doi:10.11887/j.cn.202006011　　http://journal.nudt.edu.cn

面向 RISC-V 内核的标记指令复算与纠错机制的设计*

邓　丁,郭　阳
(国防科技大学 计算机学院, 湖南 长沙　410073)

摘　要:由宇宙辐射等环境因素引起的硬件瞬时故障会极大降低计算机系统的可靠性。为了减小硬件瞬时故障对系统可靠性的影响,保证程序的正确运行,基于 RISC-V 开源内核“蜂鸟 e203”提出一种标记指令复算与纠错机制。该机制为指令码额外增加了复算标记,以较小的硬件开销实现对指定指令的复算操作。此外,该机制可以在第一次复算结果与初始运算结果不等时自动进行二次复算,并由三次运算结果的多数表决结果来纠正大部分硬件瞬时故障所引起的数据流异常。实验表明,通过与中断处理程序相结合,在随机注入瞬时故障的情况下,程序的正确执行概率平均增加了 86.67%。

关键词:RISC-V;蜂鸟;指令标记;复算;纠错
中图分类号: TP302.8　　**文献标志码**:A　　**文章编号**:1001-2486(2020)06-090-08

Recomputation and correction mechanism design for tagged instructions of the RISC-V core

DENG Ding, *GUO Yang*
(College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

Abstract: The reliability of the computer system is significantly compromised by the hardware transient faults which are mainly caused by the cosmic radiation and other environmental factors. To mitigate this undesirable impact and guarantee the correctness of the running programs, a recomputation and correction mechanism for tagged instructions for an open source core named “Humming bird e203”, which is based on the RISC-V instruction set architecture, was proposed. This mechanism adds extra flag bits for each instruction and thus enables flexible recomputation for any tagged instruction at low hardware cost. Besides, it can issue the tagged instruction again automatically if the result of the first recomputation is different from the original one. This majority voting scheme can efficiently rectify most data flow errors caused by transient hardware faults. The experimental results show that with our proposal and the interrupt handler, the average probability at which programs can operate correctly can be increased by 86.67% under the random transient fault insertion.

Keywords: RISC-V; Humming bird; tagged instruction; recomputation; correction

据统计, 1971—1986 年的 16 年期间,国外发射的 39 颗同步卫星总共出现故障 1589 次,其中 1129 次与空间辐射有关[1]。空间辐射对电子器件的影响可分为总剂量(Total Ionizing Dose,TID)效应和单粒子效应(Single Event Effect,SEE)。具体而言,单粒子效应又可细分为单粒子翻转(Single Event Upset,SEU)、单粒子闩锁(Single Event Latch,SEL)和单粒子烧毁(Single Event Burnout,SEB)三种子类。其中:后两者均会导致载体器件发生不可逆转的永久故障。而单粒子翻转仅是由于晶体管在外界带电粒子的轰击下瞬时充放电,从而造成的逻辑状态翻转。一旦粒子轰击结束,便可以自动恢复正常功能。也即,单粒子翻转所引发的是一种瞬时故障。事实上,瞬时故障在所有引起计算机系统失效的故障中所占比重接近 90%[2],是永久故障的 100 倍以上[3]。文献[4]指出,单粒子翻转是辐射效应的主要形式,在全部由宇宙辐射引发的故障中,55% 是单粒子翻转故障。

由计算机硬件故障所导致的错误可分为控制流错误和数据流错误。控制流错误主要发生在执行转移指令时,而数据流错误主要是由于存储或传输中的数据因干扰耦合等因素发生错误进而引起的系统异常。

针对控制流错误,2002 年 Standford 大学 CRC 实验室提出了基于软件签名的控制流检查

* **收稿日期**:2019-06-13
基金项目:国家自然科学基金资助项目(61832018)
作者简介:邓丁(1993—),男,四川遂宁人,博士研究生,E-mail:dengding15@nudt.edu.cn;
郭阳(通信作者),男,研究员,博士,博士生导师,E-mail:guoyang@nudt.edu.cn

(Control Flow Checking by Software Signature, CFCSS)算法,有效地将分支故障导致系统错误的概率从 33.7% 降低为 3.1%[5]。但是,CFCSS 算法仍不能检测到伪分支错误、基本块内部控制流转移错误。文献[6]通过添加硬件看门狗来检查程序的控制流错误。文献[7]通过采用市场上常见的调试接口模块能够有效及时地检测到控制流错误。

针对数据流错误,利用时间或空间上的冗余来验证数据的有效性是一种行之有效的方法。N 版本程序法(N-Version Programming, NVP)为目标功能设计了多种可能实现方式,从而保证至少有一种方式能够正确地执行[8]。数据重表达法(Data Re-expression Algorithm, DRA)采用多种不同的方式表达同样的数据,执行相同的程序,从而能够检查并恢复数据流错误。编译器级容错技术通过复制程序指令,在一定检查点插入比较指令来判断程序执行结果的正确与否,其典型的算法包括 EDDI[9]、ED4I[10]、SWIFT[11]等。

