分类号 TP301 学号 09069019

UDC 密级 公开

工学博士学位论文

面向瞬时故障的可配置容错技术研究

博士生姓名 李建立

学科专业 计算机科学与技术

研究方向 计算机软件与理论

指导教师 谭庆平 教授

国防科学技术大学研究生院

二〇一三年十二月

Research on Configurable Fault Tolerance Techniques for Transient Faults

Candidate: Li Jianli

Supervisor: Professor Tan Qingping

A dissertation

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Engineering

in Computer Science and Technology

Graduate School of National University of Defense Technology

Changsha, Hunan, P. R. China

December 9, 2013

独 创 性 声 明

本人声明所呈交的学位论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表和撰写过的研究成果，也不包含为获得国防科学技术大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文题目：面向瞬时故障的可配置容错技术研究

学位论文作者签名：李建立 日期：2013 年 11 月 4 日

学位论文版权使用授权书

本人完全了解国防科学技术大学有关保留、使用学位论文的规定。本人授权国防科学技术大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档，允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密学位论文在解密后适用本授权书。）

学位论文题目：面向瞬时故障的可配置容错技术研究

学位论文作者签名：李建立 日期：2013 年 11 月 4 日

作者指导教师签名：谭庆平 日期：2013 年 11 月 4 日

目 录

表 目 录

图 目 录

摘 要

随着处理器设计朝更小的晶体管特征尺寸、更低的工作电压和更高的频率发展，瞬时故障引发的可靠性问题已经引起整个计算市场的关注。由于不同领域的用户对系统可靠性、成本、性能、功耗等指标的要求不同，如何面向不同用户的不同需求提供可靠性和代价满足约束的可靠性解决方案，成为处理器设计者必须面对的挑战。为了应对这种挑战，本文重点研究了可配置、低代价的容错保护技术。此外，为了分析瞬时故障的影响和容错技术的可靠性，本文也研究了基于故障注入的可靠性分析技术。具体来说，本文工作可以分为以下四个方面：

1. 处理器运算单元中的故障可能导致程序运行出现数据流错误或控制流错误。其中，数据流错误检测通常基于冗余计算的方法进行，如何降低冗余计算的开销（性能、硬件开销等）是困扰容错研究至今的难点问题。为了解决这一问题，本文结合软、硬件容错技术的优势，提出了一种可配置的数据流检测技术 Epipe。Epipe 首先通过改造现有的超标量流水线处理器，提供了一个能够对指令进行选择性冗余保护的硬件平台。由于超标量处理器中有丰富的计算资源，Epipe 平台只需要很少的硬件开销。为了减少冗余保护产生的性能开销，Epipe 还基于程序分析方法评估每个指令的重要性，即指令发生故障后导致程序输出错误结果的概率。程序运行时，Epipe 根据用户的性能和可靠性要求选择保护最重要的一部分指令。Epipe 的创新点在于，Epipe 只冗余保护发生故障后导致程序输出错误结果的指令，对于导致系统异常或超时的故障则直接利用系统中的异常检测机制加以处理，而剩余的不会影响程序执行的故障（即被屏蔽的故障）则不需要任何处理。这种分类处理故障的方法有效地减少了需要冗余保护的指令，再结合时空开销较低的硬件指令保护技术，使得 Epipe 技术可以更低的开销保护程序数据流。
2. 实现控制流检测的一种有效技术是软件实现的标签分析方法。已有的标签分析技术除了存在时空开销过大和可靠性不足的问题外，还缺乏可配置性，无法满足不同用户的不同需求。此外，软件检测技术引入的冗余代码自身也有可能发生错误，现有的控制流检测技术在容错机制的自我保护方面缺乏研究。为了克服上述不足，本文提出了一种可配置的控制流检测算法 CFCES。CFCES 通过为每个程序块设计特殊格式的标签并在其中插装额外的控制流检测指令，以较少的开销有效地克服了已有算法的检测盲点。而且，CFCES 在设计检测机制时引入了一种被称为“对等性”的不变量，通过对这种不变

量进行检测，CFCES 能够以极低的代价实现检错机制的自容错保护。此外，CFCES 还通过分析函数的重要性和调节程序块的大小提供了可配置的优化方法，可以满足用户不同的时空开销和可靠性约束。CFCES 优化方法的特点在于其可以提高 CFCES 的容错效率，且可以用于优化其它基于标签分析的控制流检测算法。

3. 瞬时故障不仅可能发生在处理器运算单元，也有可能出现在处理器存储单元中。被广泛用于保护片外存储的 ECC 技术并不适合用来保护片上存储结构，原因是这些存储结构本身已经占用了大部分芯片面积，并且访问频繁，采用 ECC 保护会带来大量的面积、性能和功耗开销。鉴于现有的容错研究中十分缺乏针对片上存储结构的合理保护方案，本文针对一种特殊的片上存储结构 SPM 提出了低代价的保护技术 PPS。尽管用 ECC 对 SPM 进行完全保护的开销很高，但是对部分 SPM 存储进行 ECC 保护并进行合理分配仍是非常有价值的。PPS 技术首先设计了基于部分 ECC 保护 SPM 的存储体系结构(被保护的比例可以根据不同应用的可靠性、性能等需求决定)，然后对程序中的待分配变量进行脆弱性分析，并将 SPM 空间划分为“寄存器”，最后采取基于优先级的图着色方法将较为脆弱的变量优先分配到 ECC 保护的“寄存器”中。基于上述方法，PPS 能够以较低的开销获得较高的存储可靠性。
4. 故障注入是一种有效且广为应用的可靠性分析方法。故障注入技术面临的困难是如何平衡故障模拟速度与精度的关系。由于已有的故障注入技术还不能有效地解决上述问题，本文提出了一种新的故障注入框架 SmartInjector。SmartInjector 首先基于程序分析从故障空间中删除等价类故障和结果确定型故障。等价类故障是指发生在相似的数据流或控制流上下文环境中的故障。这类故障往往会导致系统产生相同的反应，因此只需要将它们划为等价类并从中选取代表进行模拟注入即可，等价类中其它故障则可以从故障空间中删除。结果确定型故障是指那些通过程序分析就可以确定系统反应的故障。SmartInjector 还首次开发了一种故障结果预测技术，通过预测故障产生的结果和判定结果的位置，可以在程序运行结束前提前判断故障注入的结果，从而减少单次模拟的时间开销。结合提出的故障删除技术和故障结果预测技术，SmartInjector 以少量的精度损失极大地减少了故障注入的时间开销。

关键词: **瞬时故障；程序分析；数据流检测；控制流检测**

ABSTRACT

Processor design trends toward smaller transistors, lower core voltage and higher frequency make transient faults become a critical reliability concern for the entire compute market. As the users from different fields usually have different requirements on reliability, hardware cost, performance and power consumption, how to enhance the system reliability considering different reliability and overhead constraints poses a challenge to the processor designers. To meet such a challenge, this dissertation focuses on configurable and low-cost fault tolerance techniques. In addition, to evaluate the impact of transient faults and the reliability of the proposed fault tolerance techniques, this dissertation also researches on fault injection techniques. To be specific, the main contributions of the dissertation are as follows:

1. The faults in computation units of a processor may induce data-flow errors (DFEs) or control-flow errors (CFEs) during program execution. Existing techniques usually rely on redundant computation to detect DFEs. How to reduce the considerable overheads (in terms of performance, hardware, etc.) introduced by redundant computation is an intractable issue for the DFE detection techniques. Combining the advantages of both hardware- and software-based fault tolerance solutions, this dissertation presents a configurable DFE detection technique called Epipe. By making minimal modifications to a modern superscalar processor, Epipe firstly provides a hardware platform which can selectively protect instructions by performing instruction replication. Since there exist abundant compute resources in a modern pipeline, the extra hardware cost incurred by the Epipe platform is low. To reduce the performance overheads introduced by redundant computation, Epipe analyzes the criticality of every instruction, i.e., the probability that an instruction makes the program produce incorrect results when subjected to transient faults. During program execution, Epipe chooses a subset of the most critical instructions to protect according to the reliability and the performance requirements. The novelty of Epipe lies in the fact that Epipe only protects a part of instructions which tend to cause incorrect program outputs. Meanwhile, the faults that trigger system exceptions can be detected easily by existing exception detection mechanisms in systems. The remaining masked faults can be ignored directly since they do not ultimately prop-

agate to user-visible corruptions. By handling different faults in different strategies and developing low-cost hardware protection technique, Epipe reduces the instructions which need replication significantly, and finally can detect DFEs efficiently.

2. An effective method to detect CFEs is software-implemented signature monitoring. Existing signature monitoring based techniques not only have drawbacks in terms of performance overhead, memory overhead, and reliability, but also lack configurability, thus cannot accommodate different requirements from different users. In addition, the extra instructions introduced by software solutions may also be corrupted by faults. Unfortunately, existing CFE detection techniques cannot provide self-protection. Given this situation, this dissertation proposes a configurable control-flow checking algorithm, i.e., CFCES. By allocating formatted signature for each program block and instrumenting control-flow checking instructions into the program blocks, CFCES can detect more faults than existing control-flow checking algorithms with moderate overheads. Meanwhile, CFCES embeds an invariant (i.e., equality) into the designed control-flow checking mechanism. By checking this invariant, CFCES can detect the faults happening in control-flow checking instructions under extremely low cost. Furthermore, CFCES provides a configurable optimizing approach by analyzing the criticality of each function and tuning the granularity of the program blocks, so that the specific overheads and reliability constraints of different users can be satisfied. The optimizing approach can improve the fault tolerance efficiency of CFCES, and can be applied to other signature monitoring based algorithms.
3. Transient faults not only occur in computation units of a processor, but also exist in memory units. Because the on-chip memory structures occupy most area of a processor and are frequently accessed, traditional memory protection techniques like ECC are not appropriate for the on-chip memory structures due to prohibitive hardware area, performance and power overheads. This dissertation focuses on protecting a special kind of on-chip memory structure, i.e., SPM, and proposes a low-cost fault tolerance technique named PPS. The key insight behind PPS is that although leveraging ECC to protect the whole SPM is expensive, protecting part of SPM and then making reasonable use of the protected memory units is still meaningful. PPS firstly designs a SPM-based memory architecture which only protects a part of the SPM units (the proportion of the protected units can be varied with the differ-

ent reliability, performance and power requirements of different applications), then performs vulnerability analysis for all the variables to be allocated and divides SPM into different pseudo-registers, finally, PPS allocates the most vulnerable variables into the protected pseudo-registers by performing priority-based graph coloring. In this manner, PPS can provide effective memory protection with limited overheads.

4. Fault injection is an effective and widely-used approach for reliability evaluation. The drawback of this approach is that it is difficult to trade-off between simulation speed and accuracy. This dissertation develops an efficient fault injection framework, named SmartInjector, by performing program analysis. SmartInjector firstly removes the equivalence class faults and the known-outcome faults from the initial fault space based on program analysis. The equivalence class faults are the faults which happen in similar control-flow or data-flow context. Since this kind of faults tend to influence a program similarly and induce the same fault outcomes, SmartInjector only selects a representative from each equivalence class to study through a detailed fault simulation and prunes the faults it represents. The known-outcome faults are the faults whose outcomes can be obtained by conducting program analysis only. SmartInjector also exploits a fault outcome prediction technique, which can determine the outcome of a simulation before it runs to completion by predicting the fault outcome with location. In this way, the simulation time for a single simulation can be reduced. Employing the proposed fault pruning and fault outcome prediction techniques, SmartInjector reduces the requirement of computational resources prohibitively while maintaining high accuracy of analysis results.

Key Words: Transient Faults；Program Analysis；Data-flow Checking；Control-flow Checking

第一章　绪论

1.1　研究背景

1.1.1　瞬时故障的定义

随着集成电路设计朝向更高的频率、更高的晶体管密度和更低的工作电压发展，瞬时故障（Transient Fault）引发的可靠性问题已经引起愈来愈多的关注。瞬时故障是指计算机在辐射环境或者电磁干扰等因素的影响下，内部的逻辑状态由1变为0或者由0变为1的现象。不同于计算机部件中的永久性故障（Permanent Fault）造成的硬损伤，瞬时故障具有随机发生、重写时可恢复、持续时间短等特征，因此又被称为软错误（Soft Error）。计算机集成电路中瞬时故障发生的机理如图1.1所示。计算机内部利用晶体管保持或释放一定的电荷来表示逻辑状态1或0，而当计算机处于电磁辐射环境中时，来自外界的高能粒子击打晶体管PN结，可以使其产生瞬时充放电，从而使其中保存的逻辑状态发生翻转。

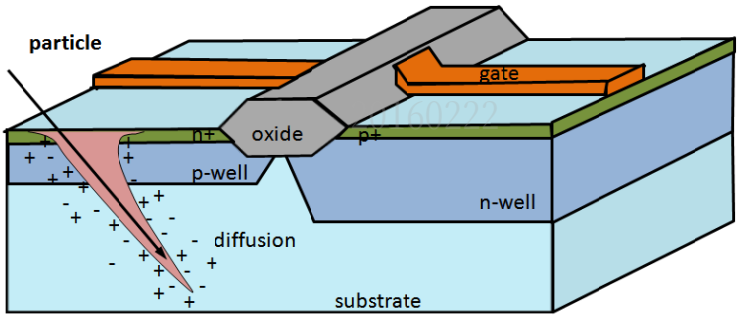


图1.1　高能粒子轰击PN结

辐射造成的瞬时故障可以分为单粒子翻转（Single Event Upset，SEU）和单粒子瞬态（Single Event Transient，SET）两种类型[1]。如果高能粒子击打造成时序逻辑中存储的逻辑值发生翻转，则称该故障为SEU故障。在组合逻辑网络中，逻辑值以信号脉冲的形式在逻辑器件之间传输。如果在信号传输过程中发生高能粒子轰击或电磁噪声干扰，则会产生宽0.35ns到1.3ns的信号毛刺[2]。这种信号毛刺一旦达到改变逻辑值的强度并被锁存器采样，则同样会造成传输的值发生错误，这种故障称为SET故障。计算机集成电路中的瞬时故障主要表现为SEU故障，对其最为敏感的部件是DRAM、SRAM、寄存器等存储部件。电子器件发生瞬时故障的概率通常用软错误率（Soft Error Rate，SER）表示。处理器中的运算单元由于占用面积较小，而且具有较强的故障屏蔽能力，其SER要比存储部件小很多[3]。概括来说，运算单元中存在以下三种故障屏蔽效应[1]：（1）逻辑屏蔽：

这种屏蔽作用是指瞬时故障在逻辑运算过程中被屏蔽，没有影响输出结果。例如，如果或门的一个输入状态是 1，则另一个输入中存在的故障将不会影响逻辑运算结果；（2）电路屏蔽：引发瞬时故障的异常脉冲在沿逻辑门传输时，有可能因为信号的衰减而不会影响最终的运算结果，从而屏蔽了瞬时故障的影响；（3）锁窗口屏蔽：运算单元中的锁存器（Latch）只有在 CPU 时钟的上升沿或下降沿才会进行数据采样。如图1.2所示 [1]，如果电路中存在的 SET 信号毛刺（glitch）没有出现在锁存器的采样窗口（Latching window）中，则不会对运算单元造成任何影响。

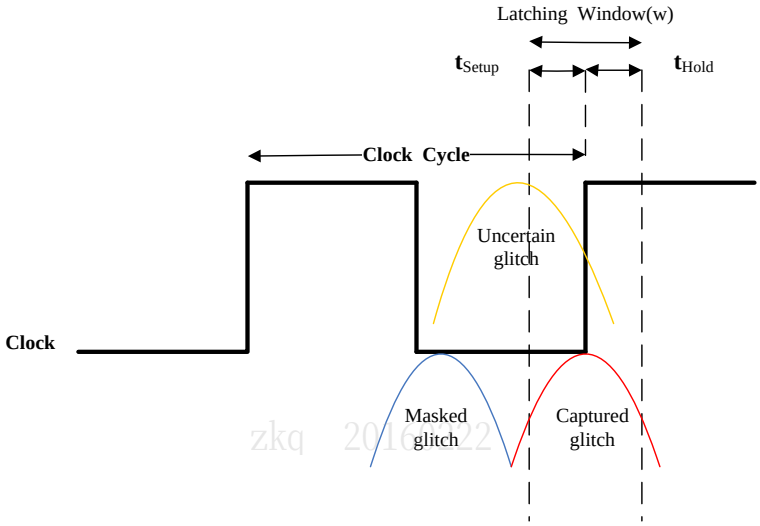


图 1.2　锁窗口屏蔽效应

尽管上述屏蔽效应使得处理器运算单元的 SER 与存储部件相比要小得多，但是，随着集成电路制造工艺水平的提高，系统工作频率不断升高、工作电压不断降低，这些因素将会削弱运算单元的故障屏蔽能力。以图1.2所示的锁窗口屏蔽效应为例，假设一个信号毛刺的持续时间 t_{glitch} 小于系统的时钟周期 T_{cycle}，则其最终导致 SET 故障的概率可以基于公式 (1.1) 估计 [4]:

$$P_{latch} = \frac{t_{glitch} + w}{T_{cycle}} \tag{1.1}$$

其中 w 表示锁存器的采样窗口。根据上述公式，运算单元中发生 SET 的概率将会随系统工作频率的升高而增加。文献 [3, 5, 6] 均证实了 SET 发生概率与系统工作频率的这种线性关系。因此，随着处理器性能的不断提高，运算单元中的 SER 将会显著上升，未来有可能达到与存储单元相当的程度 [7, 8]。所以，除了存储部件中的故障，处理器的设计也不得不开始考虑运算单元中的瞬时故障问题。

1.1.2 引发瞬时故障的原因

瞬时故障的问题几乎从计算机诞生之日起就开始出现了。早在 1954-1957 年人类进行地面核试验的过程中，就发现电子监控装备出现了很多反常行为 [9]。随后在 20 世纪 60 年代进行的太空活动中，也发现了空间电子设备存在类似的问题。只是在当时的条件下，人们还没有将这些系统失效和瞬时故障联系起来。直到 1978 年，Intel 首次发现了封装材料中的辐射粒子导致 DRAM 发生瞬时故障的证据。随后，IBM 在 1979 年发现了大气中宇宙射线导致瞬时故障的现象。今天，人们对瞬时故障的发生机理和原因的认识更为系统。总结来说，集成电路中瞬时故障产生的原因既与空间辐射和核辐射等外部环境因素有关，又与封装材料的放射性污染、电源电压不稳和电路信号耦合等内部因素相关。下面将具体分析这些影响因素。

1.1.2.1 系统工作环境中的高能粒子

首先，大气层外的太空中存在大量高能粒子，能够对各种航天器、甚至地面设备构成严重威胁。这些高能粒子主要有三种来源：宇宙射线、太阳风和范艾伦辐射带。

宇宙射线是指以不同方向进入太阳系的高能重粒子流（例如射线暴）。这些来自太阳系之外的高能粒子一般是脱离了电子的原子核，主要包括高能量的质子和粒子。宇宙射线中的高能粒子虽然密度较低（约为几个粒子 $/cm^2 \cdot s$），但是由于其能量很强（从几个 MeV 到几个 GeV），脱离地球磁场保护的航天器必须针对这些高能粒子采取加固措施进行防护。太阳风是太阳内部核聚变反应释放出的高速等离子体带电粒子流，主要由质子和电子组成（能量大约 500eV）[10]，还有少量氦核及微量重离子成分。在地球附近，太阳风速为 200-889 km/s，平均值为 450 km/s。太阳风的强度受太阳活动周期（大约为 11 年）的影响。在太阳活动的极大年，太阳风往往会影响地球轨道上的航天器，甚至穿透地球大气层，破坏地面的通讯、电力和其它电子设备。例如，1989 年强太阳风暴切断了加拿大魁北克省的电网，造成 600 万人断电，与此同时，日本的通信卫星 CS-3B 被彻底损坏，美国海军实验室的 DMSP 等 46 个卫星也出现操作异常。地球磁场可以捕获太阳风中的高能粒子，如质子、电子以及少量的重粒子。如图1.3所示，这些被捕获的粒子沿着地磁场磁力线螺旋运动，并在地磁两极之间来回振荡，同时伴随着东西方向漂移，形成内带和外带两个辐射带，统称为范艾伦带。范艾伦带由科学家詹姆斯 • 范艾伦于 1958 年发现，其内带位于地球上空 650 公里至 6300 公里，外带位于地球上空 1 万至 6.5 万公里。范艾伦带中的高能粒子是威胁人造卫星、空间站等近地空间航天器的主要辐射源。地球附近除了内带和外带之外的区域是相对

安全的区域，因此，近地空间航天器分别运行于如图1.3 所示 [7] 的近地轨道、中地轨道和同步轨道上。

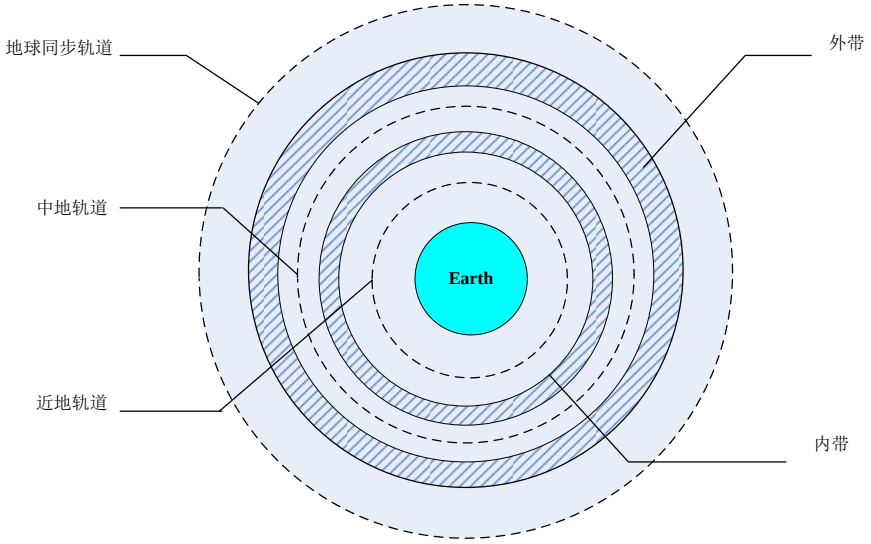


图 1.3　范艾伦辐射带

太空中的部分高能粒子可以冲破地球磁场的防护进入大气层，并和大气层中的原子核相互作用，产生以中子和介子为主的第二代、第三代粒子流。尽管这些粒子流的辐射强度会因空气的吸收作用，呈现随海拔高度降低而不断衰减的规律，但是，大气中的高能粒子流仍然能够对航空器以及地面上的电子设备造成危害。文献 [11] 表明，即便在海平面的高度上，完全消除高能粒子对电子设备的危害也需要超过 6 英尺厚的混凝土层保护。

除了自然环境中的辐射，人造的辐射源也可能对电子设备产生辐射危害，典型的例子是核爆炸环境和核反应堆环境 [12]。核爆炸会产生大量的 X 射线、γ 射线和中子 [13]。这些粒子的强度与核爆炸当量、核爆装置构造以及与爆心之间的距离有关。高空核爆炸能够形成范围广、持续时间长的人工辐射带。例如 1962 年美国进行 140 万吨级的“星鱼”核试验后一周，南大西洋上空 600 公里高度的电子通量达到了 $3\times10^5/(m^2\cdot s)$，造成 4 颗卫星不同程度地受损 [14]。核电站以及各种核动力装置中的核反应堆也会向周围环境中释放粒子流，其主要构成为低密度的 γ 射线和中子。这些粒子同样足以导致核反应堆环境中的电子设备发生瞬时故障。因此，用于各种核装置的控制、监控电子设备必须考虑核辐射带来的可靠性问题。

1.1.2.2　集成电路内部的干扰

集成电路的封装材料中含有微量放射性元素，这些放射性元素衰变时会释放出高能粒子，造成集成电路内的电子分布受到干扰，从而引发瞬时故障。IBM 公司于 1986 年到 1987 年发现，由于在生产过程中使用了受放射性污染的化学试剂，

导致集成电路出现异常 [15]。由于封装材料很难达到 100% 的纯净度（通常标准是粒子密度低于 $0.001 particles/(cm^2 \cdot hour)$），所以任何环境中的计算机都会遭受封装材料的放射性威胁 [16]。

集成电路内部还存在着多种噪声干扰，例如电磁干扰、串扰噪声以及热噪声等。通常认为电磁干扰有两种传输方式：一种是传导传输方式；另一种是辐射传输方式。传导传输必须在干扰源和敏感器之间有完整的电路连接，干扰信号沿着这个连接电路传递到敏感器，发生干扰现象。这个传输电路可包括导线，设备的导电构件、供电电源、公共阻抗、接地平板、电阻、电感、电容和互感元件等。辐射传输是通过介质以电磁波的形式传播，干扰能量按电磁场的规律向周围空间发射。集成电路中的各类引脚、接口都有可能成为像天线一样的电磁波发射源，影响其它部分电路的正常工作。串扰噪声是距离接近的信号线之间的耦合、互感和互容引起的线上噪声。热噪声又称白噪声，是在导体中由于带电粒子热骚动而产生的随机噪声，它存在于所有电子器件和传输介质中。热噪声是温度变化的结果，不受频率变化的影响。集成电路内的这些电磁噪声、干扰对系统可靠性的影响小于外界环境中的高能粒子。但是，随着芯片集成程度和系统工作频率的提高，处理器对这些干扰因素正变得越来越敏感。

1.1.3 瞬时故障的危害和发展趋势

瞬时故障虽然不会损坏硬件电路，但是却可以通过改变处理器状态或存储器内容等方式影响程序的正常运行，严重时甚至可能导致系统崩溃，带来灾难性后果。尤其对于长期处于恶劣宇宙辐射环境下的航天器，瞬时故障导致的各种灾难更为频繁，也更为严重。据有关资料 [17] 统计表明，自 1971 年至 1986 年，国外发射的 39 颗同步卫星共发生与空间辐射有关的故障 1129 次，其中，瞬时故障发生了 621 次，占辐射总故障的 55%。美国 MSTI 和 IRON9906 卫星分别于 1993 年和 1997 年因发生严重的瞬时故障而提前结束寿命 [18]。我国 1990 年发射的“风云一号”气象卫星也是由于瞬时故障造成了姿态控制系统失效，最终不得不提前退役。2010 年，美国执行星际探索任务的“旅行者 2 号”因为瞬时故障而发生了系统失效，在重启计算机系统后才恢复正常。

硬件瞬时故障同样也能导致地面计算机崩溃并造成重大损失，目前已有很多相关的案例：2004 年，Cypress 公司报告了一系列瞬时故障引发的事故，其中，发生在 2001 年底的“wake-up call”电话系统大混乱的技术原因是一次瞬时故障，它造成了一个交互系统的崩溃 [19]。2005 年，部署在 Los Alamos 国家实验室的一台 Hewlett Packard 超级服务器经常因为宇宙辐射造成的瞬时故障而崩溃 [20]。2009 年，一份研究报告指出，部署在 Lawrence Livermore 国家实验室的一

台 BlueGene/L 超级计算机平均每 4 个小时就发生一次瞬时故障 [21]。此外，最近的研究也表明 [22]，瞬时故障可以破坏 RSA 公钥加密系统 [23, 24] 的私钥，从而带来安全漏洞。

综合上述情况可以看出，瞬时故障能够对计算机系统构成严重威胁。不幸的是，处理器技术的一些发展趋势正在使瞬时故障的发生概率持续上升，这些趋势包括：(1) 为了追求更高的性能，处理器的工作频率在不断升高；(2) 处理器上的晶体管密度一直遵循着摩尔定律迅猛增长 [25]，由此导致晶体管的特征尺寸不断减小，目前已经进入纳米时代（图1.4给出了 1971 年 -2011 年处理器上集成的晶体管的数量)；(3) 现代处理器的设计还受到严格的功耗约束，因此导致电路的工作电压也在不断下降（图1.5给出了处理器的工作电压变化趋势 [22])。处理器设计的上述趋势使得处理器的性能得到了大幅提高，但是对处理器的可靠性却带来了负面影响。

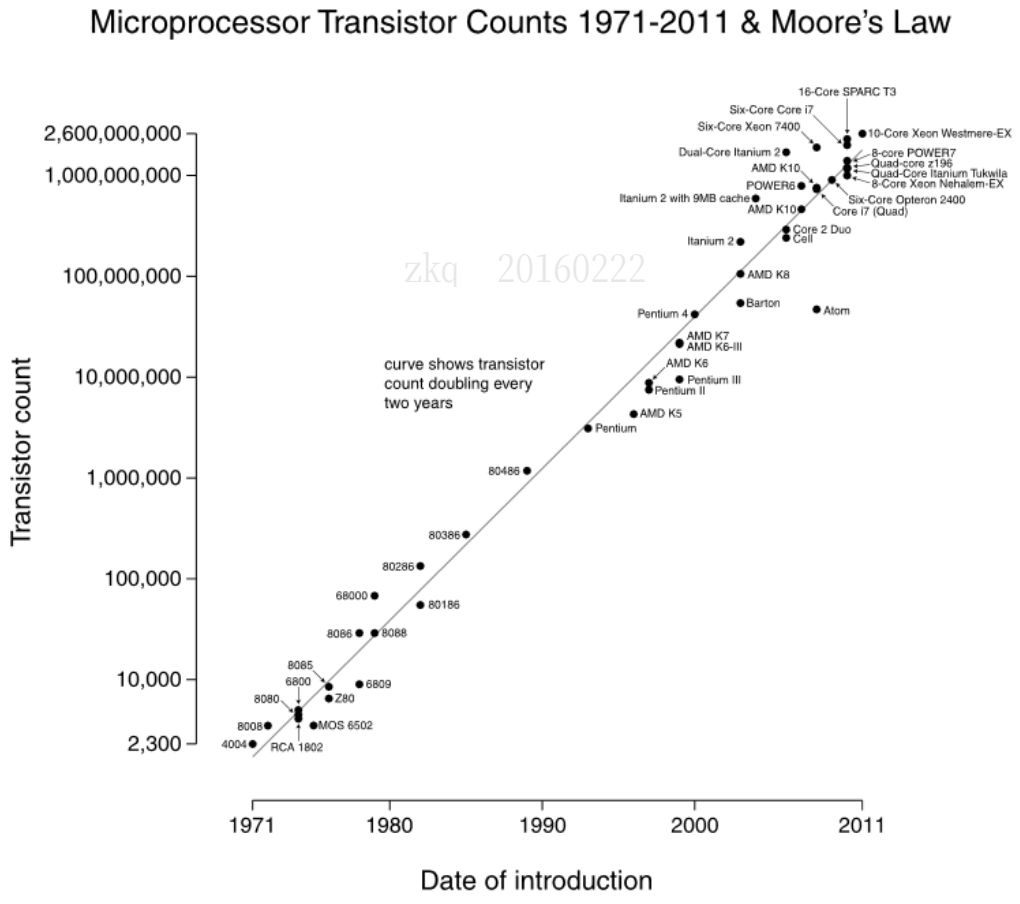


图 1.4　处理器晶体管数量与摩尔定律关系图

以存储系统的可靠性为例，其 SER 可以基于公式 (1.2) 估算 [3]:

$$SER \propto F \times A \times exp(-\frac{Q_{crit}}{Q_s}) \tag{1.2}$$

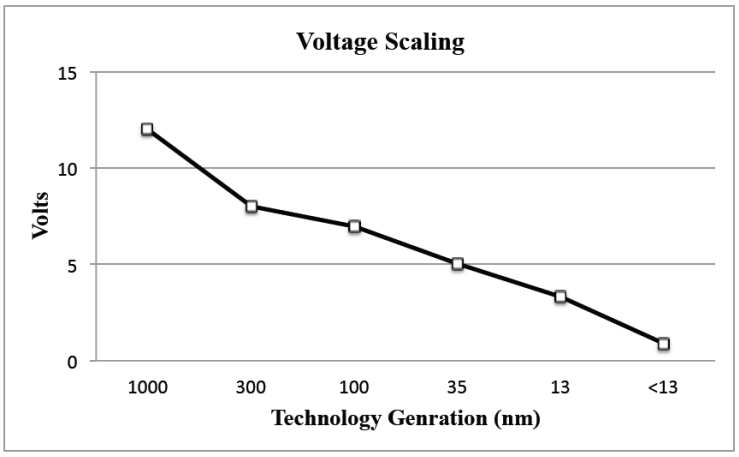


图 1.5　处理器工作电压的变化趋势

其中，F 表示能量大于 1MeV 的粒子密度。A 表示对高能粒子击打敏感的电路面积。Q_{crit} 表示导致存储单元发生瞬时故障所需的最小电量，称为临界电量 [3]。Q_s 则是高能粒子轰击晶体管导致的实际充放电量。SER 的单位是 $errors/s$。由于处理器工作电压不断降低，而临界电量 Q_{crit} 与工作电压 v 存在如下线性关系 [1]:

$$Q_{crit} \propto constant \times v \tag{1.3}$$

其中 $constant$ 表示常量值。所以，临界电量也将不断减少，这将导致能量较低的粒子、噪声也有可能引发瞬时故障，使得存储器的 SER 增加。另一方面，晶体管特征尺寸的缩小将会减少敏感电路的面积，从而降低了单个晶体管发生故障的概率。由于工作电压和特征尺寸均随制造工艺的进步而减少，所以单个晶体管发生瞬时故障的概率基本保持不变。但是由于集成电路的晶体管数量在呈指数级增长，所以存储系统的 SER 也会保持指数级增长。由于存在如1.1.1节所述的故障屏蔽效应，组合逻辑电路的 SER 和处理器工作频率、工作电压以及晶体管尺寸的关系更为复杂，但是也在不断增加 [26]。如图1.6所示 [22]，随着处理器制造工艺的不断进步，处理器总体的 SER 在迅速增加。文献 [27] 预测，采用 45nm 工艺的芯片每 100 片每月会发生一次瞬时故障，而在未来的 16nm 时代，每 100 片每天都会发生一次瞬时故障。

1.1.4　处理器可靠性设计的需求

瞬时故障的发展趋势使得传统追求高可靠性的系统（例如应用于航空、航天、战略武器等领域的系统）面临更为严重的瞬时故障问题。另一方面，对于用于大众消费电子领域的处理器芯片（例如智能手机、数码相机、平板电脑等产品中的芯片），随着 SER 的持续增加和部分对可靠性要求较高的应用（例如移动支付、

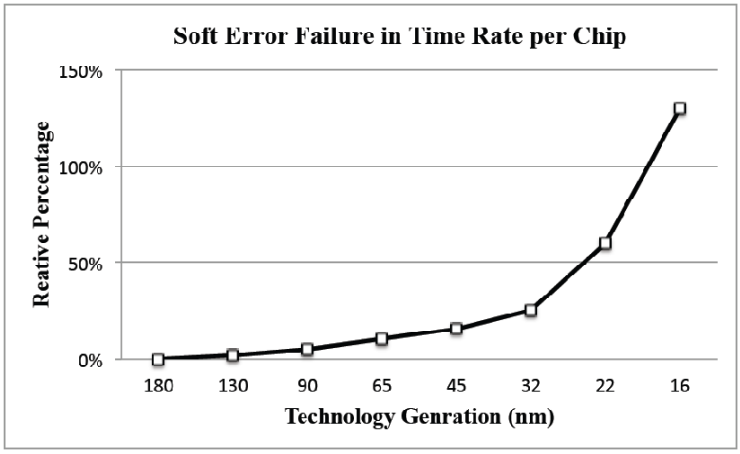


图 1.6　处理器 SER 的变化趋势

证券交易、移动导航等）逐渐推广，也不得不开始考虑瞬时故障引发的可靠性问题 [27–29]。

为了应对瞬时故障带来的问题，处理器必须进行可靠性设计。尽管可靠性设计最理想的效果是每一块芯片都是 100% 可靠的，但是从具体应用的角度来看，这种可靠性的重要程度是不同的，因此对可靠性的现实需求也是不同的。例如，与华尔街服务器出现小数点位置错误导致的后果相比，用户观看视频时个别像素出现故障造成的危害显然是微不足道的。除了可靠性，不同的系统对成本、性能等指标的需求也是不同的。总的来说，对于执行重要任务的系统来说，系统设计的首要目标是确保可靠性，而成本因素通常是次要的，因此这些系统为了追求完美的可靠性，往往会采用一些可靠性高但十分昂贵的保护技术。与之相反，主流的消费电子产品对可靠性没有严苛的需求，通常可以容忍偶尔的崩溃、挂起等。但是，这些系统对成本、性能、功耗等指标却有着严格的要求，用户不愿为可靠性设计付出高昂的代价。如何面向不同领域用户的不同需求，提供成本、可靠性满足约束的可靠性解决方案，成为处理器设计者必须面对的挑战，也是本文研究的目标。

