ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.18, No.9, September 2007, pp.2105−2116
DOI: 10.1360/jos182105

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

高效的部分冗余容错编译:复制错误流关键子图*

高 珑+, 王之元, 杨学军

(国防科学技术大学 计算机学院,湖南 长沙 410073)

Efficient Partial Redundancy Fault Tolerance Compilation: Replicating Critical Subgraph of Error Flow

GAO Long+, WANG Zhi-Yuan, 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, Wang ZY, Yang XJ. Efficient partial redundancy fault tolerance compilation: Replicating critical subgraph of error flow. *Journal of Software*, 2007,18(9):2105−2116. http://www.jos.org.cn/1000-9825/18/2105.htm

Abstract: Traditional fault tolerance compilations replicate all computations and registers to guarantee fault tolerance. But this brought great overhead in both storage utilization and performance. This paper suggestes a new concept of critical subgraph of error flow graph based on error flow analyses. Methods are given to generate critical subgraphs from critical nodes or from critical paths, and partial redundancy algorithm is suggested to replicate only critical subgraph. Partial redundancy algorithm guarantees effective fault tolerance, and greatly improves performance, reduces power dissipations and reduces storage usage. Experimental results show that, compared with full redundancy which replicates full error flow graph, partial redundancy can reduce register usage by 6.25%, reduce power dissipation by over 17%, reduces total execution cycles by nearly 26%, and improves performance by over 22%, at the cost of 6.25% lower nodes coverage.

Key words: SIHFT(software implemented hardware fault tolerance); error flow mode; critical subgraph; partial redundancy; fault tolerance compilation

摘 要: 传统的容错编译通常复制所有的计算并且使用完全冗余的存储单元来保证容错.这种完全冗余在存储空间和性能上的开销都是相当大的.在错误流分析的基础上提出错误流图的关键子图的概念以及通过关键结点和关键路径生成关键子图的方法,并设计了通过复制错误流关键子图实现部分冗余的算法.在保证有效容错能力的同时,部分冗余明显减小了经过容错编译的程序在存储空间和性能上的开销.实验显示,与复制全部错误流图的完全冗余相比,在结点覆盖率降低 6.25%的情况下,部分冗余算法最多能够减少寄存器的使用数量 6.25%,减少功耗超过 17%,减少执行时间接近 26%,同时提高性能超过 22%.

关键词: 面向硬件故障的软件容错;错误流模型;关键子图;部分冗余;容错编译

中图法分类号: TP314 **文献标识码**: A

* Supported by the National Natural Science Foundation of China under Grant No.60621003 (国家自然科学基金)
Received 2006-07-20; Accepted 2006-11-13

由于受到成本、性能和功耗等因素的影响,美国国家宇航局(NASA)和Stanford大学等机构开始尝试放弃使用专用抗辐照器件(比如,专用抗辐照CPU等)制造空间计算机,而逐步转向使用COTS (commercial off the shelf)器件制造空间计算机,原来抗辐照硬件的容错功能转而由在COTS器件上通过软件实现的容错技术(software implemented hardware fault tolerance,简称SIHFT)来完成[1−5].但是,由于飞船和卫星等载体在功耗、重量和体积等方面的限制,空间计算机的资源配置也受到很大的限制,这就要求使用软件容错技术的空间计算机尽量减小软件容错技术带来的开销.

硬件容错通过基于硬件冗余的抗辐照器件实现,虽然在成本、性能和功耗等方面开销很大,但是由于冗余的硬件并行执行指令,因而其实现容错的性能开销几乎对用户透明;相反地,软件容错通过在空间和时间上反复执行指令来实现冗余并保证容错,因而在存储空间和性能上的开销相当大.例如,EDDI的性能开销高达170%[5,6],源到源容错编译的空间开销高达 2.9 倍,性能开销高达 2.6 倍[7].

容错编译是目前软件容错中比较活跃的一个研究方向.容错编译依靠在程序中插入冗余的计算指令来实现容错,同时也带来了很大的存储和性能开销.但是,通过错误流[8,9]分析我们发现,不同的指令和存储单元对传播错误的影响是不同的.在程序中,有些存储单元累积了完全相同的错误,因而没有必要进行重复的冗余和重复地检测错误;而有些执行概率很大的指令使用的存储单元可能会累积较高的错误概率,因而需要着重检测错误.本文提出的部分冗余,通过错误流分析找出错误流中累积完全相同错误的存储单元,从而避免重复的冗余和错误检测.在损失一定错误覆盖率的情况下,仍然保证了有效的容错能力.与复制所有计算和存储单元的完全冗余相比,部分冗余在存储空间利用效率和性能上都有显著的提升.

本文第 1 节介绍相关研究.第 2 节简要介绍错误流分析的基础.第 3 节提出错误流关键子图的概念,并分别提出使用关键结点和关键路径生成关键子图的算法.第 4 节提出通过复制错误流关键子图实现部分冗余的算法.第 5 节给出模拟实验的结果和分析.第 6 节进行总结.第 7 节对未来的工作进行展望.