文献[12]将软硬件方法结合在一起,仅用一个低端现场可编程门阵列(Field Programmable Gate Array, FPGA)便极大提高了处理器的容错能力。文献[13]从综合阶段入手优化数据通路对瞬时故障的容错能力,将芯片的功耗降低了约 48%。文献[14]着重对专用于图像处理和机器学习的分布式数据流处理系统进行了容错设计。文献[15]将数据流应用映射到多个虚拟处理器上,从而减小了容错所需的总面积。

本文采用重复执行被标记指令的策略来检测并纠正由单粒子翻转效应引起的数据流瞬时故障。本文的主要创新与贡献包括:

1)实现了一种可灵活标记复算指令的机制。该机制以较小的硬件开销与性能降低为代价,能够检测出指定指令在执行期间是否发生数据流错误。

2)对于标记为复算的指令,实现了时间上的三模冗余纠错机制。若第一次复算结果与初始结果不相等,将自动执行第二次复算,并根据少数服从多数的原则,对数据流错误进行纠正。

1　标记指令复算与纠错机制

1.1　总体结构设计

单粒子翻转故障是一种瞬时故障,其故障持续时间很短。因此通常假设单粒子翻转故障发生的同时,错误也立即产生,即没有故障延迟。又因单粒子翻转故障触发的时刻、位置都是随机不可预测的,所以连续两个不同时刻,在同一位置发生同一类型的瞬时故障的概率非常微小。鉴于此,本文提出在连续的指令周期里重复执行被标记的指令来检测并纠正瞬时错误结果的容错机制。

从宏观上看,关于标记指令复算机制的改进主要集中在如图 1 所示的深灰色方框中,关于标记指令纠错机制的改进主要集中在如图 1 所示的浅灰色方框中。纠错的核心思想是:由编译器对关键或易错指令进行复算标记的赋值,并将复算标记一同存入指令存储器中;复算标记与其所对应的指令一同流入处理器内核,最终一同发射给运算逻辑单元(Arithmetic Logical Unit, ALU);ALU 部件根据所接收到的标记,由复算纠错状态机 RCC_FSM 生成反馈信号作用于 itcm_ctrl 和 ifu 两个模块,从而控制下一条指令是否流出。本文主要对以下两种情况进行纠错:①涉及写寄存器文件的指令,包括 load 指令,以寄存器文件为目标地址的逻辑运算指令、算术运算指令等;②条件分支指令。对寄存器文件写操作的纠错主要由寄存器文件备份模块 Reg_bak 来完成;对条件分支指令的纠错主要由跳转状态位的备份模块 Bjp_bak 来完成。Reg_bak 模块根据备份数据与复算数据的比较结果,控制寄存器文件的写回模块 Wbck,进而决定是否对寄存器文件进行写操作。Bjp_bak 模块根据备份跳转位与复算跳转位的比较结果,控制分支指令的跳转模块 Isjp,进而决定是否冲刷掉已经推测预取到指令缓冲 Ins_buf 中的后续指令。

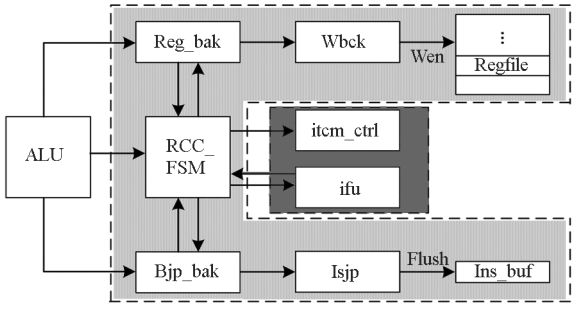

图 1　标记指令复算与纠错机制结构框图

Fig. 1　Architecture of the recomputation and correction mechanism for tagged instruction

1.2　标记指令复算机制的具体实现

在基于 RISC-V 指令集的开源内核“蜂鸟 e203”[16]上实现该标记指令复算与纠错机制,后文将以“蜂鸟 e203”内核为例,详细阐述该容错机制的具体实现过程。