1.2　面向瞬时故障的容错技术概述

1.2.1　瞬时故障的处理方法

克服瞬时故障的影响，实现可信计算有两种途径——避错（Fault Avoidance）与容错（Fault Tolerance）。其中，避错技术的目标是针对可能引发故障的各种因素，在设计或运行时采取预防性措施尽可能消除故障发生的可能，通常采取的措施有：从工艺上提高计算机元器件的可靠性，例如，提高集成电路的工作电压、减少封装材料污染的可能性等；对计算机系统进行屏蔽辐射以减少外部干扰，例

如，在航天计算机的表面覆盖铝箔。这些措施早已为计算机设计者所接受，并在提高计算机可靠性方面发挥了重要作用。

但是即使采用了避错技术，计算机系统还是有可能发生故障。因此，在设计高可靠系统时，还应当考虑采取容错技术。容错技术的目的是确保计算机在发生故障的情况下仍能继续完成指定的任务 [30]。容错技术与避错技术并非相互对立的，二者既可以单独使用，又可以相互补充。本文研究的瞬时故障处理方法即是容错技术。容错通常包括故障检测、定位、恢复等过程。其中，故障检测是故障定位及恢复的基础。而且，瞬时故障的偶发性使得执行故障恢复的几率很小，故障检测却不得不时刻执行。因此，目前大部分容错研究关注的问题都是故障检测，本文也将其作为主要的研究内容。

瞬时故障对系统可靠性的危害性在于其可以传播到应用程序中，破坏程序的运行。按照瞬时故障对程序运行的具体影响，可将系统中的瞬时故障大致分为：（1）良性故障（Benign）：这类故障在程序运行中能够被屏蔽，因此这类故障发生后程序仍然能够正常运行至结束并输出正确结果；（2）异常故障（Exception）：这类故障导致程序运行出现各种异常，如地址越界、非法指令、除 0 等；（3）超时故障（Timeout）：这类故障导致程序挂起或者运行时间远超预期；（4）结果错误故障（Silent Data Corruption，SDC）：这类故障发生后，程序表面上能够顺利运行至结束，但是输出的结果是错误的。上述四种故障类型中，第（2）、（3）类故障均导致程序运行出现反常症状，因此统称为症状型故障（Symptom）。此外，由于用户无法感知到 SDC 故障的存在，这类故障被认为是最危险的类型 [31, 32]，因此成为容错研究重点关注的目标 [27, 29, 33]。

1.2.2 容错技术的原理

容错技术的基本方法是冗余，具体来说包括以下四种形式 [30]：硬件冗余、信息冗余、软件冗余和时间冗余。其中，硬件冗余是指应用附加硬件来实现故障检测及恢复，典型的例子是三模冗余（Triple Modular Redundancy，TMR）和双模冗余（Dual Modular Redundancy，DMR）。图1.7左右两侧分别给出了 TMR 和 DMR 技术的原理图 [8]。TMR 技术同时运行三个相同的硬件模块，并通过对输出进行投票表决来选择正确的输出。TMR 技术实际上隐藏了单个模块中发生的故障，而只关心最终输出的正确。DMR 技术运行两个相同的硬件模块并比较其结果，如果结果一致则进行输出，否则报告故障并进行故障处理。

信息冗余是指通过为数据提供额外的信息实现故障检测及恢复。如图1.8所示 [8]，信息冗余需要在数据被用于运算之前预先对其进行编码，通常是检错码或纠错码，然后在运算后基于编码提供的信息进行校验，以确定运算过程中是否出

错，如果出错再根据编码类型决定是否进行恢复。信息冗余的检错、纠错效果受其编码方式、冗余位数的影响。奇偶校验编码和算术码是两种典型的编码方法。奇偶校验方法在数据中增加一个校验位，校验位的取值应使数据二进制码中含有奇数或偶数个 1。前者称为奇校验，后者称为偶校验。通过校验数据中 1 的个数是奇数还是偶数，可以达到检测故障的目的。奇偶校验可以检测奇数位故障，但是无法检测出偶数位故障，也不能进行故障恢复。算术码是专用于校验算术运算的编码方法，广泛应用于算术逻辑部件的校验。用 D 表示数据的编码，则对于算术运算 op 及其源操作数 $r1$ 和 $r2$，需满足编码规则：

$$D(r1\ op\ r2) = D(r1)\ op\ D(r2) \tag{1.4}$$

常见的信息冗余编码方法还有 n 中取 m 码 [34]、校验和 [35]、汉明码 [36]、伯格码 [37]、循环码 [38] 等。

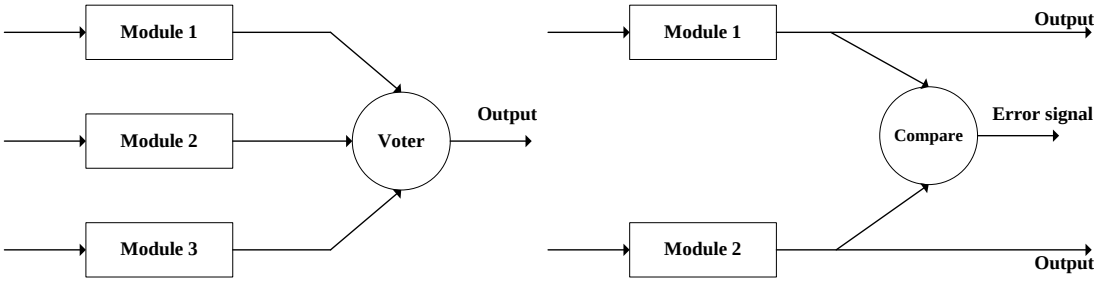


图 1.7　三模冗余和双模冗余

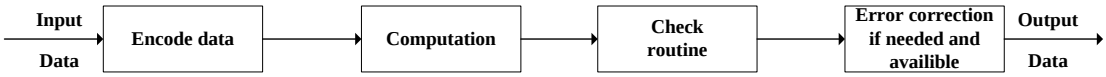


图 1.8　信息冗余

时间冗余是指通过在同一个计算机系统上多次执行程序并进行结果比对或投票实现故障检测、恢复。按照重复计算是在指令级还是程序段级，可以分为指令复执和程序卷回 [30]。由于瞬时故障具有持续时间短、相同故障重复发生概率小的特点，通过两次执行程序并比较结果可以实现故障检测。如果需要恢复故障，则可以执行程序三次，并对结果进行投票表决。图1.9给出了利用时间冗余检测故障的过程 [8]。时间冗余可以利用冗余的执行时间来减少冗余的硬件，因此时间冗余需要的成本要低于硬件冗余和信息冗余。但是时间冗余会导致较高的性能开销，因此只适用于时间资源比较充裕的应用。

软件冗余是指通过在软件中引入额外代码实现容错功能，典型的例子是不变量检测 [39] 和软件复算 [31]。不变量检测可以在程序执行前预先收集一些不变量信息，例如，特定数据的范围、控制流跳转关系等，然后在程序中插装检测代

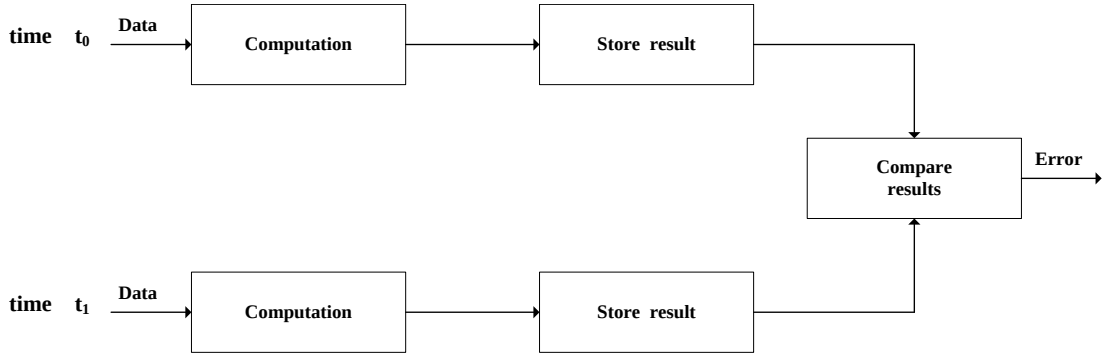


图 1.9　时间冗余

码，用以判断程序执行过程中是否保持了这些不变量特征。如果没有则说明存在故障。软件复算则通过复制程序代码并插装比较操作实现容错功能。尽管需要额外的存储开销，软件复算在本质上也属于时间冗余的方法。

过去的几十年里，容错研究者利用上述一种或几种思路，已经针对瞬时故障提出了很多容错技术。这些技术按照保护的对象可以分为存储系统容错技术和运算单元容错技术。前者的研究较早，而且已经有较为成熟的技术，例如 ECC 技术 [40, 41]、EDAC 技术 [42, 43] 等。后者的研究由于相关部件 SER 的持续上升，目前也已经引起重视。按照实现的抽象层次，现有的容错技术又可以分为硬件容错、软件容错和混合软硬件容错技术。需要特别指出的是，本文所述的软件容错专指面向硬件故障的软件容错技术。

1.3　相关研究现状

容错研究大致可分为故障的可靠性影响分析和容错技术设计两个方面。本节先介绍瞬时故障的影响评估技术，然后将现有的容错技术分为硬件容错、软件容错、混合软硬件容错三类分别进行介绍，最后对研究现状进行总结评述。

1.3.1　瞬时故障的影响评估

传统的可靠性分析关注的是软件故障或者系统永久性故障。由于瞬时故障和这些故障具有完全不同的特征，所以需要针对瞬时故障研究专门的可靠性分析方法。目前主要有三类针对瞬时故障的可靠性分析方法：空间环境或地面实验室中的辐射实验、故障分析法、故障注入法。其中，辐射实验方法可以分析系统在真实环境中的可靠性，但是成本高昂，并且很容易损伤硬件。

故障分析法 [44–48] 主要通过程序分析或概率计算分析系统的可靠性，AVF（Architectural Vulnerability Factor）分析 [44, 49] 是此类方法的典型代表。AVF 分析法在每个时钟周期对系统中所有的逻辑位进行分析，如果某个位发生翻转后会影响系统运行则将其标识为 ACE （Architectural Correct Execution）位，否则该

位在对应周期将被标识为 unACE 位。系统（或某个组件）的 AVF 可以基于公式 (1.5) 计算：

$$AVF = \frac{\sum_{cycle_i} ACE\ bits}{N \times total\ cycles} \tag{1.5}$$

其中，分子表示系统运行期间 ACE 位数的总和，N 表示系统中总的逻辑位数。AVF 值可以反映系统的脆弱性。假设系统的原始软错误率为 λ_s，则系统的失效率（Failure Rate）应为 $\lambda_s \times AVF$。系统可靠性可以用平均无故障时间（Mean Time To Failure，MTTF）评估，而 MTTF 是系统失效率的倒数，即：

$$MTTF = \frac{1}{\lambda_s \times AVF} \tag{1.6}$$

AVF 分析法先通过分析系统中存在的故障屏蔽因素（例如动态死代码 [50]）找出 unACE 位，然后将剩余位均视为 ACE 位。由于所采取的分析策略只能找出一部分 unACE 位，所以这种方法获得的只是系统可靠性的下界。

另外一种有代表性的故障分析法是徐建军等人提出的 PRASE 方法 [45]。PRASE 方法基于概率理论构建了故障产生模型和故障传播模型，对故障在程序中的分布规律及传播规律进行了定量分析，并最终得出系统的可靠性。PRASE 方法的优势在于其不再局限于在硬件层分析故障的影响，而是综合考虑了程序结构及软件行为带来的影响，因此比 AVF 分析法有更好的精度。但是，这种基于概率的分析方法在计算过程中往往需要做出不少假设，而且分析过程中缺乏程序的动态信息，因此其分析精度仍然不够。

故障注入技术采取模拟的方法人为地在系统中注入故障，然后根据系统运行结果分析故障对可靠性的影响。这种方法具有成本低、注入可控性好等优势，而且可以获得较为精确的分析结果，因此在学术界和工业界得到广泛应用 [51–53]。但是，由于故障在系统运行时发生的状态空间非常庞大，故障注入技术面临如何权衡模拟速度与分析精度之间关系的挑战 [54]。文献 [55] 采取故障注入的方法分析程序中各个变量的脆弱性，并基于分析结果选取部分最为脆弱的变量进行保护。为了获得有统计意义的分析结果，该技术需要大量的计算资源进行故障注入模拟，付出的代价对于大规模应用很难承受。

实际经验表明，注入程序的大量故障可能在电路级、微体系结构级、体系结构级或者应用级被屏蔽，不会影响系统的运行结果 [56, 57]。这些良性故障在进行容错系统测评时可以认为是无效故障，其大量存在不利于对容错系统进行高效评估。CriticalFault[58] 技术利用 AVF 分析从故障空间中删除这些故障，从而达到压缩故障空间，注入更多有效故障的目的。CriticalFault 的体系结构如图1.10 所示

[58]。其不足之处在于，仅靠删除不影响结果的故障所能获得的模拟加速是有限的，剩余的故障仍会导致大量的模拟开销。

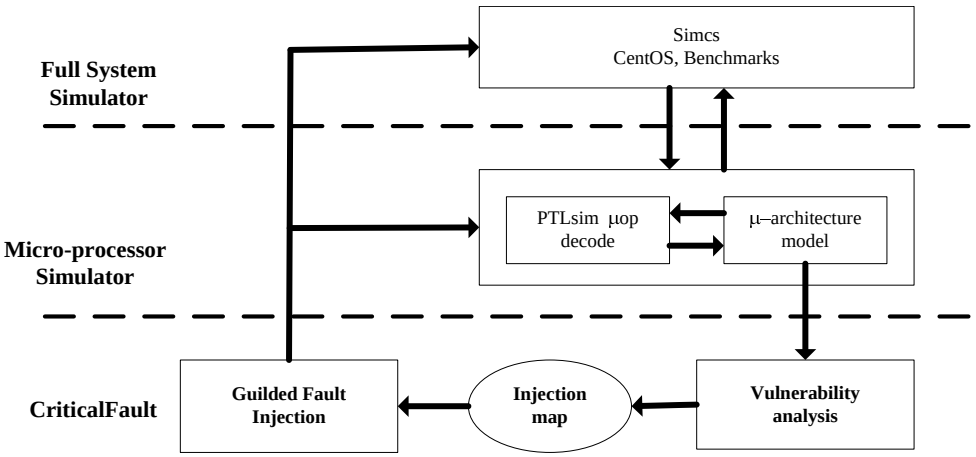


图 1.10　CriticalFault 体系结构图

Relyzer[33] 技术采取故障删除方法压缩故障空间，以达到减少模拟开销的目的。Relyzer 主要依赖提出的控制流等价策略进行故障删除，这种策略利用故障发生位置的控制流上下文环境的相似性预测故障结果的相似性，进而可以从故障空间中删除结果可能相同的故障。评估结果表明，通过这种策略可以大量删除需要注入的故障。但是，Relyzer 的控制流等价策略只考虑了指令的控制流上下文环境，忽略了数据流关系带来的影响，并且 Relyzer 只开发了同一个静态指令的不同动态实例间的等价性，没有考虑不同静态指令间存在的等价性，因此这种技术仍有不少优化的空间。

故障注入既可以在软件模拟器上进行，也可以在具体的硬件平台下完成。在实际的硬件平台下注入故障获得的结果更接近真实的应用，但是成本较高且故障模拟的可控性、灵活性不如软件模拟器。在软件模拟器上进行故障注入的缺陷在于其分析精度与软件模拟器相关，并且模拟速度不如硬件平台。

1.3.2　硬件容错技术

瞬时故障本质上属于一种硬件故障，所以相关的容错技术最初都是在硬件层面实现的。下面将已有的硬件容错技术大致分为运算单元容错技术和存储系统容错技术两类，并分别对其概述。

1.3.2.1　运算单元容错技术

传统的硬件容错技术通常以复制部分或全部处理器模块的方式实现容错。例如，在 HP NonStop 系统 [59] 中，两个相同的处理器共用同一个时钟逻辑，并在每

一个操作完成后比较结果，如果不一致则报告错误。类似的系统还有 IBM S/360 系列服务器 [60]、土星五号（Saturn V）导航计算机系统和 Boeing 777 机载计算机系统 [61]。复制处理器技术可以满足在恶劣环境下工作的系统的可靠性需求，但是这些技术的成本高昂，很难为普通用户所接受。

现代处理器提供的并发多线程机制为容错设计提供了天然的冗余资源。AR-SMT[62] 是首个利用并发多线程机制进行容错的技术。该技术利用主线程和一个副线程执行相同的程序，并通过在预定的检查点比较主线程和副线程的结果实现故障检测。SRT 技术 [63] 在 AR-SMT 的基础上，采用各种优化设计最大限度地降低了冗余线程执行对单核处理器资源的竞争，提高了并发多线程处理器的性能。SRT 提出了冗余保护域（Sphere of Replication，SoR）的概念，用以表示需要冗余执行保护的系统区域。如图 1.11 所示 [63]，SoR 区域的所有输入数据都需要复制，以使得主线程和副线程能够基于相同的输入进行冗余执行。另外，SoR 区域输出的所有结果都需要经过比较确认才能提交。SRTR[64] 在 SRT 的基础上，进一步提供了错误恢复机制。同时，SRTR 在 SMT 原有的硬件结构上添加了分支结果队列（Branch Outcome Queue，BOQ）、寄存器值队列（Register Value Queue，RVQ）和读存储器值队列（Load Value Queue，LVQ）等数据结构，用以在主线程和副线程间共享相关信息，达到减少分支预测失败和访存延迟开销的目的。总的来说，基于并发多线程的技术都存在硬件资源竞争的问题，如何减小资源冲突、降低性能损失是基于并发多线程的容错技术必须解决的问题。

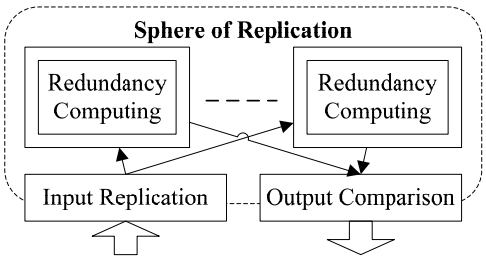


图 1.11　SoR 示意图

为了降低硬件容错的成本，一些技术尝试利用专门的检测单元实现故障检测和恢复，并取得了良好的效果。例如，DIVA[65–67] 技术在一个高性能、超标量乱序流水线的基础上，通过在提交阶段添加一个简单核实现了故障检测，其体系结构如图1.12所示 [65]。指令在乱序流水线中完成执行后将其输入和输出一并发送给简单核。简单核重新执行指令操作，并比较简单核的计算结果和指令在乱序流水线中的计算结果以确定是否存在故障。由于新引入的简单核执行的功能单一，DIVA 可以快速地完成指令结果验证，所以相比复制整个处理器的方法，DIVA 的硬件成本明显降低，且仍然保持了较高的性能。但是，DIVA 只进行了冗

余的指令执行，因此无法检测指令在流水线中取指、提交阶段发生的故障，可靠性有所下降。而且 DIVA 的硬件成本仍然无法满足主流消费市场用户的需求。

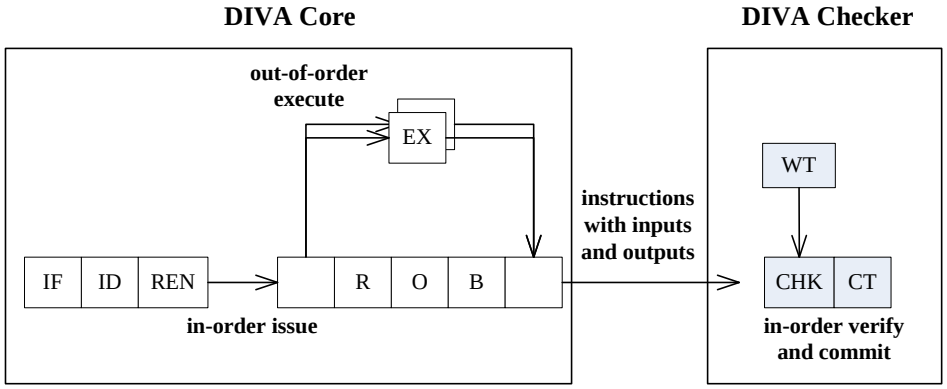


图 1.12　DIVA 体系结构图

Bulletproof[68] 针对乱序超标量流水线，通过为功能部件设计在线自检电路实现故障检测。这些自检电路利用预存的输入向量周期性地对功能单元进行测试。一旦检测到故障，系统可以基于 checkpoint 恢复机制回滚到上一个正确状态，然后利用超标量处理器中丰富的功能单元，重配置后在低性能模式下继续运行。利用在线自检逻辑可以高效地检测硬件永久性故障。但是，由于瞬时故障的瞬时性和重写可恢复性，这种基于自检逻辑的技术显然无法有效地检测系统中的瞬时故障。

文献 [69] 基于超标量流水线支持多指令执行的特点，在现有流水线处理器基础上，稍加改进实现了基于指令复算的容错方法。其方法是在取指令阶段复制源指令并将其副本指令紧随其后放入指令队列，副本指令可以像源指令一样进行分派、流出和执行。在指令提交阶段，通过比较源指令和副本指令的结果实现故障检测。该技术充分利用超标量流水线中丰富的计算资源，以较小的硬件开销提供了接近 100% 的故障覆盖率。但是，复制程序中所有指令带来了较高的性能和功耗开销。评估结果表明，即便利用了超标量处理器的并行计算能力，性能开销仍然高达 50% 以上。

1.3.2.2　存储系统容错技术

现有技术对存储结构一般采用信息冗余的方法进行保护，典型的技术是 ECC[40, 41]、奇偶校验技术等。奇偶校验技术需要的成本很低，但是只能检测奇数位故障，且无法恢复故障。ECC 技术可以有效地保护存储系统，并且已经在许多系统中推广应用。但是对于寄存器、一级缓存（L1 Cache）和便笺存储器（Scratchpad Memory，SPM）[70] 等离处理器核较近的片上 SRAM 结构，由于对这些存储结构访问比较频繁，如果使用 ECC 保护将会带来较大的性能和功耗开

销，所以并不适合采用 ECC 技术保护。即便性能开销可以采用一些并行设计进行优化，但是大量的功耗开销却无法避免。目前，功耗已经不再仅仅是高端系统（例如航天计算机）关注的问题，同样也是大众消费电子领域极为重视的问题。因此，针对寄存器、SPM 等存储结构，研究可以替代 ECC 的低代价保护技术具有重要的价值。下面将对这方面已有的研究成果进行概括说明。

ECC 技术一般需要在存储单元每次读写时进行编码或校验，由此引入了大量性能和功耗开销。为了减少 ECC 保护的开销，Shield[71] 针对流水线寄存器文件，开发了一种可动态选择部分寄存器值进行保护的技术，其体系结构如图1.13所示[71]。为了获得更高的可靠性收益，Shield 动态判断寄存器值的生存期，并优先保护生存期长的寄存器值。对于要保护的寄存器值，Shield 在写操作之前为其计算出校验值并存入 ECC 表，当需要读取该值时，Shield 再重新为其计算校验值并和 ECC 表中的校验值进行比较，以确定是否出现故障。Shield 仅从活跃的寄存器值中选取一部分进行保护，相比采取 ECC 进行完全保护的方法，明显减少了性能和功耗开销。

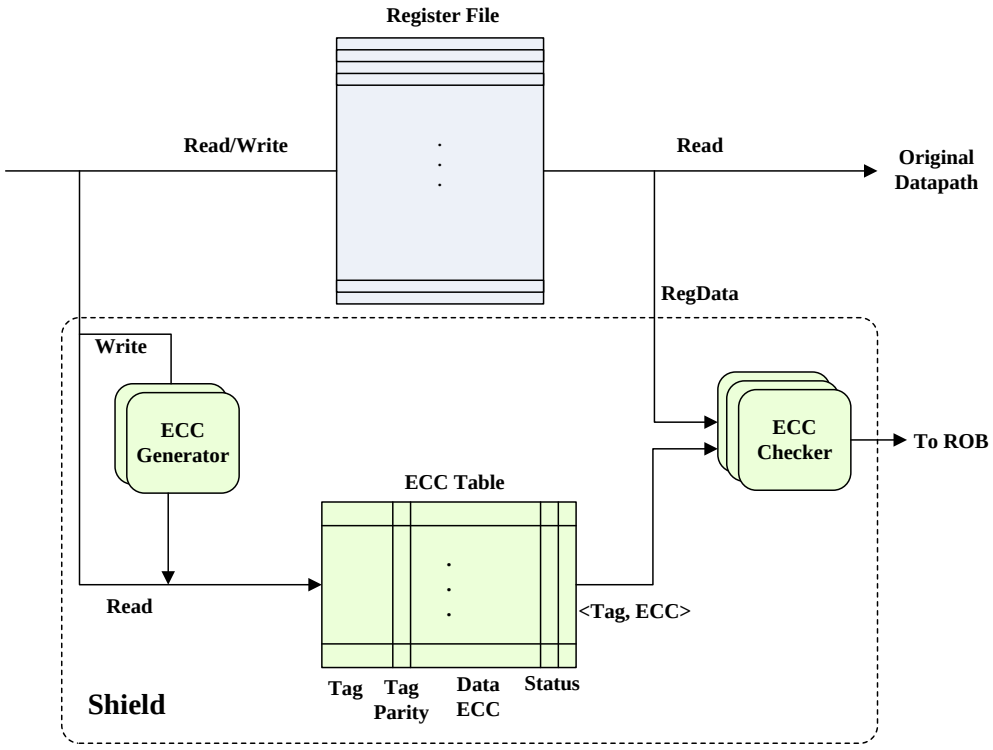


图 1.13　Shield 体系结构图

文献 [72] 针对访问频繁的 L1 Cache，在处理器中设置一个与之位置平行的有保护的 Cache，并将比较脆弱的程序代码和数据优先放入受保护的 Cache 中。由于程序中不同的代码和数据对瞬时故障的敏感性并不相同，该技术通过将最脆弱的代码和数据放入一小块受保护的 Cache 中，即可达到以较小的开销保护 Cache

结构的目的。该技术的不足在于没有提供精准的脆弱性分析方法，也没有研究如何提高受保护 Cache 的利用率。

SPM[70] 是一种和 Cache 在功能上类似、位置上平行的存储结构。但是 SPM 由软件管理分配，电路较 Cache 简单，因此其功耗、占用面积比 Cache 小。此外，基于 SPM 的系统的性能较基于 Cache 的系统更容易预测，因此许多嵌入式系统（尤其是实时系统）开始引入 SPM[70]。目前针对 SPM 进行保护的技术还极少。MM-SPM 技术 [73] 针对指令 SPM 提出了一种保护技术。由于指令 SPM 的数据在程序运行过程中不会改变，因此 MM-SPM 可以利用其在主存中的数据副本进行比较检测和恢复。MM-SPM 的体系结构如图1.14所示 [73]。MM-SPM 只是在系统已有机制的基础上稍加改进实现容错，因此这种技术不需要很高的成本。但是这种技术只解决了指令 SPM 的可靠性问题，而无法应用于数据 SPM 的保护。

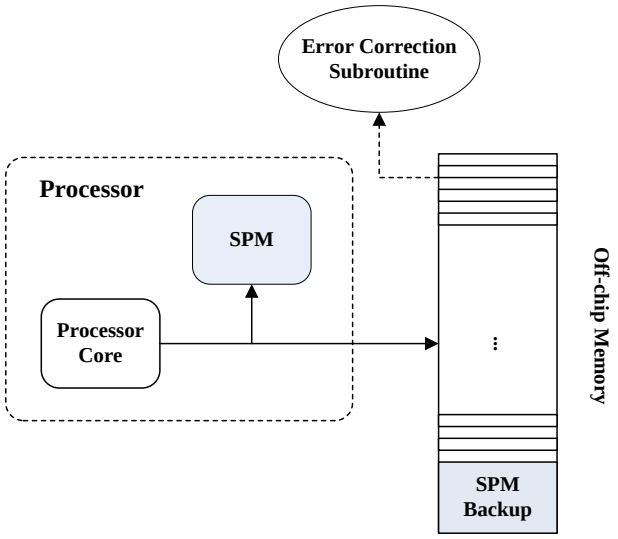


图 1.14　MM-SPM 体系结构图

1.3.3　软件容错技术

根据计算机系统结构的设计原则，很多硬件实现的功能也可以用软件方法实现。同样地，一些硬件容错技术也可以在软件层面实现。除了可以降低成本，软件容错的意义还在于其灵活性。当系统部署的环境发生变化时，如果需要更高的可靠性，只能采取软件方法对其进行可靠性增强。从程序行为的角度来看，计算机中发生的瞬时故障通常导致运行的程序发生数据流错误或控制流错误。现有的软件容错技术对这两类错误通常采取不同的思路进行检测，下面分别对其研究现状进行总结。

1.3.3.1 数据流检测技术

现有的软件容错技术对数据流错误通常采取软件复算的方法进行处理。文献[31]介绍的EDDI算法是采取软件复算的典型算法。EDDI算法在编译时对程序指令进行复制，被复制的原指令称为主指令（Master Instruction，MI），复制得到的指令则称为影子指令（Shadow Instruction，SI）。EDDI算法还在程序中预定的位置（存储和分支指令前）插装指令以比较主指令和影子指令的执行结果，如果不匹配则认为程序执行出现错误。源程序的一条加法指令经过EDDI算法处理后的结果如图1.15所示。EDDI算法可以有效地检测数据流错误，而且算法在不破坏指令的数据依赖关系的前提下，通过指令调度实现了对控制流错误的检测。但是，算法使程序中的指令增加了100%以上，由此带来大量的性能开销，对变量的复制则带来了不小的空间开销。

```
add r3, r1, r2            //主指令：r3=r1 + r2
add r32, r12, r22         //影子指令
bne r3, r32, gotoError    //比较指令
```

图 1.15　EDDI 算法示例

文献[74]提出的ED^4I算法在软件复算过程中引入了差异性变换。ED^4I程序执行时，也有源程序和副本程序两个版本，但是源程序和副本程序基于不同的输入数据，后者的输入数据是源程序输入数据的k倍（k为差异性因子）。算法在程序执行结束时比较这两个版本的程序输出结果，如果输出结果之间仍然保持k倍关系，则认为程序执行正确，否则说明系统存在故障。图1.16是ED^4I算法的示例[75]，其中差异因子k的取值为-2。通过差异性变换，ED^4I既可以检测瞬时故障，也可检测出永久性故障。差异因子的选择对ED^4I算法的检测能力有直接影响。ED^4I算法从数据完整性和故障覆盖率两项指标出发，采取概率计算的方法确定最佳的差异因子。ED^4I的缺点在于将数据进行差异性变换后可能会出现数据溢出问题。另外，这种差异性变换方法无法应用于很多逻辑操作。

其他的软件复算技术也都遵循冗余执行后比较的检错思路，不同技术的主要差别在于冗余保护的范围、设置检查点的方法、实现的层次等[7]。例如SWIFT[32]算法只复制寄存器操作数，而不复制变量数据。故障检测算法插装的比较检测指令是制约流水线性能的瓶颈，EsoftCheck[76]因此专门提出了软件复算的检查点优化技术。EsoftCheck通过安全删除部分检查点，可以在不损失故障检测能力的条件下，优化复算技术的性能。文献[77]提出的方法在源代码级实现容错，且只对保存最终结果的变量值进行比较检错操作，而忽略了中间变量的检查。这样做可以提高算法的性能，但是会加大检错延迟。ThOR[78]技术提供了一

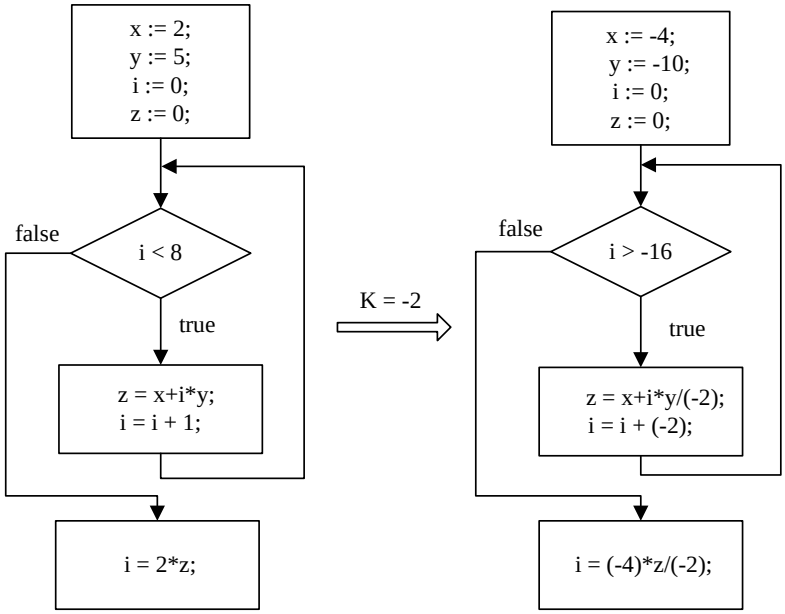


图 1.16　ED^4I 程序转换示例

个自动实现软件复算的源到源转化工具。ThOR 的优点是独立于具体应用平台和编译器，其缺点是在源代码级实现容错产生的冗余代码太多，严重增加了程序的性能开销。文献 [79, 80] 提出的 AN code 技术与 ED^4I 类似，但 AN code 技术需要对数据进行除法变换，其性能开销显然高于 ED^4I。文献 [81] 采取了将所有变量和常量值都取反的差异性变换方法，但是 ED^4I 已经证明了 -1 在很多情况下并不是最佳的差异因子。

多核平台的迅速普及为软件复算提供了新的计算资源。目前已经出现了一些多核平台下基于软件实现的冗余容错技术 [82–84]，其中具有代表性的成果是 SRMT 技术 [83] 和 DAFT 技术 [84]。SRMT 在编译阶段为程序生成一个主线程和一个副线程。主线程和副线程执行相同的操作，并在设定的检查点位置通过比较两个线程的执行结果检测故障。评估结果表明，SRMT 在可靠性方面已经达到硬件冗余技术的水平。但是，为了检测故障，SRMT 需要将主线程的结果先发送给副线程，在副线程上通过比较确认结果正确后主线程才能继续执行。SRMT 的线程间通信过程如图1.17所示 [83]。这种线程间的相互等待将通信开销和比较检测开销引入到了决定性能的关键路径中，带来高昂的性能代价。DAFT 通过提出的错误前瞻方法消除了线程间的同步检查操作，使得主线程可以顺畅运行，从而大幅降低了性能开销。针对 SPEC CPU2000 和 SPEC CPU2006 进行的实验评估结果表明，相比 SRMT 技术，DAFT 在保持故障覆盖率没有损失的同时（高达 99.93%），将性能开销从 200% 降低到了 38%。DAFT 的效果证明，在当前多核技术快速推广的技术潮流下，开发基于多核的容错技术具有很好的前景。

