ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.17, No.12, December 2006, pp.2425−2437
DOI: 10.1360/jos172425
© 2006 by Journal of Software. All rights reserved.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

高性能低功耗的容错编译技术:错误流压缩算法*

高 珑+, 杨学军

(国防科学技术大学 计算机学院,湖南 长沙 410073)

Efficient Fault Tolerant Compilation: Compress Error Flow to Reduce Power and Enhance Performance

GAO Long+, YANG Xue-Jun

(School of Computer, National University of Defense Technology, Changsha 410073, China)

+ Corresponding author: Phn: +86-731-4575808, E-mail: imgaolong@yahoo.com.cn

Gao L, Yang XJ. Efficient fault tolerant compilation: Compress error flow to reduce power and enhance performance. *Journal of Software*, 2006,17(12):2425−2437. http://www.jos.org.cn/1000-9825/17/2425.htm

Abstract: In many reliability-critical applications, computers are required to have higher performance, lower power dissipation and fault tolerance simultaneously. Traditional software fault tolerance uses a great deal of branch instructions to detect errors, thus brings great overhead in both performance and power dissipation. In this paper, an error flow model is suggested, and it is used to explain the algorithm of error flow compressing. In error flow compressing algorithm, branch instructions are reduced greatly, while total instructions remain the same. The simulated results on Wattch of FFT benchmark from project StreamIT show that compared with the traditional EDDI error detection algorithm, the EFC can reduce total branch instructions by over 24%, improve IPC by over 12%, and at the same time, reduce the power dissipation by nearly 5%, at loop parameter $n=2^{25}$. Further reasoning shows that the reduction of branch instructions can be as much as over 43% when there are 8 store instructions in the innermost iteration.
Key words: SIHFT (software implemented hardware fault tolerance); COTS; error flow model; error flow compressing algorithm; branch instruction; high performance; low power dissipation

摘 要: 在许多关键应用中,计算机的高性能、低功耗和高可靠性是必须同时满足的要求.传统的软件容错技术频繁使用和比较分支指令检测错误,带来了巨大的性能和功耗的开销.提出了基于计算数据流模型的错误流模型,并设计了错误流压缩算法.在错误流压缩算法中,利用附加计算压缩了错误流的直径,显著减少了分支指令的数量,而总指令数不变.针对 StreamIT 提供的快速傅立叶变换测试程序,采用 Wattch 对错误流压缩算法进行模拟测试.实验结果表明,当循环参数 $n=2^{25}$ 时,与传统的 EDDI 算法相比,使用错误流压缩算法可减少分支指令 24%以上,IPC 提高超过 12%,同时,功耗减少了将近 5%.给出的推算表明:在该实验中,如果内层循环体的存储指令数量为 8,分支指令的减少可以达到 43%以上.
关键词: 软件容错;COTS;错误流模型;错误流压缩算法;分支指令;高性能;低功耗

---

* Supported by the National High-Tech Research and Development Plan of China under Grant No.2002AA1Z2101 (国家高技术研究发展计划(863))

Received 2005-10-08; Accepted 2006-02-23

© 中国科学院软件研究所 http://www.jos.org.cn

中图法分类号: TP314 文献标识码: A

随着全球深空探测活动的复兴,尤其是我国若干空间探测计划的启动[1],对航天探测器的自主信息处理能力提出了更高的要求.为了智能应对各种复杂情况和实时处理空间信息,宇航用计算机必须具备高性能.为了能够在恶劣的外层空间中长时间地正常工作,宇航用计算机必须同时具备低功耗和高可靠性.

软件容错(software implemented hardware fault tolerance,简称SIHFT)技术[2,3]在COTS部件上使用冗余计算并比较结果的方法来判断错误,与硬件容错技术相比,具有成本低廉、开发快捷、高效灵活等优点.但是,分支指令在软件容错中的大量使用给高性能处理器正常发挥超长流水线性能带来了十分剧烈的损害.对分支指令的错误预测不仅会清空全部流水线,造成巨大的性能开销,而且处理器花费在乱序执行这些无用指令上的能量平均高达 28%[4].同时,分支指令频繁访问巨大的分支预测缓冲(BPB)和分支目标缓冲(BTB),消耗了处理器大约7%~10%的功耗[5].

软件容错算法EDDI[3,6]是美国Stanford大学CRC实验室的ARGOS项目的核心算法之一.EDDI算法通过在每一条存储指令之前比较冗余的计算结果来判断是否出现了错误,但其却带来了平均170%的性能开销[7],其中一大部分是由分支指令带来的.其他软件容错技术[8,9]无不依赖于大量的比较和分支指令来检测错误.例如,源到源容错编译算法[9]的空间开销为2.9倍,而时间开销高达2.6倍.

如何在提高软件容错算法性能的同时又能降低功耗,目前还没有行之有效的方法.错误在程序中的传播模型,也尚未有人涉足.本文试图对上述问题给出一个解决方案.本文第1节介绍软件容错的基本概念和原理.第2节介绍相关工作.第3节描述错误流模型的基本框架.第4节给出错误流的压缩算法.第5节给出实验设计和结果分析.第6节是结论.第7节描述未来的工作.

1 软件容错技术简介

1.1 故障模型

故障(fault)是软、硬件产生不符合其设计目的的差异,当这种差异导致计算机偏离正确的状态时,就产生了错误(error)[10].故障模型如图1所示.瞬时故障(transient fault)通常是由于带电重粒子轰击集成电路时造成瞬时充放电而导致的存储单元的逻辑状态翻转[11].这种故障原因被称为SEU(single event upset)[12].瞬时故障大约是永久性故障数量的100倍以上,占故障的绝大多数.通常我们认为,在瞬时故障发生时,错误立即发生[10].当发生瞬时故障的器件被重置后,故障都可以自动消失.

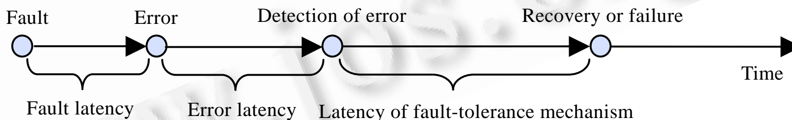

Fig.1 Fault model

图1 故障模型

1.2 冗余计算和比较

大多数软件错误检测方法采用3个步骤检测SEU导致的错误:输入数据复制、数据冗余处理、输出数据比较,如图2所示.输出数据比较的结果如果存在差异,则证明至少一路数据中存在错误.检测到错误后,由硬件或者软件发出信号,通常情况下,当前的任务被重启.