“蜂鸟 e203”内核是一款超低功耗 2 级流水线处理器核,支持 RISC-V 指令集以及 RV32I/E/A/M/C/F/D 等指令子集的配置组合。其结构框架如图 2 所示。私有的指令紧耦合存储(Instruction Tightly Coupled Memory, ITCM)与数据紧耦合存储(Data Tightly Coupled Memory, DTCM)(未画出)可在分离存储指令与数据的同时提高性能。其中 ITCM SRAM 虽然主要用于存储指令,但也可以用于存放数据并用 load 和 store 指令进行访问。其电路实例化模块是图 2 中的 u_e203_srams,位宽为 64 位,大小为 64 KB。虽然 RISC-V 指令集里指令的最大宽度是 32 位,但为减小读取功耗与延时,“蜂鸟 e203”中的 itcm_ctrl 模块依然适时地从 ITCM 中读出一整行的 64 位数据。为解决指令位宽不匹配的问题,“蜂鸟 e203”里设计了 ift2icb 模块。该模块的主要功能是把读出的 64 位指令分解成 32 位并暂存。由于“蜂鸟 e203”是单发射、顺序执行、顺序提交的体系结构,而乘法、除法等多周期指令必然导致流水线的停滞,因此,itcm_ctrl 的另一个功能是根据当前处理器所处的状态,决定输出指令来源于上一条正在执行的指令还是接受缓冲区(Ins_buf)中新的指令。由于 RISC-V 指令集是变长指令集(同时支持 32 位指令和 16 位指令),所以取指令部件 ifetch 需要对指令进行预译码,判断当前 32 位指令数据是一条完整的 32 位指令还是两条 16 位指令,抑或是上一条 32 位指令的低 16 位与下一条 32 位指令的高 16 位等。ifetch 部件最终为执行部件 u_e203_exu 提供一个 32 位“指令束”ifu_o_ir。当待发射指令是 32 位指令时,ifu_o_ir 的高 16 位来源于一个 16 位 ifu_hi_ir 触发器中的新值,低 16 位来源于一个 16 位 ifu_lo_ir 触发器中的新值。而当待发射指令是 16 位指令时,此时 ifu_o_ir 的真实有效指令只有低 16 位,其高 16 位数据此时是无关值。因此,ifu_o_ir 的低 16 位来源于 ifu_lo_ir 触发器的新值,而其高 16 位用 ifu_hi_ir 触发器的旧值进行填充。上述整个阶段是“蜂鸟 e203”流水线的第一级,可归纳为“取指”,驱动数据主要是“指令流”。指令流在第一级流水线中经历了“64 位→32 位→16 位→32 位”的变化。“蜂鸟 e203”流水线的第二级可归纳为“执行”,主要负责指令的译码、执行、写回、提交等功能,驱动数据主要是“数据流”。在本文所实现的容错机制中,标记指令复算部分的改进主要在流水线第一级进行,而纠错部分的改进主要在流水线第二级进行。

图 2 中的阴影矩形与阴影连线分别表示了指令复算标记所存储的位置与传输的路径。本文为每 16 位指令数据添加了 1 位初始标记 init_tg。因此,“蜂鸟 e203”第一级流水线里的指令流带宽被相应调整成了“68 位→34 位→17 位→34 位”。在所有指令需要暂存的地方,也都对其存储部件进行了位宽扩展。比如:u_e203_srams 从 64 位扩展为 68 位,Ins_buf 从 32 位扩展成了 34 位,ifu_hi_ir/ifu_lo_ir从 16 位扩展成了 17 位。每 1 位复算标记与其所对应的 16 位指令数据同时进入每一个模块,经历相同的路径,缓冲相同的时间,最后同时由 ifetch 模块输出给译码模块 Decode。

在 ifetch 部件最终输出的 32 位“指令束”ifu_o_ir 中包含 2 位指令复算标记 rc_tg(tg_hi,tg_lo)。最终指令的复算情况与此二位复算标记的取值存在如表 1 所示的关系。

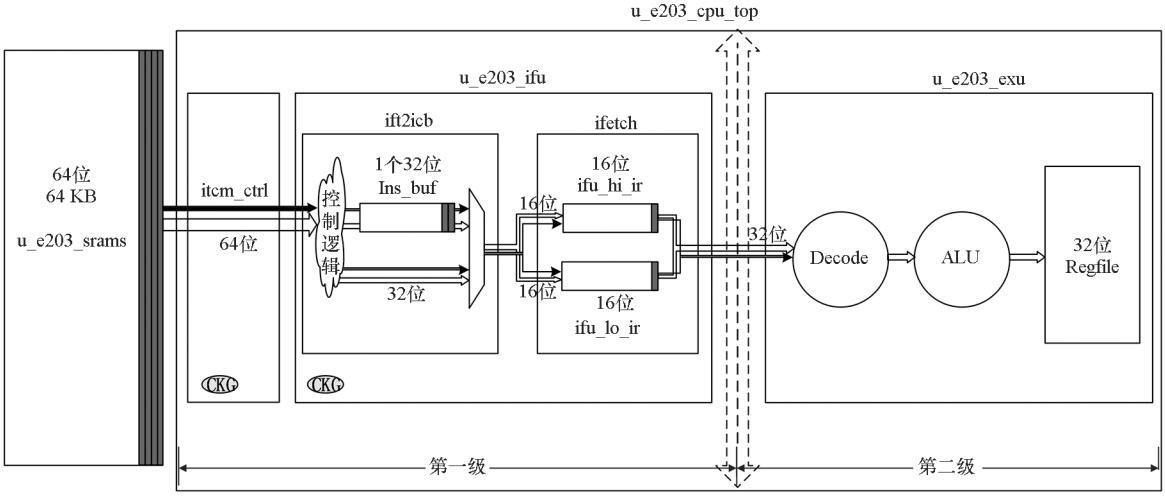

图 2 “蜂鸟 e203”内核指令流示意

Fig. 2 Instruction flow path for “Humming bird e203”