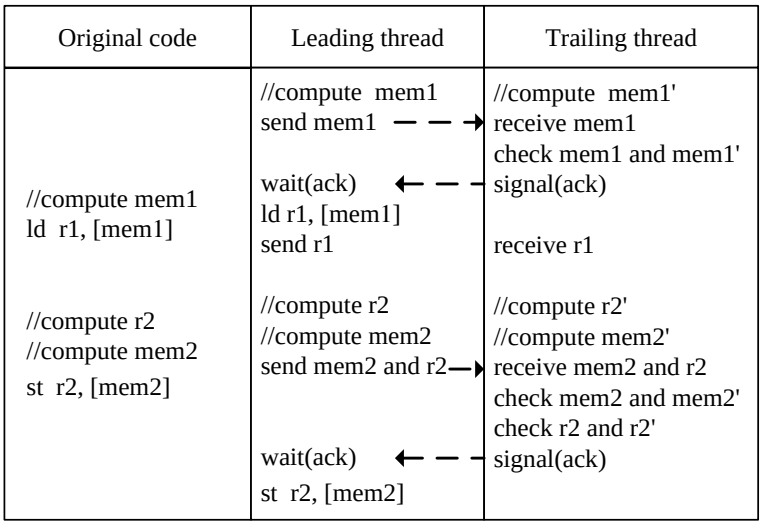


图 1.17　SRMT 技术的线程间通信示例

1.3.3.2　控制流检测技术

现有的针对控制流错误的软件检测技术大多是采用标签分析的方法。标签分析法的基本原理是：在编译时先将程序划分为基本块，并为每个基本块分配一个唯一的静态标签，然后在每个基本块中插装标签的更新和比较指令。程序运行时插装的指令可以根据当前控制流为每个基本块计算出一个动态标签，并通过比较动态标签和当前块的静态标签是否一致来检测故障，不匹配则说明控制流出现了错误 [75]。

CFCSS 算法 [85] 是采用标签分析方法的典型算法。CFCSS 在程序编译时为每个基本块分配一个静态标签 S，并计算出每个基本块和其前驱基本块的静态标签的异或运算结果 d，然后算法在每个基本块入口处插入如下两条指令：

$$\begin{aligned} &I1: G = G \oplus d \\ &I2: br\ G \neq S\ \ error \end{aligned} \tag{1.7}$$

其中，通用寄存器 G 专门用来保存每个基本块的动态标签。指令 I1 更新当前基本块的动态标签，指令 I2 比较动态标签和当前块的静态标签 S，若二者匹配则说明程序控制流正确，否则认为控制流出现错误。图1.18给出了一个 CFCSS 的例子 [75]。从图中可以看出，当程序沿合法分支从 B1 跳转到 B2 入口后，动态标签 G 的值等于 B2 的静态标签 S_2。而当程序沿非法分支从 B1 跳转到 B4 入口后，动态标签和 B4 的静态标签 S_4 不一致，据此可以判定控制流出现了错误。CFCSS 算法是已有算法中时空开销较低的算法，其缺点是存在不少检测盲点，如控制流从一个基本块内部到另一个基本块头部的跳转错误、基本块内一条指令到另一条指令的跳转错误等。

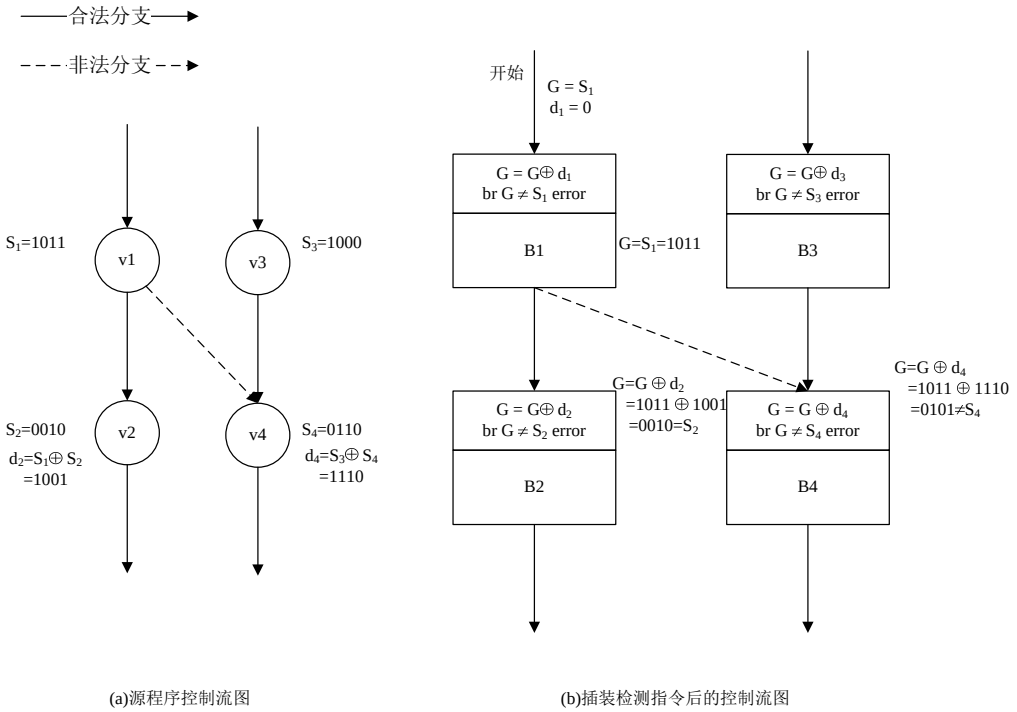


图 1.18　CFCSS 算法示例

RSCFC 算法 [86] 将程序中每个基本块对应于其静态标签 L 的二进制编码中的一位，例如，第 i 个基本块的静态标签值为 2^{i-1}。为了保存基本块之间合法的控制流关系，RSCFC 算法为每个基本块分配一个路由标签 s，s 的值为对应基本块的所有后继基本块的静态标签的按位或运算结果。RSCFC 算法通过在每个基本块入口和出口处分别插入如下断言实现动态标签的更新和比较:

$$\begin{aligned} &test: if((S = S\&L) \equiv 0)\ \ error \\ &set: S = s\&(-!(S \oplus L)) \end{aligned} \tag{1.8}$$

其中，S 是专门保存动态标签的通用寄存器。执行 set 断言前 S 的初值为 L，执行 set 断言后 $S = s\&(-!(S \oplus L)) = s\&(-1) = s$。在下一个基本块入口处，如果控制流跳转正确，则 $S\&L$ 的运算结果应该等于 L，否则 $S\&L$ 的运算结果为 0，可以利用 test 断言检测出错误。RSCFC 算法在故障覆盖率方面比 CFCSS 方法有了较大提高，但是算法也插装了更多冗余代码，时空开销明显增加。更重要的是，RSCFC 的标签设计方式所能表示的基本块的数量受限于机器字长（32 位机器字长能表示的基本块数只有 31 个），尽管 RSCFC 提出了分层嵌套的方法克服这个缺陷，但是会导致更多的性能开销。

其它控制流检测算法 [7, 87–90] 和上述算法的设计过程类似，相互之间的区别主要体现在标签的设计和插装的代码上，算法的开销和检测能力也和这些因素

有关。例如，ECCA 算法 [87] 在每个基本块入口插入 test 断言以进行标签比较，在基本块出口处插入 set 断言将标签更新为下一个块的标签。ECCA 的断言中使用了开销较高的乘法和除法操作，因此造成了较高的性能损耗。文献 [88] 提出的 DSM 算法插装了大量检测指令来克服现有算法存在的检测漏洞，但是代价非常高昂，造成程序性能下降了 3 倍，存储消耗增加了 4 倍。ECCFS 算法 [7] 通过设计格式化标签和插装一些简单的逻辑操作，实现了基本块内、基本块间和过程间控制流错误的检测，算法在容错效率方面有明显提高。但是，ECCFS 的分析结果显示，该算法仍存在部分检测漏洞。

在已有的控制流检测算法中，CEDA 算法 [91] 被认为具有最高的容错效率。CEDA 为每个基本块分配一个入口标签和一个出口标签，并在基本块中插装两条指令，分别用以将动态标签更新为所在基本块的入口或出口标签。通过插装额外的标签比较指令，可以随时检测控制流错误。CEDA 算法可以以较低的性能开销检测全部基本块间的控制流错误。但是，CEDA 算法没有解决基本块内和过程间的控制流错误检测问题，此外，和其它已有控制流检测算法一样，CEDA 也不具备可配置性和对容错机制的自我保护能力。

1.3.4 混合软硬件容错技术

在已有技术中，硬件容错技术的时空开销低，但是硬件成本高。软件容错技术不存在硬件开销，但是往往带来大量的时空开销。为了以较低的代价实现容错保护，结合硬件和软件技术的优势开发混合容错技术成为一种合理的选择。目前已经提出的混合软硬件容错技术还不多，下面列举一些有代表性的方法。

Argus[92] 在编译时为程序中每个基本块分配含有数据流和控制流信息的标签，并利用专门的指令将标签嵌入基本块中。在硬件层面，Argus 针对顺序流水线设计了专门的检测单元，可以基于程序标签进行程序数据流和控制流的正确性检查。此外，为了验证各个功能单元的正确性，Argus 还为每个功能单元设计了基于编码技术的检测逻辑。例如，Argus 采取求模编码法对乘法单元进行校验。假设乘法的输入为 A 和 B，用以校验的模为 M，则可以通过验证 $((A\%M)\times(B\%M))\%M$ 是否等于 $(A\times B)\%M$ 实现故障检测。一系列体系结构级的容错设计使得 Argus 可以有效地检测故障。但是，Argus 主要依靠硬件技术进行故障检测，并且是在较为简单的顺序流水线中实现，其硬件成本仍然相对较高。

Restore[93] 首次提出了基于症状（Symptom）[28, 94, 95] 的容错技术。该技术将系统在正常情况下很少发生的反常行为（例如非法指令码、访存越界等）看做是由瞬时故障引起的，然后通过改进系统已有的检测机制和恢复机制实现容错，其体系结构如图1.19所示 [93]。Restore 主要依靠系统已有的异常检测机制实现症

状型故障的检测，且可以以极低的代价提供较为可观的可靠性。但是，Restore 不能检测 SDC 故障，所以其所能提供的可靠性是有限的。Restore 技术的局限性还在于，当其开发利用其它症状（例如 Cache 访问未命中和分支预测失败）检测故障时，所能获得的好处有限。原因是这些症状在无故障运行时也常常发生，利用其检测故障会频繁的发生误报系统存在故障的情况（此种情形称为“假死”（false positive)），从而引入不必要的恢复开销。

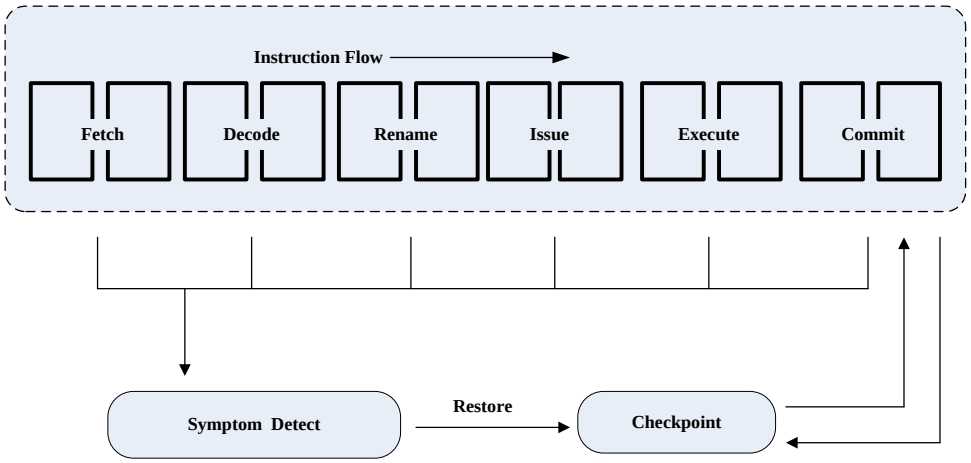


图 1.19　Restore 体系结构图

Shoestring[27] 在 Restore 技术的基础上，着力提高 SDC 故障的检测率。为此，Shoestring 提出了程序分析方法，用以识别在发生故障时有可能导致 SDC 的代码，并采用软件复算方法保护这些代码。Shoestring 以较少的性能开销提高了 Restore 的可靠性。但是 Shoestring 的程序分析方法是基于经验基础上的原则提出的，分析精度仍有很大的提升空间，而且这种技术无法满足用户对高可靠性的需求。

iSWAT[94] 技术结合了症状检测 [28] 和不变量检测技术，以实现低代价的容错保护。iSWAT 通过对硬件平台中已有的症状检测（例如异常检测）和 checkpoint 恢复机制进行简单的改造，实现对症状型故障的处理。对于 SDC 故障，iSWAT 则采取软件不变量技术进行检测。尽管 iSWAT 需要的开销很低，这种技术的检测能力有限且受具体应用影响。

文献 [96] 基于文献 [69] 提出的在超标量流水线中复制指令的方法，设计了一条专门的标记指令，用以分割需要复算保护的代码和不需要复算的代码。并改进流水线设计，使得流水线遇到这条指令时，可以在复制模式和正常模式之间切换。为了以最小的代价获得最高的可靠性，文献 [96] 也提出了基于程序分析的关键代码选择方法，用以挑选对程序可靠性影响最大的代码段进行保护。结合硬件的改进和软件的分析，该方法需要的性能开销相比完全保护整个程序明显降低，并且保持了较高的故障覆盖率。但是，该方法对症状型故障和良性故障也不加区

分地采取复制指令的方法进行检测，造成了性能浪费，因此该方法仍有不小的优化空间。

存储系统保护方面同样已经出现一些混合软硬件的容错技术。文献 [97] 对部分寄存器进行 ECC 保护，然后基于编译分析确定汇编程序中寄存器的脆弱性，并选择较为脆弱的寄存器和 ECC 保护的寄存器进行交换，从而达到以少量的硬件成本获得最大可靠性的目的。同样基于部分 ECC 保护的寄存器，文献 [7] 则在寄存器分配前对变量值进行脆弱性分析，然后改进寄存器图着色算法，将脆弱的值优先分配到有 ECC 保护的寄存器，进一步提高了 ECC 保护的寄存器的利用效率。

1.3.5 研究现状总结

在可靠性分析方面，目前主要有辐射实验、故障分析法和故障注入法三种方法。其中，辐射实验可以分析系统在真实环境中的可靠性，但是成本高昂，并且很容易损伤硬件。故障分析法由于在分析过程中缺乏程序的动态信息，往往会采取较为保守的分析策略或做出不少计算假设，导致其分析精度不足，通常只能获取可靠性的边界值。相比前两种方法，故障注入法具有成本低、注入可控性好等优势，而且可以获得较为精确的分析结果。但是，已有的故障注入技术存在无法兼顾模拟速度和分析精度的问题。

处理器中运算单元的 SER 正在快速上升，对瞬时故障的敏感度很快就会变得和存储单元一样高 [7, 8]。现有的运算单元保护技术尽管可以提供令人满意的可靠性，但是这些技术需要的代价显然是主流市场无法接受的。另一方面，传统的 ECC、奇偶校验等技术并不适合用来保护片上 SRAM 结构，目前对片上 SRAM 结构的容错技术研究却还存在明显不足。

此外，从容错实现的角度总结，硬件技术通常需要较高的硬件成本，但是需要的时空开销低。软件技术不需要硬件开销且便于分析优化，但是常常带来大量的时空开销。混合软硬件容错技术则结合了二者的优势，可以达到以较小的代价提高系统可靠性的目的。但是，混合容错技术的研究还很少，有待进一步深化。

1.4 研究内容和成果

本文研究的目标是面向不同领域用户的不同需求，提供成本、可靠性满足约束的可靠性解决方案。为此，本文首先将处理器运算单元中的故障所导致的程序错误分为数据流错误和控制流错误，并分别提出了高效、可配置的容错保护技术。然后，本文针对处理器存储单元中的故障提出了低代价的容错保护技术。最后，本文研究了基于故障注入的可靠性分析技术，用以分析瞬时故障的影响和评估提出的容错技术。具体来说，本文的研究工作和成果包括：

1．提出了一种可配置的数据流检测技术

处理器中发生的瞬时故障可能导致程序运行时发生数据流错误，这些数据流错误很容易引发程序发生危险的 SDC 错误，因此是容错技术需要优先解决的问题。然而，如何以较低的开销实现有效的数据流保护是困扰容错研究者至今的难点问题。为了应对这种挑战，本文结合软、硬件容错技术的优势，提出了一种高效的数据流检测技术 Epipe。基于改造现有的超标量流水线处理器，Epipe 提供了一个能够对指令进行冗余保护的硬件平台。由于超标量处理器中有丰富的计算部件，Epipe 平台只需要很少的硬件开销，即可实现指令的冗余保护。为了减少冗余保护产生的性能开销，Epipe 还基于程序分析方法评估每个指令的重要性，即指令发生故障后导致 SDC 的概率。程序运行时，Epipe 可以根据用户的性能和可靠性要求选择保护最重要的一部分指令。Epipe 的特点在于，该技术只冗余保护对 SDC 故障最敏感的部分指令，对于症状型故障则直接利用系统中已有的异常检测等机制加以处理，而剩余的良性故障则不需要任何处理。这种分类处理故障的方法有效地减少了需要冗余保护的指令，再结合时空开销较低的硬件指令保护技术，使得 Epipe 技术可以更低的开销保护程序数据流。

2．提出了一种可配置的控制流检测技术

处理器中的瞬时故障还有可能导致程序运行时出现控制流错误。由于已有的数据流检测技术很难处理这些控制流错误，容错计算机系统还必须另外提供对控制流错误的检测能力。实现控制流检测的一种有效方法是基于软件实现的标签分析法。已有的标签分析技术除了存在时空开销过大和可靠性不足的问题外，还缺乏可配置性，无法满足不同用户的不同需求。此外，软件检测技术引入的冗余代码自身也有可能发生错误，现有的控制流检测技术在容错机制的自我保护方面缺乏研究。为了克服上述不足，本文提出了一种基于对等标签的控制流检测算法 CFCES。CFCES 可以较少的开销有效地克服已有算法存在的检测盲点，并且首次具备了良好的可配置性和自容错能力。实现检测机制自我保护的难处在于，在对检测机制提供有效保护的同时应当避免额外产生大量的开销。为此，CFCES 在检测机制中“嵌入”了一种定义为“对等性”的不变量，通过对这种不变量进行检测，CFCES 以极低的代价实现了检错机制的自容错保护。此外，CFCES 还提供了可配置的优化方法。该优化方法可以基于函数重要性分析和新的程序块划分方法，灵活配置不同代码段的保护强度，以满足用户不同的时空开销和可靠性约束。该优化方法的特点在于其可以提高 CFCES 的容错效率，且可以用于优化其它基于标签分析的控制流检测算法。

3．提出了一种基于部分保护的片上 SPM 存储容错技术

瞬时故障不仅可能发生在处理器的运算单元中，也有可能出现在处理器的存储单元中。被广泛用于保护片外存储的 ECC 技术并不适合用来保护片上存储结构，原因是这些存储结构本身已经占用了大部分芯片面积，并且访问频繁，采用 ECC 保护会带来大量的面积、性能和功耗开销。由于现有的容错研究中十分缺乏针对片上存储结构的合理保护方案，本文针对一类特殊的片上存储结构 SPM 提出了低代价的容错保护技术 PPS。尽管用 ECC 对 SPM 进行完全保护的开销很高，但是对部分 SPM 进行 ECC 保护并进行合理分配仍是非常有价值的。PPS 技术首先设计了基于部分 ECC 保护 SPM 的存储体系结构（被保护的比例取决于可靠性、性能等需求），然后对程序中的待分配变量进行脆弱性分析，并将 SPM 根据待分配变量的大小划分为“寄存器”，最后采取基于优先级的图着色方法将较为脆弱的变量优先分配到 ECC 保护的“寄存器”中。基于上述方法，PPS 能够以较低的开销获得较高的可靠性。

4．提出了基于程序分析的故障注入技术

故障注入研究面临的挑战是如何权衡模拟速度与分析精度（或真实度）之间的关系。由于故障在系统运行时发生的状态空间非常庞大，模拟注入所有的故障将会使模拟时间变得无法控制。而直接减少注入故障的数量，则可能使分析的精度无法接受。鉴于已有的故障注入技术还不能有效地解决上述问题，本文提出了一种新的故障注入框架 SmartInjector，该框架可以在大幅度降低模拟时间开销的同时保持较高的分析精度。首先，基于发生在相似的数据流或控制流上下文环境中的故障往往会导致系统产生相同反应的认识，SmartInjector 通过程序分析将程序中的部分故障划分为不同的等价类，并从每一个等价类中选取一个代表进行模拟注入，其余的等价类故障则直接从故障空间中删除。SmartInjector 还从故障空间中删除那些仅通过程序分析就可以确定系统反应的故障。基于上述故障删除技术，SmartInjector 可以减少需要注入的故障数量，从而降低故障注入的时间开销。最后，SmartInjector 还首次提出了通过减少单次故障模拟时间降低故障注入时间开销的思路。所采取的方法是基于程序分析预测故障传播可能导致的结果和判定结果的位置，并在故障模拟时基于预测提前判断故障注入的结果。如果故障结果预测正确则可以提前结束单次故障注入，达到减少单次模拟时间的目的。

1.5 论文结构

本文内容共分为六章，论文的组织结构如图1.20所示。每章内容概括如下：

第一章是本文绪论。首先介绍了本文的研究背景，然后概述了面向瞬时故障的容错技术原理，分析了相关技术的研究现状，最后在上述内容的基础上阐明了本文的研究动机，并给出了主要的研究内容和研究成果。

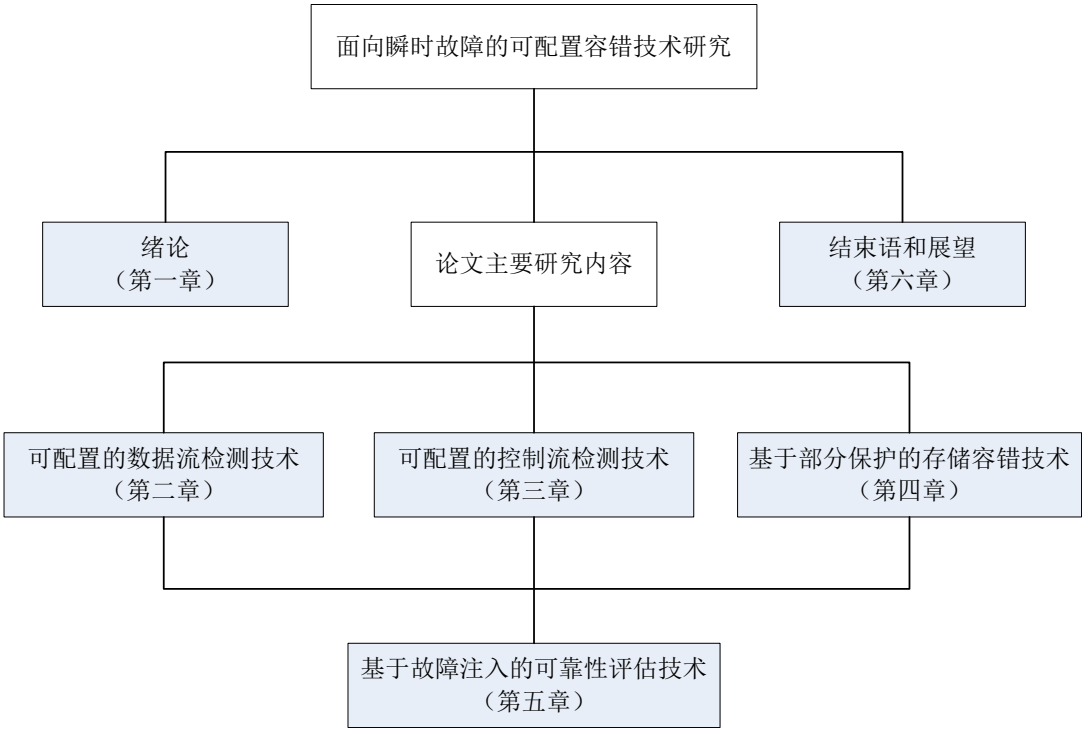


图 1.20　论文组织结构图

第二章针对处理器运算单元中的数据流错误提出了可配置的故障检测技术。该技术首先需要改造现有的超标量流水线处理器，以提供一个能够对指令进行冗余保护的硬件平台，然后基于程序分析方法评估每个指令的重要性，最后根据用户的性能和可靠性要求，配置保护最重要的一部分指令。

第三章针对处理器运算单元中的控制流错误提出了可配置的故障检测技术。该技术首先在编译时根据用户的性能和可靠性需求，划分特定粒度的程序块，然后为每个程序块分配具有对等性的标签，最后在每个程序块中插装控制流检测需要的标签运算和比较指令。

第四章针对处理器存储单元中的故障提出了低代价的容错保护技术，具体针对的是片上 SPM 存储结构。该技术首先设计一种基于部分 ECC 保护 SPM 的存储体系结构，然后对程序中的待分配变量进行脆弱性分析，最后采取基于优先级的图着色方法将较为脆弱的变量优先分配到 ECC 保护的 SPM 中。

第五章提出一种基于故障注入的可靠性评估技术，用于分析瞬时故障的影响和评估容错技术的可靠性。该技术首先基于程序分析从待注入的故障空间中删除等价类故障和结果确定型故障。然后在进行故障注入时，基于提出的故障结果预测技术在程序运行结束前提前判断故障模拟的结果，并在预测成功后提前结束单次故障模拟。

第六章对全文内容进行总结，指出了本文的主要创新点，并对下一步研究工作进行了展望。

第二章　可配置的数据流检测技术

2.1　引言

处理器中发生的瞬时故障可以导致程序运行发生数据流错误或者控制流错误。其中，数据流错误是指数据计算过程中发生瞬时故障，导致程序运行依赖的数据发生错误。由于数据流错误很容易引发程序发生危险的 SDC 错误，因此是容错技术需要优先解决的问题。

由于瞬时故障具有随机性强、持续时间短、重复发生概率小的特点，针对数据流错误的容错技术通常采取冗余计算的思路，即通过进行两次冗余计算并比较结果以实现故障检测，检测出故障后还可以通过再次计算实现故障恢复。早期的冗余计算技术主要采取硬件冗余的形式实现，即通过硬件部件的冗余实现程序的冗余运行和结果校验。典型的商业产品包括 IBM 系列服务器 [60, 98]、HP NonStop 系统 [59] 和 Boeing 777 机载计算机系统 [61]。这类技术的优点是可以提供很高的可靠性，能够满足系统在太空或其他恶劣电磁环境下运行的需要。但是，硬件冗余也带来了很高的成本开销，很难为一般系统所接受。近年来，许多研究提出了基于软件复算的方式实现冗余计算，具体方法是通过软件级的程序复制，实现程序的多次运行和结果比对。这种技术的典型例子包括 EDDI[31]、SWIFT[32] 以及 EsoftCheck[76] 算法。采用软件复算的技术可以提供接近硬件容错技术的可靠性，而且不会引入硬件成本。但是这种方法使程序代码增加了一倍以上，导致难以接受的性能、功耗和存储开销。

如何以较低的开销实现有效的数据流保护是困扰容错研究者至今的难点问题。本章结合软、硬件技术的优势提出了一种高效的数据流错误检测技术 Epipe，可以在有效保护数据流的同时大幅降低容错的开销。Epipe 首先基于改造现有的超标量流水线处理器，提供了一个能够对指令进行冗余保护的硬件平台。由于超标量处理器中有丰富的计算部件，Epipe 平台只需要很少的硬件开销。为了减少冗余保护产生的性能开销，Epipe 基于程序分析方法评估每个指令的重要性，即指令发生故障后导致 SDC 错误的可能性。程序运行时，Epipe 可以根据用户的性能和可靠性配置要求，选择保护最重要的一部分指令。

Epipe 的优势体现在两个方面：（1）Epipe 平台首次开发了一种混合同步、异步检查的方式。同步检查是指只有被保护的指令和其副本指令进行结果比较并确认一致后，该指令的结果才能提交。而基于异步检查，被保护指令则不需要等待与副本指令进行结果比较就可以提交。由于异步检查可以解耦被保护指令和其副本指令间的依赖关系，Epipe 平台对需要等待副本指令结果才能进行同步检查的

指令，改为采取异步检查的方式。由此可以提升流水线的性能，降低冗余保护的性能开销。（2）Epipe 对良性故障不加处理，对症状型故障则利用系统已有的异常、超时检测机制进行处理。Epipe 只冗余保护容易导致 SDC 故障的指令。这种分类高效处理故障的方法大大减少了需要冗余计算的指令，从而进一步有效降低了数据流检测的性能开销。

本章后续内容组织如下：2.2节将 Epipe 和相关的工作进行概括比较；2.3节介绍与 Epipe 相关的背景技术，包括基准流水线配置和基于改造超标量流水线实现的容错技术；Epipe 技术包括硬件平台构建和软件指令重要性分析两个部分，2.4节和2.5节分别提出这两个组成技术；2.6节给出本文评估 Epipe 和其它容错技术时采取的度量标准。2.7节基于故障模拟实验对 Epipe 的效能进行评估，并将其和已有类似技术进行比较。最后，2.8节对本章内容进行小结。

2.2 相关工作

现有的数据流错误检测方法既有基于硬件实现的，也有基于软件或混合软硬件实现的。硬件技术通常具有时空开销低、可靠性高的优点，但是会导致昂贵的硬件开销。软件技术不需要硬件开销，但是会带来大量的时空开销。混合软硬件技术实现的容错方案则可以结合二者的优势，达到以较低的代价获得较高的可靠性的目的。Epipe 技术即属于混合软硬件容错技术。表2.1 总结了 Epipe 和已有典型技术的区别（1.3节对这些技术进行了具体的评述）。

表 2.1　Epipe 和已有典型技术的比较

Category	Technique	Hardware cost	Perf. overhead	Mem. overhead	Configurability	Coverage
Hardware	NonStop [59]	Very high	Low	None	None	Very high
	AR-SMT [62]	Very high	Low	None	None	Very high
	DIVA [65]	High	Low	None	None	High
Software	EDDI [31]	None	High	High	None	High
	SWIFT [32]	None	High	High	None	High
	ESoftCheck [76]	None	High	High	None	High
Hybrid	Superscalar [96]	Low	Low	Moderate	High	Moderate
	Argus [92]	High	Low	Low	None	High
	iSWAT [94]	Low	Low	Low	None	Low
	Epipe	Low	Low	None	High	High

目前混合软硬件实现的数据流检错技术还很少，文献 [96] 提出的 Superscalar 技术是其中的典型代表。和 Epipe 一样，Superscalar 技术也是基于改造超标量流水线实现冗余计算，且可以通过选择保护部分代码减少冗余计算的开销。Epipe 和

Superscalar 技术主要有三个方面的区别：（1）Epipe 将 SDC 故障和良性故障、异常故障等区分开来，并分别采取高效的方式处理，因此 Epipe 比 Superscalar 技术具有更高的容错效率；（2）Superscalar 技术对保护的指令只应用同步检查，这种同步检查制约了流水线的执行效率，尤其当被保护的指令是长延迟指令（例如除法指令）且其副本指令无法同时完成执行时。Epipe 则提出一种混合同步和异步检查的方法，可以在不降低故障检测能力的前提下降低性能开销；（3）Superscalar 技术通过在被保护指令和非保护指令（即不需要冗余保护的指令）之间引入专门的 *CHECK* 指令，控制系统在冗余模式和非冗余模式之间切换，达到选择性保护程序代码的目的。而 Epipe 只需要在指令码中保留一位标记位，即可标示指令是否需要保护。由于被选择保护的代码往往不是连续存放的，Superscalar 技术需要在程序中插装较多的 *CHECK* 指令，因此引入比 Epipe 更多的存储开销。

2.3 背景技术

超标量流水线的并行计算能力也可以用来实现指令冗余保护。已有的相关研究表明，利用超标量处理器中已有的资源，以很少的硬件成本即可在流水线处理器中实现容错保护。本节将首先介绍 Epipe 采用的基准流水线配置，再基于已有研究成果说明如何基于流水线处理器改造实现容错保护。

2.3.1 基准流水线配置

为了支持指令的超标量乱序执行，流水线中设置了乱序缓冲区（Reorder Buffer, ROB）和寄存器别名表（Register Alias Table, RAT）。ROB 是存放流水线中待执行指令的队列结构。由于超标量处理器支持多指令执行，所以 ROB 中通常有大量的指令条目。此外，基准流水线中的物理寄存器（即重名寄存器）也驻留在 ROB 的指令条目中，因此物理寄存器的数量和 ROB 的大小相同。RAT 中存放的是体系结构寄存器和物理寄存器间的重名关系，其中的每个条目对应一个体系结构寄存器。程序执行过程中，如果最近的输出为体系结构寄存器 *r* 的指令 *inst* 已经将结果提交到 *r* 中，则 RAT 中对应 *r* 的条目将会存放一个特定值，表示 *r* 中的值可以使用。否则，*r* 的条目中将会存放 *inst* 在 ROB 中的索引信息，表示将 *r* 重名为 *inst* 在 ROB 中对应的物理寄存器。

基准流水线包含取指（Fetch）、分派（Dispatch）、流出（Issue）、执行（Exec）、写回（Writeback）、提交（Commit）六个阶段。这些阶段的功能为：

取指：流水线从指令缓冲区中按顺序取出待执行的指令，取出的指令将会存放在流水线的取指队列中。每个时钟周期能取的指令数目受流水线的取指带宽约束。

分派：将指令解码、重名，然后分派到 ROB 中。重名时需要查看 RAT，如果被重名的指令 *inst′* 的源操作数 *r* 已经可以使用，则直接读取 *r* 的值即可，否则说明计算 *r* 值的指令 *inst* 还在流水线中执行，此时需要将 *inst′* 的 *r* 操作数重名为 *inst* 的输出（即 *inst* 在 ROB 中的条目）。指令 *inst′* 分派到 ROB 中后，还要根据其目的寄存器更新 RAT。如果 *inst′* 的输出为 *r′*，则将 RAT 中 *r′* 对应的条目内容更新为 *inst′* 在 ROB 中的索引。

流出：从 ROB 中选取源操作数已经就绪并且对应的功能单元空闲的指令流出。每个时钟周期可以同时流出多条指令，并且支持乱序流出。

执行：被流出的指令在对应的功能单元中执行。被占用的功能单元将被设置为 busy 状态直到执行完成。

写回：将执行完成的指令的结果写回到对应的 ROB 条目中，此外还要将该结果重定向发送给依赖该结果的指令。

提交：将执行完成的指令的结果提交到体系结构寄存器中，并将指令从 ROB 中弹出，同时将 RAT 中该体系结构寄存器的状态置为可用。指令提交必须按指令存进 ROB 的顺序进行，即先进入 ROB 的指令提交后，后续指令才能提交。顺序提交的目的是确保指令之间数据依赖关系的正确性。

除了上述配置，本文还假设流水线具备对异常、超时症状的检测能力。此外，为了提高性能，现代流水线通常支持指令的前瞻执行。为了从前瞻失败中恢复，这些系统一般引入了 checkpoint 恢复机制 [93, 99]。因此，本文假设基准流水线中也具有 checkpoint 机制，能够周期性地备份系统状态。

2.3.2 基于改造超标量处理器实现的容错技术

通过改造超标量处理器，文献 [96] 提出的 Superscalar 技术实现了指令在流水线中的冗余保护。为了说明该技术的具体过程并和 Epipe 技术比较，我们在基准流水线中实现了 Superscalar 技术，图2.1给出了相应的体系结构（图中有阴影的区域表示为了实现容错所修改的流水线组件）。下面分别介绍流水线中为了实现容错功能所做的改造。