1 相关研究

1.1 典型案例

REE(remote exploration and experimentation)计划是美国国家宇航局高性能计算和通信计划(high performance computing and communications,简称HPCC)的一部分[1],其目的就是利用高性能的COTS器件和软件容错技术,实现高性能、高可靠的空间计算机.在REE计划中,计划实现超过 300MIPS/watt的性能功耗比,并且达到超过 10 年的可靠操作,以便为人类探索更遥远的太空作准备.REE计划中,空间计算机的可靠性就由在COTS器件上实现的软件容错来保证.REE计划已经完成了若干测试,并且取得了良好的效果[1,2].

ARGOS(advanced research and global observations satellite)是Stanford大学的可靠性计算中心(center for reliable computing,简称CRC)和美国国家宇航局合作的项目[4,5].在ARGOS发射的卫星上面进行了真实空间环境中的对比实验,实验的卫星上面安装了两块对照测试板,分别使用Harris公司的抗辐照中央处理器RH3000 和COTS中央处理器IDT3081[4,5].安装IDT3081 的测试板上使用了多种软件容错技术.经过长时间的实验, ARGOS得出的结论是:在COTS器件上使用软件容错技术能够保证和抗辐照器件相近的容错效果,而性能则要超过抗辐照器件一个数量级[5].最终初步验证并形成了面向硬件故障的软件容错(software implemented hardware fault tolerance,简称SIHFT)的技术路线[1−5].

1.2 软件容错的分类

以多版本程序设计(N version programming,简称NVP)[11]以及恢复模块(recovery block)[12]等为代表的软件可靠性主张依靠设计多样性(design diversity)理念,使用不同的编程语言、不同的开发团队、不同的测试方法实现多个版本的程序,共同完成同一个功能,从冗余的结果中选择出较为可靠的结果.软件可靠性主要研究如何避免软件设计中的设计错误.

而面向硬件故障的软件容错主要通过软件实现的冗余容忍发生在硬件中的瞬时错误[10].主要包括算法级容错(algorithm based fault tolerance,简称ABFT)和容错编译等.

算法级容错通过改变算法,加入冗余的数据并且重复计算,最终可以检测甚至纠正错误.比如,容错的矩阵操作[13]就增加了冗余的行和列.这些冗余的行和列也参与矩阵的运算,最终可以根据结果判断、定位和纠正错误.但是由于容错算法的设计复杂,给程序员增加了沉重的负担,也增加了在实现容错算法过程中引入新的设计错误的风险,因而,算法级容错的应用也受到了限制.

而容错编译通过在编译的时候自动地插入指令实现容错,实现简单而高效,不需要重写源代码,减轻了程序员的负担,有利于利用已有的大量程序.随着编译技术的发展,容错编译已成为软件容错领域中比较活跃的一个分支.

1.3 容错编译带来的开销

EDDI[6]和ED^{4}I[14]是ARGOS实验中的容错编译算法.EDDI和ED^{4}I在汇编语言一级复制每一条指令,在每一条存储指令之前,比较即将存储的冗余计算结果是否一致来检测错误.但是,指令的重复执行给算法带来了较大的存储和性能开销.另外,比较冗余计算结果时插入的分支指令也给流水线的性能带来显著的影响.EDDI最终大约有 170%的性能开销.

源到源容错编译[7]复制C语言中的每一条语句,随后插入条件语句比较冗余计算的结果,以判断是否出现错误.源到源容错编译能够将不含容错功能的C语言源程序自动转化成具有容错功能的C语言源程序.但是,由于每一条高级语言对应多条实际指令,因而这种高级语言级的转换比较粗糙,它的开销相当大.源到源容错编译的空间开销为 2.9 倍,运行性能开销也高达 2.6 倍[7].

上述容错编译方法中,都完全复制了每一条指令或者语句,这种完全冗余带来的开销是非常明显的.本文根据错误流分析,对这种完全冗余做出了改进,提出使用部分冗余避免复制累积相同错误的存储单元和计算,在存储空间利用率和性能上都获得了较大提升.

2 错误流分析基础

根据文献[9]中建立的错误流模型,可以对任意程序进行错误流分析,并为程序建立错误流图.在错误流图和错误流分析的基础上,能够计算出错误在程序中的传播和分布情况.在介绍本文的主要工作之前,我们先来简要介绍错误流分析的基础.

2.1 原子数据

定义 1. 原子数据是由特定的存储单元记录,在指令中不可再分的数据.

定义 2. 在指令中读取的原子数据称为源,而写入的原子数据称为目标.

同一个存储单元中的数据在程序执行过程中可能被多次重写,这样,它所代表的原子数据也就不一样.为了标志不同原子数据在时间和空间上的唯一性,我们为存储单元的名称增加上标,用以表示某时刻在该存储单元中存放的原子数据,如 f_1^2 .我们规定,在执行指令的过程中,如果存储单元的内容被改写,那么相应原子数据的上标加 1,而原子数据的初始上标为 0.如果原子数据处于我们所研究的范围以外,就使用双横线来略写原子数据.

2.2 计算关系

定义 3. 从任意指令的源到目标,都存在计算关系.

我们从每一条指令的源到目标画一条带有方向的弧,表示计算关系.图 1(a)可以表示在某时刻把f_1中的原子数据赋值给f_3.图 1(b)既可以被想象成乘法,也可以被想象成加法、减法和除法等其他二元计算.