表 1　rc_tg 取值与指令复算的关系

Tab. 1　Value of rc_tg versus recomputation

工作模式	32 位指令	16 位指令
不复算	rc_tg = 00(10,01 不存在)	rc_tg = X0
复算	rc_tg = 11	rc_tg = X1

本文的指令初始标记 init_tg 是由编译器根据目标指令在应用程序中的关键性与易错性来赋值的。指令关键性越强,易错性越高,则相应的 init_tg就置为 1,反之则置为 0。所以对于 32 位指令不复算的情况,可以由编译器在编译时指定其两段 16 位的指令数据所对应的 init_tg = 00。由表 1 可以观察得到,无论是 32 位指令还是 16 位指令,需要进行复算时,其 tg_lo = 1;不需进行复算时,其 tg_lo = 0。所以只需根据 tg_lo 的值即可判定所有指令是否应该进行复算。当 tg_lo = 0 时,按照常规步骤进行指令的译码、执行、写回等操作后将发射下一条新的指令。当 tg_lo = 1 时,当前指令提交之后,复算纠错状态机 RCC_FSM 将停止从 ITCM 中读取新的指令,并保持 ifu_o_ir 所输出的当前指令不变,重新再执行一次。实施该种机制的硬件只需两个时钟门控单元,如图 2 所示的阴影椭圆 CKG。

这种实现方式有以下三个优点:①相比于在编译器级插入复算指令的做法,该方法只引入了 1 位的复算标记,极大节省了程序存储空间;②相比于从 ITCM 中重新读取一次复算指令的做法,避免了对程序计数器使用大量复杂的控制逻辑,同时也节省了将复算指令从 ITCM 读到 ift2icb 再到经 ifetch 整合并输出给执行部件所消耗的功耗与延时;③相比于引入一个额外的指令缓冲区来存储已发射并需要复算的指令的做法,本文节省了一个至少 32 位宽的指令缓冲区面积。

1.3　标记指令纠错机制的具体实现

如图 1 所示,标记指令的纠错机制主要由复算纠错状态机 RCC_FSM 和备份冗余模块(Reg_bak 与 Bjp_bak)构成。

复算纠错状态机 RCC_FSM 的主要功能是:根据指令的标记情况、提交情况决定是否执行复算;根据原指令结果与第一次复算结果的比较情况,决定是否进行第二次复算。其状态转移如图 3所示。

有 3 个信号决定 RCC_FSM 的状态转移:

1)Tag 信号:Tag = 1 时,表示被标记指令需要进行复算;Tag = 0 时则不进行复算。

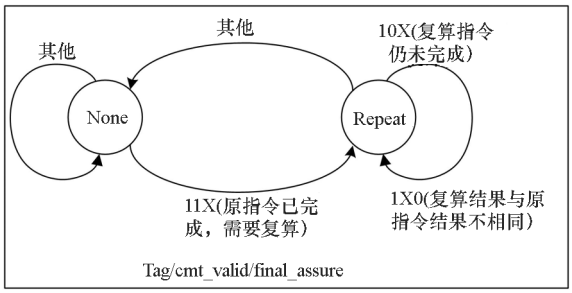

图 3　RCC_FSM 的状态转移

Fig. 3　State transition diagram of RCC_FSM

2)cmt_valid 信号:cmt_valid = 1 时表示原指令已经执行完毕,下一周期将取下一条指令。

3)final_assure 信号:final_assure = 0 时,表示复算结果与备份结果不相等。

默认情况下,RCC_FSM 处于 None 状态,即不进行复算状态。在此种状态下,只有当 Tag = 1(也即 tg_lo = 1)且 cmt_valid = 1 时,才会跳转到 Repeat 状态。其他情况下,都将保持 None 状态不变。

当正在进行复算或者已经进行过一次复算时,RCC_FSM 则处于 Repeat 状态。在此状态下,只有两种类型的输入能够使 RCC_FSM 继续保持在 Repeat 状态:

1)Tag = 1 且 final_assure = 0。此种情况的含义是:第一次复算已经完成并已与原始指令的结果进行了比较,发现连续两次执行的同一条指令结果不相同,因此需要对该指令进行第二次复算,通过三模冗余的方式进行纠错处理。

2)Tag = 1 且 cmt_valid = 0。此种情况常发生在被复算的指令是多周期指令时。

其他输入模式下,状态机都将跳回到 None 状态。

Reg_bak 与 Bjp_bak 模块其实都是备份冗余模块 rddt 的实例化,只是各自备份的数据位宽不同而已。rddt 模块主要由两个数据缓存单元、两个比较器、一个计数器、一个 3 路选择器,一个或门构成。因为 Reg_bak 需要备份将要写进寄存器文件的 32 位数据,所以其数据位宽是 32 位;而 Bjp_bak 只需要备份指示条件分支指令是否跳转的 1 个状态位,所以其数据宽度是 1 位。