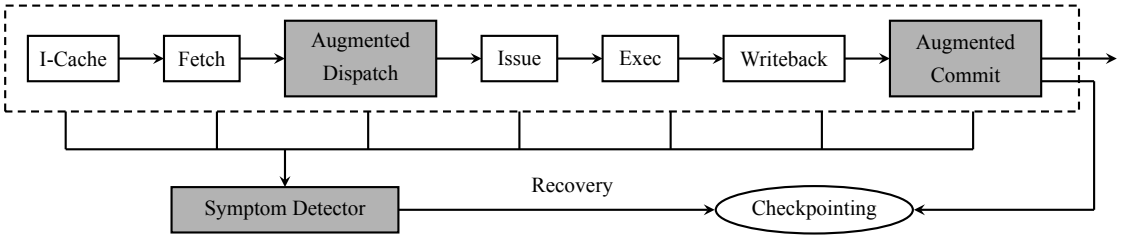


图 2.1　基于流水线改造实现的容错处理器

症状检测器（Symptom Detector）：症状检测器是指系统中的异常检测机制和超时检测机制。一般的流水线处理器中，这些机制发现问题后会立即停止执行并报告问题。在容错处理器中，则先将发现的问题视为由瞬时故障引起的，并利用系统中的 checkpoint 机制进行恢复执行。如果检测到的问题不再出现，则说明故障恢复成功。否则认为该问题是系统固有的缺陷造成的，采取原有的处理措施即可。

扩展的分派单元（Augmented Dispatch）：为了在流水线中实现指令冗余保护，需要改造原有的分派单元。扩展后的分配单元根据处理器模式来决定指令的分派操作。处理器模式包括冗余模式和非冗余模式，由专门的 ***CHECK*** 指令控制处理器的模式切换。如果处理器处于非冗余模式，则指令的分派操作不变。反之，则需要将指令复制得到其副本，并将指令和其副本均分派到 ROB 中。副本指令在 ROB 中必须存放在源指令的下一个条目中。

冗余模式下，指令分派的挑战在于如何确保源指令之间数据流依赖关系及副本指令之间数据流依赖关系的独立性。在进行重名时，如果源指令 *inst* 需要读取 *r* 的值作为源操作数，则需要在 RAT 中查看 *r* 的状态。若 *r* 已经有可用的值，则 *inst* 和其副本均使用 *r* 的值作为源操作数即可。否则，输出 *r* 值的指令 *inst′* 仍在执行中：如果 *inst′* 是受冗余保护的指令，则 *inst* 和其副本分别从 *inst′* 和 *inst′* 的副本的 ROB 条目读取 *r* 的值。如果 *inst′* 没有被复制，则 *inst* 和其副本均从 *inst′* 的 ROB 条目读取 *r* 的值。

扩展的提交单元（Augmented Commit）：在非冗余模式下，指令的提交过程不变。在冗余模式下，如果一条源指令和其副本指令均已完成执行，则需要比较二者的结果。如果结果一致，则可以提交源指令并将源指令和其副本指令从 ROB 中弹出，否则说明系统出现了故障，需要启动 checkpoint 机制进行恢复。如果源指令和其副本指令还没有全部完成执行，则提交操作需要推迟到二者全部完成执行后才能进行。

流水线的其他阶段不需要修改。在冗余保护的指令中，访存指令需要特殊处理。流水线将访存操作分解为两个操作：地址计算和访问存储空间。冗余模式下，只有源指令进行访问存储空间操作，副本指令则只进行地址计算。提交时，需要比较源指令和副本指令地址计算的结果，以确认访存地址是正确的。这种处理减少了冗余保护的存储带宽压力，但是无法检测访存过程中出现的故障。另外，流水线从冗余模式切换到非冗余模式前，必须要确保流水线中所有指令完成提交。原因是非冗余模式下执行的指令 *inst* 可能依赖于冗余模式下的指令 *inst′*，如果 *inst′* 还没有完成提交操作，则无法确保其结果是正确的，*inst* 读取的值可能因此存在错误。

2.4 Epipe 平台技术

为了以较低的开销实现程序数据流保护，Epipe 首先基于改造现有的超标量处理器，提供了能够对指令进行选择性保护的平台。Epipe 平台尽可能利用超标量流水线中已有的硬件资源，只需要简单的硬件扩展即可实现指令的冗余保护。图2.2给出了 Epipe 平台的体系结构，图中阴影区域为 Epipe 技术在基准流水线中进行扩展的模块。下面从指令复制、同步检查、异步检查三个方面说明 Epipe 对基准流水线所做的扩展。

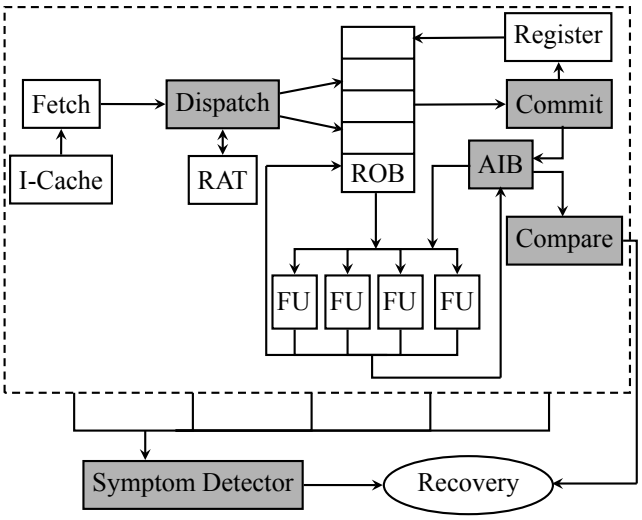


图 2.2 Epipe 平台体系结构

指令复制：指令复制的过程和 Superscalar 技术 [96] 类似，也是在分派阶段进行。但是，Epipe 平台基于指令码中的标记位进行指令的选择性复制。所有的指令码中均包含一位冗余标记位，用以标示需要冗余保护的指令。在指令解码的时候，如果一条指令的冗余标记位为 0，则说明该指令不需要复制，将该指令直接重名、分派即可。如果指令的冗余标记位为 1，则需要对该指令进行冗余保护。另外，在对受保护的指令 *inst* 进行重名时，如果 *inst* 需要依赖指令 *inst′* 的结果，则即便 *inst′* 是受保护指令，*inst* 和其副本指令的输入也都将重名为 *inst′* 的输出。这种设计可以提高流水线的性能。虽然 *inst′* 中的故障会传播给 *inst* 及其副本指令，但是由于 *inst′* 先于 *inst* 提交，流水线提交 *inst′* 时可以检测出该故障，所以这种情况并不会影响系统的可靠性。

为了便于指令集扩展，很多指令集格式中存在一些未使用的保留位 [92, 100]。对于这些指令集，指令的冗余标记位可以存放于保留位中，不会导致额外的存储开销。2.7节进行 Epipe 评估时采用的 PISA 指令集即属于此种情况。而对于不存在保留位的指令集，则需要将指令的冗余标记位单独存放，处理器取指令的同时

还要取出相应的标记位。文献 [101, 102] 给出了在处理器中引入额外标记位的方法，其方法也可应用于存放冗余标记位。首先，在存储空间中需要为每个指令引入一个冗余标记位。此外，数据总线也要额外增加一位，使得处理器在取指令的同时也能取出其冗余标记位。最后，ROB 和各种指令队列中也应增加一位，用以存储冗余标记位。假设底层支持的是 32 位指令集，则由冗余标记位导致的额外存储开销仅仅是 1/32=3.125%。上述设计的主要问题是增加了处理器的设计复杂度。但是，考虑到这种设计能够获得的性能、存储方面的好处，所花的代价仍然是值得的。

同步检查：在指令流出、执行阶段，对于被复制的指令，流水线优先流出、执行源指令。在指令提交阶段，未保护的指令可以正常提交。对于被标记为需要保护且已经完成执行的指令 *inst*，则要分三种情况处理：（1）如果 *inst* 的副本指令也已经完成执行，则将 *inst* 和其副本进行同步检查，如果二者结果一致，则可以提交 *inst*。否则说明指令执行中出现了故障，需要启动恢复机制进行故障恢复。（2）如果 *inst* 的副本指令已经流出但还没有完成执行，则需要延迟提交 *inst* 直到其副本指令完成执行。（3）如果 *inst* 的副本指令尚未流出，则可以直接提交 *inst*，但是需要将 *inst* 的副本指令和 *inst* 的结果拷贝到流水线中一个新引入的指令队列 AIB（Asynchronous Instruction Buffer）中，并将 *inst* 的副本指令从 ROB 中移除。由于 *inst* 和其副本指令依赖于相同的指令，且 *inst* 已经完成执行，所以 *inst* 的副本指令的输入均是可用的。因此，AIB 中的指令均处于流出就绪状态。三种情况中，前两种情形的 *inst* 指令定义为同步检查指令，最后一种情形的 *inst* 指令定义为非同步检查指令。

AIB 的结构类似于 ROB，唯一的扩展是在每个条目中需要保存非同步检查指令的结果。对于非同步检查指令 *inst*，若 AIB 中没有剩余的空间可以存放其副本指令和结果，*inst* 的提交操作将会延迟，直到 AIB 中有可用的空间或者其副本指令开始流出、执行。如果 AIB 先出现可用空间，则直接提交 *inst*，并将 *inst* 的副本指令和结果放入 AIB 即可。否则，*inst* 的结果将会通过同步检查进行校验。AIB 的空间不足将会制约非同步检查的性能优化效果，但是通过配置合适的 AIB 大小，可以有效地避免上述情形。假定 AIB 的大小为 k，对于不同配置的流水线，k 有不同的最优值（2.7节的实验评估结果可以证明，对于 ROB 大小为 64 的基准流水线，k 的最优值是 8）。

非同步检查：为了减少非同步检查指令的故障检测延迟，AIB 中的指令在流出时比 ROB 中的指令拥有更高的优先级。在写回阶段，对于已经完成执行的 AIB 指令 *inst′*，需要进行非同步检查，即将其结果和 AIB 中对应的非同步检查指令的结果进行比较。如果比较结果一致，说明程序运行是正确的，可以将 *inst′* 从 AIB

中移除。否则说明已经提交的非同步检查指令是不可靠的，需要利用 checkpoint 机制进行恢复。由于非同步检查操作可以和写回阶段的其它操作并行处理，所以非同步检查并不需要一个额外的阶段专门执行。

如上所述，Epipe 平台将非同步检查指令的结果预测为正确的并对其执行了非同步检查。这样做的原因是，如果非同步检查指令要等待进行同步检查，将会带来提交延迟。由于基准流水线采取的是顺序提交规则，非同步检查指令的延迟提交将会导致其后执行的指令也无法提交，从而造成流水线阻塞。所以，对非同步检查指令执行同步检查操作将会带来明显的性能损耗（尤其对于浮点指令、除法指令等长延迟指令)。既然瞬时故障在一次程序运行过程中发生的概率仍是极低的，将非同步检查指令的结果预测为正确的是高度可信的。利用这一高可信的预测，可以直接提交非同步检查指令的结果，从而获取更好的性能。非同步检查的另外一个好处是，当 ROB 中的指令因为 Cache 访问未命中而出现阻塞时，AIB 中的指令仍可以正常流出。已有的研究结果表明 [103]，开发利用这种 Cache 访问未命中带来的延迟可以明显提高系统性能。

由于 Epipe 平台只对少量必要的指令采取非同步检查，AIB 并不需要很大的空间，同时也不会带来很大的故障检测延迟。另外需要指出的是，由于 I/O 操作的错误结果将会导致用户可见的失效，这类操作必须进行同步检查。

2.5 指令重要性分析

尽管可以利用超标量流水线的并行计算能力，在 Epipe 平台上对程序进行完全冗余保护仍然会导致明显的性能开销。而且相当一部分的冗余保护是并不必要的，例如，发生良性故障的指令不会影响程序结果，因此不需要任何处理；指令中的症状型故障则可以用低代价的症状检测机制加以解决，并不需要采取昂贵的冗余保护。为了减少冗余保护的开销，Epipe 基于提出的指令重要性分析技术，将冗余保护的目标集中到防止 SDC 故障上。指令重要性（Instruction Criticality, IC）的定义是指令发生故障后导致程序发生 SDC 错误的可能性。基于指令重要性分析结果，Epipe 还可以根据用户的性能和可靠性需求，配置保护最重要的一部分指令。下面将介绍提出的 IC 分析方法。

IC 分析的目的是评估每条指令对程序输出的影响（本文假设存储映射 I/O 模型)。IC 分析首先需要对程序进行数据流分析，获取指令之间的数据依赖关系。基于指令和程序输出之间的数据流传播关系，可以将程序中的指令分为三类：(1）指令中的故障可以直接传播到输出数据，即待存储数据；(2）指令中的故障可以直接传播到输出数据的地址，即存储操作的地址；(3）指令中的故障不能直接影响输出操作，但可以传播到分支跳转操作，间接影响输出的正确性。

在程序中提取输出操作和分支操作的后向切片，可以达到识别上述三类指令的目的。提取程序后向切片的方法参见5.3.2节内容。图2.3 给出了一个基于数据流分析识别第一类指令的例子，图中加星号标注的均为识别出的指令。首先，指令 I_6 是输出操作（该操作将寄存器 $r8$ 的值存入 memory），基于数据流分析结果可知，指令 I_5 对 $r8$ 进行了定值，所以 I_6 依赖于指令 I_5，因此 I_5 被识别为第一类指令。同样地，由于 I_5 依赖于 I_3 和 I_2（I_3 和 I_2 分别对 $r5$ 和 $r6$ 进行了定值），I_3 和 I_2 也被识别为第一类指令。最后，由于指令 I_2 依赖于 I_1，所以 I_1 也被识别为第一类指令。

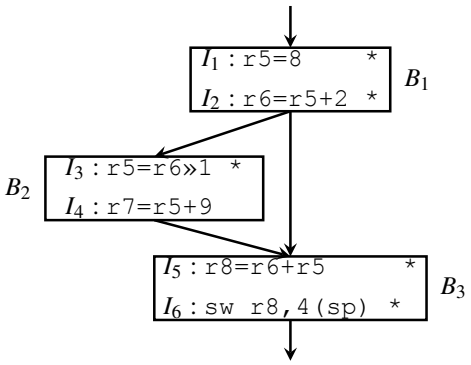


图 2.3　识别第一类指令示例

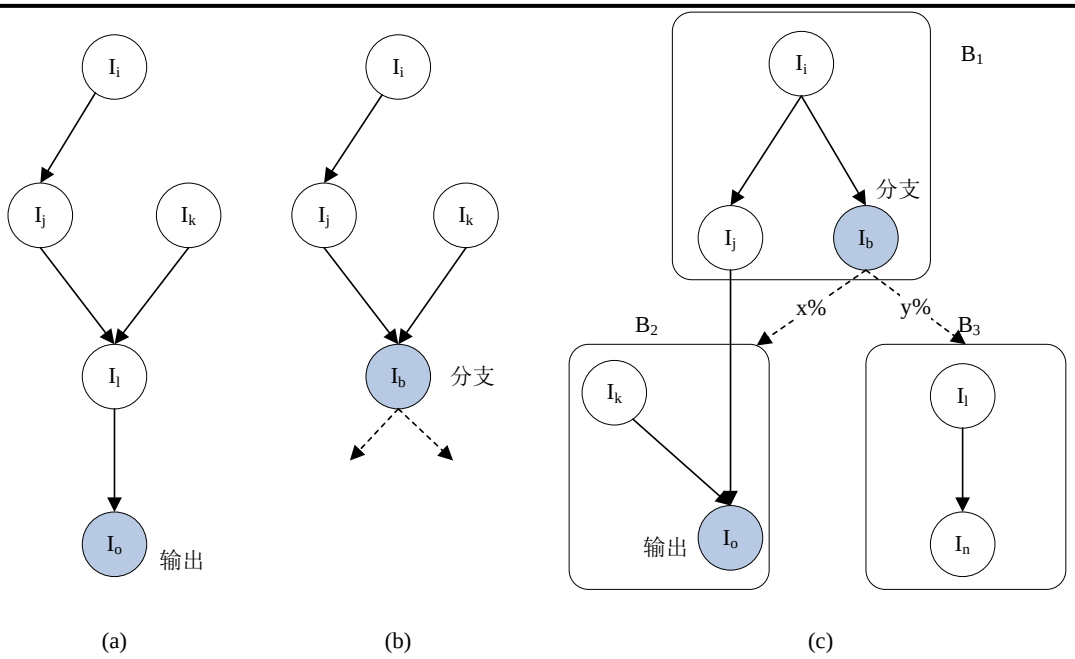
对于第一类指令 I_i，假设其故障能够传播到输出操作 I_o 的待存储数据中，则其重要性可以基于以下等式评估（此种情形如图2.4(a) 示例所示）：

$$IC(I_i) = P_{SDC}^{data}(I_o) \times \Pi_{j=i}^{l}(1 - P_{mask}(I_j) - P_{excp}(I_j)) \tag{2.1}$$

其中，$P_{SDC}^{data}(I_o)$ 表示待输出数据存在错误时，程序发生 SDC 的概率。若 I_o 是半字长或单字节存储操作，$P_{SDC}^{data}(I_o)$ 的值分别为 0.5 和 0.25，否则 $P_{SDC}^{data}(I_o)$ 的值设定为 1。I_j 为从 I_i 到 I_o 的数据传播路径上的指令。发生在 I_i 中的故障，在传播到 I_o 之前，可能被 I_j 屏蔽或者在 I_j 执行时发生系统异常。因此，评估 I_i 对 I_o 的影响需要考虑上述两个因素。$P_{mask}(I_j)$ 表示指令 I_j 屏蔽故障的概率，$P_{excp}(I_j)$ 则表示 I_j 发生故障后直接触发异常的概率。$P_{mask}(I_j)$ 和 $P_{excp}(I_j)$ 可以基于程序 profile 信息分析获得，或者通过向 I_j 注入故障后统计其结果获得，具体分析方法将在2.7 节实验方法部分介绍。

若指令 I_i 中的故障可以传播到多个输出（假设为 d 个），则其重要性计算公式为：

$$IC(I_i) = \sum_{m=1}^{d}(P_{SDC}^{data}(I_{o_m}) \times \Pi_{m_j=i}^{l_m}(1 - P_{mask}(I_{m_j}) - P_{excp}(I_{m_j}))) \tag{2.2}$$



(a) (b) (c)

图 2.4 指令重要性分析示例

对于第二类指令 I_i，假设其故障能够传播到输出操作 I_o 的地址中，则其重要性可以基于以下等式评估：

$$IC(I_i) = P_{SDC}^{addr}(I_o) \times \Pi_{j=i}^{l}(1 - P_{mask}(I_j) - P_{excp}(I_j)) \tag{2.3}$$

其中，$P_{SDC}^{addr}(I_o)$ 表示输出操作的地址存在错误时，程序发生 SDC 的概率。由于地址存在错误时，程序绝大部分情况下会发生 SDC 错误或地址访问越界异常，所以通过先获取程序访存空间的 profile 信息，计算出地址访问越界的概率，即可估计出 $P_{SDC}^{addr}(I_o)$ 的值。例如，假设表示程序访存空间的最大地址需要用 L 位二进制数据，系统字长为 32 位，则发生地址访问越界的概率可以估计为 $(1 - L/32)$，相应地，$P_{SDC}^{addr}(I_o)$ 的值应为 $L/32$。

对于第三类指令 I_i，即影响程序分支跳转的指令，其对程序输出的影响可以基于以下等式评估（此种情形如图2.4(b) 示例所示）：

$$IC(I_i) = P_{SDC}^{br}(I_b) \times \Pi_{j=i}^{l}(1 - P_{mask}(I_j) - P_{excp}(I_j)) \tag{2.4}$$

其中，I_j 为从 I_i 到分支跳转操作 I_b 的数据传播路径上的指令，$P_{SDC}^{br}(I_b)$ 表示 I_b 在输入出错时导致程序最终发生 SDC 的概率。$P_{SDC}^{br}(I_b)$ 参数可以通过对分支指令 I_b 进行故障注入分析获得。

当 I_i 和其能够影响的输出操作或分支操作不在同一个基本块时，IC 估计需要进一步考虑从 I_i 到目标指令间的路径执行概率（图2.4(c) 给出了一个相关例子）。此时，指令重要性的计算等式为：

$$IC(I_i) = \sum_{p \in paths} (IC_p(I_i) \times P_{path}(p)) \tag{2.5}$$

其中，p 为从 I_i 到输出的一条路径，$P_{path}(p)$ 为 p 执行的概率，$IC_p(I_i)$ 为在路径 p 上计算出的 IC 值。

算法2.1给出了指令 IC 分析的具体过程。算法首先对每个指令的 IC 值进行初始化（第 1-10 步），并针对每个输出操作或条件分支 I_i，基于数据流分析从程序中提取其后向切片 $BS(I_i)$（第 11-12 步），以达到将指令分为三类的目的。然后按照距离目标指令 I_i 从近到远的顺序，逐个为每个 $BS(I_i)$ 中的指令计算出相对于 I_i 的重要性 $IC^i(I_{i_j})$（第 13-17 步）。若 I_{i_j} 和 $I_{i_{j-1}}$ 不属于同一个基本块，还要考虑二者之间的路径概率 $P_{path}(I_{i_j} \rightarrow I_{i_{j-1}})$（第 19 步）。最后，算法为影响多个输出的指令计算出其最终的 IC（第 23-25 步）。

为所有待保护指令计算出 IC 值后，需要将指令按 IC 的大小进行排序。编译时可以根据具体的性能和可靠性需求，将最重要的部分指令标记为需要冗余保护的指令。

2.6 系统可靠性度量标准

目前已经存在许多评估系统可靠性的度量标准，但是已有的度量大多是考虑软件失效或者硬件永久性失效，这些标准很难直接用来度量瞬时故障对可靠性的影响。本节将对瞬时故障背景下系统的可靠性度量进行分析，并以此为基础介绍容错技术的评估方法。

基于本文第一章介绍的 MTTF 度量（公式 (1.6)）可以评估系统的可靠性。但是，该度量没有考虑到性能和可靠性的关系，因此并不能准确地评估系统可靠性。假设有需要比较的两个系统 A 和 B，其可靠性（$MTTF_A$、$MTTF_B$）和性能（P_A、P_B）分别满足下列等式：

$$MTTF_A = \frac{MTTF_B}{2} \tag{2.6}$$

$$P_A = 2P_B \tag{2.7}$$

基于上述等式，系统 B 的平均无故障时间是 A 的两倍，但是其执行相同的程序所需要的时间也是 A 的两倍，因此 A 和 B 的可靠性应该是相等的。但是，基于

算法 2.1 CalculateIC ( )

输入: 程序 P 中每条指令的 P_{mask}、P_{excp} 参数，条件分支指令的 P_{SDC}^{br} 参数
输出: 每条指令的 IC

for each instruction $I_i \in P$ **do**
    if I_i is a conditional branch **then**
        $IC(I_i) = P_{SDC}^{br}(I_i)$;
    else if I_i is an output operation **then**
        $IC^{data}(I_i) = P_{SDC}^{data}(I_i)$;
        $IC^{addr}(I_i) = P_{SDC}^{addr}(I_i)$;
    else
        $IC(I_i) = 0$;
    end if
end for
for each conditional branch or output operation $I_i \in P$ **do**
    Identify the backward slice of I_i and save it as an instruction link $BS(I_i)$;
    for each $I_{i_j} \in BS(I_i)$ (iterate I_{i_j} in the ascending order of the distance between I_{i_j} and I_i) **do**
        if j=0 **then**
            $IC^i(I_{i_j}) = IC(I_i)$;
        else if I_{i_j} and $I_{i_{j-1}}$ are in the same basic block **then**
            $IC^i(I_{i_j}) = IC^i(I_{i_{j-1}}) \times (1 - P_{mask}(I_{i_j}) - P_{excp}(I_{i_j}))$;
        else
            $IC^i(I_{i_j}) = P_{path}(I_{i_j} \to I_{i_{j-1}}) \times IC(I_{i_{j-1}}) \times (1 - P_{mask}(I_{i_j}) - P_{excp}(I_{i_j}))$;
        end if
    end for
end for
Let n be the sum of conditional branches and output operations in P;
for each $I_k \in P$ **do**
    $IC(I_k) = \sum_{i=1}^{n} IC^i(I_k)$;
end for

MTTF 度量，却只能判定为 B 是更可靠的。针对 MTTF 的不足，文献 [104] 提出了平均无故障执行指令数（Mean Instructions To Failure，MITF），其计算公式为:

$$MITF = \frac{number\ of\ committed\ instructions}{number\ of\ errors\ encountered} = \frac{number\ of\ committed\ instructions}{\frac{execution\ time\ in\ cycles}{frequency \times MTTF}} = IPC \times frequency \times MTTF = \frac{frequency}{\lambda_s} \times \frac{IPC}{AVF} \tag{2.8}$$

其中，IPC（Instructions Per Cycle）表示系统每个时钟周期执行的指令数。MITF考虑了 IPC 和系统频率（frequency）这些硬件平台特征，却没有考虑程序相关的因素，因此只适合用来评估硬件容错技术，而不适用与软件容错技术的评估。

Reis[104] 针对 MITF 的不足提出了新的度量标准，即平均无故障完成任务量（Mean Work To Failure，MWTF），其计算公式如下：

$$MWTF = \frac{amount\ of\ work\ completed}{number\ of\ errors\ encountered} = \frac{1}{\lambda_s} \times \frac{1}{AVF \times execution\ time} \tag{2.9}$$

该度量考虑了完成一个任务需要的时间（execution time）和在此期间系统发生的失效数量。这里的任务是一个通用的概念，程序、事务、线程等均可以视为任务。由于 MWTF 既考虑了软件的因素，又考虑了硬件的特征，所以可以作为软件容错和硬件容错技术的通用评估标准。计算 AVF 通常需要在模拟器下进行逐位的分析，需要大量的计算资源。容错研究领域一个更为普遍的做法是对程序进行故障注入实验，然后统计故障覆盖率（Fault Coverage，FC），并将 FC 作为 AVF 的近似值。FC 的计算公式为：

$$FC = \frac{tolerated\ faults}{total\ injected\ faults} \tag{2.10}$$

其中，tolerated faults 指的是被容错技术成功处理的故障。基于容错技术的故障覆盖率，可以估计出系统的 AVF 和 MWTF，其计算公式分别为：

$$AVF = 1 - FC \tag{2.11}$$

$$MWTF = \frac{1}{\lambda_s} \times \frac{1}{(1 - FC) \times execution\ time} \tag{2.12}$$

根据等式 (2.12)，容错技术需要的性能开销越低，能够提供的故障覆盖率越高，则系统的可靠性也就越高。对于混合软硬件实现的容错技术，或者硬件容错技术，可以采用 MWTF 进行评估（例如本章技术和第五章技术的评估）。而对于软件容错技术的评估或比较（例如第四章技术的评估），由于不涉及硬件设计的修改，一个更为简单和广泛采用的评估标准是容错效率（Fault-tolerance Efficiency，FE）[91]，FE 可以基于容错技术的性能开销和故障覆盖率计算，具体方法为：

$$FE = \frac{1}{(1 - FC) \times PO} \tag{2.13}$$

其中，PO 表示容错技术的性能开销（相对于源程序执行时间增加的比例）。从定义可以看出，容错效率随着故障覆盖率的增加而增加，随着性能开销的增加而减

少。在提供的故障覆盖率相同的条件下，容错效率较高的容错技术所产生的性能开销会较低。基于公式 (2.12) 可以获得 FE 和 MWTF 的关系：

$$MWTF = \frac{1}{\lambda_s} \times \frac{1}{\frac{1}{FE \times PO} \times T_{source} \times (1 + PO)} = \frac{1}{\lambda_s} \times \frac{FE}{T_{source} \times (1 + \frac{1}{PO})} \tag{2.14}$$

其中 T_{source} 表示源程序的执行时间。从公式 (2.14) 可以看出，性能开销相同的前提下，容错效率高的技术所提供的可靠性更高。

2.7 实验评估

本节在模拟器上对 Epipe 进行了性能评估和可靠性评估。为了和已有工作比较，本节也评估了 Superscalar[96] 技术的效果。本节内容组织如下：2.7.1节介绍实验评估的方法，包括采用的 benchmark、故障注入方法等；Epipe 平台中 AIB 的大小将会影响平台的性能，因此2.7.2节评估不同 AIB 大小对 Epipe 平台性能的影响，并基于评估结果选择最优的 AIB 配置；Epipe 和 Superscalar 均属于软、硬件结合的技术，2.7.3节先对两种技术的硬件平台进行评估比较；最后，结合软件可配置性，2.7.4节分析比较 Epipe 和 Superscalar 技术的整体效果。

2.7.1 实验方法

我们通过改造 SimpleScalar[105] 的 sim-outorder 乱序流水模拟器实现了 Epipe 平台，并基于模拟器运行获取程序的 profile 信息。SimpleScalar 是一种广泛使用的体系结构模拟器，其开发工具集可以配置成许多不同体系结构的模拟器，如 PISA、Alpha、PowerPC 和 X86 等。其中，PISA 指令集是一种类似 MIPS 的 RISC 指令集，也是实验使用的指令集。实验从 Mibench[106] 测试套件中选取了 9 个 benchmark 作为测试用例。由于 PISA 指令集中包含 16 位保留位，Epipe 使用其中的一位作为冗余标记位，用来标记指令是否需要冗余保护。

在对指令进行重要性分析前，首先需要获取每个指令的 P_{mask} 和 P_{excp} 参数。为了减少分析的开销，实验只具体分析了逻辑操作和亚字长操作（例如半字加、半字乘等）的 P_{mask} 参数。分析方法为给定 20 组不同的输入，然后分别运行程序并获取上述操作的参数值，最后基于这些值计算出指令的平均 mask 概率。其它指令的 P_{mask} 简单地设定为 0。同样地，实验只具体分析了乘除法操作的 P_{excp} 参数。分析方法为对每个上述操作进行 20 次故障注入实验，基于指令触发异常的次数计算出 P_{excp} 的近似值。其它指令的 P_{excp} 参数也直接设定为 0。对于分支跳转指令 I_b，需要提前获取其 $P^{br}_{SDC}(I_b)$ 参数，获取方法为向其输入（即源操作数）中注入 20 次故障，然后将程序发生 SDC 的比率作为 $P^{br}_{SDC}(I_b)$ 的近似值。

实验采用本文第5章开发的 SmartInjector 故障注入工具评估 Epipe 的故障检测能力。注入的故障按照程序运行的结果可以分为五类，即良性故障（Benign）、异

常故障（Exception）、超时故障（Timeout）、结果错误故障（Silent Data Corruption，SDC）、检出故障（Detected）。其中，Detected 故障是指被 Epipe 成功报告检出的故障。此外，实验采用正常运行时间的 5 倍值做为超时故障的判断标准。注入的故障中，Benign 故障不会影响程序结果，因此不需要容错处理。Exception 和 Timeout 故障属于症状型故障，可以被系统中设置的症状检测器处理。Detected 故障是 Epipe 可以处理的故障。SDC 故障既没有被检出，又导致程序输出结果错误，是本文提出容错技术着力解决的问题。基于上述分类和公式 (2.10)，可以将容错技术的故障覆盖率定义为：

$$FC = \frac{Benign + Exception + Timeout + Detected}{total\ injected\ faults} \tag{2.15}$$

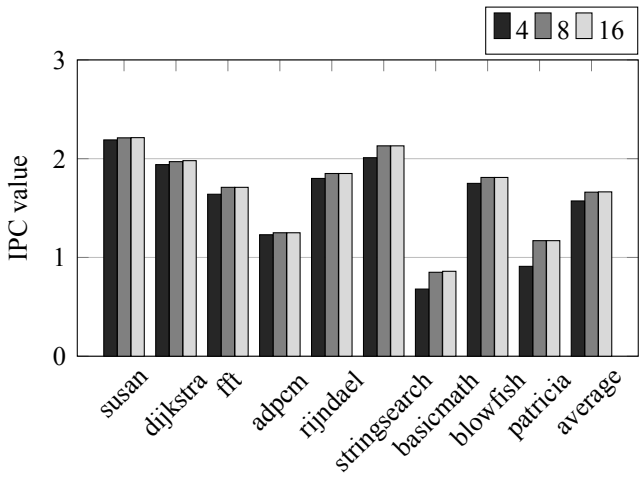


图 2.5　不同 AIB 大小时系统的 IPC 结果

2.7.2　选择最优的 AIB 大小

AIB 的大小可以影响 Epipe 平台的性能。为了确定最优的 AIB 大小 k，实验分别评估了 k 的值取 4、8、16 时 Epipe 平台的性能。图2.5 给出了性能评估的结果，用 IPC （Instructions Per Cycle）表示。一般来说，随着 AIB 不断增大，更多的指令可以放进 AIB 进行非同步检查，从而可以获得更多的性能提升。但是，由于非同步检查指令的数量是有限的，所以实际需要的 AIB 大小存在一个上限。如图 2.5所示，当 k 从 4 增加到 8 时，IPC 的平均值从 1.572 增加到了 1.661。然而，当 k 从 8 进一步增加到 16 时，IPC 仅仅有稍微增加（k 为 16 时 IPC 的平均值为 1.664）。因此，8 个 AIB 条目已经足够满足提升系统性能的需要，同时又不会带来较大的硬件开销。后续的 Epipe 评估实验结果均是将 k 设定为 8 的条件下获得的。

2.7.3 Epipe 平台评估

为了验证基于改造超标量流水线实现的容错技术，实验首先在保护整个程序的前提下对 Epipe 平台的性能进行了评估，并和 Superscalar 平台 [96] 进行了比较。图2.6给出了 Epipe 平台和 Superscalar 平台对程序进行冗余保护的性能开销。如本章2.4节所述，Epipe 平台采取的混合检查方式可以减少同步检查和 Cache 访问未命中带来的性能开销，因此可以提升系统性能。性能评估结果显示，Epipe 平台下进行冗余保护的性能开销比 Superscalar 平台平均减少了 8.0% （平均性能开销分别为 46.4% 和 54.4%)。

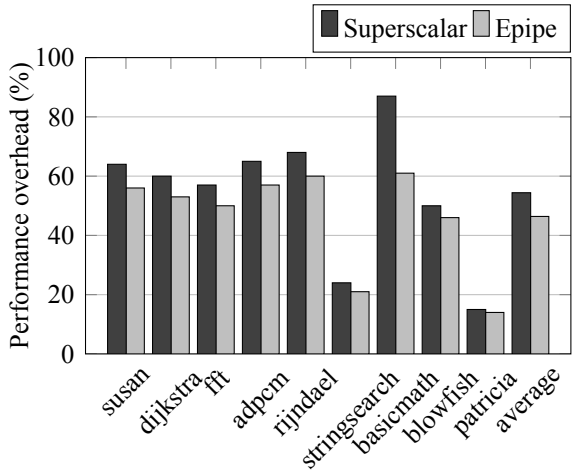


图 2.6 Epipe 平台和 Superscalar 平台的性能开销

由于 Epipe 平台和 Superscalar 平台均是通过在流水线中复制指令实现故障检测，所以二者理论上具备相同的可靠性。实验对两种平台的可靠性下界和上界进行了评估。可靠性下界是在不进行任何冗余保护时（即仅依靠症状检测器进行保护）获得的可靠性。可靠性上界则是指对整个程序进行冗余保护时所能达到的可靠性。图 2.7 给出了可靠性的上、下界评估结果（L 表示下界，U 表示上界)。从图中可以看出，在不进行任何冗余保护时，系统的平均故障覆盖率仍然达到了 67.8%，包括 48.7% 的良性故障，16.6% 的异常故障和 2.5% 的超时故障。这说明在不付出任何性能代价的情况下，上述平台仍然可以提供相当的可靠性。但是，不进行任何冗余保护也造成 32.2% 的故障最终导致了 SDC，使得系统可靠性难以满足用户需求。在进行全冗余保护后，Epipe 平台（或 Superscalar 平台）能够提供的可靠性则达到了 99.1%，大大提高了系统可靠性。但是，全冗余保护在有效检测 SDC 故障的同时，也将良性故障的比例从 48.7% 大幅降低到了 5.5%，这意味着系统报告了大量的“假死”（false positive)。这些“假死”现象是由过度的冗

