试统计后的结果如图 6 中的上半部分所示,最容易被触发的异常类型是 FPF 异常和 INV 异常,都达到了 90%的可能.以 INV_EDOM 为例,返回值为 *NaN* 的异常类型触发的可能最大,达到了 86%(INV_NaN).

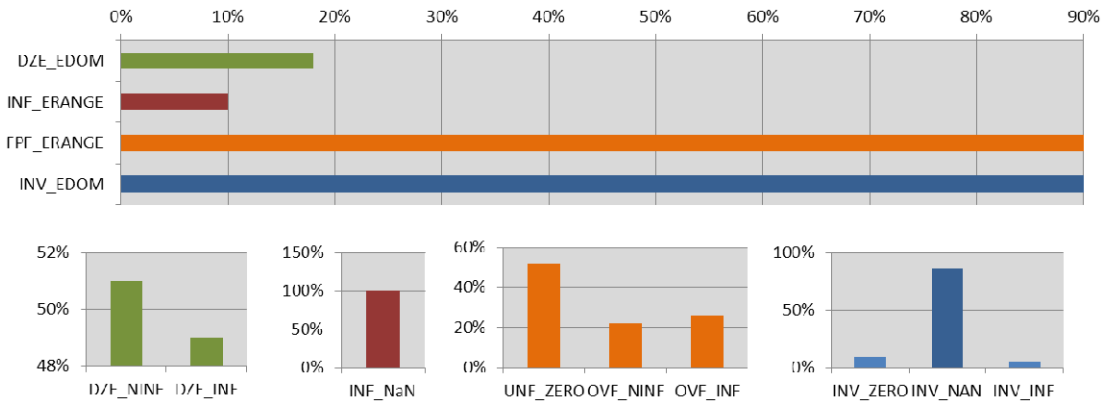


Fig.6 Possibilities of exception results

图 6 异常结果的可能性

### 3.4 异常对性能的干扰

虽然对于浮点函数有高可靠和高性能的要求,但高可靠往往比高性能更重要,只有保证函数高可靠的基础上,才能去不断实现对高性能的需求.判断异常处理方法是否有效,除了满足高可靠的要求,还需要满足对函数性能干扰尽量少的要求,干扰越少,说明方法越高效.本文通过对浮点函数使用异常处理方法前后的运行性能的对比测试,反映出异常处理方法的高效性.

虽然浮点函数支持所有的浮点数,保证函数在执行过程中不会出现因浮点异常导致的程序中断,但是在正常的应用过程中,还是以正常区间的常用浮点数作为函数最频繁的输入.这些输入的运行路径都集中在函数的关键路径上,通过测试关键路径的运行性能,才能反映函数性能的真实运行情况.鉴于此,为了简化测试工作,保证每个函数都能尽可能地运行在关键路径,本文选取(0,1)之间的浮点数(对函数数学意义进行分析总结后确定的区间)对 Mlib 函数进行性能测试,通过对函数重复 10 万次的运行,计算函数的平均性能.测试结果如图 7 所示.