备份冗余模块运行过程的时序情况如图 4 所示。若目标指令不是复算指令(即图 4 所示的单周期指令 Inst1),写使能信号 Wen = 0、比较使能信号 CP_en = 0,备份冗余模块并不对备份数据信号 Din 进行暂存和比较。计数使能信号 Cnt_en 一直等于 0,因此提交确认信号 cmt_assure 一直

为 1。只要该指令执行完毕,便可立即提交。

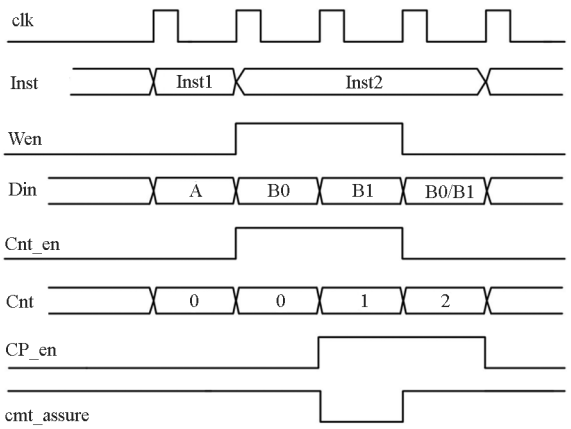

图 4 rddt 模块时序图

Fig. 4 Timing diagram of rddt

若目标指令是复算指令(即图 4 所示的单周期指令 Inst2),则:①在原指令执行期间,内部计数值 Cnt = 0, CP_en = 0, 因此 cmt_assure = 1。Wen = 1,所以 Din 将原始指令执行的结果存入第一级缓冲中。②在第一次复算期间,CP_en 由 0 变为 1, Cnt 此时由 0 变为 1,因此 cmt_assure = CP1。若 CP1 = 1,说明复算结果与备份数据相等,从而可以顺利提交该指令;若 CP1 = 0,说明复算结果与备份数据不相等,需要继续执行第二次复算。此时 Wen = 1,原始执行结果被传到第二级缓存,第一次复算结果被存进第一级缓存。③在第二次复算期间,Cnt 由 1 变为 2。第二次复算的结果将分别与原始结果和第一次复算结果进行比较,并把 CP1|CP2 赋给 cmt_assure。

2 理论分析

本文提出的标记指令复算纠错机制主要是针对由宇宙辐射所导致的数据流瞬时错误。因此,对于某一个特定的指令,在不同的时刻发生单粒子翻转效应的概率相等。对于某条执行周期为 n 的指令,假设其每个周期中数据流出错的概率为 p。

在不复算的情况下,该指令执行正确的概率:

$$P_{\mathrm{r}} = (1-p)^n \tag{1}$$

在不复算的情况下,该指令执行错误的概率:

$$P_{\mathrm{f}} = 1 - P_{\mathrm{r}} = 1 - (1-p)^n \tag{2}$$

在复算的情况下,两次均执行正确的概率:

$$P_{\mathrm{rr}} = (1-p)^{2n} \tag{3}$$

在复算的情况下,一次执行错误、一次执行正确的概率:

$$P_{\mathrm{fr}} + P_{\mathrm{rf}} = 2[1-(1-p)^n](1-p)^n \tag{4}$$

在复算的情况下,两次均执行错误的概率:

$$P_{\mathrm{ff}} = [1-(1-p)^n]^2 \tag{5}$$

在不利用中断程序辅助错误处理的情况下,采取本文提出的标记指令复算纠错机制,其最终能得到正确结果的概率为:

$$\begin{aligned} P_{\mathrm{new}} &= P_{\mathrm{rfr}} + P_{\mathrm{frr}} + P_{\mathrm{rr}} \\ &= 2[1-(1-p)^n](1-p)^{2n} + (1-p)^{2n} \end{aligned} \tag{6}$$

在不复算也不利用中断程序处理错误的情况下,其最终能得到正确结果的概率:

$$P_{\mathrm{old}} = P_{\mathrm{r}} = (1-p)^n \tag{7}$$

所以指令 i 正确执行的概率增加了:

$$\begin{aligned} \Delta P &= P_{\mathrm{new}} - P_{\mathrm{old}} = 2[1-(1-p)^n](1-p)^{2n} + \\ &\quad (1-p)^{2n} - (1-p)^n \\ &= -2t^3 + 3t^2 - t = f_1(t) \end{aligned} \tag{8}$$

其中,$t = (1-p)^n$。

$f_1(t)_{\max} = f_1((3+\sqrt{3})/6) = 0.096\ 2$,即利用本文提出的标记指令复算纠错机制,在没有中断程序辅助错误处理的情况下,最多可将单一指令的执行正确率提高 9.62%。