余保护引起的，浪费了大量的系统性能。因此，实验结果也证明了 Epipe 技术通过分析指令的重要性，重点保护对 SDC 敏感的指令的合理性。

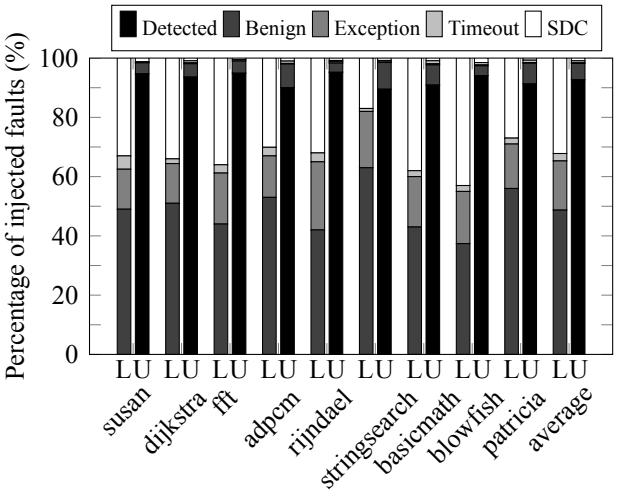


图 2.7　Epipe 平台可靠性上、下界评估结果

严格来说，系统可靠性的评估不仅要考虑到故障覆盖率，也要考虑到性能开销、硬件开销等，因为后者会增加系统发生故障的概率。本文基于公式 (2.12) 定义的 MWTF 度量评估容错系统的可靠性。Epipe 平台全冗余保护时的可靠性（$MWTF_E$）相对于基准平台（$MWTF_B$）的增加幅度可以用下列等式评估:

$$\frac{MWTF_E}{MWTF_B}-1=\frac{\frac{1}{\lambda_s^E}\times\frac{1}{(1-FC_E)\times T_E}}{\frac{1}{\lambda_s^B}\times\frac{1}{(1-FC_B)\times T_B}}-1=\frac{\lambda_s^B}{\lambda_s^E}\times\frac{(1-FC_B)\times T_B}{(1-FC_E)\times T_E}-1 \tag{2.16}$$

其中，λ_s^B 和 λ_s^E 分别表示基准平台和 Epipe 平台的原始故障率，T_B 和 T_E 分别表示程序在基准平台和 Epipe 平台下的执行时间，FC_B 和 FC_E 则分别表示两种平台下的故障覆盖率。由于 Epipe 平台在基准平台的基础上做了硬件扩展，所以其原始故障率会相应地增加。本文将原始故障率增加幅度估计为 15%，即 $\frac{\lambda_s^B}{\lambda_s^E}=\frac{1}{1.15}$。图2.8给出了基于上述评估方法计算出的 Epipe 平台可靠性增加幅度。从中可以看出，程序在 Epipe 平台上的 MWTF 相对于基准平台均有明显提高，平均增加了 21.4 倍。

2.7.4　Epipe 技术可配置性评估

为了以用户可以接受的性能开销获得最大的可靠性，Epipe 技术基于指令重要性分析提供了可配置的保护方法。实验对 Epipe 技术的可配置性进行了评估。评估方法为从程序中选取最重要的部分指令进行保护，通过调节保护的指令数量，使其性能开销分别为 Epipe 全冗余保护性能开销（T_E）的 1/4、2/1 和 3/4（即平均开销分别为 11.6%、23.2% 和 34.8%），然后在这三种配置下分别评估系统

的可靠性。由于 Superscalar 技术也包含一个可配置的选择性保护方法，本节将 Epipe 技术和其进行了比较。为了公平比较，实验配置 Superscalar 技术的保护力度使其和 Epipe 技术的性能开销相同，然后分析其可靠性。

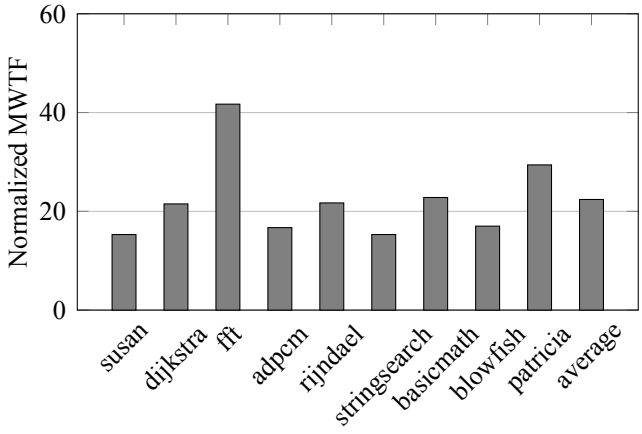


图 2.8　Epipe 平台相对于基准平台的可靠性提高幅度

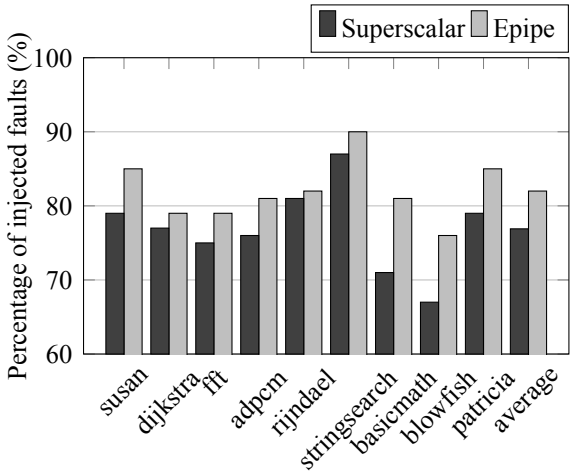


图 2.9　性能开销为 $T_E/4$ 时的故障覆盖率

两种技术在三种性能约束配置下的故障覆盖率如图2.9-2.11所示。结果显示，Epipe 技术的平均故障覆盖率分别达到了 82%、92.6% 和 96.8%，SDC 故障相比无冗余保护的情况下分别减少了 44%、77% 和 90%。从该结果可以看出，Epipe 技术通过对故障分类处理，即利用症状检测技术处理异常、超时故障，而只对容易导致 SDC 的指令进行冗余保护，最大限度减少了冗余保护的范围，达到了以较小的性能开销获取最大可靠性的目的。另一方面，评估结果也说明了程序中不同指令的重要性有明显不同，只保护最重要的部分指令即可显著提高可靠性。

另外，从图中可以看出，在性能开销相同的前提下，Epipe 技术的故障覆盖率均比 Superscalar 技术高。在三种性能约束配置下，Epipe 技术的故障覆盖率分别

比 Superscalar 技术高 5.2%、6.6% 和 5% （后者的平均故障覆盖率分别为 76.8%、86% 和 91.8%）。Epipe 技术的优势来自于两个方面：（1）Epipe 平台采取了混合检查方式，具有更高的性能，因此在性能约束相同的前提下，可以保护更多的指令；（2）Epipe 技术通过指令重要性分析挑选最有可能导致 SDC 的指令进行保护，其它指令中的故障则采用轻量级的方法处理，减少了不必要的冗余保护，而 Superscalar 技术在选择部分代码保护时并没有进行这种区分。综合上述因素，Epipe 技术比 Superscalar 技术具备了更高的容错效率。

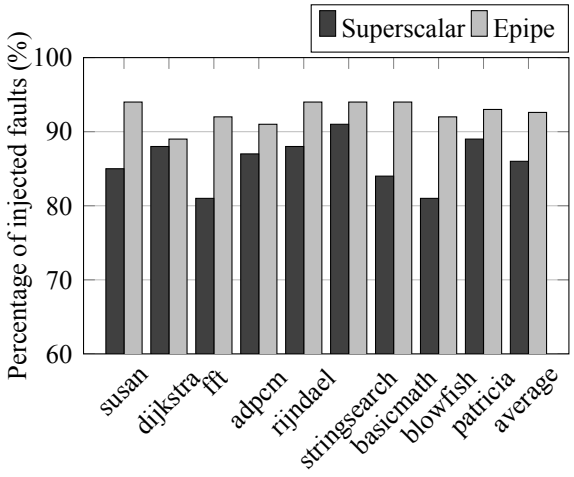


图 2.10　性能开销为 $T_E/2$ 时的故障覆盖率

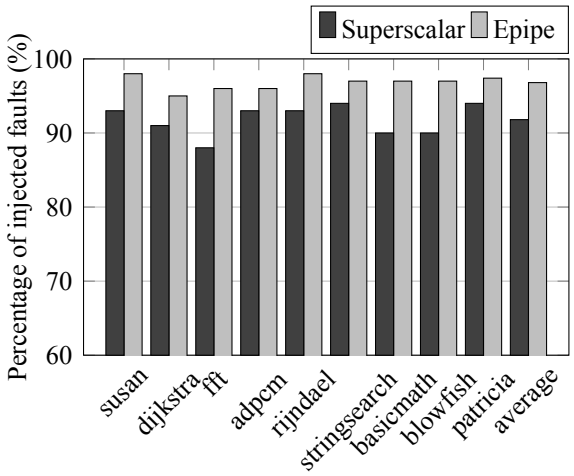


图 2.11　性能开销为 $3T_E/4$ 时的故障覆盖率

2.8　本章小结

处理器中的瞬时故障很容易导致程序运行发生数据流错误。本章针对这类错误，提出了软、硬件结合的可配置容错技术 Epipe。Epipe 技术通过对现有的超标

量流水线进行简单扩展，提供了一个可配置的容错平台，并通过软件层面的指令分析技术评估、选择最重要的部分指令进行保护，从而达到了以较少的性能开销获取最大可靠性的目的。评估结果显示，Epipe 技术在性能开销分别为 11.6%、23.2% 和 34.8% 的配置下，故障覆盖率分别达到了 82%、92.6% 和 96.8%。

第三章　可配置的控制流检测技术

3.1　引言

处理器中的瞬时故障不仅会引发程序运行发生数据流错误，还很有可能导致程序运行出现控制流错误。文献 [107, 108] 的实验结果表明，控制流错误占瞬时故障引发错误的 33%—77%。出现控制流错误的原因在于，发生在程序计数器(PC)、总线、地址计算单元等位置的瞬时故障导致指令的地址错误，或者破坏指令操作码，将非跳转指令修改为跳转指令或者将跳转指令修改为非跳转指令。计算机程序运行是构成程序的指令序列按照预定顺序执行的过程，其中正确的执行顺序即控制流是程序正确执行的基础，如果控制流出现错误，则程序很可能得到错误的结果 [7]。由于已有的数据流错误检测技术很难处理 PC、地址计算单元等处的故障，容错计算机系统还必须另外提供对控制流错误的检测能力。

现有的针对控制流错误的检测技术按实现可以分为硬件控制流检测技术和软件控制流检测技术。硬件检测技术主要是通过设置专门的 Watchdog 协处理器实现 [109–111]。Watchdog 协处理器可以对主处理器产生的总线事务进行监测，以监视程序的指令流是否正确。这种方法对程序性能影响较小，但是需要修改底层硬件，成本较高。另外，这种方法无法应用于具有 Cache 的系统，因为这种系统的地址总线在主处理器内部，处理器直接从 Cache 中取指令执行，使得协处理器无法监控主处理器执行的程序流。

软件检测技术大多是采用标签分析的方法，其基本原理是，在编译时为程序中每个基本块分配不同的静态标签，并插装标签运算和比较指令到基本块中，程序运行时插装的冗余指令将根据当前控制流计算出一个动态标签，并比较当前块的动态标签和静态标签是否一致来确定控制流是否出错。相比硬件方法而言，软件检测方法不需要修改硬件设计，同时又可以提供较高的故障覆盖率，因此成为控制流错误检测的理想方法。但是，软件检测方法引入的额外指令很容易给程序运行带来大量的性能和存储开销。由于不同的应用对这些开销的容忍度存在差异(例如，嵌入式系统对程序的时空开销有着严格的约束)，控制流检测技术必须具备良好的可配置性，以满足用户不同的需求。此外，软件检测技术引入的冗余代码自身也有可能发生错误，影响系统的可靠性，因此软件检测技术还应具有自我保护能力。遗憾的是，现有的软件检测技术在可配置性和自容错保护方面还缺乏研究。

为了克服已有控制流检测技术的上述不足，本章提出了一种基于对等标签的控制流检测算法 CFCES （Control-flow Checking based on Equality Signature)。

CFCES 可以较少的开销有效地克服已有算法存在的检测盲点，并且首次具备了良好的可配置性和自容错能力。实现检测机制自我保护的难点在于，在对检测机制提供有效保护的同时应当避免额外产生大量的开销。为此，CFCES 在检测机制设计时引入了一种定义为“对等性”的不变量，通过对这种不变量进行检测，CFCES 以极低的代价实现了检错机制的自容错保护。此外，CFCES 还提供了可配置的优化方法。基于函数重要性分析和新的程序块划分方法，该优化方法可以灵活配置不同的保护强度，以满足用户不同的时空开销和可靠性约束。该优化方法的特点在于其可以提高 CFCES 的容错效率，且可以用于优化其它基于标签分析的控制流检测算法。

本章后续内容组织如下：3.2节对相关的软件实现的控制流错误检测方法进行概括分析；3.3节提出 CFCES 的基本检测算法，包括标签设计、不同类型控制流错误的处理、自容错保护机制等；3.4节基于定理证明的方法对 CFCES 的检错能力进行分析；CFCES 的可配置优化方法则在第3.5节给出；最后，3.6 节和3.7节分别通过故障注入模拟实验对 CFCES 的基本检测算法和配置优化方法进行有效性评估；3.8节对本章内容进行总结。

3.2 相关工作

软件实现的控制流检测方法相比硬件检测方法具有明显的成本优势，且可以获得较高的故障覆盖率。因此，软件检测方法的研究受到了广泛的关注，并已经取得不少研究成果。如1.3节内容所述，现有的软件检测技术在时空开销和可靠性方面各有不足，并且不具备可配置性和自容错保护能力。在已有的控制流检测算法中，CEDA 算法 [91] 被认为具有最高的容错效率。CEDA 为每个基本块分配一个入口标签和一个出口标签，并在基本块中插装两条指令（插装的指令如图3.1(a)所示），分别用以将动态标签更新为所在基本块的入口或出口标签。通过插装额外的标签比较指令，可以随时检测控制流错误。如图3.1(b) 所示，为了克服多扇入基本块（例如 B2）的别名问题，CEDA 在设计标签时为其前驱基本块设计部分位相同的出口标签，并在 B2 入口处插装按位与逻辑操作，取出前驱块出口标签的相同部分作为 B2 的入口标签。这种设计确保了非法控制流跳转 B1→B4 不会被漏检。CEDA 算法可以以较低的性能开销检测全部基本块间的控制流错误。但是，CEDA 算法没有解决基本块内和过程间的控制流错误检测问题，例如，如果发生如图3.1(b) 中虚线所示的基本块内控制流错误，CEDA 将无法检出。此外，和其它已有算法一样，CEDA 也不具备可配置性和自容错保护能力。本章提出的 CFCES 算法和 CEDA 在基本块间控制流错误检测方面有类似之处，但是 CFCES 对基本块内和过程间控制流错误检测、检测机制的自我保护、配置优化等问题进

行了专门研究，有效克服了 CEDA 的上述不足，实现了更为高效、实用的控制流检测。

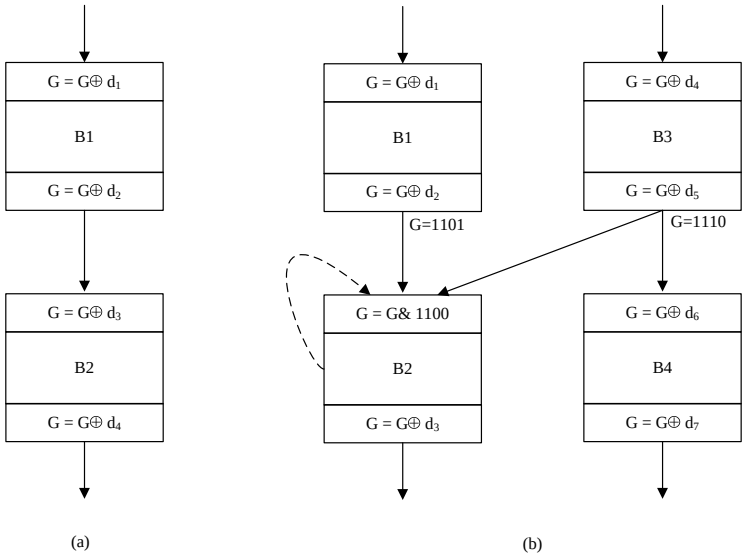


图 3.1　CEDA 算法示例

3.3　CFCES 的基本检测算法

为了实现高效的控制流保护，本章提出了一种基于对等标签的可配置控制流检测算法 CFCES。本节先介绍 CFCES 的基本检测方法，其优化配置方法在下一节提出。本节内容组织如下：先在3.3.1节给出控制流检测的相关定义，然后在3.3.2节介绍对等标签的设计方法，随后，3.3.3节详细介绍 CFCES 的基本块间和过程间控制流错误检测机制，3.3.4节阐述 CFCES 检测基本块内控制流错误的机制，3.3.5节对 CFCES 的自容错保护方法进行分析介绍。最后，由于用以检测错误的比较指令对性能影响较大，3.3.6节提出检测点优化的方法。

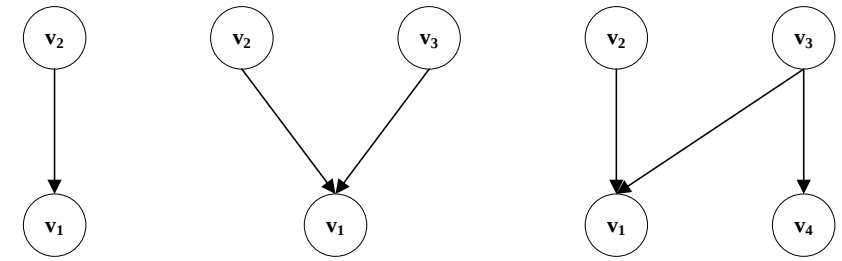
3.3.1　相关定义

CFCES 算法首先需要将程序汇编代码划分成基本块，并构造程序的全局控制流图。相关的概念定义如下：

定义 3.1 基本块（Basic Block）：基本块是程序中一个能够顺序执行的最大指令序列，这组指令只有一个入口和一个出口，入口就是第一条指令，出口就是最后一条指令。基本块中除了最后一条指令可以是转移指令外，其它指令都不能是转移指令。转移指令包括直接跳转、条件跳转和过程调用等。

定义 3.2 全局控制流图（Global Control-flow Graph, GCFG）：程序的全局控制流图可以用一个二元组 (V，E) 表示：

- $V=\{v_1, v_2, \ldots, v_n\}$ 为 *GCFG* 的节点集合，每个节点 v_i 对应程序的一个基本块 B_i（为方便描述，下面的内容中对基本块和其对应的节点这两种概念不再区分）；
- $E=\{e_1, e_2, \ldots, e_m\}$ 为 *GCFG* 的边集，如果 $v_i \in V$，$v_j \in V$ 且 v_i 到 v_j 存在控制流转移关系，则有一条有向边 $<v_i, v_j> \in E$。



(a) I型节点　　(b) Y型节点　　(c) N型节点

图 3.2　GCFG 中各种类型的节点示例（以 v_1 为例）

GCFG 中也包含过程调用的控制流转移关系，即过程调用发生的基本块（简称函数调用块）到被调用过程的入口基本块之间存在一条有向边，被调用过程的出口基本块到函数调用块的下一个块之间也有一条有向边。基于程序的全局控制流图，又有如下定义：

定义 3.3 前驱：定义 *GCFG* 中节点 v_i 的前驱节点集合为 $pre(v_i)$。对于 $\forall v_j$，如果 $<v_j, v_i> \in E$ 则 $v_j \in pre(v_i)$。

定义 3.4 后继：定义 *GCFG* 中节点 v_i 的后继节点集合为 $suc(v_i)$。对于 $\forall v_j$，如果 $<v_i, v_j> \in E$ 则 $v_j \in suc(v_i)$。

定义 3.5 I 型节点：如果节点 v_i 在 *GCFG* 中只有一个前驱节点，则称 v_i 为 I 型节点。此外，程序入口节点也定义为 I 型节点。I 型节点集合定义为 *Iset*。I 型节点的示例如图3.2(a) 所示。

定义 3.6 Y 型节点：如果节点 v_i 在 *GCFG* 中有多于一个前驱节点，且 $\forall v_j, v_k \in pre(v_i)$ 满足 $suv(v_j) = suc(v_k)$，则称 v_i 为 Y 型节点。Y 型节点集合定义为 *Yset*。Y 型节点的示例如图3.2(b) 所示。

定义 3.7 N 型节点：如果节点 v_i 在 *GCFG* 中有多于一个前驱节点，且 $\exists v_j, v_k \in pre(v_i)$ 满足 $suv(v_j) \neq suc(v_k)$，则称 v_i 为 N 型节点。N 型节点集合定义为 *Nset*。N 型节点的示例如图3.2(c) 所示。

按照节点间的控制流关系，*GCFG* 中所有节点可以分为上述三种类型。基于 *GCFG* 的定义，从一条指令 I_s 到另外一条指令 I_d 的合法控制流转移 $I_s \rightarrow I_d$ 应该满足如下条件：

- 若 I_s 不是所属基本块 B_i 的最后一条指令，则 I_d 应为 I_s 在 B_i 内的下一条指令；
- 若 I_s 是所属基本块 B_i 内最后一条指令，则 I_d 应为 $GCFG$ 中 B_i 的一个后继基本块的第一条指令。

何违反上述条件的控制流转移都可以认为是非法控制流，当然这里的非法是指语法层面的。按照跳转的跨度范围，可以将程序中的控制流错误分为三种类型：

定义 3.8 基本块间控制流错误（Inter-block Control-flow Error）：如果非法控制流从 $GCFG$ 中同一个过程内部的一个节点跳到另外一个节点，则称其为基本块间控制流错误。

定义 3.9 基本块内控制流错误（Inter-block Control-flow Error）：如果非法控制流从基本块中一条指令跳到同一基本块中的另外一条指令，则称其为基本块内控制流错误。

定义 3.10 过程间控制流错误（Inter-procedure Control-flow Error）：如果非法控制流从 $GCFG$ 中一个过程跳到另外一个过程内，则称其为过程间控制流错误。

3.3.2 标签分配

CFCES 算法需要为每个基本块分配一个入口标签（SY）和一个出口标签（ST）。假设系统机器字长为 $2m$ 位，则入口标签和出口标签的二进制位宽均为 m。入口标签的分配规则如下：

规则 3.1：若 $v_i \in Nset$，则为 v_i 分配一个唯一的标签值作为 $SY(v_i)$，$SY(v_i)$ 的后两位的值必须为 0。

规则 3.2：若 $v_i \in Yset \vee v_i \in Iset$，则 $SY(v_i)$ 需具有唯一性，即 $SY(v_i) \neq ST(v_i)$ 且 $\forall v_j \in V(j \neq i)$，$SY(v_i) \neq SY(v_j) \wedge SY(v_i) \neq ST(v_j)$。$SY(v_i)$ 还需要满足：$\forall v_n \in Nset$，$D_{m-2}(SY(v_i)) \neq D_{m-2}(SY(v_n))$，其中 $D_{m-2}(SY(v_i))$ 表示 $SY(v_i)$ 的前 $(m-2)$ 位的值。

每个基本块还要分配一个出口标签 ST。出口标签的分配需要先后应用下列规则实现：

规则 3.3：若 $v_i \in Nset$ 满足 $|pre(v_i)| = 2$（$|pre(v_i)|$ 表示 v_i 的前驱节点数），且对于 $\forall v_j \in Nset$，$pre(v_i) \cap pre(v_j) = \emptyset$，则 v_i 的前驱节点将分别分配如图3.3(a) 和图3.3(b) 所示的两个标签。其中，两个标签的前 $(m-2)$ 位值相等，记为 $D_{m-2}(SY(v_i))$。

规则 3.4：若 $v_i \in Yset$，其所有前驱节点共用一个标签值 S，即对于 $\forall v_j \in pre(v_i)$，$ST(v_j) = S$。S 需要满足以下条件：（1）$\forall v_n \in V$，$SY(v_n) \neq S$；（2）$\forall v_k \notin pre(v_i)$，$ST(v_k) \neq S$；（3）$\forall v_l \in Nset$，$D_{m-2}(S) \neq D_{m-2}(SY(v_l))$。

规则 3.5：若 $v_i \in Iset$ 或 v_i 为程序出口节点，则其前驱节点 v_j 的出口标签 $ST(v_j)$ 应满足以下条件：（1）$ST(v_j) \neq SY(v_j)$；（2）$\forall v_k \in V(k \neq j)$，$ST(v_j) \neq ST(v_k) \wedge ST(v_j) \neq SY(v_k)$；（3）$\forall v_n \in Nset$，$D_{m-2}(ST(v_j)) \neq D_{m-2}(SY(v_n))$。

(a) (b)

图 3.3　为 N 型节点 v_i 的前驱块分配的出口标签

上述规则中，规则 3.3 假设 N 型节点 v_i 的前驱只有两个且对于 $\forall v_j \in Nset$，$pre(v_i) \wedge pre(v_k) = \emptyset$。这种情形的典型例子如图3.2(c) 所示，按照规则 3.3，节点 v_2 和 v_3 将会分配前 $(m-2)$ 位值相等但最后两位不等的出口标签。但是，并非所有 N 型节点都符合应用规则 3.3 的条件，例如图3.4(a) 中，N 型节点 v_1 因为 $|pre(v_i)| > 2$ 无法直接应用规则 3.3，N 型节点 v_2 因和 v_1 有共同的前驱也无法直接应用规则 3.3。对于这类 N 型节点，算法需要先通过预处理使其满足规则 3.3 的条件，再根据规则进行标签分配。为 N 型节点分配出口标签的具体步骤为：

Step 1：基于程序的 $GCFG$ 构造别名冲突图（Alias Interference Graph，AIG）。$AIG(V', E')$ 是一个无向图，其节点集合 $V'=\{v_i|\exists v_j \in Nset$，$v_i \in pre(v_j)\}$，边集 $E'=\{< v_i, v_k >|\exists v_j \in Nset$，$v_i \in pre(v_j) \wedge v_k \in pre(v_j) \wedge (|suc(v_i)| \neq 1 \vee |suc(v_k)| \neq 1)\}$。图3.4(b) 给出了图3.4(a) 中 $GCFG$ 对应的 AIG。

Step 2：$\forall v_i, v_j \in V'$，如果满足 $suc(v_i) = suc(v_j) \wedge |suc(v_i)| = 1$，则将 v_i 和 v_j 合并为一个新节点，并将 AIG 中连接到 v_i 和 v_j 的边修改为连接到新节点（如图3.4中 Iteration1-Step2 所示）。节点合并后，判断 AIG 中是否存在节点 v_i 满足 degree(v_i)>1，其中 degree(v_i) 表示 v_i 在 AIG 中的度数，如果存在则选择和 v_i 邻接的一个节点 v_j，并选择 $v_k \in suc(v_i) \cap suc(v_j)$，然后在 $GCFG$ 中插入一个虚拟基本块 vb 到 v_i 和 v_k 之间，最后重新转到 Step 1 对新的 $GCFG$ 进行迭代处理（如图3.4中 Iteration2-Step1 所示）。

虚拟基本块是一个空块，引入虚拟基本块的目的是改变程序的控制流结构，使 N 型标签满足规则 3.3 要求的条件。当上述迭代过程进行到 AIG 中不存在节点 v_i 满足 $degree(v_i) > 1$ 时则说明预处理过程完成（如图3.4中 Iteration2-Step2 所示），此时将 AIG 中度数为 1 的节点和其邻接的节点按照规则 3.3 进行标签分配即可，其中合并得到的新节点的标签即为其每个组成节点的标签。

上述处理过程中部分 N 型节点由于控制流结构变化而变为 Y 型或 I 型节点，最终的 N 型节点均可以按照规则 3.3 为其前驱分配出口标签，其余节点均可利用规则 3.4、3.5 进行标签分配。基于图3.4(a) 中 $GCFG$ 进行出口标签分配获得的最

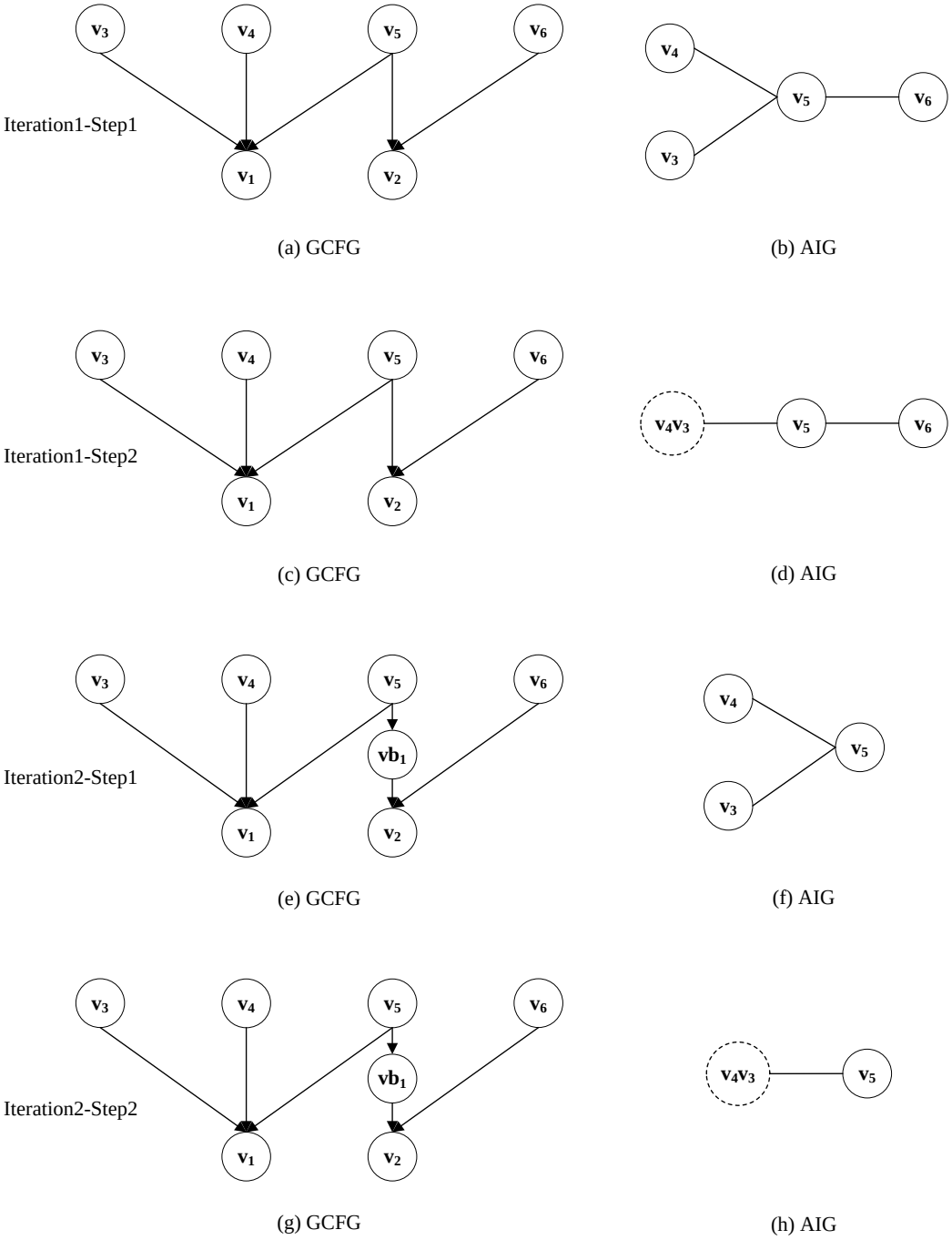


图 3.4　N 型节点预处理过程

终结果如图3.5所示（标签采用二进制码表示）。*GCFG* 中新引入的虚拟基本块也和其它标签一样分配标签并插装控制流检测指令，并且可以证明如下结论成立：

定理 3.1：插装虚拟基本块前后的程序在语义上是等价的。

证明：假设一个虚拟基本块 vb_i 被插到节点 v_j 和 v_k 之间，则原 *GCFG* 中的边 $v_j \rightarrow v_k$ 将会被两条边 $v_j \rightarrow vb_i$ 和 $vb_i \rightarrow v_k$ 取代，而 E 中其它边均不变，因此，插入 vb_i 只会影响 v_j 和 v_k 之间的控制流，程序其它部分的控制流和语义均不受影

响。而对于 v_j 和 v_k，由于插入的虚拟基本块中只包含控制流检测引入的冗余指令，不含改变源程序语义的操作，所以程序从节点 v_j 执行到 v_k 的过程中，程序的语义也不受 vb_i 影响。综上，插装虚拟基本块前后的程序在语义上是等价的。

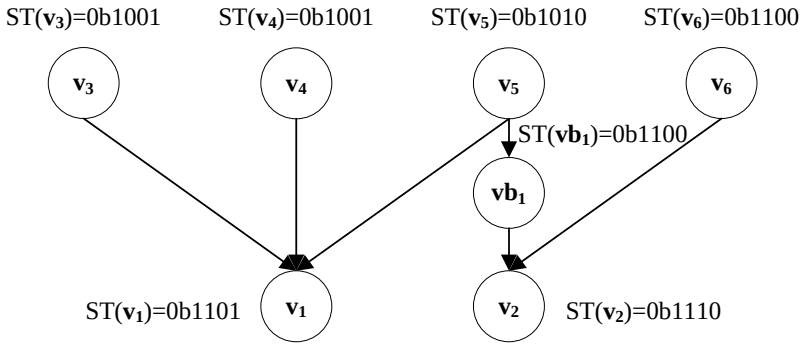


图 3.5　出口标签分配结果

(a) 对等入口标签　　(b) 对等出口标签

图 3.6　为节点 v_i 分配的对等标签

在分配的入口标签和出口标签基础上，CFCES 算法需要基于已分配的标签扩展得到相应的对等标签，分别称为对等入口标签（ESY）和对等出口标签（EST）。对于一个已分配的入口或出口标签 S，其对等标签为 $2m$ 位，且对等标签的高 m 位和低 m 位均为 S。图3.6给出了为节点 v_i 分配的对等标签格式。此外，CFCES 算法还需要计算出当前节点的 ESY 和 EST 标签差异值 d_{yt}，以及当前节点的 ESY 标签和其前驱节点的 EST 标签的差异值 d_{ty}。节点 v_i 的 d_{yt} 计算公式为：

$$d_{yt}(v_i) = ESY(v_i) \oplus EST(v_i) \tag{3.1}$$

当 $v_i \in Yset \vee v_i \in Iset$ 且 $v_j \in pre(v_i)$ 时，节点 v_i 的 d_{ty} 计算公式为：

$$d_{ty}(v_i) = EST(v_j) \oplus ESY(v_i) \tag{3.2}$$

由于 N 型节点的前驱节点的 EST 标签不同，不能为 N 型节点计算差异值 d_{ty}。N 型节点被分配一个掩码标签 M，M 的二进制值为 $0b\underbrace{11\ldots11}_{m-2}00\underbrace{11\ldots11}_{m-2}00$。

3.3.3 基本块间和过程间控制流错误检测

基于分配的标签，CFCES 通过在基本块中插入标签运算和比较指令，实现控制流的动态监控和检测。CFCES 的块间控制流检测机制类似于 CEDA 算法 [91]，但是由于在 *GCFG* 中基本块间和过程间控制流错误均表现为图中两个块间的跳转错误，所以 CFCES 基于 *GCFG* 设计的块间检测机制不仅可以检测基本块间跳转错误，还可以检测过程间跳转错误。