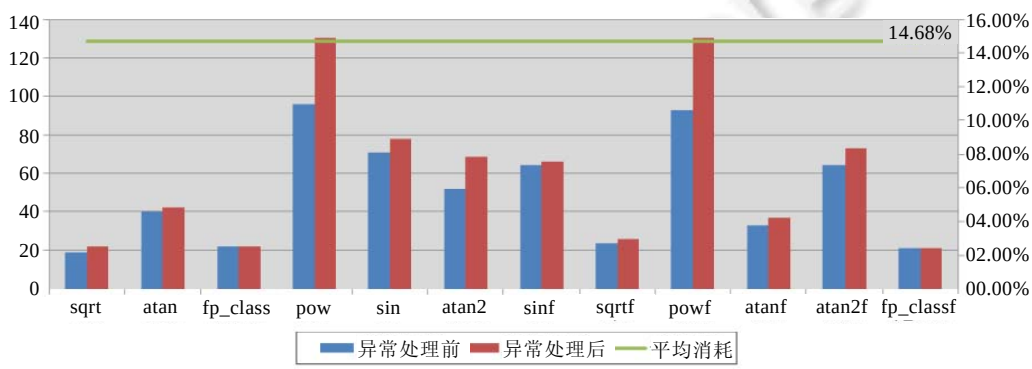


Fig.7 Performance comparisons before and after exception handling

图 7 异常处理前后的性能对比

测试结果显示:经过异常处理后,函数的平均性能降低了 14.68%((异常处理后-异常处理前)/异常处理前);同时,对于多数函数而言,异常处理也只带来了几拍到十几拍的性能消耗,完全在可接受的范围内.通过测试,进一步体现了本文方法的高效性.

## 4 总 结

本文实现了一个分段式的异常处理方法,该方法面向汇编语言实现的浮点函数,并成功应用于 Mlib 函数库中.通过测试,该方法能够有效地提高浮点函数的可靠性,同时对浮点函数的性能干扰较少,能够满足浮点函数的高效性要求.相对于实验测试,理论验证总是显得更加可靠,虽然本文方法成功应用于浮点计算相对密集的浮点函数,并利用浮点计算应用最广泛的高性能计算平台进行测试,且在主从核共进行了 600 多个函数的测试,可依然无法百分之百地肯定本文方法完全正确.下一步,将尝试从理论验证的角度全面证明本文方法的可靠性,同时,将该异常处理方法扩展到更多的浮点计算型数值软件中.

**致谢** 在此,向评阅本文的审稿专家表示衷心的感谢,向对本文工作给予支持和建议的同行表示感谢.

**References**:

[1] Kahan W. IEEE standard 754 for binary floating-point arithmetic. Lecture Notes on the Status of IEEE, 1996,754:94720−1776.

[2] Wikipedia. Ariane 5 flight 501. http://en.wikipedia.org/wiki/Ariane_5_Flight_501

[3] CNN. Toyota: Software to blame for Prius brake problems. http://edition.cnn.com/2010/WORLD/asiapcf/02/04/japan.prius.complaints/

[4] Kahan W, Darcy JD. How Java's floating-point hurts everyone everywhere. In: Proc. of the Talk Given at the ACM '98 Workshop on Java for High-Performance Network Computing. 1998.

[5] Liu CG. Floating Point Calculation: Programming Principle, Realization and Application. Beijing: China Machine Press, 2008. 6−8 (in Chinese).

[6] Goldberg D. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys, 1991,23:5−48. [doi: 10.1145/103162.103163]

[7] Hauser JR. Handling floating-point exceptions in numeric programs. ACM Trans. on Programming Languages and Systems (TOPLAS), 1996,18(2):139−174. [doi: 10.1145/227699.227701]

[8] Cadar C, Dunbar D, Engler DR. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI). 2008. 209−224.

[9] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. Proc. of the 26th ACM SIGPLAN Symp. on Programming Language Design and Implementation (PLDI). 2005,40(6):213−223. [doi: 10.1145/1065010.1065036]

[10] Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C. In: Proc. of the 10th European Software Engineering Conf. on Held Jointly with 13th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (ESEC-FSE). 2005. 263−272. [doi: 10.1145/1081706.1081750]

[11] Boldo S, Melquiond G. Flocq: A unified library for proving floating-point algorithms in Coq. In: Proc. of the 20th IEEE Symp. on Computer Arithmetic (ARITH). 2011. 243−252. [doi: 10.1109/ARITH.2011.40]

[12] De Dinechin F, Lauter C, Melquiond G. Certifying the floating-point implementation of an elementary function using Gappa. IEEE Trans. on Computers, 2011,60(2):242−253. [doi: 10.1109/TC.2010.128]

[13] Xia H, Wang CY, Yan JY. Analysis and research of floating-point exceptions. In: Proc. of the 2nd Int'l Conf. on Information Science and Engineering. 2010. 1851−1854. [doi: 10.1109/ICISE.2010.5690343]

[14] Goodenough JB. Exception handling: Issues and a proposed notation. Communications of the ACM, 1975,18(12):683−696. [doi: 10.1145/361227.361230]