© 中国科学院软件研究所 http://www.jos.org.cn

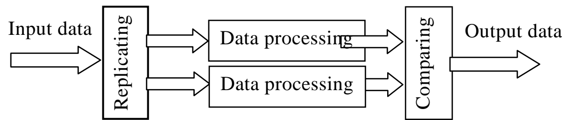

Fig.2 Principles of fault tolerance system
图 2 容错系统原理图

2 相关工作和算法概要

在计算机诞生之初,Neumann 和 Shannon 都研究过如何用不可靠的部件搭建可靠的计算机[13],其中包括 Neumann 提出的 TMR[13]思想.随着计算机的发展,有些存储模块中固化了 ECC(error correcting code)[14]进行硬件数据校验,但这样也会降低存储器的访问速度.随后出现了系统级容错平台,如 Stratus[15]和 NonStop Himalaya[16]等.Himalaya 通过实时地比较冗余双处理器每一个管脚的信号来判断处理器在运算过程中是否出现了错误.Watchdog[17]则是用来检测总线上数据正确性的专用处理器.

随着现代体系结构中的并行性越来越高,也出现了利用并行体系结构进行冗余计算的容错方法,比如 CRTR[18]和 SRTR[19]等.在这些方法中,由于程序中的并行性不可能被完全开发,所以加入相互间相关性很小的冗余容错处理,可以更充分地利用体系结构中的并行处理能力.然而,这些容错方法需要对原体系结构进行修改,成本和开发周期都相应地增加.其中,CRTR 和 SRTR 中使用 DBCE 和 DDBCE 策略检测前后相接的几条微指令之间的数据依赖关系,并由此仅仅比较依赖关系链中最后一条微指令中的数据,认为如果该数据正确,则该依赖链上的所有微指令都已经执行正确.这样减少了比较的次数,提高了性能.但是,由于受到硬件成本和空间的限制,DBCE 和 DDBCE 只能检测很少的微指令之间的依赖关系,大约为几条~十几条.

早期的软件容错技术也有所发展.NVP(N version programming)[20]和 Rocovery Block[21]通过投票机制来选出多个冗余结果中最可能正确的结果.ABFT(algorithm based fault tolerance)[8]提出,在算法设计中,程序员应该从编写程序的一开始就加入容错处理.但是,这样也会给程序员增加额外的负担.源到源编译[9]方法采用转换 C 程序的方法,将程序员书写的源程序自动预处理为具有容错功能的 C 程序.然而在源到源编译中,高级语言的每一条语句都被复制并且逐条语句地进行比较,所以这种方法的效率很低.

随着性价比较高的商用 COTS 部件的出现并广泛应用,在没有进行专门容错保障的 COTS 部件上,通过软件实现容错的技术逐渐成为主流.McCluskey 领导的 Stanford 大学的 ARGOS 项目,在卫星上进行了广泛的基于 COTS 部件的软件容错的对比实验.他提出的 EDDI[3,6]和 ED4I[2]算法通过编译的方法实现程序内部指令级的软件容错.尽管仍然存在平均 170%的性能开销,但仍比同时代经过硬件容错加固的专用部件的性能要高一个数量级[7].美国国家宇航局喷气推进实验室进行的许多基于 COTS 的软件容错实验[22,23]也得到了类似的结论.

3 基于计算数据流模型的错误流模型

在可靠性工程[24]当中,系统可靠性的基本模型有串联模型、并联模型、表决模型、储备模型、一般网络模型等.主要的分析方法有故障树分析等.但是,这些模型和方法都是针对机械电子、电力网等硬件大系统的,无法描述和解决计算机程序中出现的可靠性问题.

传统的数据流模型可以用来分析程序的结构.但是,在对纯粹的数据进行分析和比较时就显得有些力不从心了.因为在传统的数据流中,分析的基本单位是基本块,它由多条或者一条指令组成.指令在传统的数据流分析中是不可再分的最小单位,我们不可能直接用它来分析比指令更小的成分——数据.但是,容错算法中最终比较的却是数据而不是指令.

在下面几节中,我们首先提出一种新型的描述计算活动的计算数据流模型,进而在这种计算数据流模型上建立了错误流模型,用来描述错误在计算活动中的传播规律.在本文后面的实验中,我们的错误流模型成功地描述了一种新的高效、低功耗的编译容错算法.

© 中国科学院软件研究所 http://www.jos.org.cn

3.1 计算数据流模型

我们把每一条指令抽象为两部分内容:原子数据和原子计算.

定义 **1**. 原子数据是指由特定的硬件器件记录的、在指令运算中不可再分的数据.

原子数据包括在指令的操作数段能够出现的各种寻址类型的数据.比如,参与浮点计算的浮点寄存器中的数据、Load 指令的地址指向的内存数据等.

某一个寄存器中的数据在整个运算过程中可能被多次重写,那么它所代表的数据也就不一样.在 SSA (static single-assignment)[25,26]中,被不同赋值语句赋值的程序变量被赋予不同的下标,最后通过选择函数来为其确定终值.然而,SSA 并没有摆脱以指令为基本单位的定位并且主要对高级语言的变量分析.而我们需要的是一种能够直接观察到数据(如寄存器)的新模型.为了达到这一点,我们引入了原子数据.同时,为了标识不同原子数据在时间上的唯一性,我们为存储器件的名称增加了上标.对于寄存器而言,我们规定:在指令执行的过程中,如果寄存器的内容被重写,那么寄存器标志的上标加 1.例如,浮点寄存器 f_1 中的原子数据 f_1^m 被重写,则新的原子数据标志为 f_1^{m+1},分析过程中初始的原子数据上标为 0.

对于内存中的原子数据,我们可以按上述方法标明它的实际地址,也可以仅仅指出它们是从内存中取出的,而不具体指明地址.如图 3(b)所示,使用双横线表示从内存中存取.图中指令为 SimpleScalar/PISA 汇编指令[27].在有些容错算法中,内存中的数据也是冗余的,在正常情况下,它们的值相等,如果不相等,就表明出现了错误.

定义 **2**. 原子计算是由原子数据参与的、由一条指令完成的计算.

n 元原子计算 g 就是原子数据集合 V 上的一个 n 元函数,$g:V^n\rightarrow V$.对于参与原子计算的原子数据,我们把被读取的原子数据称为源,而把写入的原子数据称为目标.对于某一次原子计算,我们从源到目标画一条带有方向的弧,表示数据之间的计算关系,如图 3(a)所示.