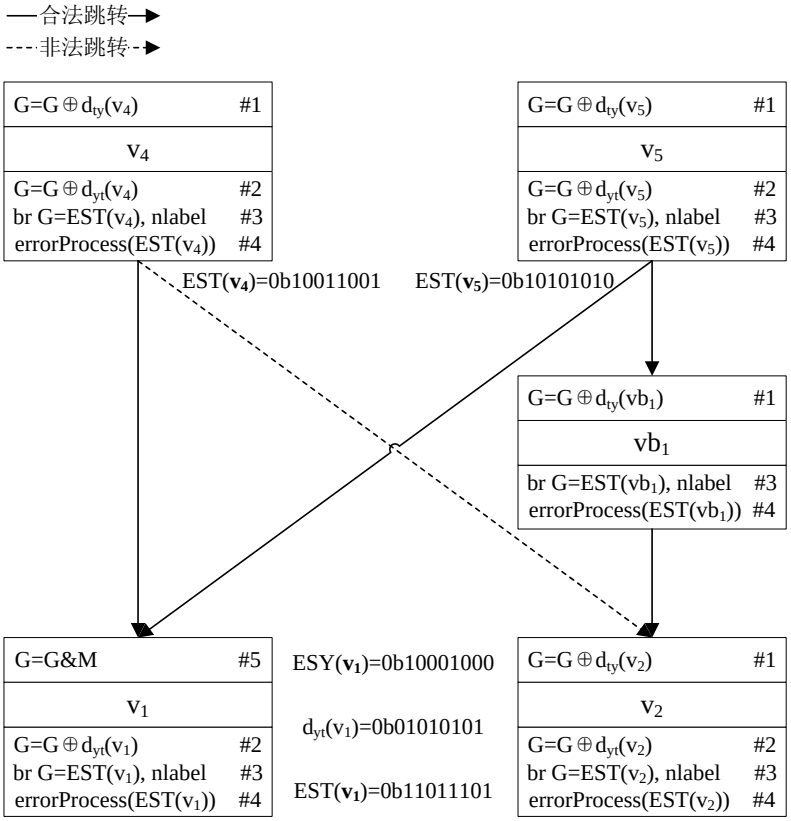


图 3.7 基本块间控制流错误检测示例

图3.7给出了图3.5中部分节点插装额外的指令实现基本块间控制流检测的例子。如图3.7所示，CFCES 在基本块入口和出口处共插装 5 种类型的指令（#1—#5），实现基本块间（过程间）控制流错误的检测。不同类型的节点插装指令的类型和数量是不同的。其中，v_2、v_4 和 v_5 中的指令代表了 I 型或 Y 型节点需要插装的指令，v_1 中的指令代表了 N 型节点需要插装的指令，vb_1 中的指令则代表了虚拟基本块需要插装的指令。算法用一个通用寄存器 *G* 专门存放节点的动态标签。程序运行进入一个节点 v_i 之前，*G* 中存放的是 v_i 的前驱节点的 EST 标签。进入节点 v_i 之后，指令 #1（异或操作）或 #5 （按位与操作）可以将 *G* 更新为 v_i 的 ESY 标签。当程序运行至 v_i 的出口处时，指令 #2 将 *G* 更新为 v_i 的 EST 标签。指令 #3 比较 *G* 中标签是否和预存的 v_i 的 EST 标签相同，如果相同则说明程序控制流正

常，跳转到 nextlabel 处继续正常的程序执行，否则说明程序控制流存在错误，需要转入错误处理例程（#4）。需要特殊说明的是，由于虚拟基本块中内部没有需要保护的指令，没有必要像其它基本块一样同时在入口和出口处进行标签校验，因此不需要插入指令 #2。

以图中节点 v_4 为例，如果 v_4 沿合法分支跳转到节点 v_1，显然在执行到 v_1 中的指令 #3 处时，G 的值等于 $EST(v_1)$，可以确认程序控制流正常。如果 v_4 沿虚线所示的非法分支跳转到节点 v_2，执行 v_2 中的指令 #1 后，$G = EST(v_4) \oplus EST(vb_1) \oplus ESY(v_2)$，由于 $EST(v_4) \neq EST(vb_1)$，所以 $G \neq ESY(v_2)$。接着执行完 v_2 中的指令 #2 后，$G = G \oplus ESY(v_2) \oplus EST(v_2)$，显然 $G \neq EST(v_2)$，经指令 #3 比较可以检测出该控制流错误。

3.3.4 基本块内的控制流错误检测

除了基本块间和过程间控制流错误，程序也有可能发生基本块内控制流错误。块内控制流错误检测的难点在于如何和块间的控制流校验机制结合，以最少的额外开销实现块内控制流错误的检测。CFCES 算法在块间控制流检测机制的基础上，通过进一步分配标签和插装指令实现了块内控制流错误的检测。

首先，CFCES 算法需要为每个基本块额外分配一些块内标签（SI）。用 $SI_j(v_i)$ 表示节点 v_i 的第 j 个块内标签。$SI_j(v_i)$ 为 m 位标签，且具有唯一性，即 $SI_j(v_i)$ 不等于其它块内标签、入口标签和出口标签。而且，$SI_j(v_i)$ 需满足：$\forall v_n \in Nset$，$D_{m-2}(SI_j(v_i)) \neq D_{m-2}(SY(v_n))$。进一步地，将块内标签对等化，得到 $2m$ 位的对等块内标签 (ESI)。此外，也要计算出同一个节点的序号相邻的两个 ESI 标签的差异值。假设节点 v_i 有 n 个对等块内标签，当 $0 < j < n$ 时，差异值 d 的计算公式为：

$$d_{j,j+1}(v_i) = ESI_j(v_i) \oplus ESI_{j+1}(v_i) \tag{3.3}$$

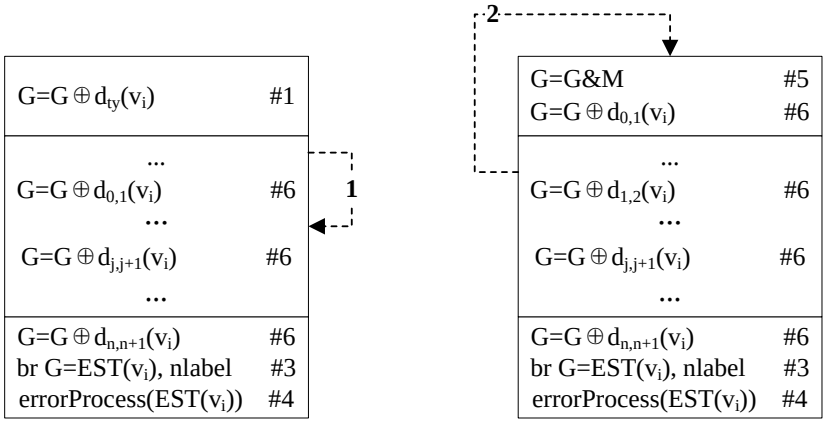
当 j=0 或 n 时，$d_{j,j+1}(v_i)$ 分别表示第 1 个 ESI 标签和 ESY 标签的差异值、第 n 个 ESI 标签和 EST 标签的差异值，即：

$$d_{0,1}(v_i) = ESY(v_i) \oplus ESI_1(v_i) \tag{3.4}$$

$$d_{n,n+1}(v_i) = ESI_n(v_i) \oplus EST(v_i) \tag{3.5}$$

如图3.8所示，基于新分配的 ESI 标签，算法在基本块内部等间距插入 n 条指令 #6 实现块内控制流错误的检测。块内插入的 #6 指令可以将基本块分割成若干子区域（Basic Region，BR），每个 BR 块对应一个 ESI 标签和一条 #6 指令。指令 #6 可以将 G 中标签从前一个 BR 块的 ESI 标签更新为当前 BR 块的 ESI 标签。程序控制流正常的情况下，执行完所有的 #6 指令后，

$G = G \oplus d_{0,1}(v_i) \oplus d_{1,2}(v_i) \ldots \oplus d_{n,n+1}(v_i) = G \oplus d_{yt}(v_i) = EST(v_i) \oplus d_{yt}(v_i) = EST(v_i)$。如果发生了跨越 BR 块的基本块内控制流错误，使得部分 #6 指令的执行受到影响（如图中虚线 1 所示），则执行比较指令 #3 时 $G \neq EST(v_i)$，因此可以判断出控制流错误。如图3.8所示，和 I/Y 型节点不同，N 型节点需要在基本块入口（#5 后）插入一条指令 #6，原因是指令 #5 具有可重复执行性，如果不在其后插入 #6，将会漏检虚线 2 所示的非法跳转。此外，根据基本块的大小，不同基本块可以设定不同的 n 值，以平衡块内控制流检测能力和时空开销。



(a) I/Y型节点　　(b) N型节点

图 3.8　基本块内控制流错误检测机制

3.3.5　检测机制的自容错保护

软件检测算法插装的指令会增加程序运行的时空开销。假设瞬时故障的发生概率服从时间和空间的均匀分布 [45, 46]，则故障有相当可能性会发生在插装的冗余检测代码中。发生在检测机制中的故障可能导致控制流错误，也可能破坏标签数据，引发数据流错误。前者会破坏程序的正常运行，后者则一般只会破坏检测机制，而不影响程序原有代码的执行。检测机制的可靠性包含两个方面的要求：(1) 若检测机制中的错误会破坏程序原有代码的执行，即检测机制出现了控制流错误，则应当检测出这种错误，以便对程序进行恢复处理；(2) 若检测机制中的错误不影响程序原有代码的执行，即只是发生了数据流错误，则应当避免报告这种不影响程序结果的错误（即“假死”错误），以免带来不必要的程序恢复开销。由于3.4节的内容可以证明 CFCES 的检测机制内发生控制流错误后，算法均能够成功检出错误，所以本节关注的目标是如何处理检测机制中的数据流错误。

由于上述两种错误都会导致动态标签和预存的静态标签不匹配，所以控制流检测算法会把这两种错误均作为“控制流错误”检出，并对程序进行错误恢复处

理。本节的难点在于如何从检出的“控制流错误”中识别出数据流错误，以及如何针对检测机制中的数据流错误进行专门的低成本恢复处理。和已有技术一样[85, 86]，CFCES 算法假设的故障模型是单粒子翻转模型（SEU），即一次程序运行过程中只发生一次翻转错误。基于上述假设，CFCES 利用设计的对等标签和插装的对等操作实现数据流错误的判别和快速恢复。对等操作的定义为：

定义 3.11 对等操作：如果一条运算指令的所有源操作数均是对等的，则其结果（即目的操作数）也必定是对等的，称这样的运算操作为对等操作。

定理 3.2：CFCES 插装的运算指令均为对等操作。

证明：如图3.7和3.8所示，CFCES 插装的标签运算指令分为两类：（1）异或指令：包括 #1、#2 和 #6；（2）按位与指令：指令 #5。由于两类指令都为按位逻辑操作，如果其源操作数均是对等的，则结果也必定是对等的。因此 CFCES 插装的运算指令均是对等操作。

定理 3.3：存储在 G 中的动态标签在控制流正常的情况下一直是对等的。

证明：假设程序控制流正常情况下算法插装的运算指令执行的顺序为 I_0，I_1，$I_2 \ldots I_n$，可以用归纳法证明此序列中任意一条指令 I_k 计算出的动态标签均满足对等性：

（1）k=0 时，I_0 为程序入口节点 v_1 中的标签运算指令 #1，基于初始值，G 将被更新为 $ESY(v_1)$，所以满足对等性。

（2）假设 k=j 时结论成立，即 I_j 计算得到的动态标签 G 是对等的，则 I_{j+1} 的一个源操作数 G 是对等的。I_{j+1} 的另一个操作数，在 I_{j+1} 的类型是 #1、#2、#6 和 #5 的情形下均是对等的。因为 I_{j+1} 是对等操作，所以 I_{j+1} 计算出的动态标签 G 也是对等的，即 $k = j + 1$ 时结论成立。

综上，控制流正常的情况下，存储在 G 中的动态标签一直是对等的。

定理 3.4：如果程序运行中发现动态标签是不对等的，则程序一定出现了数据流错误。

证明：要证该结论成立，只需证明程序发生控制流错误后，动态标签仍是对等的。证明如下：

非法跳转对标签运算指令的影响分为三种情况：（1）非法跳转发生在一个基本块内或 BR 块内，标签运算指令的执行不受影响；（2）非法跳转导致部分标签运算指令没能被执行；（3）非法跳转导致部分标签运算指令被多次执行。显然第一种情况下，动态标签不受影响，仍是对等的。第二、三种情形下，可以假设非法跳转 $I_s \rightarrow I_d$ 发生在两个标签运算指令之间。基于定理 3.3 可知 I_s 计算出的动态标签 G 是对等的，所以 I_d 的源操作数（包括 G 和 d、M 标签）均是对等的，因此 I_d 计算出的动态标签也是对等的。综上可知，程序发生控制流错误后，动态标签仍是对等的。原结论成立。

```
errorProcess:
1.  br G ≠ EST(v_i), templabel
2.  return
templabel:
3.  D=G ≻ m
4.  br G=D, errorRecovery
5.  G=EST(v_i)
6.  return
```

图 3.9　错误处理例程

由于检测机制中的数据流错误不会破坏程序原有的语义，基于定理 3.4 的结论，可以在发现动态标签不对等时直接恢复控制流检测机制的标签数据，然后继续运行程序。基于上述原理，CFCES 设计了如图3.9 所示的错误处理例程，以识别、恢复检测代码中的数据流错误。首先，指令 1 将 G 和 $EST(v_i)$ 再次比较。如果比较结果相等，则说明算法插装的指令 #3 因自身执行出错而误报了错误，此时直接返回报错的基本块往下执行即可（指令 2)。否则，利用指令 3、4 判断动态标签 G 是否具有对等性（指令 3 将 G 循环右移 m 位)，如果动态标签是对等的则说明程序存在控制流错误，需要对程序进行错误恢复，否则说明检测机制存在数据流错误，利用指令 5 将动态标签的值恢复，然后返回报错的基本块继续执行即可（指令 6)。

从设计的错误处理例程可以看出，当检测机制出现不影响被保护程序执行的数据流错误后，CFCES 可以以极低的代价判断这种错误并将标签恢复至正常状态，成功避免了检测机制出错后直接报告“假死”而导致的大量程序恢复开销。尽管采用传统的指令复算技术，例如 EDDI、ED^4I 等，也可以对 CFCES 的检测机制进行保护，但这样会使控制流检测的开销大幅增加。CFCES 的自容错保护机制在程序无故障运行状态下不会引入额外的性能开销，在检测机制发生数据流错误后则只需执行少量指令即可恢复。CFCES 能够以低代价实现自我保护的关键在于在检测机制中引入了对等性，这里的对等性实质上是一种不变量。利用这种不变量，可以方便地识别出检测机制中的数据流错误并进行处理。

3.3.6　检查点优化放置

CFCES 插装的指令中，检查点指令（即 #3 和 #4）对程序的性能开销影响最大。为了在不影响算法检测能力的前提下减少性能开销，本节提出了 CFCES 的检查点优化放置方法。

定义 3.12 数据依赖：如果在一条程序路径 p 上存在指令序列 $I_1 : varB_1 = op(varA)$，$I_2 : varB_2 = op(varB_1)$，$I_3 : varB_3 = op(varB_2) \ldots I_n : varB = op(varB_{n-1})$，

则称变量 *varB* 在 *p* 上依赖于 *varA*，并称 I_1，I_2，$I_3 \ldots I_n$ 为 *varA* 和 *varB* 之间的数据依赖链。

定义 3.13 检查点的覆盖：针对两个数据 *varA*、*varB* 的检查点 cmp(*varA*) 和 cmp(*varB*)，如果从 cmp(*varA*) 到 cmp(*varB*) 的任何路径上 *varB* 均依赖于 *varA*，则称 cmp(*varB*) 是 cmp(*varA*) 的一个覆盖。特别地，如果从 cmp(*varA*) 到 cmp(*varB*) 的任何路径上，*varA* 和 *varB* 之间的数据依赖链均不含具有故障屏蔽能力的指令，则称 cmp(*varB*) 是 cmp(*varA*) 的一个安全覆盖。

上述定义中，有故障屏蔽能力的指令 $varB_i = op(varB_j)$ 是指，当源操作数 $varB_j$ 中出现错误时，指令仍然可能获得正确的结果 $varB_i$。例如，CFCES 插装的按位与指令 #5 可以屏蔽源操作数 *G* 中第 0、1、*m* 和 $(m+1)$ 位的故障，原因是另外一个源操作数 *M* 的对应位为 0。显然，如果 cmp(*varB*) 是 cmp(*varA*) 的一个安全覆盖，则 cmp(*varA*) 能够检测的错误，cmp(*varB*) 也一定能够发现，因此将 cmp(*varA*) 从程序中删除不会影响算法的检测能力。

CFCES 算法的初始设计是通过在每个基本块中插装指令不断更新动态标签，并在基本块出口处插装检查点 cmp(*G*) 实现控制流错误的检测。由于算法插装的标签运算指令之间均存在依赖关系，任何一个基本块 v_i 及其检查点 $cmp_i(G)$ 满足：$\forall v_j \in pre(v_i)$，$cmp_i(G)$ 是 $cmp_j(G)$ 的覆盖。但是，$cmp_i(G)$ 不一定是 $cmp_j(G)$ 的安全覆盖，可以分两种情形讨论：

- 如果 v_i 是 I 型或 Y 型节点，因为 v_i 和 v_j 中的标签运算指令均为异或运算，不具备故障屏蔽能力，所以 $cmp_i(G)$ 一定是 $cmp_j(G)$ 的安全覆盖；
- 如果 v_i 是 N 型节点，因为 v_i 中的指令 #5 可能屏蔽标签中的错误，所以 $cmp_i(G)$ 不是 $cmp_j(G)$ 的安全覆盖。

基于上述结论，检查点优化方法可以将 I 型或 Y 型节点的前驱节点中的检查点删除，但是 N 型节点的前驱节点中的检查点则必须保留。此外，检查点优化过程中还应考虑一些其它需求，例如，I/O 操作前的检查点应该保留。根据故障恢复机制对故障检测延迟的约束，程序中每隔一定距离也要保留必要的检查点。

3.4 CFCES 检测能力证明

本节从安全性和完全性两个角度对 CFCES 的检测能力进行分析。安全性是指在程序运行没有发生故障的情况下，检测算法不会报告错误。完全性是指如果程序运行出现控制流错误，则检测算法一定能够检出错误。下面首先证明 CFCES 的安全性。

定理 3.5：如果程序运行中不存在错误，则算法不会报告错误。

证明：欲证该结论成立，只需证明在每个基本块中，每个标签运算指令（指令 #1、#2、#5 和 #6）执行完成后，*G* 的值都等于标签运算指令的预期结果（指

令 #1、#2、#5 和 #6 的预期执行结果分别为所在基本块的 ESY 标签、EST 标签、ESY 标签和 ESI 标签)。下面用归纳法证明:

(1) 在程序的入口基本块 v_1 中，指令 #1: $G = G \oplus d_{ty}(v_1)$ 可以基于初始值 (G=0，$d_{ty}(v_1) = ESY(v_1)$) 将 G 更新为 v_1 的 ESY 标签，接下来执行的指令 #2: $G = G \oplus d_{yt}(v_1) = G \oplus ESY(v_1) \oplus EST(v_1) = EST(v_1)$ 将 G 更新为 v_1 的 EST 标签。如果 v_1 中插入了块内标签运算指令 #6: $G = G \oplus d_{j,j+1}(v_i)$，则同样可以将 G 更新为相应的 ESI 标签。综上分析，在入口基本块 v_1 中，每个标签运算指令执行完成后，G 的值都等于预期的结果。

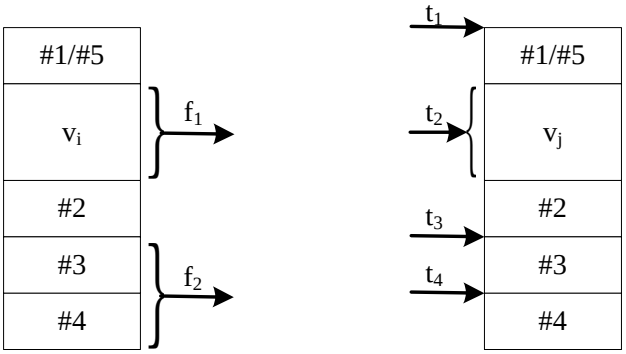
(2) 假设在基本块 v_i 中结论成立，则 v_i 的出口处，G 的值等于 v_i 的 EST 标签。对于 $\forall v_j \in suc(v_i)$，可以分为以下两种情况讨论:

- 若 v_j 是 I 型或 Y 型节点，则执行其中的指令 #1 后，$G = G \oplus d_{ty}(v_j) = EST(v_i) \oplus EST(v_i) \oplus ESY(v_j) = ESY(v_j)$，即 #1 可以将动态标签更新为期待的 ESY 标签。基于 G 中的 ESY 标签，v_j 中后续执行的指令 #2: $G = G \oplus d_{yt}(v_j) = G \oplus ESY(v_j) \oplus EST(v_j) = EST(v_j)$，可以将动态标签更新为预期的 EST 标签。如果 v_j 中插入了块内标签运算指令 #6: $G = G \oplus d_{j,j+1}(v_j)$，则同样可以将 G 更新为期待的 ESI 标签。
- 若 v_j 是 N 型节点，则执行 v_j 中第一条指令 #5 后，$G = G\&M = EST(v_i)\ \&0b\underbrace{11\ldots11}_{m-2}00\underbrace{11\ldots11}_{m-2}00$，由于 v_j 的入口标签的前 $(m-2)$ 位和 v_i 的出口标签的前 $(m-2)$ 位相同，且后两位为 0，所以上述运算的最终结果为 $G = ESY(v_j)$，同样地，v_j 中后续执行的指令 #6 或 #2 可以将动态标签更新为预期的标签。

所以，假设在基本块 v_i 中结论成立，则 v_i 的所有后继基本块也满足该结论。综合 (1)(2) 可知，程序无故障运行的情况下，在每个基本块中，每个标签运算指令执行完成后，G 的值都等于标签运算指令的预期结果，插装的检查点不会报告错误。

算法的完全性包含两个方面:(1) 基本块 (过程) 间控制流错误的完全检测;(2) 基本块内控制流错误的完全检测。需要特别指出的是，CFCES 假设每次程序运行最多只发生一次控制流错误，并且算法只将违背程序 $GCFG$ 的跳转错误视作基本块 (过程) 间的控制流错误，而不关注“if-then-else”分支问题 (即程序在 then 和 else 之间选择了错误的分支方向)。后者实质上是由于程序原有代码中的数据流错误引起的，可以假设源程序中已经存在数据流检测机制处理这种问题。如图3.10 所示，基本块间控制流检测机制在每个块中插装 4 条指令，这些指令将基本块分割成不同的区域，假设在基本块 v_i 和 v_j 之间存在一个控制流错误，则该非法跳转的起始位置可以分为两种情况：f_1 和 f_2。其中，f_1 表示在 #1 或 #5

之后、#2 之前的位置，f_2 表示在 #2 之后的位置。非法跳转在 v_j 中的终止位置可以分为 t_1、t_2、t_3、t_4 四种情形，其中 t_1 表示跳转到 v_j 的入口，t_2 表示跳转到 #1 或 #5 之后、#2 之前的位置，t_3 表示跳转到 #3，t_4 表示跳转到 #4。下面先基于不同的非法跳转起始位置，分步证明算法在块间控制流检测方面的完全性。



(a)非法跳转的起始位置　　(b)非法跳转的终止位置

图 3.10　块间非法跳转的起始和终止位置

定理 3.6：从 f_1 起始的非法块间跳转都可以被算法检测。

证明：假设非法跳转发生在基本块 v_i 和 v_j 之间。从 f_1 跳转之前，$G = ESY(v_i)$。下面根据不同的终止位置分别进行证明：

（1）首先考虑非法跳转为 $f_1 \rightarrow t_1$ 的情况：

- 如果 v_j 是 I 型或 Y 型节点，v_j 的一个前驱节点是 v_k，则执行 v_j 中的指令 #1 后，$G = G \oplus d_{ty}(v_j) = ESY(v_i) \oplus EST(v_k) \oplus ESY(v_j)$，根据标签设计规则，$ESY(v_i) \neq EST(v_k)$，所以 $G \neq ESY(v_j)$，继续执行 v_j 中的指令 #2 后，$G = G \oplus d_{yt}(v_j) = G \oplus ESY(v_j) \oplus EST(v_j)$，由于输入 $G \neq ESY(v_j)$，所以执行完 #2 后 $G \neq EST(v_{j)}$，最终指令 #3 可以通过比较 G 和 $EST(v_j)$ 检测出该非法跳转。
- 如果 v_j 是 N 型节点，则执行 v_j 中第一条指令 #5 后，$G = G\&M = ESY(v_i)\&0b\underbrace{11\ldots11}_{m-2}00\underbrace{11\ldots11}_{m-2}00$，根据标签设计规则，$D_{m-2}(SY(v_i)) \neq D_{m-2}(SY(v_j))$，所以执行 #5 后 $G \neq ESY(v_j)$。同样地，继续执行 #2 后 $G \neq EST(v_j)$，最终指令 #3 可以检测出该非法跳转。

（2）当非法跳转为 $f_1 \rightarrow t_2$ 时，执行 v_j 中的指令 #2 后，$G = G \oplus dyt(v_j) = ESY(v_i) \oplus ESY(v_j) \oplus EST(v_j)$，根据 ESY 标签的唯一性规则，$ESY(v_i) \neq ESY(v_j)$，所以 $G \neq EST(v_j)$，最终指令 #3 可以检测出该非法跳转。

（3）当非法跳转为 $f_1 \rightarrow t_3$ 时，跳转后程序执行指令 #3：$brG = EST(v_j), nlabel$，由于此时 $G \neq EST(v_j)$，所以可以比较判断出此非法跳转。

（4）当非法跳转为 $f_1 \rightarrow t_4$ 时，跳转后程序直接转入如图3.9所示的错误处理例程。由于错误处理例程会再次执行和 #3 相同的操作，所以同情形（3）一样，可以检测出该控制流错误。

综合以上证明结果可知，从 f_1 起始的非法块间跳转都可以被算法检测。

定理 3.7：从 f_2 起始的非法块间跳转都可以被算法检测。

证明：假设非法跳转发生在基本块 v_i 和 v_j 之间。非法跳转开始之前，$G = EST(v_i)$。下面根据不同的终止位置分别进行证明：

（1）当跳转为 $f_2 \rightarrow t_1$ 时，如果 $v_j \in suc(v_i)$，则此跳转语义上等价于从 v_i 出口跳转至 v_j 入口，因此并不违反程序的 $GCFG$，可以视为合法跳转。如果 $v_j \notin suc(v_i)$，$f_2 \rightarrow t_1$ 是非法跳转，对其检测的过程需要分两种情况讨论：

- 如果 v_j 是 I 型或 Y 型节点，v_j 的一个前驱节点是 v_k，则执行 v_j 中的指令 #1 后，$G = G \oplus d_{ty}(v_j) = EST(v_i) \oplus EST(v_k) \oplus ESY(v_j)$，由于 $v_i \notin pre(v_j)$，$EST(v_i) \neq EST(v_k)$，所以 $G \neq ESY(v_j)$，继续执行 v_j 中的指令 #2 后，$G = G \oplus d_{yt}(v_j) = G \oplus ESY(v_j) \oplus EST(v_j)$，由于输入 $G \neq ESY(v_j)$，所以执行 #2 后 $G \neq EST(v_j)$，最终指令 #3 可以检测出该非法跳转。
- 如果 v_j 是 N 型节点，则执行 v_j 中第一条指令 #5 后，$G = G\&M = EST(v_i)\& 0b\underbrace{11\ldots11}_{m-2}00\underbrace{11\ldots11}_{m-2}00$，由于 $v_i \notin pre(v_j)$，根据标签设计规则应该有，$D_{m-2}(ST(v_i)) \neq D_{m-2}(SY(v_j))$，所以执行 #5 后 $G \neq ESY(v_j)$。同样地，继续执行 #2 后 $G \neq EST(v_j)$，最终指令 #3 可以检测出该非法跳转。

（2）当跳转为 $f_2 \rightarrow t_2$ 时，执行 v_j 中的指令 #2 后，$G = G \oplus d_{yt}(v_j) = EST(v_i) \oplus ESY(v_j) \oplus EST(v_j)$，由于 $EST(v_i) \neq ESY(v_j)$，所以 $G \neq EST(v_j)$，最终指令 #3 可以检测出该非法跳转。

（3）当跳转为 $f_2 \rightarrow t_3$ 时，如果 v_i 和 v_j 均为同一个 Y 型节点的前驱，根据标签设计规则，$EST(v_i) = EST(v_j)$。因为跳转发生时，$G = EST(v_i)$，所以跳转到 t_3 后执行 v_j 中的指令 #3：$br\ G = EST(v_j)$, nlabel 不会报告错误。但是由于 v_i 和 v_j 的后继节点相同，所以该跳转在语义上等价于从 v_i 直接跳往 v_i 的后继节点，因此可以视为合法跳转。如果 v_i 和 v_j 之间不存在上述关系，则 $EST(v_i) \neq EST(v_j)$，显然，执行 v_j 中的指令 #3 将可以检出非法跳转 $f_2 \rightarrow t_3$。

（4）当跳转为 $f_2 \rightarrow t_4$ 时，跳转后程序执行直接进入错误处理例程。错误处理例程会再次执行和 #3 相同的操作，此时算法的检测情况和情形（3）相同。

综合以上证明结果可知，从 f_2 起始的非法块间跳转都可以被算法检测。基于定理 3.6 和定理 3.7 可以得出结论，所有基本块间的控制流错误均可以被 CFCES 算法检测。

CFCES 通过在基本块内插装标签运算指令实现了对块内控制流错误的检测。图3.11给出了一个块内检测机制的例子。图中块内检测机制插装的指令将基本块

分割成不同的区域，显然，I/Y 型节点的块内控制流错误将发生在 p_1、p_2、p_3、p_4、p_5、p_6 之间，N 型节点的块内控制流错误将发生在 p_1、p_2、p_3、p_4、p_5、p_6、p_7 之间。对于发生在一个 BR 块内（例如，3.11(a) 中 p_2 和 p_2 之后的 #6 之间的区域）的跳转错误，由于这些错误不会影响算法插装指令的执行，CFCES 不能检测到这些错误。但是，在基本块内插装更多的 #6 指令可以有效克服这个问题。下面通过定理 3.8 证明 CFCES 可以检测除上述错误之外的所有块内控制流错误。

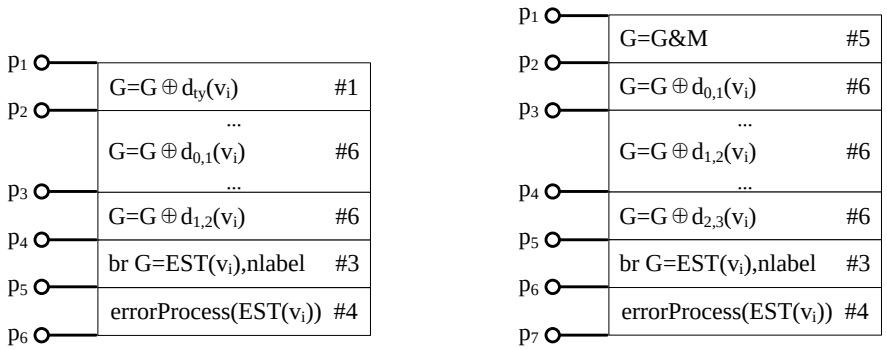


图 3.11　块内控制流错误的起始和终止位置

定理 3.8：CFCES 算法可以检测所有基本块内的控制流错误。

证明：当 v_i 为 I 型或 Y 型节点，可以根据不同的起始位置分别进行证明：

（1）从 p_1 到块内其它位置的跳转错误可以视为是块间跳转错误，定理 3.6 和 3.7 已经证明可以检测这些错误。

（2）从 p_2 到其它位置的非法跳转（跳转前 $G = ESY(v_i)$）：

- 如果非法跳转为 $p_2 \rightarrow p_1$，将会导致指令 #1 被再次执行，假设 v_i 的一个前驱节点为 v_k，再次执行 #1 后，$G = G \oplus d_{ty}(v_i) = ESY(v_i) \oplus EST(v_k) \oplus ESY(v_i) = EST(v_k)$，由于 G 的值不再是预期的 $ESY(v_i)$，后续的指令 #6 基于 G 更新得到的动态标签也将不等于预期的结果，指令 #3 通过标签比较可以检测出该非法跳转。
- 如果非法跳转为 $p_2 \rightarrow p_3$，跳转后执行指令 #6：$G = G \oplus d_{1,2}(v_i) = ESY(v_i) \oplus ESI_1(v_i) \oplus EST(v_i)$，由于 ESI 标签具有唯一性，即 $ESY(v_i) \neq ESI_1(v_i)$，所以 $G \neq EST(v_i)$，继续执行到 #3 时可以检测到该非法跳转。
- 当非法跳转为 $p_2 \rightarrow p_4$ 时，跳转后执行指令 #3，此时 $G = ESY(v_i)$ 且 $ESY(v_i) \neq EST(v_i)$，所以可以通过比较检测出该非法跳转。
- 当非法跳转为 $p_2 \rightarrow p_5$ 时，跳转后程序直接执行错误处理例程，错误处理例程通过再次比较可以检测出该非法跳转。

- 非法跳转 $p_2 \rightarrow p_6$ 等同于块间跳转错误，定理 3.6 和 3.7 已经证明可以检测这种错误。

（3）从 p_3 到其它位置的非法跳转和从 p_2 到其它位置的非法跳转类似，均可以被成功检测。

（4）从 p_4 到 p_1、p_2、p_3 的跳转错误和非法跳转 $p_2 \rightarrow p_1$ 类似，均可以被成功检测。从 p_4 到 p_5、p_6 的错误跳转不会被检出，但是这些非法跳转并不影响程序的执行结果，因此可以被安全忽略。

（5）从 p_5、p_6 到其它位置的非法跳转和从 p_2 到其它位置的非法跳转类似，也可以被成功检测。

当 v_i 为 N 型节点时，只需要分析目标为 p_1 的块内非法跳转即可，其它情形和 I/Y 型节点类似。

（1）当非法跳转为 $p_2 \rightarrow p_1$ 时，指令 #5 将被重复执行，此非法跳转不会被检出，但是由于 #5 是按位与运算，此非法跳转不会影响程序的运行结果，因此可以安全忽略。

（2）当非法跳转为 $p_3 \rightarrow p_1$ 时，跳转前 $G = ESI_1(v_i)$，跳转后先执行指令 #5，$G = G\&M$，根据设计规则，$D_{m-2}(SI_1(v_i)) \neq D_{m-2}(SY(v_i))$，所以执行指令 #5 后，$G \neq ESY(v_i)$。继续执行后续三条 #6 指令后，$G = G \oplus d_{0,1}(v_i) \oplus d_{1,2}(v_i) \oplus d_{2,3}(v_i) = G \oplus d_{y,t}(v_i) \neq EST(v_i)$，所以执行指令 #3 可以检测出该错误。

（3）从 p_4、p_5、p_6、p_7 到 p_1 的非法跳转和 $p_3 \rightarrow p_1$ 类似，均可以通过指令 #3 检出。

综合以上结论，算法可以检测所有基本块内的控制流错误。

3.5 CFCES 的可配置优化方法

控制流检测技术插装的冗余代码会引入大量的时空开销，而直接减少插装的检测代码将会导致检测能力的明显下降。因此，对控制流检测进行的配置优化应当实现以较小的可靠性损失换取较大的开销优化，或者以较小的时空开销获得较大的可靠性提升。综合来说，即是优化之后控制流检测的容错效率得到提高。本节为 CFCES 提出了可以提高容错效率的优化配置方法。该方法首先基于函数重要性分析对不同函数实施不同程度的保护，然后对重要的函数进行重新划分程序块并重构控制流图。通过配置不同函数的保护强度和程序块的大小，该方法可以灵活调节 CFCES 的时空开销和可靠性。

本节内容组织如下：3.5.1节介绍函数的重要性分析方法，并提出基于函数重要性的分类保护思路；3.5.2节阐述基于重构控制流图进行检测优化的原理；3.5.3、3.5.4和3.5.5节给出重构控制流图的主要步骤，分别为划分逻辑块、复制基本块和分割逻辑块。