[15] Cristian F. Exception handling and software fault tolerance. IEEE Trans. on Computers, 1982,100(6):531−540. [doi: 10.1109/TC.1982.1676035]

[16] Garcia AF, Rubira CMF, Romanovsky A, Xu J. A comparative study of exception handling mechanisms for building dependable object-oriented software. Journal of Systems and Software, 2001,59(2):197−222. [doi: 10.1016/S0164-1212(01)00062-0]

[17] Cabral B, Marques P. Exception handling: A field study in Java and Net. In: Proc. of the 2007 European Conf. on Object-Oriented Programming. 2007. 151−175. [doi: 10.1007/978-3-540-73589-2_8]

[18] Brumley D, Chiueh T, Johnson R, Lin H, Song D. RICH: Automatically protecting against integer-based vulnerabilities. Department of Electrical and Computing Engineering, 2007. 28.

[19] Lu XC, Li G, Lu K, Zhang Y. High-Trusted-Software-Oriented automatic testing for integer overflow bugs. Ruan Jian Xue Bao/Journal of Software, 2010,21(2):179−193 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3785.htm [doi: 10.3724/SP.J. 1001.2010.03785]

[20] Rodrigues RE, Sperle Campos VH, Quintao Pereira FM. A fast and low-overhead technique to secure programs against integer overflows. In: Proc. of the 11th IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO). 2013. 1−11. [doi: 10.1109/CGO.2013.6494996]

[21] Barr ET, Vo T, Le V, Su ZD. Automatic detection of floating-point exceptions. In: Proc. of the 40th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). 2013. 549−560. [doi: 10.1145/2429069.2429133]

[22] CRlibm: Correctly rounded mathematical library. http://libforge.ens-lyon.fr/www/crlibm/index.html

[23] Chinchani R, Iyer A, Jayaraman B, Upadhyaya S. ARCHERR: Runtime environment driven program safety. In: Proc. of the 9th European Symp. on Research in Computer Security (ESORICS). 2004. 385−406. [doi: 10.1007/978-3-540-30108-0_24]

[24] Dietz W, Li P, Regehr J, Adve V. Understanding integer overflow in C/C++. In: Proc. of the 2012 Int'l Conf. on Software Engineering (ICSE). 2012. 760−770.

[25] Chen P, Wang Y, Xin Z, Mao B, Xie L. Brick: A binary tool for run-time detecting and locating integer-based vulnerability. In: Proc. of the 2009 Int'l Conf. on Availability, Reliability and Security (ARES). 2009. 208−215. [doi: 10.1109/ARES.2009.77]

[26] Molnar D, Li XC, Wagner DA. Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proc. of the 18th Conf. on USENIX Security Symp. 2009. 67−82.

[27] Wang T, Wei T, Lin Z, Zou W. IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution. In: Proc. of the 16th Annual Network & Distributed System Security Symp. (NDSS). 2009.

[28] Sunway blue-ray. 2013 (in Chinese). http://baike.baidu.com/view/6780250.htm

## 附中文参考文献:

[5] 刘纯根.浮点计算编程原理、实现与应用.北京:机械工业出版社,2008.6−8.

[19] 卢锡城,李根,卢凯,张英.面向高可信软件的整数溢出错误的自动化测试.软件学报,2010,21(2):179−193. http://www.jos.org.cn/1000-9825/3785.htm [doi: 10.3724/SP.J.1001.2010.03785]

[28] 神威蓝光.2013. http://baike.baidu.com/view/6780250.htm

**许瑾晨**(1987−),男,江苏泰州人,博士生,主要研究领域为高性能计算.

**王磊**(1977−),男,副教授,主要研究领域为高性能计算.

**郭绍忠**(1964−),女,副教授,主要研究领域为高性能计算,分布式系统.

**周蓓**(1977−),女,讲师,主要研究领域为高性能计算.

**黄永忠**(1968−),男,博士,教授,博士生导师,主要研究领域为高性能计算,大数据处理.