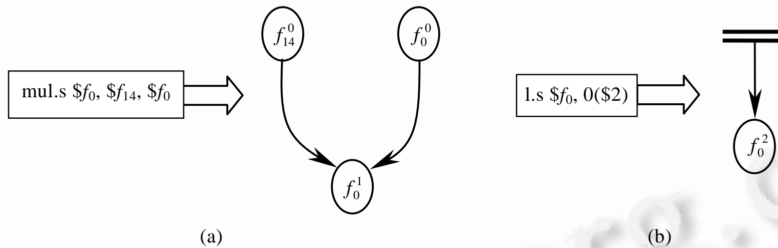

Fig.3 Spot, path and basic computing

图 3 原子数据和原子计算

定义 **3**. 如果 g 是原子数据集合 V 上的 n 元原子计算,则计算关系的集合 E 是一个与 g 以及 V 相关的二元关系,$E=\{\langle x,y\rangle|x,y\in V,y=g(\langle x_1,x_2,\ldots,x_n\rangle),x=x_i,1<i<n\}$.

具体的指令类型可以是算术计算、Load/Store、移动、比较、类型转换等指令.我们可以省略不关心的指令类型.

对于分支指令,我们可以认为数据分别以分支条件发生和不发生的概率及不同分支路径上的其他数据产生计算关系.这时的计算关系具有分支概率加权.如果 E 是所有计算关系的集合,则分支概率函数 P 是从 E 到[0,1]上的一个映射 $P:E\rightarrow[0,1]$.如果与计算关系 e 相关的指令一定执行,则 $P(e)=1$;如果存在分支指令,则 $P(e)$的值取决于分支的概率.在必要时,我们可以在表示计算关系的弧上标注这种概率.图 4 列出了左侧程序中有关浮点计算的计算关系和分支概率函数.在编译时,我们可以静态预测分支成功和失败的概率.使用较为准确的静态预测算法可以提高预测的成功率,比如使用 SCBP[28],Profile 指导的预测技术[29]和值域传播[30]等方法.

这样,计算活动基本上都可以表示为以原子数据为节点、以计算关系为边的图了.

© 中国科学院软件研究所 http://www.jos.org.cn

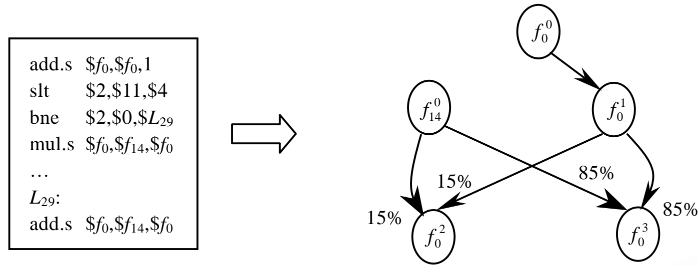

Fig.4 Computational data flow graph with conditional branch

图 4 包含分支指令的计算数据流图

定义 **4**. 如果 V 是原子数据的集合,E 是与 V 相关的计算关系的集合,我们就称以 V 为节点、以 E 为边的图 $G(V,E)$为计算数据流图.

当我们不标明存储数据的具体位置时,计算数据流图是一种有向无环图.

3.2 错误流模型

我们根据错误的原因把原子数据 v 的错误 $A(v)$分成两类:固有错误 $A_L(v)$和感染错误 $A_T(v)$.

固有错误 $A_L(v)$是由于器件 v 不可靠引起的,属于偶然失效期以内的可靠性问题[24].对于瞬时错误来说,主要决定于物理环境中的致错因素和器件 v 对于这些致错因素的敏感程度.屏蔽恶劣的环境或者采用较为可靠的器件都可以降低这种固有错误发生的概率.发生固有错误的概率 $P(A_L)$可以通过可致错粒子流的密度在时间和空间上的积分求出,这就是器件 v 的累积失效概率.由于每个原子数据仅存续很短的时间,大约就是相关的原子计算持续的一个或者几个时钟周期;并且也仅占据很小的空间,仅为几个字节.所以我们可以使用式(1)来估计固有错误概率 $P(A_L)$:

$$P(A_L)=D(t,a)\cdot \Delta t\cdot \Delta a \tag{1}$$

其中,$D(t,a)$是可致错粒子流的密度函数在时间 t 和空间 a 处的值.IBM 在 20 世纪 70 年代在地面和空间中进行了广泛的实验,在各种不同的时间和地点测量了不同尺寸电子器件的 SER(soft error rate)[11],为我们计算固有错误概率提供了大量的数据.

对于任何原子数据 u,如果存在从 u 指向 v 的计算关系 $e=\langle u,v\rangle$,那么,u 的错误就有可能导致 v 产生感染错误 $A_T(v)$.也就是说,如果参与计算 v 的原子数据错误,那么 v 也很可能错误.这时,沿着计算关系 $e=\langle u,v\rangle$传染到 v 的错误就是 v 的一个感染错误分量 $A_T(v)(e)$.v 的感染错误 $A_T(v)$就是 v 的所有的感染错误分量的总称.只要存在计算关系 $e=\langle u,v\rangle$,我们就称 e 的源 u 是 v 的一个传染源.感染错误和本地的固有错误无关,而与传染源的错误有关.

我们可以给出下面两个假设:

假设 **1**. 如果计算关系 e 的源错误,那么 e 的目的也很可能错误.

我们把假设 1 中的这种可能性定义为计算关系 e 传播错误的传染概率函数 $f:E\rightarrow[0,1]$.所以与 $e=\langle u,v\rangle$相关的 v 的感染错误分量 $A_T(v)(e)$发生的概率可以用式(2)计算:

$$P(A_T(v)(e))=P(A(u))\cdot f(e) \tag{2}$$

假设 **2**.

1) v 不存在感染错误 $A_T(v)$的充要条件是:不存在任何有关 e 的感染错误分量 $A_T(v)(e)$;

2) v 不存在错误 $A(v)$的充要条件是:既不存在固有错误 $A_L(v)$,也不存在感染错误 $A_T(v)$.

由假设 2 我们可以得到

$$P(A_T(v))=1-\prod_{e=\langle u,v\rangle}^{e\in E}(1-P(A_T(v)(e))) \tag{3}$$

$$P(A(v))=1-(1-P(A_L(v)))\cdot(1-P(A_T(v))) \tag{4}$$

由式(2)~式(4)可得

© 中国科学院软件研究所 http://www.jos.org.cn

$$P(A(v)) = 1 - (1 - P(A_L(v))) \cdot \prod_{e=\langle u,v \rangle}^{e \in E} (1 - P(A(u)) \cdot f(e)) \tag{5}$$

假设 2 中唯一的例外发生在必要条件中:多个错误叠加反而产生了正确的结果.这是一个概率非常小的小概率事件,在一次实验中是不可能发生的.所以,我们仍然不妨认为假设 2 成立.

在定义错误流图之前,我们先定义节点 v 的错误概率函数 $F:V\to[0,1]$,表示 v 发生错误的概率,所以 $F(v)=P(A(v))$.我们现在拥有了分别定义在计算数据流图的节点和边上的两个函数:$F:V\to[0,1]$和 $f:E\to[0,1]$.