3.5.1 基于函数重要性评估的分类保护

将函数 f_i 的重要性定义为 f_i 发生控制流错误并导致失效（即 SDC 错误）的概率，其计算公式如下：

$$F_{crt}(f_i) = FV(f_i) \times P_{SDC}(f_i) \tag{3.6}$$

其中，$FV(f_i)$ 表示 f_i 的脆弱性，即 f_i 运行期间程序发生控制流错误的可能性。$P_{SDC}(f_i)$ 表示 f_i 的 SDC 故障率，即 f_i 发生控制流错误的情况下程序结果出现 SDC 错误的概率。P_{SDC} 参数通过故障注入的方式获得，函数的脆弱性则基于函数的执行时间进行估计，其估计公式如下：

$$FV(f_i) = \frac{T_{f_i} \times Freq(f_i)}{T_{avg}} \tag{3.7}$$

其中，T_{f_i} 表示 f_i 的执行时间，$Freq(f_i)$ 表示 f_i 的执行频率，T_{avg} 表示程序的平均执行时间。这些参数均可以通过程序执行剖面信息获得。基于程序中各个函数的重要性评估结果，可以得出每个函数在程序中的相对重要性，其计算公式为：

$$F_{rc}(f_i) = \frac{F_{crt}(f_i)}{\sum F_{crt}(f_i)} \times 100\% \tag{3.8}$$

其中，$\sum F_{crt}(f_i)$ 表示程序中各个函数的重要性总和。

由于程序的运行往往具有明显的局部性，即程序大部分的执行时间消耗在一小部分代码上（这也是现代系统中引入 Cache 的原因），所以不同函数的脆弱性有很大的差别。此外，文献 [29] 的研究结果表明，不同的代码段导致 SDC 错误的概率也有明显不同。综合上述原因，可以认为不同函数的重要性也具有明显差别。基于这个认识，CFCES 的优化方法将函数根据重要性分为需要重点保护的和可以放松保护的两类。分类的方法是，根据设定的阈值标准 K，从程序中选取相对重要性总和高于 K 的最小函数集合，作为需要重点保护的对象。程序中剩余的函数则作为可以放松保护的对象。对于可以放松保护的函数，实现控制流检测技术时将其整体上视为一个程序块，即只在函数入口处和出口处插装控制流检测指令。这样可以减少部分性能开销和大量的存储开销。由于这部分函数对程序可靠性影响很小，所以对其放松保护不会明显降低程序可靠性。对于需要重点保护的函数，则需要根据用户对时空开销和可靠性的约束，进一步精细地配置保护的力度（下一小节将介绍针对这部分函数提出的重构控制流图优化方法）。基于函数分类进行不同强度的保护，可以提高控制流检测技术的容错效率。

3.5.2 重构控制流图优化方法的原理

现有的控制流检测技术通常在所有基本块内插装相同的检测指令，这种做法会导致算法的容错效率下降，原因是：对于较小的基本块，如只包含 1 条跳转指令的基本块，发生控制流错误的概率很小，但是却需要在基本块内插装多条指令实现控制流检测，检错的时空开销很大；对于部分较大的基本块，发生各种控制流错误的概率很大，插装同样数量的检测指令却难以保证故障覆盖率，尤其是现有的绝大多数控制流检测方法不能检测基本块内的控制流错误，这些算法受这种问题的影响更为明显。为了克服基本块大小差异带来的容错效率降低问题，CFCES 针对需要重点保护的函数提出了重构控制流图的优化配置方法。该方法先改变程序块的划分方法，使得划分出的程序块大小更加均匀，然后基于新划分的程序块重构控制流图并实现控制流检测机制，以此达到提高容错效率的目的。这种方法的理论依据为：

定理 3.9：基于均匀划分的程序块实现的控制流检测技术具有更高的容错效率。

证明：从公式 (2.13) 可以看出，欲证结论成立，只需证明在相同的性能开销前提下，将程序块划分均匀后实现控制流检测技术可以减少漏检的错误即可。

假设一个程序共有 L 条指令，所采取的控制流检测技术对块间控制流错误的检测率为 P，对块内控制流错误没有检测能力，即检测率为 0。将程序划分为 k 个程序块，第一种划分方式是非均匀的，各个块的大小分别为 n_1，n_2，$n_3 \ldots n_k$，显然 $L = n_1 + n_2 + n_3 + \ldots + n_k$ 且 $\exists n_i \neq n_j (1 \leq i, j \leq k)$。第二种划分方式为均匀的，即每个块的大小为 $\frac{L}{k}$ 。由于划分的块数相同，两种划分下实现的控制流检测需要的性能开销是相同的。此外，第一种划分方式下程序发生的控制流错误是块内控制流错误的概率为 $\frac{C_{n_1}^2+C_{n_2}^2+\ldots+C_{n_k}^2}{C_L^2}$ ，是块间控制流错误的概率为 $(1-\frac{C_{n_1}^2+C_{n_2}^2+\ldots+C_{n_k}^2}{C_L^2})$，所以控制流检测技术漏检错误的概率为：

$$\begin{aligned}&\frac{C_{n_1}^2+C_{n_2}^2+\ldots+C_{n_k}^2}{C_L^2}+(1-\frac{C_{n_1}^2+C_{n_2}^2+\ldots+C_{n_k}^2}{C_L^2})\times(1-P)=\\&1-P+P\times\frac{C_{n_1}^2+C_{n_2}^2+\ldots+C_{n_k}^2}{C_L^2}\end{aligned} \tag{3.9}$$

第二种划分方式下块内控制流错误的发生概率为 $\frac{k\times C_{\frac{L}{k}}^2}{C_L^2}$，块间控制流错误的发生概率为 $(1-\frac{k\times C_{\frac{L}{k}}^2}{C_L^2})$，所以控制流检测技术漏检错误的概率为：

$$\frac{k\times C_{\frac{L}{k}}^2}{C_L^2}+(1-\frac{k\times C_{\frac{L}{k}}^2}{C_L^2})\times(1-P)=1-P+P\times\frac{k\times C_{\frac{L}{k}}^2}{C_L^2} \tag{3.10}$$

两种划分下漏检错误概率的差值为：

$$
\begin{aligned}
&1-P+P\times\frac{C_{n_1}^2+C_{n_2}^2+\ldots+C_{n_k}^2}{C_L^2}-(1-P+P\times\frac{k\times C_{\frac{L}{k}}^2}{C_L^2})=\\
&P\times(\frac{C_{n_1}^2+C_{n_2}^2+\ldots+C_{n_k}^2}{C_L^2}-\frac{k\times C_{\frac{L}{k}}^2}{C_L^2})=P\times\frac{n_1\times(n_1-1)+n_2\times(n_2-1)+\ldots+n_k\times(n_k-1)-k\times\frac{L}{k}(\frac{L}{k}-1)}{2C_L^2}\\
&=P\times\frac{n_1^2-n_1+n_2^2-n_2+\ldots+n_k^2-n_k-\frac{L^2}{k}+L}{2C_L^2}=P\times\frac{n_1^2+n_2^2+\ldots+n_k^2-\frac{L^2}{k}}{2C_L^2}\\
&=P\times\frac{\frac{1}{k}\times(k\times n_1^2+k\times n_2^2+\ldots+k\times n_k^2-L^2)}{2C_L^2}=P\times\frac{\sum_{i\neq j}(n_i-n_j)^2}{2k\times C_L^2}>0
\end{aligned}
\tag{3.11}
$$

因此，均匀划分的情况下实现的控制流检测技术漏检错误的概率更小，所以可以得出结论，基于均匀划分的程序块实现的控制流检测技术具有更高的容错效率。

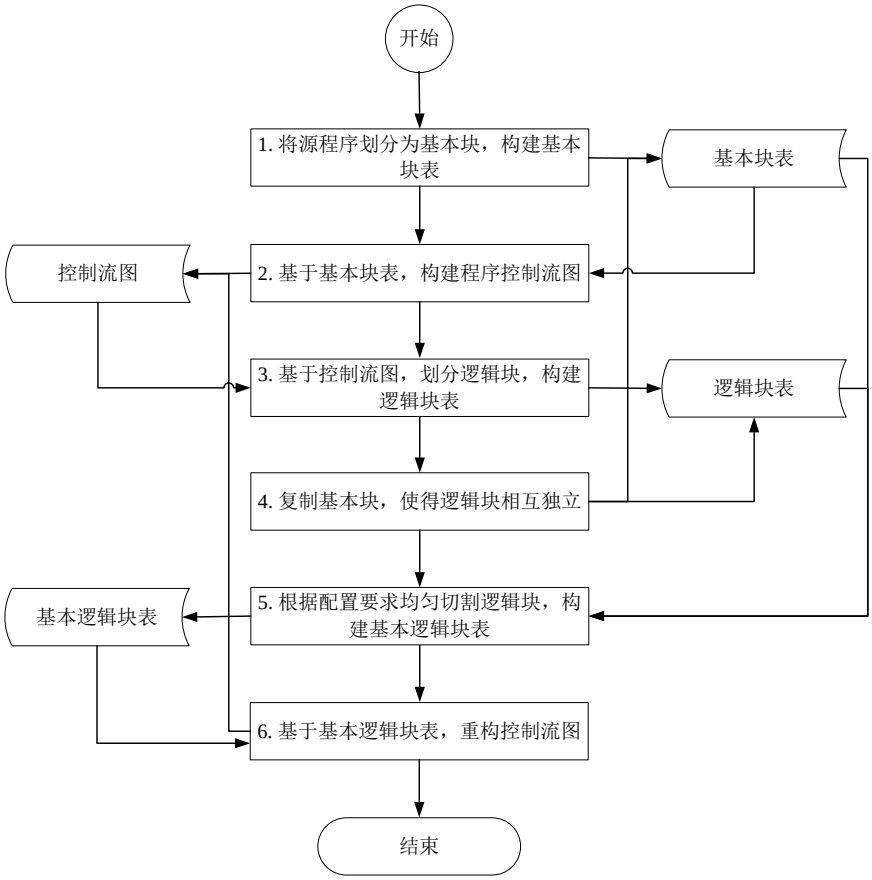


图 3.12　重构程序控制流图的流程

重构控制流图优化的流程如图3.12所示。首先将程序划分为基本块并构建控制流图（第 1-2 步），然后在基本块的基础上划分出更大的逻辑块（第 3 步）。如果不同的逻辑块包含共同的基本块，还要复制这部分基本块，以使得逻辑块相互独立（第 4 步）。最后，根据配置的大小要求均匀切割逻辑块得到基本逻辑块（第 5 步），并基于基本逻辑块重新构造程序的控制流图（第 6 步）。通过配置基本逻

辑块的大小，可以调节控制流检测技术的检测能力和开销。下面分别介绍 3-6 步涉及的概念和算法。

3.5.3 划分逻辑块

为了划分出均匀的程序块，首先需要在基本块基础上合并得到较大的逻辑块。下面介绍逻辑块及其它相关的定义：

定义 3.14 多扇出块（Fan-out Block）：如果控制流图中的一个基本块 v_i 满足 $|suc(v_i)| > 1$，则称 v_i 为多扇出块。

定义 3.15 分界块（Boundary Block）：定义多扇出块、函数出口块、函数调用块的前驱和函数调用块为分界块。程序分界块的集合定义为 $BoundarySet$。

定义 3.16 逻辑块（Logic Block）：逻辑块是一个顺序执行的基本块序列，逻辑块的最后一个基本块是分界块，第一个基本块在控制流图中的前驱至少有一个是分界块。将逻辑块 LB_k 用 n 元序偶 $< v_{k_1}, v_{k_2}, \ldots, v_{k_n} >$ 表示，则 $v_{k_n} \in BoundarySet \wedge \exists v_j (v_j \in pre(v_{k_1}) \wedge v_j \in BoundarySet)$，且序列中任意两个相邻的块 v_{k_i}，$v_{k_{i+1}}$ 满足：$suc(v_{k_i}) = \{v_{k_{i+1}}\}$。

定义 3.17 基本逻辑块（basic logic block）：基本逻辑块是将逻辑块按照一定的大小标准平均分割后得到的程序块。

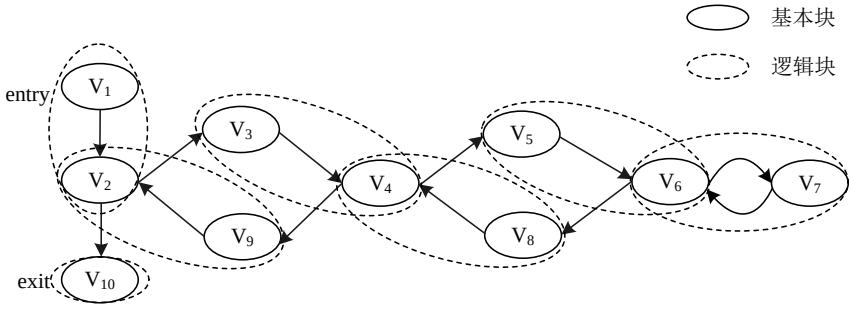


图 3.13　矩阵乘法程序的逻辑块划分结果

图3.13给出了基于矩阵乘法程序的控制流图划分出的逻辑块：$< v_1, v_2 >$、$< v_3, v_4 >$、$< v_5, v_6 >$、$< v_7, v_6 >$、$< v_8, v_4 >$、$< v_9, v_2 >$、$< v_{10} >$。由于逻辑块是基于基本块之间的控制流关系合并得到的，逻辑块内部的各个基本块在物理存储上不一定是连续的。例如对于逻辑块 $< v_5, v_6 >$ 和 $< v_7, v_6 >$，其中必定有一个是不连续的。

算法3.1给出了基于程序的控制流图划分逻辑块的具体过程。算法首先分析程序中的分界块（第 3-7 步），然后把程序入口块和分界块的后继块放入临时集合 $HeaderSet$ 中（第 8-14 步）。针对 $HeaderSet$ 中每一个块 v_n，算法为 v_n 创建一个以 v_n 为链首的链表 $Blink$（第 15-17 步），并通过不断将当前链尾节点 $Vtemp$ 的后

算法 3.1 DivideLB ( )

输入: 程序 P 的全局控制流图 $GCFG(V, E)$

输出: 程序 P 划分出的逻辑块集合 $LBset$

```
1:  BoundarySet=∅;
2:  LBset=∅;
3:  for each v_i ∈ V do
4:      if v_i is a boundary block then
5:          BoundarySet = BoundarySet ∪ {v_i};
6:      end if
7:  end for
8:  Create block set HeaderSet;
9:  HeaderSet = HeaderSet ∪ {v_1}; //provided that v_1 is the first block of P
10: for each v_j ∈ BoundrySet do
11:     for each v_k ∈ suc(v_j) do
12:         HeaderSet = HeaderSet ∪ {v_k};
13:     end for
14: end for
15: for each v_n ∈ HeaderSet do
16:     Create block link Blink;
17:     Put v_n into the tail of Blink;
18:     Define temporal variable Vtemp = v_n;
19:     while Vtemp ∉ BoundrySet do
20:         Let v_m ∈ suc(Vtemp);
21:         Vtemp = v_m;
22:         Put Vtemp into the tail of Blink;
23:     end while
24:     LBset = LBset ∪ {Blink};
25: end for
26: return LBset;
```

继插入 $Vtemp$ 后面来扩展 $Blink$，对 $Blink$ 的扩展直到 $Vtemp$ 是分界块时停止（第 18-23 步）。扩展后的 $Blink$ 就是以 v_n 为入口的逻辑块，算法将其加入逻辑块集合 $LBset$（第 24 步）。最后，基于 $HeaderSet$ 中每个基本块构建逻辑块并加入 $LBset$ 后，算法将 $LBset$ 作为程序的逻辑块集合返回（第 26 步）。

3.5.4 复制基本块

源程序划分出的逻辑块中往往包含共同的基本块（例如，图3.13中的逻辑块 $<v_3, v_4>$ 和 $<v_8, v_4>$），这使得逻辑块具有多个入口且入口有可能出现在块内。由于

目前的控制流检测技术都是基于基本块设计的（包括 CFCES 的基本检测算法），而逻辑块的特征和基本块明显不同（基本块只能有一个入口），所以不能直接基于逻辑块应用目前的控制流检测算法。为了解决上述问题，需要对逻辑块中公共的基本块进行复制，直到所有的逻辑块都不再含有共同的基本块。

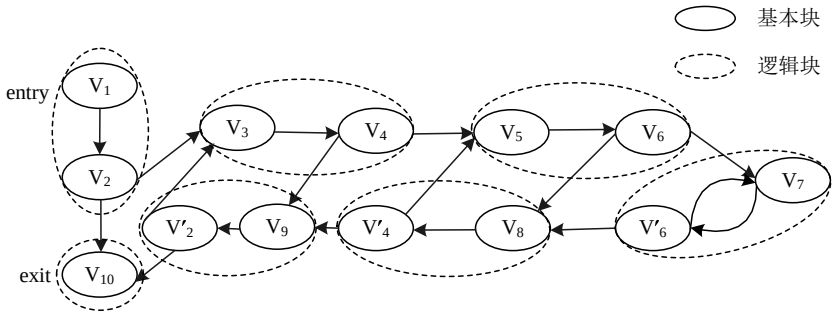


图 3.14　复制逻辑块的公共基本块

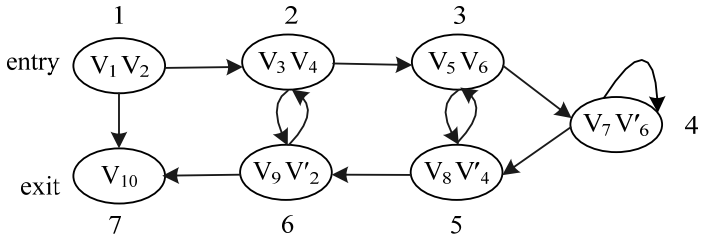


图 3.15　基于逻辑块构建的控制流图

图3.14给出了复制图3.13中部分基本块后的结果。图3.13中 v_2、v_4 和 v_6 都是公共基本块，图3.14中分别对其复制得到了副本节点 v'_2、v'_4 和 v'_6，并通过修改基本块间的控制流关系，使得逻辑块之间不再包含共同的基本块。如图3.14所示，经过基本块复制后的逻辑块也只有一个入口和一个出口，且入口就是其中的第一个基本块，出口就是其最后一个基本块，从而具备了与基本块相似的控制流特征，可以基于逻辑块实现现有的所有控制流检测算法。图3.15给出了基于逻辑块构建的控制流图。由于复制基本块并没改变程序执行时的操作，所以不会影响程序的性能。复制基本块会带来额外的存储开销，但是由于复制基本块只针对需要重点保护的函数进行，且只限于含共同基本块的逻辑块，这种存储开销是有限的，此外，由于针对非重点保护函数的轻量级保护可以大大减少存储开销，总体上 CFCES 的优化方法仍可以降低控制流检测的存储开销。

3.5.5 分割逻辑块

将程序分割成相互独立的逻辑块后，CFCES 优化的最后一步是根据设定的大小标准对逻辑块进行平均分割，从而得到大小相近的基本逻辑块。假设设定的基本逻辑块大小上限是 N，一个逻辑块 LB_i 的长度为 n。当 $N \geq n$ 时可以直接将 LB_i 视为一个基本逻辑块。当 $N < n$ 时，则可将 LB_i 平均分割为 $\lceil n/N \rceil$（$\lceil \rceil$ 表示上取整运算）个基本逻辑块。例如，设定每个基本逻辑块包含的指令数量 $N \leq 5$，则可将大小为 12 的逻辑块分割为大小均为 4 的 3 个基本逻辑块，将大小为 10 的逻辑块分割为 2 个大小为 5 的基本逻辑块。显然，这样分割后得到的基本逻辑块大小接近。此外，同一个逻辑块内分割出的基本逻辑块之间是顺序的控制流关系。图3.13中控制流图重构的最终结果如图3.16所示。图3.15中的逻辑块 3 和 4 在图3.16中分别被分割为 2 个（即 L_{3-1} 和 L_{3-2}）和 5 个基本逻辑块（即 L_{4-1}、L_{4-2}、L_{4-3}、L_{4-4}、L_{4-5}），基于这些基本逻辑块可以重构控制流图并实现控制流检测技术。

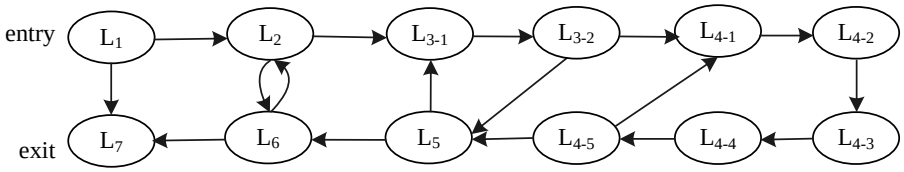


图 3.16 基于基本逻辑块重构后的控制流图

通过配置基本逻辑块的大小标准，可以调节 CFCES 的控制流检测能力和时空开销。当 N 的值设定为较大值时，由逻辑块切割得到的基本逻辑块的总数就会较少，由于 CFCES 在每个基本逻辑块中插入相同的指令，所以插装指令带来的开销就会减小，但是同时算法的检测能力也会下降。反之，当 N 的值较小时，插装指令带来的开销就会增加，但是算法的检测能力则会增强。不管 N 的配置大小，重构控制流图过程中分割出的基本逻辑块都是大小相近的，因此可以提高控制流检测的容错效率。

3.6 CFCES 的基本检测算法评估

为了评估 CFCES 的基本检测算法的有效性，本节分别对其进行了性能开销评估和检测效能评估。为了进行比较，本节也对两个已有的典型算法 CFCSS[85] 和 CEDA[91] 进行了评估。下面首先介绍故障模拟实验的方法，然后分别分析控制流检测算法的性能开销、故障覆盖率和容错效率，最后对 CFCES 的自容错能力进行评估。

3.6.1 实验方法

CFCES 的评估实验是在 SimpleScalar 模拟器 [105] 上进行的，模拟器的具体版本是 3.0，部署在 Red Hat Linux 9.0 上运行。实验中硬件运行平台采用 4 核 2.53GHz 的 CPU 和 4G 的内存。实验采用矩阵乘法（mm）、冒泡排序（bsort）、斐波那契（fib）、蒙特卡罗法求 π 值（pi）、随机数测试（shuf）和汉诺塔（hanoi）六个基准测试程序作为测试用例（这些测试用例曾被用作 CFCSS[85]、RSCFC[86]等已有算法的评估）。如图3.17所示，本文采用改变编译过程的方式在程序中实现控制流检测算法，其过程是：先将源程序用 SimpleScalar 内置的 GNU GCC 交叉编译环境编译得到汇编代码（实验使用的指令集为 PISA 指令集），然后利用开发的容错转换系统在汇编代码中插入控制流检测指令，得到具有容错功能的汇编程序，最后将容错汇编程序编译成可在 SimpleScalar 模拟器上运行的目标代码。

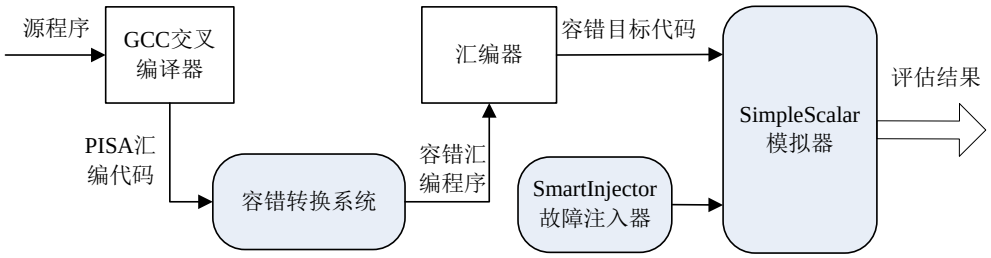


图 3.17　实验运行环境图

如图3.17所示，为了评估算法的控制流检测能力，实验在程序运行过程中利用本文第5章开发的 SmartInjector 故障注入工具向程序中专门注入控制流错误。注入故障的位置包括程序计数器（PC）、跳转指令以及操作码被翻转后将会变为跳转指令的指令。由于测试用例的规模较小，为了避免产生大量的地址越界异常，实验将 PC 故障的注入位置限制在了低 10 位。此外，基于单粒子翻转模型（SEU）假设，在一次程序运行中只注入一位翻转故障。注入故障的结果分类和2.7.1 节相同，故障覆盖率也可基于等式 (2.15) 计算。控制流检测算法的容错效率则可以通过以下等式计算得出：

$$FE = \frac{1}{(1-FC)\times PO} = \frac{1}{\frac{SDC}{total\ injected\ faults}\times PO} \tag{3.12}$$

其中，FC 和 PO 分别表示算法的故障覆盖率和性能开销。PO 的计算方法为:

$$PO = \frac{execution\ time\ of\ fault\ tolerance\ program}{execution\ time\ of\ source\ program}\times 100\% - 1 \tag{3.13}$$

3.6.2 性能开销评估

CFCES 实现了检查点优化机制，并在优化后保持每 50 条指令至少有一个检查点。此外，CFCES 在大于 7 的基本块中插装指令实现块内检测机制（保证每个 BR 块大小不超过 7)。CFCSS、CEDA 和 CFCES 算法的性能开销评估结果如图3.18所示。从图中可以看出，控制密集型程序产生的性能开销较高，例如 fib 和 hanoi。而运算密集型的程序产生的性能开销则较低，如 mm 和 shuf。这是由于前者内部的控制流跳转频繁，划分出的基本块较小，而后者划分出的基本块则较大。因为控制流检测算法在每个基本块中插装相同的检测指令，所以基本块较小时，算法的性能开销相对较大，基本块较大时，性能开销则相对较小。由于 CFCSS 算法在每个基本块中插装 3-5 条指令（PISA 指令集中的比较指令不支持立即数操作数，所以标签检查时需要额外引入一条指令加载静态标签)，CEDA 算法在每个基本块中插装 4 条指令，二者的性能开销比较接近。CFCSS 和 CEDA 的平均性能开销分别为 37.6% 和 38.3%。尽管 CFCES 插装的块内检测指令会导致更多性能开销，但是算法的检查点优化同时降低了性能开销。最终，在三个算法中 CFCES 的性能开销是最小的，平均为 32.6%。

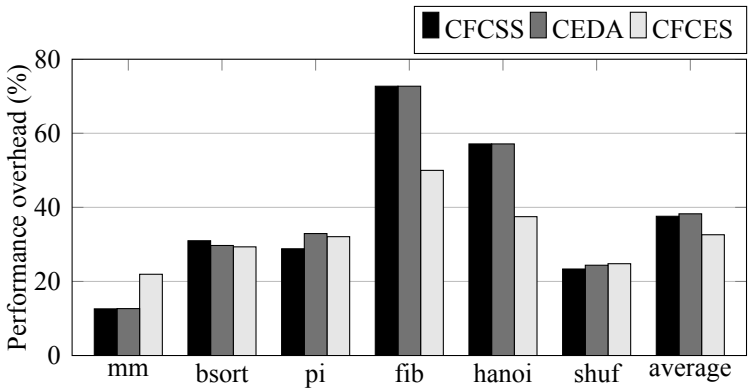


图 3.18 性能开销评估结果

3.6.3 检测效能评估

图3.19给出了对 CFCSS、CEDA 和 CFCES 的故障注入结果（分别在图中 C、D 和 E 列)。为了评估算法的有效性，实验还对源程序进行了故障注入（S 列)。从 S 列结果可以看出，Benign 故障的平均比例高达 34.6%，这意味着程序在没有算法保护的情况下，仍有相当大的概率输出正确结果。程序注入故障后能够输出正确结果的原因包括：（1）PISA 指令格式中包含部分保留位，发生在这些位中的故障不会影响指令语义；（2）当故障注入到条件分支的目标地址时，如果跳转操作并没有发生，则注入的故障不会影响程序结果；（3）注入到 PC 的故障可能导

致程序发生后向的非法跳转，这种错误会导致部分指令被重复执行。如果被重复执行的指令并没有改变程序语义，则程序的最终结果也不会受到影响。此外，平均 26.8% 的故障导致了程序发生异常（Exception），9.2% 的故障导致程序执行出现了超时（Timeout）。更严重的是，由于缺乏检测保护，平均 29.4% 的注入故障导致程序输出了错误结果（SDC）。

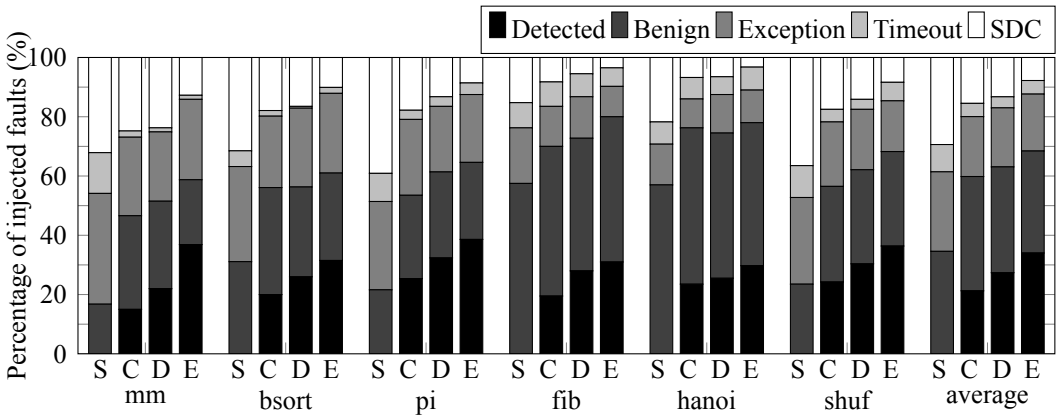


图 3.19　故障注入结果（S、C、D、E 分别表示源程序、CFCSS 程序、CEDA 程序和 CFCES 程序)

CFCSS 算法的故障覆盖率平均为 84.5%，包括 21.3% 的 Detected 故障、38.5% 的 Benign 故障、20.2% 的 Exception 故障和 4.5% 的 Timeout 故障。由于应用算法保护后，程序规模和条件分支指令都有增加，导致出现 Benign 故障的因素（第（1）、（2）点因素）增加，使得 Benign 故障的比例高于源程序。从图中可以看出，程序应用 CFCSS 算法保护后，未检出的故障即 SDC 的比例明显降低，平均为 15.5%，证明了控制流检测算法的有效性。

CEDA 算法在设计上克服了 CFCSS 的一些检测漏洞，因此其故障覆盖率提高到 86.7%，包括 27.4% 的 Detected 故障、35.7% 的 Benign 故障、19.9% 的 Exception 故障和 3.7% 的 Timeout 故障。而 SDC 故障的比例则降低到 13.3%，这些未检测故障大部分导致了程序发生块内或过程间控制流错误，而 CEDA 算法不具备检测这些错误的能力。因为出现块内控制流错误而导致漏检的典型程序是 mm，该程序中一个最大的基本块占了程序规模的 52%，导致 SDC 的比例高达 23.8%。

CFCES 算法在 CEDA 基础上，增强了对块内和过程间控制流错误的检测能力。其故障覆盖率进一步提高到了 92.2%，包括 34% 的 Detected 故障、34.4% 的 Benign 故障、19.2% 的 Exception 故障和 4.6% 的 Timeout 故障。剩余的 SDC 故障比例为 7.8%，这部分故障没被检出的原因是，它们导致了 BR 块内的控制流错误

或者产生了一些边界控制流错误（例如，直接跳到程序结束处)。相比 CFCSS 和 CEDA，CFCES 具有最强的检测能力。

评价算法的优劣不仅要比较其故障覆盖率，还要综合考虑性能的开销。容错效率指标将这两个因素结合在了一起。CFCSS、CEDA 和 CFCES 算法的容错效率如图3.20所示。从图中可以看出，对于每个程序，CFCES 的容错效率都是最高的，CEDA 的容错效率则高于 CFCSS 算法。最终，CFCSS 的平均容错效率为 22.8，CEDA 的平均容错效率为 26.3，CFCES 的平均容错效率则为 48.8，远高于 CFCSS 和 CEDA。

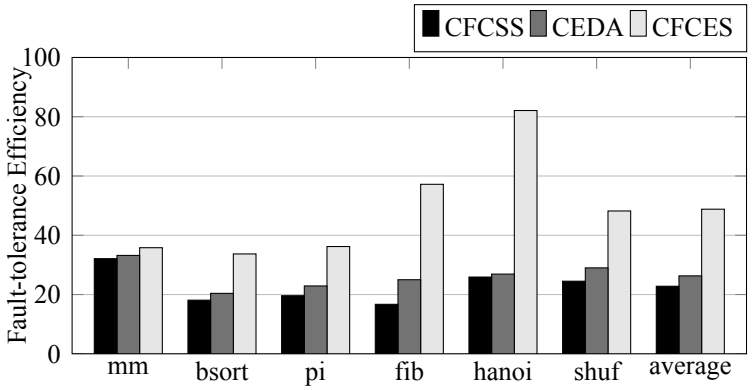


图 3.20　容错效率比较

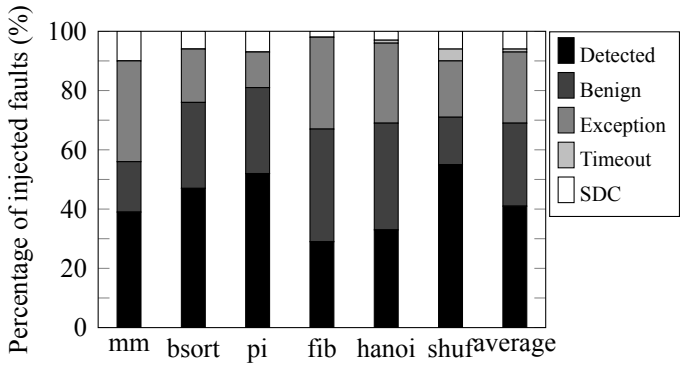


图 3.21　检测代码中注入控制流错误的结果

3.6.4　CFCES 的自容错能力评估

为了评估 CFCES 检测机制的自我保护能力，实验分别向检测代码中注入了控制流错误和数据流错误。控制流错误的注入方式和检测效能评估时相同，数据流错误则专门注入到检测机制使用的寄存器中。控制流错误的注入结果如图3.21所示。结果显示，检测机制的故障覆盖率平均为 94%。同样由于部分故障导致了 BR 块内的控制流错误或者产生了一些边界控制流错误，有 6% 的故障导致了 SDC 错

误。通过插装更多指令加强基本块内控制流错误检测可以减少这种情况。数据流错误的注入结果如表3.1 所示。由于注入的数据错误 100% 被识别并成功恢复，所以注入结果全为 Benign，消除了报告“假死”的可能。综上，CFCES 可以有效检出插装代码中破坏程序原有语义的故障，同时可以处理自身出错导致的“假死”问题，具有很强的自容错能力。

表 3.1　检测代码中注入数据流错误的结果

程序	mm	bsort	pi	fib	hanoi	shuf
Benign(%)	100	100	100	100	100	100

3.7　CFCES 的可配置优化方法评估

本节在 SimpleScalar 模拟平台上评估了 CFCES 的配置优化效果。3.5.1节进行函数重要性评估所需要的参数利用模拟器和故障注入获得。实验从 SPEC 2000 中选取了五个 benchmark 作为测试用例，分别为 gcc、twolf、ammp、vortex 和 gzip。实验中通过两个参数实现 CFCES 的配置优化：相对重要性比例 K 和基本逻辑块的大小 N。为了评估不同参数的调节效果，实验设定了四种配置方案：

（1）设定 K=95%，将非重点保护的函数视为单个节点进行保护，对重点保护的函数应用 CFCES 的基本块间和过程间检测机制进行保护。此种配置下，对重点保护的函数不进行重构控制流图优化，也不实现块内检测机制；

（2）设定 K=90%，实现方案同上；

（3）设定 K=90%，将非重点保护的函数视为单个节点进行保护，对重点保护的函数先进行重构控制流图，再应用 CFCES 的基本块间、过程间以及基本块内检测机制进行保护。设定基本逻辑块的大小 N=5。当一个逻辑块包含多个基本逻辑块时，利用块内检测机制保护这些基本逻辑块；