2.3 执行概率

由于条件分支指令的存在,在程序的某一次运行过程中,某些指令可能被执行,也可能不被执行.因而,相应的计算关系也以一定的概率成立.

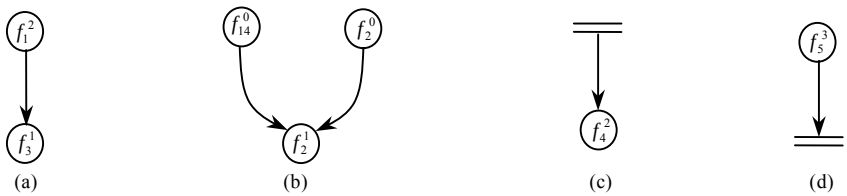


Fig.1 Atomic data and computational relation

图 1 原子数据和计算关系

定义 4. 计算关系的执行概率等于相应指令获得执行的概率.

我们在表示计算关系的有向边上标注该计算关系的执行概率,表示该计算关系成立的概率,如图 2 所示.图中左边的程序使用SimpleScalar/PISA指令[15],假定分支成功的概率为p,失败的概率为 $1-p$.

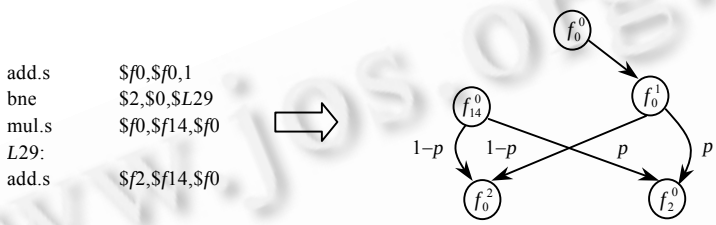


Fig.2 Conditional branch

图 2 分支指令

2.4 错误流图

根据文献[9]中的算法,可以为任意程序建立错误流图,其中,每个结点上有一个表示该结点发生错误的错误概率函数,每一条有向边上有一个表示该有向边传播错误的传播概率函数.以StreamIt项目中的快速傅里叶变换程序[16]为例,可以画出最内层循环体的错误流图,如图 3(a)所示.图中没有标出错误概率函数和传播概率函数.

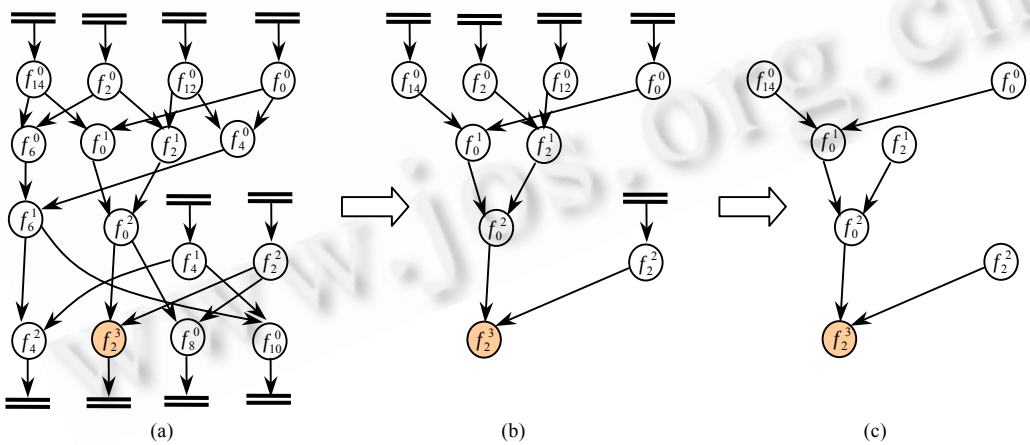


Fig.3 Critical subgraph from critical node

图 3 从关键结点生成关键子图

定义 5. 如果在错误流图中存在一条从结点 u 到结点 v 的路径,那么称 u 在 v 的上游,v 在 u 的下游.

定义 6. 起始结点指错误流图中入度为 0 的结点,终止结点是指错误流图中出度为 0 的结点.

对于出度为 0 的双横线结点,由于已经离开了我们研究的范围,难以对其进行再次操作,所以在确定终止结点之前,通常去掉所有的双横线结点以及与这些双横线结点相邻的有向边;相反地,入度为 0 的即将进入我们研

究范围的双横线结点,通常可以立即加以利用,所以,这些入度为 0 的双横线结点可以充当起始结点的角色.例如,在图 3(a)中,起始结点为全部 6 个入度为 0 的双横线结点;而终止结点为 f_4^2, f_2^3, f_8^0, f_{10}^0.

使用错误的数据进行计算,计算的结果也很可能是错误的.

规则 1. 在错误流图中,错误以一定的概率沿着计算关系的方向传播,而不是反方向.

由于计算关系的传递性,错误也以一定概率沿着计算关系连成的路径传播.

错误被计算关系屏蔽的例外情况如 *a*=*b*&0*xff*00 会屏蔽存在于 *b* 中低 8 位的错误传播到 *a*,而 *c*=*d*<*e* 会屏蔽任何满足 *d* 小于 *e* 的错误传播到 *c*.在文献[9]中详细地讨论了错误在错误流图中的传播,并且给出了计算任意结点错误概率的方法.