定义 **5**. 如果 $G(V,E)$是计算数据流图,$F:V\to[0,1]$是定义在 V 上的错误概率函数,$f:E\to[0,1]$是定义在 E 上的传染概率函数,那么我们把错误流图记为 $G(V,E,F,f)$.

如果我们仅研究错误流图局部的特点,那么可以研究它的投影.

定义 **6**. 如果 $G(V,E,F,f)$是错误流图,W 是原子数据的集合,g 是 W 上的原子计算的集合,R 是与 g 相关的计算关系集合,那么称 $G'=(V\cap W,E\cap R,F\uparrow_{V\cap W},f\uparrow_{E\cap R})$为错误流图 $G(V,E,F,f)$在 W 上的投影.其中,$F\uparrow_{V\cap W}$为 F 在 $V\cap W$ 上的限制,$f\uparrow_{E\cap R}$为 f 在 $E\cap R$ 上的限制.

我们继续定义错误流图的流直径.

定义 **7**. 在错误流图 $G(V,E,F,f)$中,任何入度为 0 的节点都是 G 的源头,任何出度为 0 的节点都是 G 的入海口;如果存在一条路径从节点 u 到节点 v,那么称 u 在 v 的上游,v 在 u 的下游.

定义 **8**. 对于错误流图 $G(V,E,F,f)$,如果集合 D 满足以下条件:

(1) $D\subseteq V$;

(2) 对于任意从源头到入海口的路径 p,$D\cap p\neq\varnothing$;

(3) 对于任意满足条件(1)和条件(2)的集合 C,$D\subseteq C$,

则称 D 为错误流图 $G(V,E,F,f)$的流直径集,记为 $D(G)$.$D(G)$中元素的个数称为错误流图 $G(V,E,F,f)$的流直径.在很多情况下,错误流图的源头是 load 指令中的原子数据,入海口是 store 指令中的原子数据;处理器内部的错误都会流过错误流图在寄存器上的投影的流直径集.

4 错误流压缩算法

4.1 EDDI算法的错误流

EDDI 算法通过在汇编代码中插入冗余计算和比较分支指令来检测错误.在 EDDI 算法中,基本块(basic block)被进一步划分为无存储基本块(storeless basic block)[6].无存储基本块的最后一条指令除了分支指令外,还可以是 store 指令.EDDI 算法检测每一个无存储基本块的最后一条指令是不是存储指令.如果是,就为该存储指令增加一对比较和分支指令.这样的开销是容易估算的:根据统计,store 指令占程序的 10%,分支指令占 20%,那么,EDDI 算法将会使程序的分支指令增加大约 50%.

我们在图 5 中画出了 EDDI 算法的错误流图.

图 5 中仅画出了来自麻省理工学院 StreamIT 项目[31]的快速傅立叶变换程序的最内层循环体在经过 EDDI 算法编译后的情况.图中左半部分为原程序的错误流图,右半部分为插入的冗余计算.冗余计算中的下标 N'和原程序中的下标 N 按照 $N'=30-N$ 来设定.这主要是为了减少寄存器使用的冲突;使用 30 而不是 31 使寄存器为偶数编号的,这样就为双精度浮点数提供了可移植性.菱形节点 FCC^n 表示浮点比较指令的结果寄存器 FCC 中的原子数据.

可以看到,对于我们的测试程序而言,EDDI 算法需要进行 4 次比较测试才能覆盖整个最内层循环体,并且每一次比较后都必须根据测试结果执行一条分支指令,以便决定是继续正常执行还是启动错误处理子程序.我们清楚地知道,这些分支指令的插入是非常不利于超长流水线高效、稳定地发挥性能.

© 中国科学院软件研究所 http://www.jos.org.cn

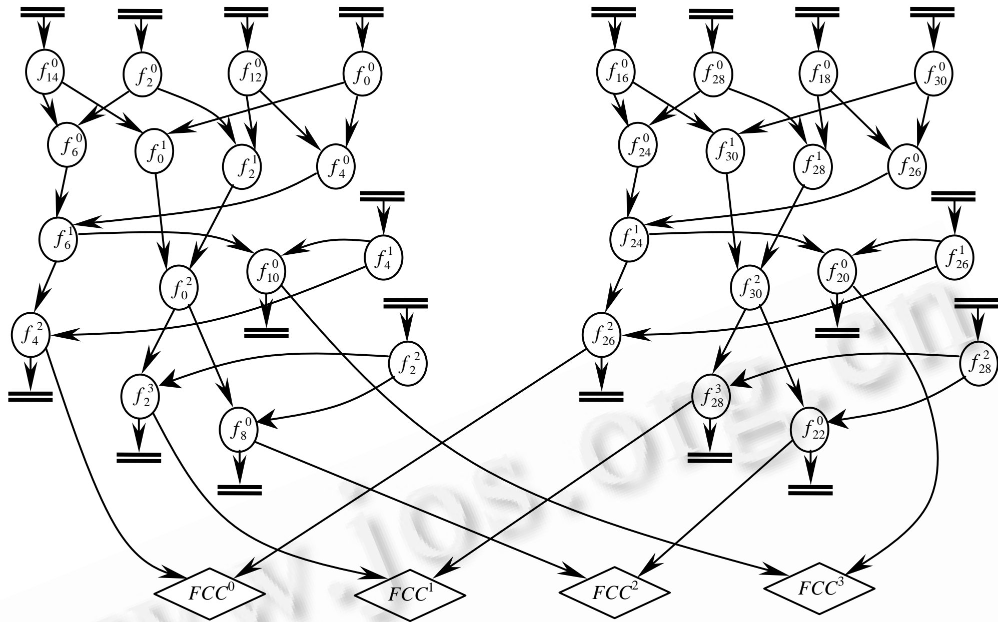

Fig.5 Error flow of EDDI
图 5 EDDI 算法的错误流图

为了今后叙述方便,我们将原子数据和插入的冗余副本合称为原子数据对,原子数据对之间互称为共轭原子数据.比如 f_6^0 和 f_{24}^0 就是一对原子数据对,可以简称为数据对; f_6^0 和 f_{24}^0 互为对方的共轭原子数据.比较指令比较的对象都是数据对,它们是否相等直接影响到分支指令分支的方向.

4.2 错误流压缩算法

为了减少容错算法中的比较和分支指令,我们提出了错误流压缩算法.与 EDDI 算法相比,错误流压缩算法在所有冗余的计算完成之后并不立即进行比较,而是使用附加的算术运算,将需要比较的数据对减少到 1 对.这时,我们只需要 1 对比较和分支就可以检测出错误,比 EDDI 减少了 3 对.错误流压缩算法的错误流图如图 6 所示,虚线是附加的浮点算术运算.在比较运算前,错误流图的流直径被压缩到了 2,流直径集是 $\{f_8^2, f_{22}^2\}$;而原计算和插入的冗余计算的错误流子图的流直径分别被压缩到了 1,流直径集分别是 $\{f_8^2\}$ 和 $\{f_{22}^2\}$.