若与中断机制相结合,则当三次计算结果均不一样时,可以通过中断程序报错,让操作系统进行相应的错误处理。此种情况的概率为:

$$\begin{aligned} P_{\mathrm{int}} &= 2(1-p)^n[1-(1-p)^n] - \\ &\quad 2(1-p)^{2n}[1-(1-p)^n] - \\ &\quad 2(1-p)^n[1-(1-p)^n]\cdot q^n \end{aligned} \tag{9}$$

其中:$2(1-p)^n[1-(1-p)^n]$ 表示的是标记指令被执行 3 次的概率;$2(1-p)^{2n}[1-(1-p)^n]$ 表示的是 3 次执行中 2 次正确、1 次错误的概率;$2(1-p)^n[1-(1-p)^n]\cdot q^n$ 表示的是 3 次执行中 2 次错误、1 次正确,且错误结果相同的概率。q 表示出现错误且错误结果与之前的错误结果相同的概率,其数值远远小于 p。比如,对某个深度为 32、位宽为 32 的寄存器文件而言,若寄存器 R0 某个数据位发生单粒子瞬时故障的概率为 p,则对于指定位(如最高位)发生单粒子瞬时故障的概率 $q = p/32$。

因此,可以忽略 $2(1-p)^n[1-(1-p)^n]\cdot q^n$ 项,则 $P_{\mathrm{int}} \approx 2(1-p)^n[1-(1-p)^n][1-(1-p)^n]$。

同理,设 $t = (1-p)^n \in (0,1)$,则:

$$P_{\mathrm{int}} \approx f_2(t) = 2(t^3 - 2t^2 + t) \tag{10}$$

所以 $P_{\mathrm{int}} + \Delta P \approx f(t) = -t^2 + t$。

$f(t)_{\max} = f(1/2) = 0.25$,即:若与中断错误处理程序配合使用,标记指令复算纠错机制可把单

个指令的检错能力最多提高 25%。

对于一个包含 s 条指令的程序,假设没有复算纠错机制,每一条指令执行正确的概率是 $t_i = (1-p_i)^{n_i}$,其中 p_i 是第 i 条指令发生错误的概率,n_i 是执行第 i 条指令所需的时钟周期。则整个程序最终能正确执行结束的概率是:

$$P_{\text{ori}} = \prod_{i=0}^{i=s} t_i \tag{11}$$

若采用指令复算纠错机制,则整个程序最终能够正确执行结束的概率是:

$$P_{\text{rcc}} = \prod_{i=0}^{i=s} (t_i + \Delta t_i) \tag{12}$$

所以对于整个系统而言,最终目标程序的容错能力提高了:

$$\Delta P_{\text{total}} = P_{\text{rcc}} - P_{\text{ori}} = \prod_{i=0}^{i=s} (t_i + \Delta t_i) - \prod_{i=0}^{i=s} t_i \tag{13}$$

以本文所做的实验为例,假设整个程序中共 $s = 160$ 条指令,所有指令都是涉及寄存器文件操作的单周期指令,任一时刻寄存器文件中的某个寄存器发生故障的概率 $p_i = 1/32$,则对于每一条指令,其能正确执行的概率 $t_i = (1-1/32)^1 = 31/32$,代入式(8)可得 $\Delta t_i = \Delta P_i = 2.83\%$。

在没有复算纠错机制时,整个程序最终得以正确执行的概率 $P_{\text{ori}} = (31/32)^{160} = 0.62\%$。

若采用复算纠错机制(不考虑中断辅助),程序正确执行的概率为:

$$P_{\text{rcc}} = (31/32 + 2.83\%)^{160} = 62.33\% \tag{14}$$

所以对于整个程序而言,本文提出的复算纠错机制将系统可靠性提高了:

$$\Delta P_{\text{total}} = P_{\text{rcc}} - P_{\text{ori}} = 61.71\% \tag{15}$$

3　实验结果

本文用 6 个来自 riscv-test 项目中的基准测试集对复算纠错机制进行测试。riscv-test 是由 RISC-V 架构开发者在 Github 平台上维护的公共项目,其中包含一些测试处理器是否符合 RISC-V 指令集架构定义的测试程序,它们均由汇编语言编写。本文的重点是从硬件上建立指令复算和纠错机制,而非在软件上优化编译器使之能根据指令的重要性和易错性自动添加复算标记。所以在整个实验过程中,复算标记是在 ITCM 初始化文件中手动添加的。

本文对逻辑运算测试集 or、and、xor 和算数运算测试集 add、mul、div 分别进行了 100 次容错实验。因为在上述 6 个测试集中也包含有分支跳转指令,因此通过对其中的分支跳转指令添加复算标记,也可以顺便完成对 Bjp_bak 纠错模块的测试。其中故障注入是通过在测试文件 tb_top. v 中利用 force 和 release 语句,在指定时刻对随机信号产生随机干扰完成的。为降低故障注入的复杂度并便于调试,本文在每个时钟周期上升沿之后,只针对 32 个通用寄存器文件进行随机故障注入,每次故障持续的时间都最长不超过一个时钟周期。虽然只针对寄存器进行了故障注入,但是该方法可以模拟整个寄存器文件读、写以及运算执行过程中所覆盖的硬件的全部故障,即达到了本文想要检测并纠正数据流瞬时故障的初衷。因此,对于每个时钟周期,每个寄存器文件出错的概率 $p = 1/32$。每个“蜂鸟 e203”寄存器文件的数据位宽是 32 位,因此对于指定寄存器的指定位发生错误的概率 $q = p/32$。