根据规则 1,终止结点将汇聚所有到终止结点存在路径的结点的错误.如果错误流图是连通图,那么终止结点的集合将可能汇聚所有结点的错误.

3 错误流关键子图

定义 7. 错误流关键子图是指错误流图中对错误的传播和分布具有关键影响的子图.

可以通过多种方式衡量错误流子图对于错误传播的影响:覆盖的结点数目比较多的错误流子图比较关键;另外,计算关系的执行概率比较大的错误流子图也比较关键.我们可以通过关键结点和关键路径生成错误流关键子图.一般情况下,我们要求的关键子图是比错误流图本身更小的一个真子图.

3.1 由关键结点生成关键子图

定义 8. 关键结点是指错误流图中对错误的传播和分布具有关键影响的结点.

根据错误流分析,所有能够到达关键结点的结点对关键结点的错误都具有影响,所以这些结点也比较关键,因而,由这些比较关键的结点组成的子图就是关键子图.我们可以由关键结点生成关键子图.

3.1.1 完全关键子图

在错误流图中,完全关键子图由所有到关键结点存在路径的结点以及这些路径上的有向边组成.对于错误流图G和指定的关键结点集合V_0,可以调用图 3 中的算法$Full(G,V_0)$从关键结点集合V_0生成完全关键子图$G_{crit}(V_0)$.在算法 1 中,递归地找到能够到达任意$v\in V_0$的所有结点以及经过的有向边,这些结点和有向边就构成了由关键结点v_0生成的完全关键子图.在图 3 中,图 3(b)就是在图 3(a)中由关键结点集合{ f_2^3}生成的完全关键子图.

算法 1.

```
PROCEDURE Full(G,V0)
INPUT:错误流图G,指定的关键结点集合V0;
OUTPUT:由关键结点集合V0生成的完全关键子图Gcrit(V,E).
BEGIN
    FOR (V0中的每一个结点v ) DO
        FOR( G 中每一条指向 v 的有向边 e=⟨u,v⟩) DO
        BEGIN
            V=V∪{v}
            E=E∪{e}
            Full(G,{u})
        END
END
```

3.1.2 部分关键子图

当我们需要精简的关键子图时,可以根据特定的选择条件在完全关键子图上裁减部分结点和有向边,生成

部分关键子图.如果G是完全关键子图,$Choose(v)$是用来判断是否需要对v进行裁减的函数,那么,通过算法 2 中的算法$Part(G,Choose)$就可以裁减生成部分关键子图 G'_{crit}.在算法$Part(G,Choose)$中,如果选择了某个结点v_0作为裁减的对象,就把除v_0外所有能够到达v_0的结点和有向边全部删除.这样,对于关键子图 G'_{crit} 中除起始结点外的任意结点,计算该结点的所有操作数都是具备的,不会由于缺少操作数而不能通过计算得出.图 3 中,图 3(c)就是由完全关键子图 3(b)经过裁减而成的部分关键子图.对于一般的错误流图G,我们也可以通过$Part(G,Choose)$进行适当的裁减.

算法 2.

PROCEDURE $Part(G,Choose)$

INPUT:完全关键子图 G,选择函数 $Choose$;

OUTPUT:根据选择函数 $Choose$ 由 G 裁减成的部分关键子图.

```
BEGIN
    FOR (G 中每一个结点 v) DO
        IF (Choose(v)) DO
            FOR (Full(G,{v})中每一个结点 x)
                IF (x≠v) DO
                    裁减掉 x 和所有与 x 相邻的有向边
END
```

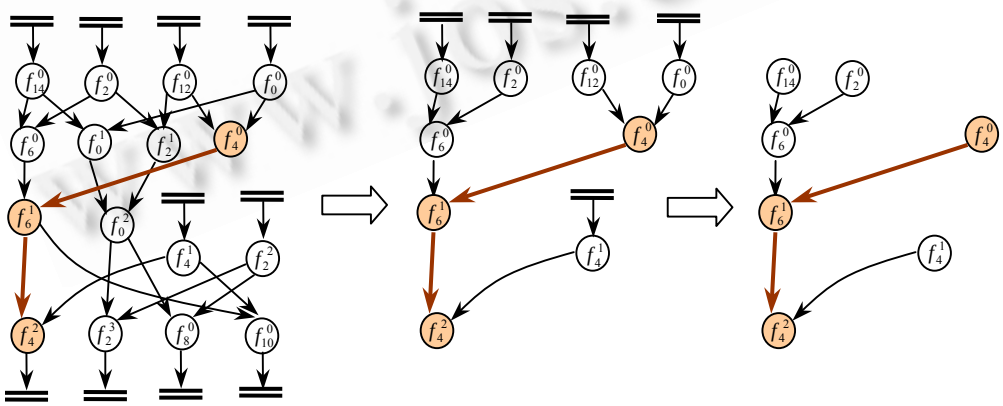
3.2 由关键路径生成关键子图

定义 9. 关键路径是指错误流图中对错误的传播和分布具有关键影响的路径.