算法 **1**. 错误流压缩算法.

1 对原程序建立错误流图 $G(V,E,F,f)$;
2 for (错误流图 G 的每一条弧 $e=v_sv_d;e\in E,v_s,v_d\in V$){
3 if (e 是 Load 指令)加载 v_s 在内存中的共轭原子数据 v_s' ,复制 v_d 的共轭原子数据 v_d' ;
4 else 复制 v_s,v_d 的共轭原子数据 v_s',v_d'
5 令 $e'=v_s'v_d'$; $V'=V'\cup\{v_s',v_d'\}$; $E'=E'\cup\{e'\}$; $f'(e')=f(e)$; $F'(v_s')=F(v_s)$; $F'(v_d')=F(v_d)$;
6 if (e 是 Store 指令) $S=S\cup\{v_s\}$, $S'=S'\cup\{v_s'\}$,把 v_d 和 v_d' 存在内存的邻近位置;
7 } /*$G'(V',E',F',f')$就是 $G(V,E,F,f)$的共轭错误流图*/
8 while (S 中元素的个数大于等于 2){
9 选定 S 中的任意两个节点 v_1,v_2,并且 v_1',v_2' 分别是 v_1,v_2 的共轭原子数据;
10 选定适当的二元原子计算 $c(x,y)$;
11 $S=S\cup\{c(v_1,v_2)\}-\{v_1,v_2\}$; /*压缩 $G(V,E,F,f)$的流直径,直到 1*/
12 $V=V\cup\{c(v_1,v_2)\}$;$E=E\cup\{e_1,e_2|e_1=v_1c(v_1,v_2),e_2=v_2c(v_1,v_2)\}$;

© 中国科学院软件研究所 http://www.jos.org.cn

13 计算 $F(c(v_1,v_2))$,$f(e_1)$和 $f(e_2)$;

14 $S'=S'\cup\{c(v_1',v_2')\}-\{v_1',v_2'\}$; /*压缩 $G'(V',E',F',f')$的流直径,直到 1*/

15 $V'=V'\cup\{c(v_1',v_2')\}$; $E'=E'\cup\{e_1',e_2' \mid e_1'=v_1'c(v_1',v_2'),e_2'=v_2'c(v_1',v_2')\}$;

16 计算 $F'(c(v_1',v_2'))$, $f'(e_1')$ 和 $f'(e_2')$;

17 } /*S 和 S'中最后都只剩下一个元素*/

18 对于 $v\in S,v'\in S'$,添加检错比较操作 compare(v,v')以及出错处理函数.

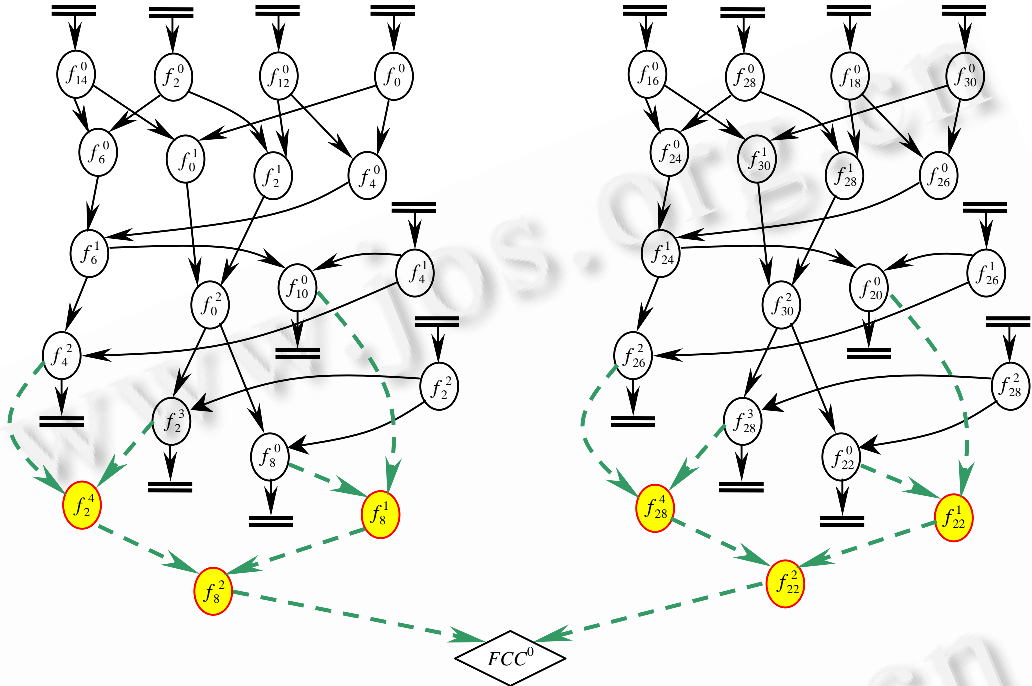

Fig.6 Error flow of error flow compressing

图 6 错误流压缩算法的错误流图

4.2.1 指令分析

假设原程序最内层循环体有 s 条存储指令.在 EDDI 容错算法中,由于需要新增 s 对比较和分支,指令总数增加 $2\cdot s$ 条;在错误流压缩算法中,仅需 1 对比较和分支指令进行比较,代价是在原计算和插入的冗余计算中各增加了 $s-1$ 条浮点计算指令,所以错误流压缩算法增加的指令数为 $2\cdot 1+2\cdot(s-1)=2\cdot s$ 条.最终经过 EDDI 算法和错误流压缩算法编译后的程序指令数是相等的.

表 1 列出了快速傅立叶变换测试程序的 PISA 汇编代码中最内层循环体在容错编译前后的指令变化情况.fft.s 为原程序 fft.c 编译产生的无容错功能的汇编代码;baseIS.s 为使用 EDDI 容错算法产生的汇编代码;fcompIS.s 为使用错误流压缩算法产生的汇编代码.