图 5 展示了在 100 次随机故障注入的实验中,引入标记指令复算与纠错机制前后各测试集错误执行的次数。其中,空白条 BF 表示未引入复算纠错机制前,测试程序能够顺利完成但得到的是错误结果的次数;灰色填充条 BD 表示未引入复算纠错机制前,测试程序由于数据流错误,导致其中某些分支跳转指令跳错方向,从而陷入死循环无法正常退出的次数,即伪分支错误;绿色填充条 AF 表示引入复算纠错机制后,测试程序能够顺利完成但得到的是错误结果的次数;红色填充条 AD 表示引入复算纠错机制后,测试程序由于数据流错误,其中某些分支跳转指令跳错方向,从而陷入死循环无法正常退出的次数。

从 or、and、xor 的实验数据中可以看出,未改进之前,逻辑运算内在的容错能力十分微弱,在 100 次随机故障注入的实验中,没有一次能够正确运行完毕。对于算数运算测试集 add、mul、div,在未引入复算纠错机制之前,出错概率也为 100%。当引入复算纠错机制和中断处理机制之后,从图 5 可以看出错误执行概率大大减小。其中,执行完毕但得到错误结果的概率(彩色条长度相对于灰白条长度)平均减小了约 89.82%,说明标记指令复算与纠错机制能够有效地检查并校正数据流错误。另外,伪分支错误的概率(红色填充条长度相对于灰色填充条长度)平均减小了 40%。

图 6 展示了在 100 次故障注入实验中,在引入本文所提出的标记指令复算与纠错机制之后,程序最终得以正确执行的次数。从中可以看出,在本文所提出的标记指令复算与纠错机制和中断

处理程序的配合下,6 个测试集平均有 86.67% 的概率能够正确完成程序的指定任务。对于单周期指令 or、and、xor 测试集而言,实验数据表明在只有复算纠错机制的情况下,其正确执行的概率平均为[(65 + 77 + 61)/300] × 100% = 67.67%,与第 2 节理论分析中得出的 62.33% 相吻合。

对于 mul、div 等多周期指令,随着指令周期数 n 的增大,单条指令受到随机故障影响的概率也越大。因此单条 mul、div 指令正确执行的概率将变小,从而导致通过两次复算进行纠错的成功率也将大大降低。尽管如此,本文提出的复算与纠错机制可以根据原始计算与两次复算结果的不同,检查出数据流错误并报中断程序进行系统级纠错,从而成功保证程序得以正常执行。从图 6 的实验数据可以发现,对于 mul 测试集和 div 测试集,其通过两次复算成功纠错的次数分别只有 27 次和 1 次,但分别有 62 次和 99 次通过报中断处理程序得以纠正错误,保证程序的正确执行。

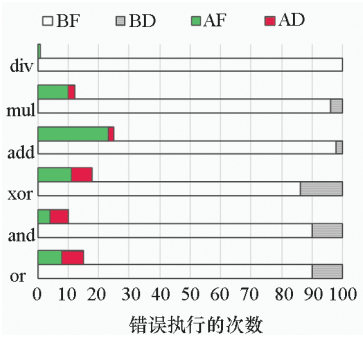

图 5 程序执行错误的次数

Fig. 5 The number of program failures

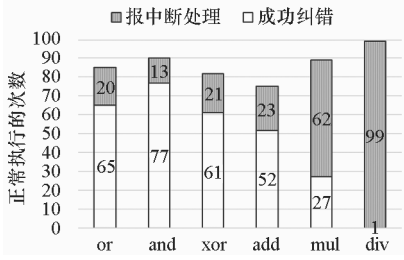

图 6 程序正常执行完毕的次数

Fig. 6 The number of program successes

4 结论

本文提出了一种标记指令复算与纠错机制。通过编译器对每 16 位指令数据插入 1 位复算标记,并引入一个复算纠错状态机 RCC_FSM 来监控指令流中的标记,可以对标记为复算的指令进行复算。通过引入 Reg_bak 和 Bjp_bak 两个备份纠错模块,若写入寄存器文件的数据或者分支跳转的方向在第一次复算时的结果与原始运算结果不同,可自动进行第二次复算。根据少数服从多数的原则,可自动纠正错误从而得到正确结果。若三次运算结果均各不相同,则可自动触发中断机制,交由系统级容错软件进行处理。

所提出的标记指令复算与纠错机制的优点有:

1)对软件程序所需的内存开销较小。因为没有直接插入完整的冗余指令,而只是插入了是否复算的标记位。

2)对芯片的硬件开销较小。采用时钟门控的做法,避免了在原有的硬件基础上增加复杂的控制逻辑。整个内核只需引入一个复算纠错状态机 RCC_FSM。可根据具体的需要,针对易错部件灵活地插入备份纠错模块。

3)对程序执行时间的开销较小。在程序复算的过程中免去了重新取指的过程,可以直接重新发射。而且仅在第一次复算结果与原始结果不同时,才进行第二次复算。

然而,本文的工作也仍存在不足:

1)虽然用额外添加标记位的方式来表达指令复算的需求可以节省存储空间,但同时也要求软件工程师能分析出易出错的指令,抑或是将容错能力纳入编译器中,使编译器能够对易错指令进行自动标记。下一步研究将对 RISC-V 编译器进行优化和改进,使之能自动分析出指令的易错性和重要性。

2)虽然二次复算可以避免传统硬件三模冗余结构带来的面积与功耗开销,但是对于出错率高、执行周期长的指令而言,可能会降低整体程序的性能,因此,本文提出的结构可能更适合于出错率相对较低的环境。

3)最终实验时,只选取了功能单一的测试向量并且最终的故障注入实验只针对寄存器文件中的 32 个通用寄存器进行了等概率的随机故障注入,其主要原因是缺乏具有标记容错能力的编译器与真实的辐射实验环境。事实上,不同指令所调用的部件、执行周期长短都各不相同,因此其真实的出错概率并不一样。后续将研究建立更精确的故障注入模型,利用改进后的编译器产生更多贴近真实情况的程序,从而更好地模拟复算纠错机制对系统容错能力的提升。

参考文献(References)

[1] 王长河. 单粒子效应对卫星空间运行可靠性影响[J]. 半

导体情报, 1998, 35(1): 1 – 8.
WANG Changhe. The influence with reliability of motional satellite by the single-event phenomena[J]. Semiconductor Information, 1998, 35(1): 1 – 8. (in Chinese)

[2] Sosnowski J. Transient fault tolerance in digital systems[J]. IEEE Micro, 1994, 14(1): 24 – 35.

[3] Clark J A, Pradhan D K. Fault injection: a method for validating computer-system dependability[J]. Computer, 1995, 28(6): 47 – 56.

[4] Normand E. Single-event effects in avionics[J]. IEEE Transactions on Nuclear Science, 1996, 43(2): 461 – 474.

[5] Oh N, Shirvani P P, McCluskey E J. Control-flow checking by software signatures[J]. IEEE Transactions on Reliability, 2002, 51(1): 111 – 122.

[6] Khosravi F, Farbeh H, Fazeli M, et al. Low cost concurrent error detection for on-chip memory based embedded processors[C]//Proceedings of IFIP International Conference on Embedded and Ubiquitous Computing. IEEE, 2011.

[7] Du BY, Reorda M S, Sterpone L, et al. Online test of control flow errors: a new debug interface-based approach[J]. IEEE Transactions on Computers, 2016, 65(6): 1846 – 1855.

[8] Avizienis A. The N-version approach to fault-tolerant software[J]. IEEE Transactions on Software Engineering, 2006, SE – 11(12): 1491 – 1501.

[9] Oh N, Shirvani P P, McCluskey E J. Error detection by duplicated instructions in super-scalar processors[J]. IEEE Transactions on Reliability, 2002, 51(1): 63 – 75.

[10] Oh N, Mitra S, McCluskey E J. ED4I: error detection by diverse data and duplicated instructions[J]. IEEE Transactions on Computers, 2002, 51(2): 180 – 199.

[11] Reis G A, Chang J, Vachharajani N, et al. SWIFT: software implemented fault tolerance[C]//Proceedings of International Symposium on Code Generation and Optimization. IEEE, 2005: 243 – 254.

[12] Lindoso A, Entrena L, García-Valderas M, et al. A hybrid fault-tolerant LEON3 soft core processor implemented in low-end SRAM FPGA[J]. IEEE Transactions on Nuclear Science, 2017, 64(1): 374 – 381.

[13] Sengupta A, Kachave D. Spatial and temporal redundancy for transient fault tolerant datapath[J]. IEEE Transactions on Aerospace and Electronic Systems, 2018, 54(3): 1168 – 1183.

[14] Xu C, Holzemer M, Kaul M, et al. On fault tolerance for distributed iterative dataflow processing[J]. IEEE Transactions on Knowledge and Data Engineering, 2017, 29(8): 1709 – 1722.

[15] van Kampenhout R, Stuijk S, Goossens K. Fault-tolerant deployment of dataflow applications using virtual processors[C]//Proceedings of 21st Euromicro Conference on Digital System Design (DSD). IEEE Computer Society, 2018: 77 – 84.

[16] Hu Z B. Hummingbird E203 opensource processor core[EB/OL]. (2018 – 09 – 27) [2019 – 04 – 13]. https://github.com/SI-RISCV/e200_opensource.