对于存在分支的程序而言,某些路径的执行概率可能会很大.比如,控制循环的分支指令,分支成功的概率通常远大于失败的概率.这些经常执行的路径就会对错误传播产生较为关键的影响,通过关键路径生成的子图,对于错误传播的影响也会较为关键.下面我们将要介绍通过关键路径生成关键子图的方法.

3.2.1 完全关键子图

当把任何指定的关键路径看作是错误流图时,它也会有终止结点.我们可以通过关键路径的终止结点来生成完全关键子图.我们可以首先找到关键路径l的终止结点v_0,然后以v_0为关键结点调用第 3.1.1 节生成完全关键子图算法$Full(G,\{v_0\})$.这样生成的完全关键子图必然包含指定的关键路径l,因为关键路径上的每一个结点到关键路径的终止结点v_0都存在路径.图 4 中,图 4(b)就是在错误流图 4(a)中由指定的关键路径 $f_4^0 f_6^1 f_4^2$ 生成的完全关键子图.



(a) (b) (c)

Fig.4 Critical subgraph from critical path

图 4 从关键路径生成关键子图

3.2.2 部分关键子图

与第 3.1.2 节中生成部分关键子图的方法类似,我们可以通过适当裁减完全关键子图得到部分关键子图.唯一不同的是,如果我们根据关键路径生成了关键子图,那么我们不能裁减任何属于该关键路径的结点和有向边.图 4 中,图 4(c)就是裁减完全关键子图 4(b)而成的一个部分关键子图.

4 基于复制错误流关键子图的部分冗余

因为关键子图对于错误传播的影响比较关键,所以仅复制关键子图即能保证比较有效的容错能力;而错误流关键子图较小,因而仅复制错误流关键子图的存储和性能开销可能会相对较小.与复制全部错误流图的完全冗余相对应,我们把仅复制错误流关键子图的软件冗余称为部分冗余.

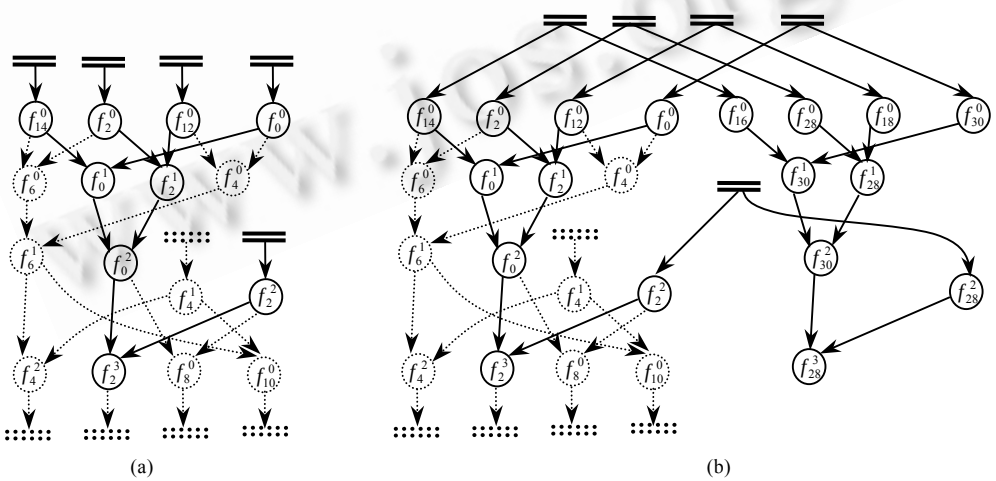
4.1 关键子图的复制

在复制关键子图时,除了起始结点以外的所有结点和有向边都被复制,而起始结点则被共享.这样,在复制出的关键子图的副本 G'中的任何一个结点都是可以得到的:要么是可以计算得到的,要么就是不需要计算而与关键子图 G 共享的.这样,在不同的时空中对相同的数据进行冗余计算,就可以检测和避免计算中的错误.

关键子图的复制算法如算法 3 中 *Copy*(*G*,*g*)所示.*Copy*(*G*,*g*)中按照映射函数 *g*,复制 *G* 中除起始结点以外的所有结点和有向边,而所有的起始结点则被关键子图 *G* 和副本关键子图 *G*′共享.图 5(a)表示复制之前的错误流图,其中的实线表示完全关键子图 *G*.图 5(b)表示复制后的错误流图.图 6 表示复制部分关键子图情形.事实上,任何错误流图都是自身的一个关键子图,所以 *Copy*(*G*,*g*)也可以用来复制一般的错误流图.

算法 3.

PROCEDURE *Copy*(*G*,*g*)
INPUT:错误流关键子图 *G*,结点映射函数 *g*;
OUTPUT:复制错误流关键子图 *G* 后具有冗余容错功能的错误流图.
BEGIN
  FOR (*G* 中每一个结点 *u* 和有向边⟨*u*,*v*⟩) DO
    IF (*u* 是起始结点) DO
      生成结点 *g*(*v*)和有向边⟨*u*, *g*(*v*)⟩
    ELSE
      生成结点 *g*(*v*)、结点 *g*(*u*)和有向边⟨*g*(*u*),*g*(*v*)⟩
END



(a) (b)