Table 1 Instruction summary

表 1 指令类型特性

Instruction type	fft.s	baseIS.s	fcompIS.s
Load	6	6	6
Store	4	4	4
FP arithmetic	10	20	26
FP compare	0	4	1
FP cond. branch	0	4	1
Summary	31	53	53

www.jos.org.cn

假设原程序最内层循环体有 s 条存储指令,那么每执行最内层循环体 1 次,错误流压缩算法比 EDDI 减少的分支指令数量就是 $s-1$.设 $f(n)$表示最内层循环体执行的总次数,n 是循环变量的终值.对于快速傅立叶变换而言, $f(n)=\frac{n}{2}\log_2(n)$,$u(n)$为原程序最终执行的分支指令总数,$U(n)$是 EDDI 算法中执行的分支指令总数.于是,错误流压缩算法比 EDDI 算法减少的分支指令比例为

$$b(s,n)=\frac{(s-1)f(n)}{U(n)}=\frac{(s-1)\cdot f(n)}{s\cdot f(n)+u(n)}=\frac{s-1}{s+\frac{u(n)}{f(n)}}=\frac{s-1}{s+BPL(n)} \tag{6}$$

其中, $BPL(n)=\frac{u(n)}{f(n)}$ 是原程序中执行的分支指令总数与最内层循环体执行次数的比值,是一个与原程序特性以及 n 有关的函数,但却与 s 以及内层循环体的大小无关.由于在指令的推断执行中可能分支预测失败,因而实际执行的分支指令数要略高于预测完全正确的理想状态.我们通过分析原程序的运行数据得到了 $BPL(n)$的部分数值,在 8~12 之间,并且 $BPL(n)$随着 n 的增大而减少,见表 2.

Table 2 $BPL(n)$

表 2 $BPL(n)$

n	$BPL(n)$	
	Simulated (%)	Expected (%)
2^{13}	11.74	11.47
2^{15}	10.69	10.46
2^{17}	9.90	9.70
2^{19}	9.28	9.10
2^{21}	8.78	8.61
2^{23}	8.36	8.21

可以得到

$$BPL(n)'_n<0 \tag{7}$$

$b(s,n)$对 n 求导,不妨假设 $s>1$,同时将式(7)带入,可得

$$b(s,n)'_n=\left(\frac{s-1}{s+BPL(n)}\right)'_n=\left(\frac{s-1}{s+BPL(n)}\right)'_{s+BPL(n)}\cdot(s+BPL(n))'_n=-\frac{s-1}{(s+BPL(n))^2}\cdot BPL(n)'_n>0 \tag{8}$$

由式(8)可知,$b(s,n)$对 n 单调递增.也就是说,当原程序中循环次数增多时,错误流压缩算法比 EDDI 减少的分支指令比例也增多.举例来说,在我们的实验中,当 $s=4,n=2^{23}$ 时,查表 2 可知,理想情况下,$BPL(n)=8.21$,代入式(6)可以算出 $b(s,n)$的理想预期值为 $b(s,n)=24.57\%$,实验中测得 $b(s,n)=24.27\%$.

$b(s,n)$对 s 求导,因为 $BPL(n)$与 s 无关,可得

$$b(s,n)'_s=\left(\frac{s-1}{s+BPL(n)}\right)'_s=\left(1-\frac{BPL(n)+1}{s+BPL(n)}\right)'_{s+BPL(n)}\cdot(s+BPL(n))'_s=\frac{BPL(n)+1}{(s+BPL(n))^2}>0 \tag{9}$$

由式(9)可知,$b(s,n)$对 s 单调递增.也就是说,当原程序中内层循环的存储指令数量增多时,错误流压缩算法比 EDDI 减少的分支指令比例也增多.例如,当 $s=8,n=2^{23}$ 时,错误流压缩算法比 EDDI 算法预期的分支指令减少的比例预期将可以达到 $b(s,n)=43.18\%$.

综合式(8)、式(9)可知,错误流压缩算法具有良好的可扩展性.我们使用理想情况下的 $BPL(n)$计算了部分的 $b(s,n)$值,预测值和实验结果十分接近,见表 3.

Table 3 $b(s,n)$

表 3 $b(s,n)$

s	n	$b(s,n)$		s	n	$b(s,n)$	
		Simulated (%)	Expected (%)			Simulated (%)	Expected (%)
4	2^{13}	19.06	19.39	4	2^{19}	22.59	22.90
4	2^{15}	20.42	20.75	4	2^{21}	23.48	23.78
4	2^{17}	21.58	21.90	4	2^{23}	24.27	24.57

4.2.2 容错分析

计算类型的指令基本上都是二元计算,我们以二元计算为例进行简化的错误传播分析.对于图 7 中的二元计算关系,根据式(5)有

$$P(A(v))=1-(1-P(A_L(v)))\cdot(1-P(A(u))\cdot f(e_1))\cdot(1-P(A(w))\cdot f(e_2)) \tag{10}$$

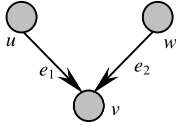

Fig.7 Sample of error propagation

图 7 错误传播示例

令 $P(A_L(v))=a,P(A(u))=b,P(A(w))=c$,同时简化地认为如果参与计算的数据错误,那么结果一定错误,则 $f(e_1)=1,f(e_2)=1$.所以,式(10)变为 $F(v)=a+(1-a)(b+c(1-b))=b+(1-b)(a+c(1-a))=c+(1-c)(a+b(1-a))$.又因为 $0<a,b,c<1$,所以 $a,b,c<F(v)$.也就是说,在 v 处发现错误的概率要比在 u,w 处发现错误的概率都大.因而我们在 v 处比较并检测错误,能够更容易地发现错误.

因此简要地说,错误流压缩算法在错误流的下游检测错误,不会影响到检测错误的覆盖率;容错延迟的开销与 EDDI 相比,也不会多于一个最内层循环体执行的时间.

5 实验设计和分析

5.1 实验设计

我们使用 Wattch 1.02 模拟器[32],指令集为 SimpleScalar/PISA 指令集[27].出于节能的考虑,我们采用了功耗较低的配置,最大限度地模拟了具备多种节能技术的 IBM G4 高性能处理器[33].模拟器的功耗和性能与 G4 处理器相近,平均功耗约为 10W~14W,最大功耗约为 25W~30W;平均 IPC 约为每周期一条指令.分支预测部件所消耗的功耗约为全部功耗的 9%.运算部件为 4 个整数 ALU,1 个整数乘除部件,2 个浮点 ALU,1 个浮点乘除部件;分支预测器采用 Combined 策略,选择器为 4K 大小,bimodal 表为 4K 大小,2-level 表为 4K 大小,2 位历史信息;访存延迟为 100 个时钟周期;处理器主频为 1.2GHz.实验中采用 Wattch 中的 cc3 能量模型:处理器中部件消耗的能量和访问该部件的次数成正比,空闲时每个部件仍有相当于最大功耗 10%的功耗泄漏.

在我们的实验中,采用的是麻省理工学院 StreamIT 项目[31]提供的 C 版本的快速傅立叶变换测试程序.我们针对最内层循环体中的浮点计算进行容错编译.我们在 RedHat Linux 9.0 上先使用修改过的 gcc 2.7.2.3 编译生成无容错功能的 PISA 汇编代码,编译优化选项为-O3;然后再对汇编代码进行容错优化;最后使用 gcc 从具有容错功能的汇编代码生成目标代码.