Fig.5 Duplicating full critical subgrap

图 5 复制完全关键子图

映射函数 g 的确定要考虑到寄存器分配时的冲突和使用效率以及对性能的影响.如果可供分配的寄存器从 0 到 m 连续编号,我们可以简单地选取结点映射函数 g 为 $g(n)=m-n$,其中,n 是待复制关键子图中使用的寄存器编号,这主要是由于从寄存器文件的末尾开始分配可以减小寄存器的使用冲突.在我们的实验中,SimpleScalar/PISA 体系结构有 32 个单精度浮点寄存器,从 0 到 31 编号;有 16 个双精度浮点寄存器,从 0 到 30 偶数编号.为了提供未来对双精度浮点数的扩展,我们采用 $g(n)=30-n$.在我们的例子中,复制前没有使用编号超过 16 的单精度浮点寄存器,所以也可采用 $g(n)=16+n$.但是考虑到关键子图和副本关键子图有可能同时生成,所以不能预先知道已经使用的寄存器最大编号,故采用 $g(n)=30-n$ 较为稳妥.

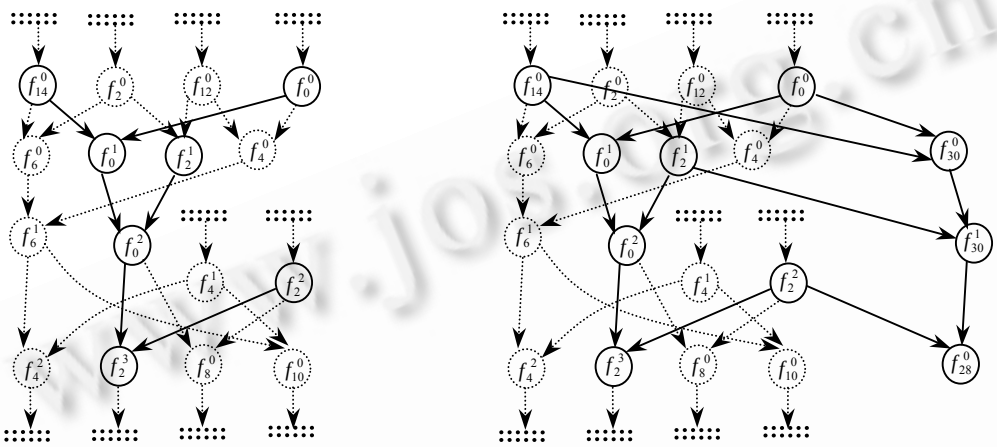


Fig.6 Duplicating partial critical subgrap

图 6 复制部分关键子图

4.2 部分冗余算法

部分冗余算法通过仅复制关键子图实现冗余和容错.部分冗余算法 *PartCopy*(*G*,*Evaluate*)见算法 4,其中的关键子图通过选取恰当的关键结点集合生成.

算法 4.

```
PROCEDURE PartCopy(G,Evaluate)
INPUT:程序流图 G;
OUTPUT:经过部分冗余算法编译后的错误流图 H.
BEGIN
   FOR (所有的 T⊆Out(G)) DO
      IF (Evaluate(Full(G,T),G)) DO
         BREAK
   H1←Copy(Full(G,T),g)
   H←G∪H1
   FOR (所有的 t∈T) DO
   BEGIN
      选定比较计算 e(x,y)
      V(H)←V(H)∪{e(t,g(t))}
      E(H)←E(H)∪{⟨t,e(t,g(t))⟩,⟨g(t),e(t,g(t))⟩}
   END
END
```

在 *PartCopy*(*G*,*Evaluate*)中,G 是要进行部分冗余的错误流图.在 *Evaluate*(*X*,*Y*)中,X 是 Y 的子图,

Evaluate(*X*,*Y*)从容错的角度评测 *X* 在 *Y* 中的关键程度.算法中,*Out*(*Z*)表示错误流图 *Z* 的终止结点集合.算法中使用的 *Full*(*G*,*T*)就是第 3.1 节提出的通过关键结点集合生成关键子图的算法,*Copy*(*G*,*g*)就是第 4.1 节中提出的关键子图复制算法.

在算法运行过程中,先遍历错误流图 *G* 终止结点集合的所有子集,并通过 *Evaluate* 函数评测由这些子集生成的关键子图.如果某个关键子图符合要求,就复制该关键子图,并加入检测错误的比较指令,最终形成我们需要的部分冗余的错误流图.

4.2.1 关键结点集合的选取

部分冗余算法中的关键一点在于,如何选取恰当的关键结点集合来生成关键子图.一般情况下,我们都从终止结点集合中选取关键结点,因为根据第 2.4 节的分析,终止结点集合可能汇聚所有结点的错误,所以对错误的传播和分布具有关键性的影响.但是,究竟选取哪些终止结点作为关键结点集合,还需要从结点的错误概率、结点生成关键子图的覆盖率和复制后的运行开销等方面综合考虑和判断:1) 我们可以根据关键结点集合本身的错误概率来判断它们的关键程度,错误概率大的结点关键程度相对较高,否则相反;2) 对于由关键结点集合生成的关键子图 *X*,我们也可以通过 *X* 的覆盖率来判断相应关键结点集合的关键程度,覆盖率高的关键子图对应的关键结点集合关键程度较高,否则相反;3) 也可以根据在部分冗余算法中,复制关键子图后的程序是否有利于寄存器分配、是否有利于提高性能以及降低功耗等原则来修正这种判断.