5.2 实验分析

5.2.1 实验结果

图 8 显示了 EDDI 算法比原程序以及错误流压缩算法比 EDDI 算法增加的分支指令、功耗和 IPC 的比值.图中横坐标表示快速傅立叶变换中循环变量的终值,数值取以 2 为底的对数.除了循环部分以外,实验程序中还有初始化等工作.为了能够提高循环部分在程序运行过程中占的比例,实验中循环参数最大达到 2^{25},这样更有利于准确地判断循环体本身的特点.

数据显示:EDDI 算法比原程序的分支指令增加最多接近 50%,平均增加 40.32%;IPC 降低最多超过 20%,平均降低 7.99%;功耗增加最多超过 50%,平均增加 37.14%.可见,EDDI 算法的性能开销和功耗开销都是相当大的.而采用错误流压缩算法后,分支指令比 EDDI 算法减少最多接近 25%,平均减少 21.43%;IPC 提高最多超过 12%,平均提高 5.11%;功耗减少最多接近 5%,平均减少 1.86%.错误流压缩算法在性能和功耗上都明显优于 EDDI 算法.

5.2.2 性能分析

图 9 和图 10 都显示出错误流压缩算法比 EDDI 算法增加的比值.我们看到,错误流压缩算法减少了分支指令的数量,使动态预测分支的时机减少,所以没有提交的乱序执行的无用指令迅速减少,而提交的正确指令数不变,最终使得程序执行的总指令数减少,如图 10 所示.从图 9 中可以看到:总周期数减少最多超过 10%,平均减少

© 中国科学院软件研究所 http://www.jos.org.cn

5.04%.所以错误流压缩算法的 IPC 得到了明显的提高,如图 8 所示,最多提高 12%,平均提高 5.11%.

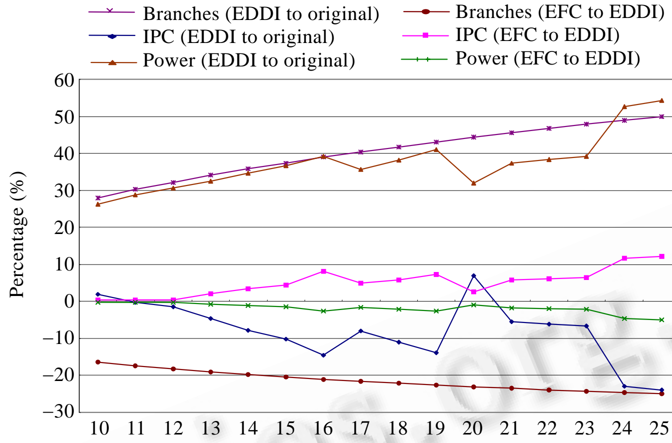

Fig.8 Performance and power

图 8 性能和功耗

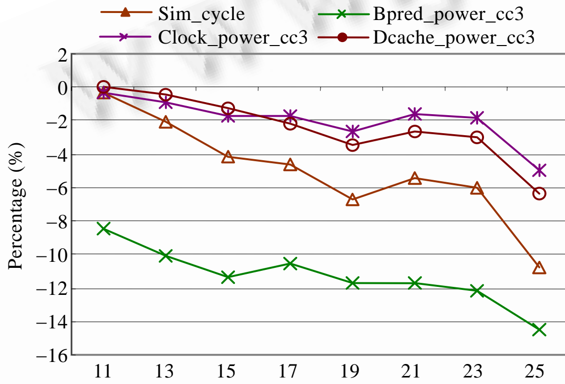

Fig.9 Branches and power

图 9 分支和功耗

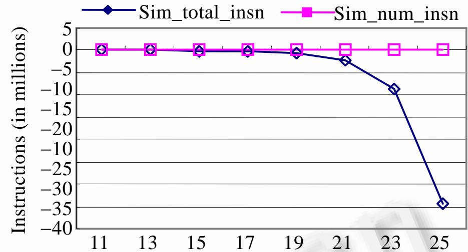

Fig.10 Instructions executed and committed

图 10 执行和提交的指令总数

5.2.3 功耗分析

分支指令数量的减少直接促使分支预测部件的功耗减少,如图 9 所示,分支预测部件的功耗减少最多超过 14%,平均减少 11.31%.因为分支预测部件的功耗占总功耗的 10%左右,所以分支部件的节能直接使总功耗最多减少 1.4%,平均减少 1.1%;时钟部件功耗减少最多超过 5%,平均减少 1.97%.因为时钟部件占总功耗的 30%左右,所以时钟部件的节能使总功耗减少最多达到 1.5%,平均减少 0.6%.以上两项使总功耗减少最多接近 3%,平均减少 1.7%.最终各部件的能耗变化使总功耗减少最多接近 5%,平均减少 1.86%.

事实上,现代的高性能处理器无一例外地使用乱序推断执行指令来提高性能,大量指令在指令窗口中被推断执行.由于分支预测可能失败,执行的指令中有许多是无用的指令,因此而消耗的能量平均可以占到总能量消耗的 28%[4].错误流压缩算法减少了分支指令数量,因而减少了由于执行无用指令而浪费的能量.

图 11 中列出了当循环变量终值 $n=2^{25}$ 时,使用错误流压缩算法比 EDDI 减少的功耗在处理器和 ALU 内部的分布情况.我们可以看到,减少分支指令给低功耗带来的好处是多方面的.另外,我们看到整数 ALU 部件减少的功耗掩盖了我们插入的附加浮点运算所带来的功耗开销,最终 ALU 整体功耗还有所下降.这与部件的能量泄漏有关,当我们在 Wattch 中使用没有能量泄漏的 cc2 理想模型时,ALU 的总功耗反而增加了 4%.

© 中国科学院软件研究所 http://www.jos.org.cn

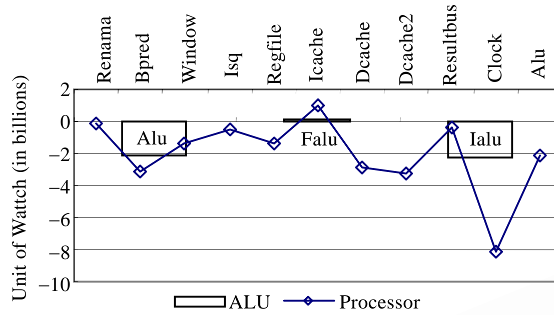

Fig.11 Power distribution

图 11 能量分布

6 结 论

减少软件容错算法中的分支指令数量能够同时减少功耗并且提高性能.本文通过错误流模型表述了错误流压缩算法.错误流压缩算法通过附加计算,压缩了错误流的直径,因而明显减少了用于检测错误的分支指令数量.实验数据表明:与 EDDI 算法相比,错误流压缩算法在保持容错能力的同时,既提高了性能又降低了功耗.错误流压缩算法同时具有良好的可扩展性.我们可以展望,未来的高性能、低功耗的软件容错计算平台将会给低成本、高效率的深空探测带来更强有力的支持.