对于由关键结点集合 *T* 生成的关键子图 *Full*(*G*,*T*),在部分冗余算法中,通过 *Evaluate*(*Full*(*G*,*T*),*G*)判断 *T* 的关键程度.*Evaluate*(*X*,*Y*)最终选取这样的终止结点集合作为关键结点集合:它所生成的关键子图覆盖了更多的结点,同时,复制该关键子图后还能够有利于提高性能和降低功耗.详见第 5.1 节的实验分析.

在分支指令较多、有向边的执行概率各不相同的情况下,我们可以通过选取恰当的关键路径生成关键子图.通常,我们选取的关键路径应该包含关键结点,同时应该优先选取其中执行概率较大的路径.

5 模拟实验

实验使用Wattch 1.02 模拟器[17],指令集为SimpleScalar/PISA指令集[15].出于节能上的考虑,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[18].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.

我们的实验采用的是麻省理工学院StreamIt项目[16]提供的C版本的快速傅里叶变换测试程序.我们针对最内层循环体中的浮点计算进行容错编译,其最内层循环体的错误流图*G*在图 3~图 6 中都已作为图(a)出现过,后文中图 7 的左半部分表示的也是同一个错误流图*G*.

我们在 RedHat Linux 9.0 上使用专门为 SimpleScalar/PISA 指令集优化过的一个版本的 gcc 2.7.2.3 交叉编译生成无容错功能的PISA汇编代码,编译优化选项为*–O*3;然后再对汇编代码进行容错优化;最后从具有容错功能的汇编代码生成目标代码.

5.1 实验分析

实验中,通过分析我们发现,由终止结点集合{ f_4^2, f_2^3 }作为关键结点集合生成的完全关键子图,与由终止结点集合{ f_{10}^0, f_8^0 }生成的完全关键子图,除关键结点集合本身以外完全相同,如图 7 中的斜线和横线结点所示.所以,在算法中我们选取{ f_4^2, f_2^3 }作为关键结点集合生成完全关键子图.得到的好处是:1) 能够覆盖绝大多数的结点;2) 不必为 f_8^0 和 f_{10}^0 生成副本结点,减少了两次计算,少分配两个浮点寄存器,同时减少了两对用于检测错误的比较和分支指令,如图 7 中的虚线所示.实验数据表明,这些变化能够有效地提高性能并减小功耗.表 1 列出

了部分冗余(partial)和完全冗余(full)相比指令数量的变化情况.

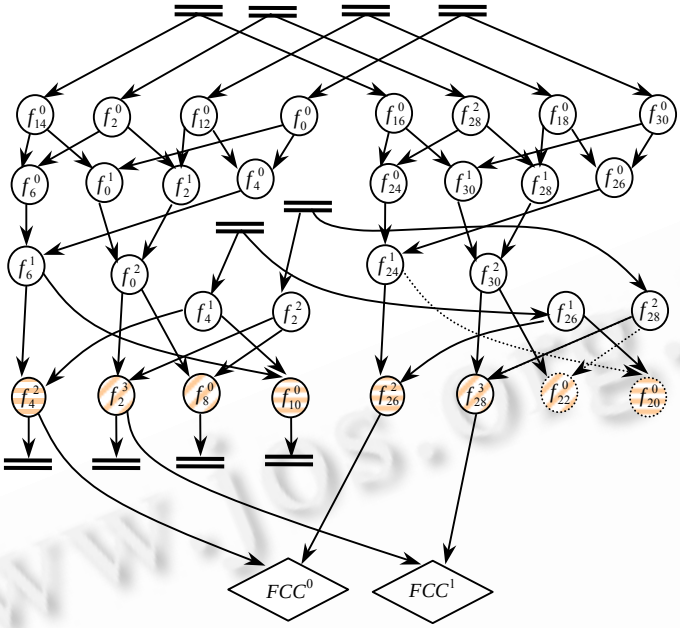


Fig.7 Critical subgraph from critical node

图 7 从关键结点生成关键子图

Table 1 Instruction summary

表 1 指令类型特性

Instruction type	None	Full	Partial
Load	6	6	6
Store	4	4	4
FP arithmetic	10	20	18
FP compare	0	4	2
FP cond. branch	0	4	2
Summary	31	53	47

5.2 实验结果

实验数据表明,部分冗余与完全冗余相比,最多可以分别减少功耗(total_power_cycle_cc3)超过 17%,减少执行总周期数(sim_cycle)接近 26%和提高性能(sim_IPC)超过 22%,如图 8 所示.部分冗余算法的优化效果,甚至要好于错误流压缩算法[8].

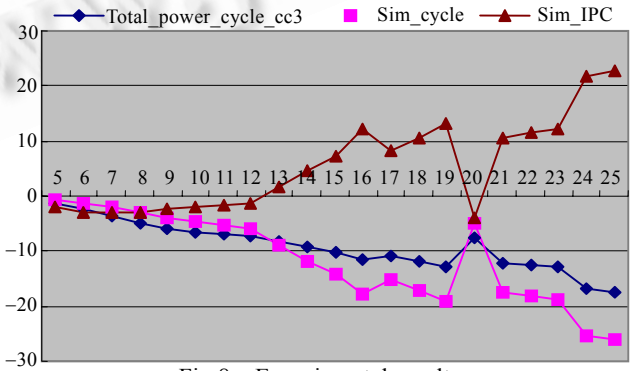


Fig.8 Experimental results

图 8 实验结果

图中的横坐标表示快速傅里叶变换的循环参数的对数.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所占的比例,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.

5.3 评测分析

5.3.1 存储和性能分析

部分冗余少使用了 2 个浮点寄存器,比完全冗余的最内层循环减少了 6.25%;减少功耗超过 17%,减少执行总周期数接近 26%,并且提高性能超过 22%.可见,部分冗余在存储和性能方面的提升是非常显著的.

5.3.2 容错分析

与完全冗余相比,部分冗余对于结点错误的覆盖率有所下降.如图 7 所示,由于没有为终止结点 f_8^0 和 f_{10}^0 生成副本结点,也没有进行比较,所以,直接发生于 f_8^0 和 f_{10}^0 中的错误难以检测.但是,由于除了 f_8^0 和 f_{10}^0 以外,其他结点的错误都汇聚于另外两个终止结点 f_4^2 和 f_2^3,所以,通过检测 f_4^2 和 f_2^3 中是否发生错误,是可以覆盖其余所有结点的错误的.我们损失的结点覆盖率仅为 f_8^0 和 f_{10}^0 在错误流图中所占的比例,也就是 6.25%.

6 结 论

在我们的实验中,部分冗余算法以损失 6.25%结点覆盖率为代价,使得减少功耗超过 17%,减少执行总周期数接近 26%,提高性能超过 22%,存储开销减少 6.25%.可以预计,部分冗余能够有效减少完全冗余带来的存储和性能开销.

7 未来的工作和展望

在裁减完全关键子图时,可能需要选择错误概率较低的结点进行裁减,以保证裁减后的部分关键子图仍然保留大部分的关键属性,这需要在 *Part*(*G*,*Choose*)算法中设计更为细致的 *Choose* 函数.而在复制错误流关键子图的算法 *Copy*(*G*,*g*)中,也可以根据需要选取更为综合考虑的结点映射函数 *g*,甚至在程序运行过程中动态改变映射的规则.使用更多的测试用例,也将能够更好地验证部分冗余算法的效果.我们也正在开发一款基于 SimpleScalar、能够进行错误注入(fault injection)和容错验证的模拟器,这也将有利于我们从实验的角度验证算法的容错性能.

References:

[1] Some RR, Ngo DC. REE: A COTS-based fault tolerant parallel processing supercomputer for spacecraft onboard scientific data analysis. In: Proc. of the 18th Digital Avionics Systems Conf., Vol.2. St. Louis: IEEE CS, 1999. 7.B.3-1−7.B.3-12.

[2] Madeira H, Some RR, Moreira F, Costa D, Rennels D. Experimental evaluation of a COTS system for space applications. In: Proc. of the Int'l Conf. on Dependable Systems and Networks (DSN 2002). Bethesda: IEEE CS, 2002. 325−330.

[3] Katz DL, Springer PL, Granat R, Turmon M. Applications development for a parallel COTS spaceborne computer. In: Proc. of the 3rd High Performance Embedded Computing (HPEC'99). Lexington: IEEE CS, Lincoln Laboratory, MIT, 1999.

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

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

[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] 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.

[8] 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 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/17/2425.htm

[9] Gao L, Yang XJ. Error flow model: Error propagation modeling and analysis based on computational data flow model. Journal of Software, 2007,18(4):808−820 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/18/808.htm

[10] Gao L, Wang ZY, Jia J, Yang XJ. Promote performance of highly reliable space computer by software implemented hardware fault tolerance based on COTS. Journal of Computer Research and Development, 2007,44(Suppl.):133−139 (in Chinese with English abstract)

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

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

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

[14] Oh N, Mitra S, McCluskey EJ. ED^{4}I: Error detection by diverse data and duplicated instructions. IEEE Trans. on Computers, 2002, 51(2):180−199.

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

[16] http://cag.csail.mit.edu/streamit

[17] 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 (ISCA 2000). Vancouver: IEEE CS, 2000. 83−94.

[18] Freescale Semiconductor Inc. MPC7447A RISC microprocessor hardware specifications. Technical Data, Rev. 3,08/2005, Chandler: Freescale Semiconductor Inc., 2005.

附中文参考文献:

[8] 高珑,杨学军.高效的容错编译技术:通过压缩错误流减小分支指令的功耗和性能开销.软件学报,2006,17(12):2425−2437. http://www.jos.org.cn/1000-9825/17/2425.htm

[9] 高珑,杨学军.错误流模型:基于计算数据流模型的错误传播建模与分析.软件学报, 2007,18(4):808−820. http://www.jos.org.cn/1000-9825/18/808.htm

[10] 高珑,王之元,贾佳,杨学军.通过基于 COTS 器件的软件容错技术提高空间高可靠计算机的性能.计算机研究与发展,2007,44(增刊):133−139.

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.

王之元(1982－),女,博士生,主要研究领域为光计算,遥感图像处理.

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.