7 未来的工作

目前的错误流模型依赖于若干假设条件,我们将把错误流模型建立在更为公理化的逻辑体系之上.错误流模型目前只用于描述计算过程中的错误传播规律,然而我们相信,错误流模型具有描述更为复杂结构的潜力.

容错延迟以及错误覆盖率等问题仍然有待于进一步的研究,尤其需要使用错误注入(fault injection)的技术进行验证.至于如何在多核处理器上利用过剩的并行处理能力进行容错,以及深度并行计算机通信过程中的容错问题等,都是我们未来的研究目标.

References:

[1] 2006. http://www.people.com.cn/GB/keji/25509/32328/

[2] Oh N, Mitra S, McCluskey EJ. ED4I: Error detection by diverse data and duplicated instructions. IEEE Trans. on Computers, 2002, 51(2):180−199.

[3] Oh N. Software implemented hardware fault tolerance [Ph.D. Thesis]. Stanford: Stanford University, 2000.

[4] Juan LA, Jose G, Antonio G. Power-Aware control speculation through selective throttling. In: Proc. of the 9th Int'l Symp. on High-Performance Computer Architecture. Washington: IEEE Computer Society, 2003. 103.

[5] Dharmesh P, Kevin S, Yan Z, Mircea S. Power-Aware Branch prediction: Characterization and design. IEEE Trans. on Computers, 2004,53(2):168−186.

[6] Oh N, Shirvani PP, McCluskey EJ. Error detection by duplicated instructions in super-scalar processors. IEEE Trans. on Reliability, 2002,51(1):63−75.

[7] Shirvani P. Fault tolerant computing for radiation environment [Ph.D. Thesis]. Stanford: Stanford University, 2001.

[8] Huang KH, Abraham JA. Algorithm-Based fault tolerance for matrix operations. IEEE Trans. on Computers, 1984,33(6):518−528.

[9] Maurizio R, Matteo SR, Massimo V, Marco T. A source-to-source compiler for generating dependable software. In: Proc. of the 1st IEEE Int'l Workshop on Source Code Analysis and Manipulation. Washington: IEEE Computer Society, 2001. 33−42.

[10] Clark JA, Pradhan DK. Fault injection: A method for validating computer-system dependability. IEEE Computer, 1995,28(6): 47−56.

[11] Ziegler JF. IBM experiments in soft fails in computer electronics (1978−1994). IBM Journal of Research and Development, 1996,40(1):3−18.

[12] Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Reorda MS, Violante M. Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. IEEE Trans. on Nuclear Science, 2000,47(6):2231−2236.

© 中国科学院软件研究所 http://www.jos.org.cn

[13] Lyons RE, Vanderkulk W. The use of triple-modular redundancy to improve computer reliability. IBM Journal of Research and Development, 1962,6(2): 200−209.

[14] Chen CL, Hsiao MY. Error-Correcting codes for semiconductor memory applications: A state-of-the-art review. IBM Journal of Research and Development, 1984,28(2):124−134.

[15] 2006. http://www.stratus.com/

[16] HP NonStop S88000, S78000, and S780 servers data sheet. Hewlett-Packard Development Company, 2004.

[17] Lu DJ. Watchdog processor and structural integrity checking. IEEE Trans. on Computers, 1982,C-31(7):681−685.

[18] Gomaa M, Scarbrough C, Vijaykumar TN, Pomeranz I. Transient-Fault recovery for chip multiprocessors. In: Proc. of the 30th Annual Int'l Symp. on Computer Architecture. New York: ACM Press, 2003. 98−109.

[19] Vijaykumar T, Pomeranz I, Cheng K. Transient-Fault recovery using simultaneous multithreading. In: Proc. of the 29th Annual Int'l Symp. on Computer Architecture. Washington: IEEE Computer Society, 2002. 87−98.

[20] Avizeinis A. The N-version approach to fault-tolerant software. IEEE Trans. on Software Engineering, 1985,SE-11(12):1491−1501.

[21] Randell B. System structure for software fault tolerance. IEEE Trans. on Software Engineering, 1975,SE-1(2):220−223.

[22] Alkalai L, Tai A, Chau S. COTS-Based fault tolerance in deep space: Qualitative and quantitative analyses of a bus network architecture. In: Proc. of the 4th IEEE Int'l Symp. on High Assurance. Washington: IEEE Computer Society, 1999. 97 −104.

[23] Equils DJ. Method for enhancing the process of software tool evaluation and selection: COTS, heritage, and custom software reviewed. In: Proc. of the SpaceOps 2004. Pasadena: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2004. 1−10.

[24] Liu P. Reliability Engineering Principles. Revised ed., Beijing: Measurements Press, 2002 (in Chinese).

[25] Alpern B, Wegman MN, Zadeck FK. Detecting equality of values in programs. In: Proc. of the 15th ACM Symp. on Principles of Programming Languages. New York: ACM Press, 1988. 1−11.

[26] Marc MB, Hanspeter M. Single-Pass generation of static single-assignment form for structured languages. ACM Trans. on Programming Languages and Systems, 1994,16(6):1684−1698.

[27] Burger DC, Austin TM. The SimpleScalar tool set, version 2.0. ACM SIGARCH Computer Architecture News, New York: ACM Press, 1997,25(3):13−25.

[28] Cliff Y, Michael DS. Static correlated branch prediction. ACM Trans. on Programming Languages and Systems, 1999,21(5): 1028−1075.

[29] Wu Y, Larus JR. Static branch frequency and program profile analysis. In: Proc. of the 27th Annual Int'l Symp. on Microarchitecture. New York: ACM Press, 1994. 1−11.

[30] Jason RC, Patterson. Accurate static branch prediction by value range propagation. In: Proc. of the ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation. New York: ACM Press, 1995. 67−78.

[31] 2006. http://cag.csail.mit.edu/streamit

[32] Brooks D, Tiwari V, Martonosi M. Wattch: A framework for architectural-level power analysis and optimizations. In: Proc. of the 27th Annual Int'l Symp. on Computer Architecture. New York: ACM Press, 2000. 83−94.

[33] Freescale Semiconductor Inc. MPC7447A RISC microprocessor hardware specifications. technical data. Chandler: Freescale Semiconductor Inc., 2005.

附中文参考文献:

[24] 刘品.可靠性工程基础.修订版,北京:计量出版社,2002.

高珑(1978－),男,山东胶州人,博士生,主要研究领域为操作系统,编译器,嵌入式系统.

杨学军(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为并行与分布式处理,超大规模并行计算.