（4）设定 K=90%，N=10，实现方案同上。

通过比较第（1）种和第（2）种配置的结果，可以评估基于函数重要性进行分类保护法的优化效果。通过比较第（2)、（3）和（4）种配置的结果，可以评估重构控制流图方法的优化效果。下面分别从性能开销、存储开销、故障覆盖率和容错效率四个方面评估上述配置下的优化结果。

3.7.1　时空开销优化效果评估

性能开销的优化结果如图3.22所示。图中 95%-S、90%-S、90%-5、90%-10 分别对应上述四种配置方案。从 95%-S 和 90%-S 两种配置下的结果可以看出，通过对部分非重要函数进行轻量级保护，可以减少控制流检测的性能开销（相对于完

全的重点保护)。而且，由于程序中重要的代码分布具有局部性，性能开销的下降幅度较为明显（两种配置下平均分别减少了 9.6% 和 16.5%)。

在 90%-5、90%-10 两种配置下，性能开销平均分别减少了 16.8% 和 33.3%。从图中可以看出，部分程序在 90%-5 配置下的性能优化结果差于在 90%-S 配置下的优化结果，例如 twolf、ammp 和 gzip。这是因为在这些程序中，重点保护函数的基本块较大，使得重构控制流图后划分出的基本逻辑块数超过原有的基本块数，最终导致实现控制流检测时需要插装更多的指令。而在 90%-10 配置下，重构控制流图后划分出的基本逻辑块数均少于原有的基本块数，使得所有程序的性能优化结果好于程序在 90%-S 配置下的优化结果。

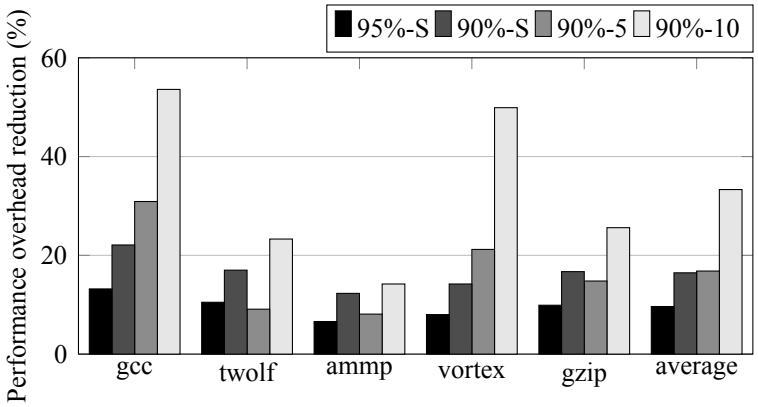


图 3.22　性能开销优化结果

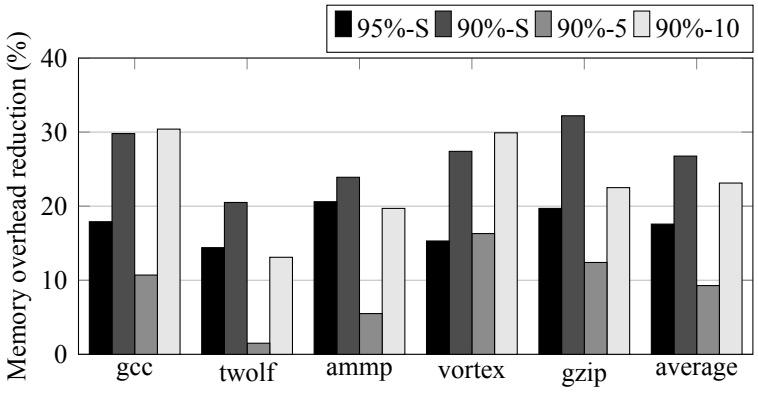


图 3.23　存储开销优化结果

存储开销的优化结果如图3.23所示。同性能开销一样，在 95%-S 和 90%-S 两种配置下，控制流检测的存储开销明显降低，平均降低幅度分别为 17.6% 和 26.8%。不同程序的优化效果和具体程序的局部性有关。在 90%-5、90%-10 两种配置下，存储开销的优化效果受两个方面的因素影响：(1) 在重点保护的函数中进行重构控制流图优化时，需要复制部分基本块，从而产生额外的存储开销；(2)

基本逻辑块的大小影响插装指令的数量，导致存储开销增加或减少。由于这两个方面的因素影响，两种配置下存储开销的平均下降幅度分别为 9.3% 和 23.1%。

3.7.2 检测效能评估

在不同的配置下，控制流检测算法的故障覆盖率也发生变化。实验以未检出故障（即 SDC）的比例变化情况评估控制流检测优化对故障覆盖率的影响。图3.24 给出了在四种配置下未检出故障的变化幅度。由于在非重点保护的函数中应用了轻量级保护，仍然可以有效地检测过程间的控制流错误，所以在 95%-S 和 90%-S 两种配置下，未检出故障的比例增加很少，平均分别为 2% 和 3.5%。在 90%-5 和 90%-10 两种配置下，由于调节了重点保护函数中基本逻辑块的大小，未检出故障的比例变化较为明显，在 90%-5 配置下平均减少了 1.1%，在 90%-10 配置下则平均增加了 15.2%。在 90%-5 配置下，部分程序划分出的基本逻辑块比其原有的基本块更小（twolf、ammp 和 gzip），所以控制流保护的力度更强，使得未检出故障的比例反而减少。

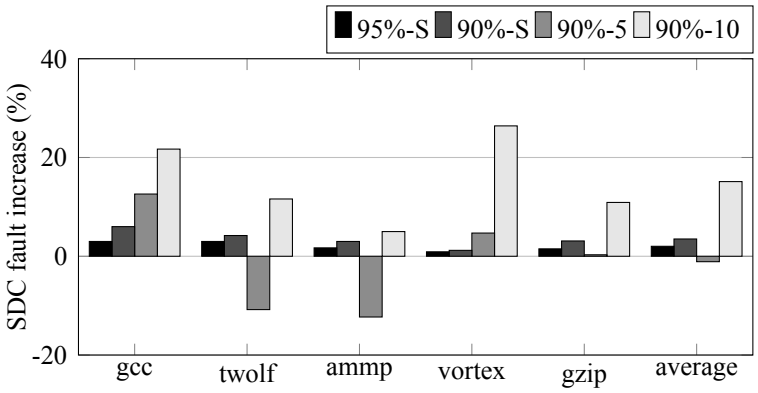


图 3.24　未检出故障变化情况

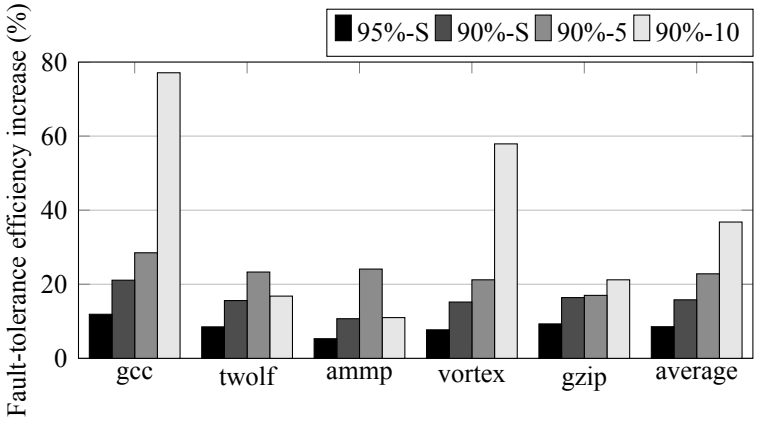


图 3.25　容错效率优化结果

CFCES 的配置优化方法的优势在于，其调节控制流检测的开销和故障覆盖率的同时，还可以提高检测算法的容错效率，即可以以较少的开销为代价，获取较大的可靠性收益。图3.25给出了容错效率的变化情况。从图中可以看出，在四种配置下，容错效率相对于优化前均有提高，平均提高幅度分别为 8.5%、15.8%、22.8% 和 36.8%。在 95%-S 和 90%-S 两种配置下，容错效率得到提高的原因是按照函数的重要性不同进行了分类保护。而在后两种配置下，容错效率得到优化的原因还包括划分了均匀的基本逻辑块，并应用了高效的块内检测机制。

3.8 本章小结

容错计算机系统必须具备对控制流错误的容忍能力。本章提出了基于软件实现的控制流检测算法 CFCES。CFCES 基于标签分析的思想对基本块内、基本块间和过程间控制流错误分别进行了处理，以较少的开销克服了已有算法存在的检测漏洞。此外，通过在控制流检测机制中引入“对等性”，CFCES 有效解决了检测机制的自容错保护问题。最后，CFCES 还通过函数重要性分析和重构控制流图，实现了算法的可配置优化。CFCES 克服了软件控制流检测技术在应用时面临的一系列障碍，具有更强的实用价值。本章通过理论分析和模拟实验证明了 CFCES 的有效性。

第四章　基于部分保护的片上 SPM 存储容错技术

瞬时故障不仅可能发生在处理器运算单元中，也有可能出现在处理器存储单元中。本章重点关注处理器存储单元中的故障。尽管 ECC 保护技术已被大量应用于保护片外存储结构，并被证明是一种行之有效的方法，但是这种技术并不适合用来保护片上存储结构。因为这些存储结构本身已经占用了大部分芯片面积，并且访问频繁，如果采用 ECC 保护会带来大量的面积、性能和功耗开销。本章针对一类特殊的片上存储结构，即便笺存储器（Scratchpad Memory，SPM），提出了一种低代价的容错保护技术。

4.1　引言

为了填补 CPU 和主存在速度上的巨大差距，现代计算机通常在处理器核和主存之间设置高速、小容量的片上 SRAM 存储结构，主要包括 Cache 和 SPM[70]。其中，SPM 是一种软件管理的快速存储器，设计 SPM 的目的是在嵌入式处理器中替代现有的 Cache。相比 Cache，SPM 具有如下优势：

- 更低的面积和功耗开销：Cache 中需要设计复杂的标记解码逻辑（tag-decoding）和比较逻辑，而 SPM 则不需要实现这些逻辑，因此采用 SPM 可以降低面积和功耗开销。研究表明 [112]，同等容量的前提下，SPM 需要的面积和功耗分别比 Cache 少 34% 和 40%。
- 更高的性能：尽管 SPM 和 Cache 具有相似的访问延迟（Cache 访问命中的情况下），但是由于 Cache 是由硬件管理，在访问未命中的情况下会产生大量的额外性能开销，而 SPM 则不存在这种问题。因此，对于访问模式比较规则的应用（例如多媒体应用），应用 SPM 的系统性能要高于 Cache[113, 114]。
- 更好的时间可预测性（timing predictability）：时间可预测性是实时系统极为关注的问题。时间行为的不确定性使得 Cache 难以在实时系统中大规模应用 [73]，而基于软件管理的 SPM 则可以提供良好的时间可预测性。

鉴于上述优势，越来越多的嵌入式系统开始采用 SPM 结构取代 Cache 或者设计混合 SPM 和 Cache 的存储体系结构 [70]。例如，Motorola M-core MMC221 和 TI TMS370Cx7x 利用 SPM 取代了 Cache，ARM10E 和 ColdFire MCF5 则采取了混合 SPM 和 Cache 的存储体系结构。除了嵌入式系统，SPM 还越来越多地被应用在 GPU、通用处理器甚至高端处理器上 [115]。

因为片上存储结构占用了大部分的芯片面积和晶体管数量，而且对瞬时故障的屏蔽能力较弱（相对于组合逻辑电路），在处理器中这种结构对瞬时故障是最为敏感的 [116]。另一方面，片上存储在计算机存储体系中属于离处理器核较近的存

储结构，使用访问频繁，发生在这些结构中的故障很容易传播到系统其它部位，导致错误的计算结果或系统崩溃。因此，发生在片上存储中的瞬时故障是影响系统可靠性的关键因素之一，必须采取容错措施加以解决。已有的技术主要采取奇偶校验或 ECC 编码的方式保护存储结构。其中，奇偶校验技术只能检测奇数位故障且不能恢复故障，因而无法提供理想的可靠性，只能和其他技术结合应用。ECC 技术则通常可以实现检二纠一，在目前单次程序运行中瞬时故障的发生概率仍然很小的情况下，可以有效地保护存储结构 [117]。但是，大量的面积、性能和功耗开销使得 ECC 技术仅能应用于保护片外存储（例如主存），却不适合应用于片上存储结构，尤其是一级 Cache 和 SPM。例如，已有研究结果表明 [118, 119]，在一级 Cache 中实现 ECC 技术带来的性能、功耗和芯片面积开销分别达到 95%、22% 和 18%。

现有的容错研究中十分缺乏针对片上存储结构的合理保护方案，因此本章针对 SPM 提出了一种低代价的保护技术 PPS（Data Allocation in Partially Protected SPM for Improving System Reliability）。尽管用 ECC 对 SPM 进行完全保护的开销很高，但是对部分 SPM 存储进行 ECC 保护并进行合理分配仍是非常有价值的。PPS 技术首先提供了一种基于部分 ECC 保护的 SPM 结构，即在处理器中设计两块并行放置的 SPM，其中一块采取 ECC 技术保护，而另一块则不进行 ECC 保护。然后对程序中的待分配变量进行脆弱性分析，并根据待分配变量大小将 SPM 划分为若干个“寄存器”，最后采取基于优先级的图着色方法将较为脆弱的变量优先分配到 ECC 保护的“寄存器”中。基于上述方法，PPS 能够以较低的开销获得较高的可靠性。

本章后续内容组织如下：第4.2节介绍与本章研究相关的工作；第4.3节提出 PPS 技术，具体包括 PPS 存储体系结构、变量脆弱性分析以及面向可靠性需求的 SPM 图着色分配等；第4.4节通过模拟实验对 PPS 技术的效果进行评估，并和传统的 SPM 分配技术进行比较；最后，第 4.5节对本章内容进行小结。

4.2 相关工作

尽管本章关注的是 SPM 的可靠性，但是由于功能的相似性，一些 Cache 保护技术也具有借鉴意义。另外，SPM 是由软件管理分配的，SPM 对系统可靠性的影响与分配到 SPM 中的数据有关。为了达到最优的系统可靠性，应当结合 SPM 的分配策略研究 SPM 的保护问题。最终，本节将相关的工作分为 SPM 分配技术、SPM 保护技术和 Cache 保护技术，并分别对这三类技术进行概述。

SPM 分配技术：SPM 的分配是在应用级或编译过程中实现的。目前针对 SPM 分配的研究都是以提高系统性能为目标，这些技术可以分为静态分配和动态分配

[115]。静态分配的核心思想是运行前在变量和 SPM 空间之间建立固定的映射关系，在整个程序运行期间，SPM 中只保存这些预分配的变量。动态分配则通过在程序中插装数据移动语句，在满足时序约束的前提下，将同一块 SPM 空间动态分配给不同的变量。文献 [115] 提出的方法是静态分配的典型方法。该方法提出了一些静态划分的启发式策略。例如，将标量数据分配到 SPM；将大小超过 SPM 的数组分配到片外存储器；对于可以放进 SPM 的数组，则基于提出的启发式策略计算优先级，并将优先级高的放进 SPM。显然这种简单的分配策略对存储空间的利用率不高，能够带来的性能提升有限。文献 [113] 提出的存储着色技术（Memory Coloring，MC）属于典型的动态分配方法。MC 技术先将 SPM 划分成不同的寄存器文件，然后利用一般化的图着色寄存器分配算法进行分配。这种动态分配方法可以提高 SPM 的利用率，对于有多个“热点”的大规模应用，可以获得的性能收益尤其明显。但是已有的动态分配方法和静态分配方法一样，只考虑了系统的性能需求，对 SPM 中瞬时故障的影响则缺乏关注。

SPM 保护技术：目前面向 SPM 中的瞬时故障的容错技术还极少。SPM 按照具体用途来分，可分为指令 SPM 和数据 SPM。Hamed 针对指令 SPM 中的瞬时故障进行了研究，提出了 MM-SPM （Memory Mapped SPM）方法 [73]。MM-SPM 基于指令 SPM 中的数据不会被修改这一特征，对指令 SPM 进行奇偶校验检错，并利用存放于片外存储的原始数据进行故障恢复。评估结果表明，相比 ECC 编码技术，MM-SPM 方法在性能、功耗和占用芯片面积方面均有明显优势。但是，由于数据 SPM 不存在类似指令 SPM 的访问特征，MM-SPM 并不适用于保护数据 SPM。和 MM-SPM 方法不同，PPS 技术则针对数据 SPM 提出了有效的容错方案。

Cache 保护技术：EWB（Early Write-back）[116] 技术将脏块周期性地写回到下一级存储，使得每个被写回的块在下一级存储中都拥有一个副本，然后采取奇偶校验结合副本恢复的方法保护被写回的脏块，从而达到保护部分 Cache 数据的目的。这种技术的缺点是可靠性的提高是以直接降低性能为代价的，不利于提高容错的效率。文献 [120] 为一级 Cache 设置了一个全相关、并列放置的冗余 Cache（R-cache），用以专门存放脏块的副本。写入 Cache 的数据也将同时被写入到 R-cache。这种方法不会导致大量的性能和功耗开销，而且将脏块复制率提高到了 97.3%，有效提高了 Cache 可靠性。但是这种方法采取复制的方式保护脏块，需要的存储开销偏高。

4.3 基于部分保护的 SPM 容错技术

片上存储是影响系统可靠性的关键因素，实现可信计算必须为片上存储提供有效的容错保护。本节面向 SPM 的可靠性问题，提出了一种低代价的容错保护技

术 PPS。基于提出的部分 ECC 保护的存储体系结构，PPS 先将 SPM 空间划分为若干个伪寄存器，然后改造传统的图着色寄存器分配方法，将对性能影响最大的数据分配到 SPM 中，最后再基于脆弱性分析重新着色分配，将 SPM 中对可靠性影响较大的数据调整到有 ECC 保护的伪寄存器中。经过多次图着色分配，PPS 可以以较低的开销获得较高的系统可靠性。

4.3.1 PPS 存储体系结构

基于 ECC 技术可以有效地对存储结构进行故障检测和恢复，但是采用 ECC 技术保护整个 SPM 空间将会带来很高的性能、功耗和面积开销。由于很多研究的分析结果表明 [72, 121]，不同的数据对系统可靠性的影响并不相同，实现低代价存储保护的一个自然做法是对存储结构进行部分 ECC 保护，然后将重要的数据优先分配到受保护的存储空间。基于这一原理，PPS 技术首先提出了部分 ECC 保护的 SPM 存储体系结构。如图4.1所示，PPS 存储体系结构包含两块并列放置的 SPM，其中一块 SPM 不受任何保护，另一块 SPM 则实现了 ECC 编码保护技术。另外，因为已经有大量商业产品采用 ECC 保护片外存储，PPS 假设片外存储也有 ECC 保护。

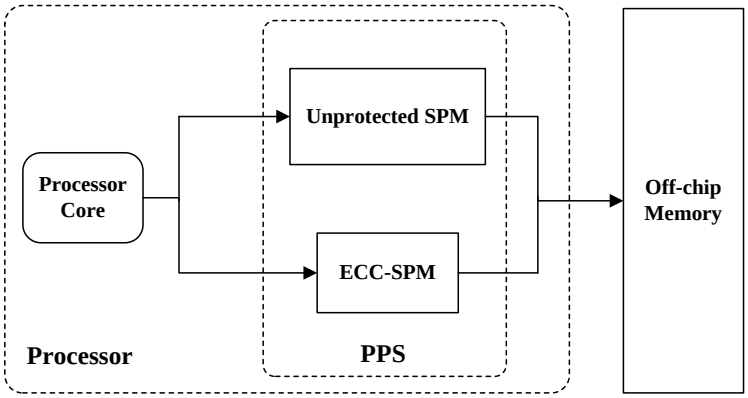


图 4.1 PPS 存储体系结构

4.3.2 待分配数据选择

基于提出的存储体系结构，PPS 技术的目标是合理选择数据分配到不同的存储器中，使系统的可靠性和性能达到最优。为此，PPS 尽量将访问频繁并且对系统可靠性影响较大的数据分配到 ECC 保护的 SPM 中，将访问频繁但对可靠性影响相对较小的数据存放到不受保护的 SPM 中，而访问次数很少的数据则可以留在 ECC 保护的片外存储中。由于程序中的数组变量往往占用大量存储空间且访问频繁，是影响性能和可靠性的主要因素，所以 PPS 技术选择数组变量作为分配到

SPM 的候选对象。而对于程序中的标量数据，则可以专门为其在 SPM 中预留一部分空间，采取类似 PPS 的方法进行分配。此外，由于 SPM 分配需要在编译时插装代码将变量分配到 SPM 中的具体位置，很难处理迭代函数中的局部变量，所以 PPS 和已有 SPM 分配技术一样 [113] 只考虑不含迭代调用的程序。

为了提高 SPM 的利用率，PPS 采取的是动态分配法，需要将数据在片外存储和 SPM 之间动态传输，确保系统频繁访问的数据存放在 SPM 中。定义变量从第一次访问到最后一次访问的区间为其生命周期。一个变量在其生命周期的不同阶段常常具有不同的访问频率，如果在其访问不频繁的阶段仍将其存放在 SPM 中，则有可能使得访问更为频繁的其他变量无法调入 SPM，从而不利于提高系统的性能。因此，PPS 技术基于数组变量的访问一般集中在循环内的特点，只考虑将数组变量在循环内的生命周期段分配到 SPM。为了便于分配，PPS 在开始存储分配之前，先将数组在循环内的生命周期段独立分割出来。分割方法和已有 SPM 分配技术一样 [113]，即通过插装数组拷贝代码为循环内的数组产生一个副本，并在循环体内用副本将其临时替代。PPS 技术将待分配对象最终确定为这些新产生的副本变量。图4.2(a) 和图 4.2(b) 分别给出了一段源程序在进行数组生命周期分割前后的代码示例。该例子通过在循环体外插装数组拷贝代码，将原数组 *a* 在循环内的生命周期分割转移到 *aCopy*，并在循环体内用副本变量 *aCopy* 取代了对原数组 *a* 的访问。如图中所示，如果副本变量在循环内被修改，还要在循环后插装写回代码将其拷贝给原数组。完成存储分配后，如果一个副本变量最终被选择分配到 SPM 中，则对应的拷贝和写回代码将被替换为片外存储和 SPM 之间的传输代码。否则，需要撤回上述分割数组生命周期的操作。

```
for (j=1; j<=4; j++)
{
  a[j]=ietmp1 & 0x3f;
  ietmp1 >>= 6;
 }
```

(a)

```
aCopy = a;  //循环前基本块
for (j=1; j<=4; j++)
{
  aCopy[j]=ietmp1 & 0x3f;
  ietmp1 >>= 6;
 }
 a = aCopy; //循环后基本块
```

(b)

图 4.2　数组变量生命周期分割示例

算法4.1给出了生成待分配对象的过程。算法首先从外层到内层遍历每个循环嵌套，并由子函数 *GenerateCopy*（如算法4.2所示）为每个在循环中被访问的数组生成副本变量（第 2-13 步）。*GenerateCopy* 将副本变量的拷贝和写回代码分别放在所属循环 *L* 前的基本块（pre-header）和循环后的基本块（outgoing block）中。

如果 L 属于内层循环，则这些拷贝代码在程序运行时会被执行多次，产生大量不必要的性能开销。为此，算法 14-20 步从内层到外层再次遍历循环嵌套，如果当前循环 $L1$ 只有一个被插装变量拷贝和写回代码的子循环，则将其子循环前、后基本块中的插装代码转移到 $L1$ 的前、后基本块中。函数 *GenerateCopy* 先在循环 L 前的基本块中插装拷贝代码将原数组 a 拷贝到副本 *aCopy* 中（第 3 步），然后遍历 L 的内层循环，并用 *aCopy* 取代所有对 a 的访问（4-11 步）。最后，如果发现 a 在 L 内被修改过，还要在 L 后的基本块中插装写回代码将修改后的 *aCopy* 传送到 a（第 12-14 步）。

算法 4.1 PotentialAllocatedVar()

输入： 程序控制流图 CFG
输出： 待分配对象集合 V

```
1:  V = φ;
2:  for each loop nest LN in the program do
3:      for each loop L starting from outermost to innermost do
4:          for each array variable a accessed in L do
5:              if a ∈ V then
6:                  continue;
7:              else
8:                  v = GenerateCopy(a, L);
9:                  V = V ∪ {v};
10:             end if
11:         end for
12:     end for
13: end for
14: for each loop nest LN in the program do
15:     for each loop L1 starting from innermost to outermost do
16:         if only one loop L2 in L1 contains new instrumented array operation
            codes then
17:             Move copy codes in the pre-header of L2 to the pre-header of L1;
18:             Move writeback codes in the outgoing block of L2 to the outgoing
                block of L1;
19:         end if
20:     end for
21: end for
22: return V;
```

算法 4.2 GenerateCopy(a, L)

输入: 循环 L，L 中的数组 a
输出: 生成的待分配对象 $aCopy$

```
1: bool modifyFlag = false;
2: Create new array aCopy;
3: Insert copy code aCopy = a in the pre-header of L;
4: for each loop from L to L's innermost loop do
5:     if a is accessed then
6:         Replace a with aCopy;
7:         if a is modified then
8:             modifyFlag = true;
9:         end if
10:    end if
11: end for
12: if modifyFlag = true then
13:    Insert writeback code a = aCopy in the outgoing block of L;
14: end if
15: return aCopy;
```

4.3.3 SPM 空间划分

PPS 采取图着色分配方法将待分配变量分配到 SPM 中。为此，PPS 技术首先需要将 SPM 空间划分为与待分配变量大小匹配的伪寄存器。SPM 空间的划分采取了文献 [113] 提出的方法，即先将待分配变量按照大小进行分类，大小接近的变量被划为一类，然后按照每一类的大小标准分别对 SPM 进行划分。

SPM 划分的具体过程如算法4.3所示。受 ECC 保护的 SPM 和无保护的 SPM 均采取这一算法进行划分。算法首先需要利用预先设置的对齐因子对每个待分配变量的大小进行对齐，并将对齐后的大小进行分类（4-6 步），然后基于对齐后的大小，对 SPM 进行多次划分，最终获取按对齐大小分类的变量集合和伪寄存器集合（9-23 步）。设置对齐因子是为了将大小相近的变量放到相同大小的伪寄存器中，减少伪寄存器的数量，降低图着色分配的开销。对齐因子设置的越大，则划分出的伪寄存器越少，但是，SPM 的空间利用率也会降低。

表4.1给出了一个待分配变量大小对齐的示例，对齐因子为 32 字节。经过对齐后，变量大小分为 32、64 和 128 三种。基于这三种大小分割 SPM 后得到的伪寄存器如图 4.3所示（图中 SPM 的大小为 256 字节）。划分 SPM 后，相同大小的伪寄存器之间不存在别名关系，但是不同大小的伪寄存器之间则有可能互为别名。例如，由于占用的空间存在重叠，$R_{128,0}$ 和 $R_{64,0}$ 互为别名。显然，互为别名

算法 4.3 SPMPartitioning (V, $alignSize$)

输入： 待分配数组变量集合 V，对齐因子 $alignSize$

输出： SPM 划分结果：按大小分类的变量集合 VC 和伪寄存器集合 RC

1: $vSizeClass = \phi$; $n = 0$; //$vSizeClass$ 表示变量大小类别集合，n 表示变量对齐后的大小
2: **for** each variable $v_i \in V$ **do**
3: Let s_i be the size of v_i (in bytes);
4: $n_i = \lceil \frac{s_i}{alignSize} \rceil \times alignSize$; //$n_i$ is the aligned size of v_i
5: **if** $n_i \notin vSizeClass$ **then**
6: $vSizeClass = vSizeClass \cup \{n_i\}$;
7: **end if**
8: **end for**
9: **for** each $n \in vSizeClass$ **do**
10: Define $vClass_n = \{v_i | v_i \in V \wedge n_i = n\}$;
11: Define $rClass_n$;
12: Let $mSize$ be the size of SPM;
13: **for** each j from 0 to $\lfloor \frac{mSize}{n} \rfloor$ **do**
14: Let $mStartAddr$ be the start address of SPM;
15: Define register $R_{n,j}$;
16: $R_{n,j}.startAddr = mStartAddr + j \times n$;
17: $R_{n,j}.endAddr = R_{n,j}.startAddr + n$;
18: $R_{n,j}.size = n$;
19: $rClass_n = rClass_n \cup \{R_{n,j}\}$;
20: **end for**
21: $RC = RC \cup \{rClass_n\}$;
22: $VC = VC \cup \{vClass_n\}$;
23: **end for**

的伪寄存器不能分配给生命周期存在冲突的变量。对于对齐大小为 n 的变量 v，所有大小为 n 的伪寄存器组成其可分配寄存器集合，记为 $availr(v)$。例如，对于表4.1中的变量 B，有 $availr(B) = \{R_{64,0}, R_{64,1}, R_{64,2}, R_{64,3}\}$。

表 4.1 待分配变量大小对齐示例

待分配变量	大小（字节）	对齐后大小（字节）
A	16	32
B	48	64
C	60	64
D	104	128

<table>
<tr><td colspan="4">$R_{128,0}$</td><td colspan="4">$R_{128,1}$</td></tr>
<tr><td colspan="8"></td></tr>
<tr><td colspan="2">$R_{64,0}$</td><td colspan="2">$R_{64,1}$</td><td colspan="2">$R_{64,2}$</td><td colspan="2">$R_{64,3}$</td></tr>
<tr><td colspan="8"></td></tr>
<tr><td>$R_{32,0}$</td><td>$R_{32,1}$</td><td>$R_{32,2}$</td><td>$R_{32,3}$</td><td>$R_{32,4}$</td><td>$R_{32,5}$</td><td>$R_{32,6}$</td><td>$R_{32,7}$</td></tr>
</table>

图 4.3　SPM 划分示例

4.3.4　构建冲突图

由于只有不被同时访问的两个变量才能先后动态存放到同一块存储空间，PPS 需要为待分配的变量构建冲突图（Interference Graph，IG），以获取变量分配时的这种“互斥”约束关系。下面先介绍冲突图及其它相关的定义：

定义 4.1 广义活跃：如果数组变量 v_i 在程序点 *Loc* 被访问或存在一条从 *Loc* 开始的路径使得 v_i 还将被访问（读或写），则称 v_i 在 *Loc* 处广义活跃。

定义 4.2 访问期：将 v_i 广义活跃的程序点集合定义为 v_i 的访问期（Access Range），记为 AR(v_i)。

定义 4.3 冲突图 $IG(V,E)$：针对程序 P 中的待分配变量，其冲突图 $IG(V,E)$ 可以定义为：

- V 为 P 的待分配数组变量集合；
- 如果变量 $v_1 \in V$，$v_2 \in V$ 且 $AR(v_1) \cap AR(v_2) \neq \phi$，则有一条无向边 $< v_1, v_2 > \in E$，并称 v_1 和 v_2 是冲突的。

构建冲突图需要先对待分配变量进行广义活跃性分析。此分析可以基于标准的数据流分析实现。令 $LiveIn(B)$ 为在基本块 B 入口处广义活跃的待分配变量集合，$LiveOut(B)$ 为在 B 出口处广义活跃的待分配变量集合。则有，

$$LiveOut(B) = \bigcup_{B_i \in S(B)} LiveIn(B_i) \tag{4.1}$$

其中 $S(B)$ 为 B 的后继基本块集合。定义 $Def(B)$ 为基本块 B 中被创建的待分配变量集合，$LiveUse(B)$ 为 B 中被访问的待分配变量集合。则有，

$$LiveIn(B) = (LiveOut(B) - Def(B)) \cup LiveUse(B) \tag{4.2}$$

上述数据流分析是针对单个函数的局部分析。对于存在子函数调用的情况，则还要在局部分析的基础上进行全局分析。全局分析针对每个函数调用块 B_{call} 和其调用的函数 f，简单地认为 $LiveOut(B_{call})$ 中的变量在 f 中是一直活跃的。基于数据流分析结果，为待分配数组变量构建冲突图的过程如算法4.4 所示。算法首先

将程序 P 中所有待分配数组变量加入冲突图的节点集合 V（第 1 步），然后逐个遍历程序中的函数 f，并基于 f 的局部数据流分析结果，遍历 f 中每个基本块的 *LiveIn* 集合，对于任何同在一个 *LiveIn* 集合的两个节点，添加一条边到冲突图的边集 E 中（3-11 步）。算法最后逐个遍历 P 中的函数调用块 B_{call} 和被 B_{call} 调用的函数 f_{callee}，并在 $LiveOut(B_{call})$ 中的每个变量和 f_{callee} 中每个待分配变量之间添加一条边到 E 中（12-20 步）。最终得到的 $IG(V,E)$ 即为 P 的冲突图。

算法 4.4 ConstructIG (N, *LiveIn*, *Liveout*)

输入： 程序 P 中待分配变量集合 N，P 中每个基本块的局部数据流分析结果 *LiveIn*, *Liveout*

输出： P 的冲突图 $IG(V,E)$

```
1:  V = N; E = φ;
2:  for each function f ∈ P do
3:      for each basic block B ∈ f do
4:          for each v_i ∈ LiveIn(B) do
5:              for each v_j ∈ LiveIn(B) − {v_i} do
6:                  if < v_i, v_j > ∉ E then
7:                      E = E ∪ {< v_i, v_j >};
8:                  end if
9:              end for
10:         end for
11:     end for
12:     for each calling basic block B_call ∈ P do
13:         for each v_i ∈ LiveOut(B_call) do
14:             for each function f_callee called by B_call do
15:                 for each v_j in f_callee do
16:                     E = E ∪ {< v_i, v_j >};
17:                 end for
18:             end for
19:         end for
20:     end for
21: end for
22: return IG(V, E);
```

4.3.5 提高性能的图着色分配过程

基于划分出的伪寄存器和冲突图，可以采取类似寄存器图着色分配的方法对 SPM 进行分配。由于 SPM 是影响系统性能的关键部件，PPS 技术的首要目标是

通过将片外存储中的数据分配到 SPM 中，尽量提升系统性能。在此基础上，再通过重新分配，将 SPM 中对可靠性影响较大的数据调整到 ECC 保护的伪寄存器中，从而将 SPM 中瞬时故障的危害降到最低。PPS 技术通过多次图着色过程实现上述目标，本节先介绍提高性能的图着色分配过程。由于分配的对象和目标均和传统的寄存器分配不同，PPS 需要对传统的图着色寄存器方法 [122] 进行改造。如图4.4所示，提高性能的图着色分配的各个步骤如下：

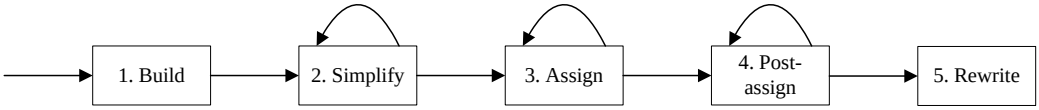


图 4.4　提高性能的图着色分配过程

1. Build：这一步需要为待分配变量构建冲突图 $IG(V,E)$，并将每个变量 v 的 $availr(v)$ 值赋为和其对齐大小相同的伪寄存器集合，包括有 ECC 保护的伪寄存器和无保护的伪寄存器。
2. Simplify：在程序的冲突图中寻找符合下述条件的节点 v：$\sum_{<v',v>\in E} alias(v',v) <| availr(v) |$，其中，$alias(v',v)$ 表示和 v' 的一个可分配寄存器别名的 v 的可分配寄存器的最大个数。以表4.1中的变量 D 和 A 为例，显然有 $alias(D,A) = 4$。$| availr(v) |$ 表示集合 $availr(v)$ 中元素的个数。如果找到符合上述条件的 v，则将 v 和与其邻接的边从冲突图中删除，并将 v 压栈。如果无法找到符合上述条件的节点，则需要根据启发式策略选择一个节点标记为“潜在溢出”并压入栈中。同样地，还要将“潜在溢出”节点和与其邻接的边从冲突图中删除。启发式函数的细节在后续部分介绍。重复上述搜索处理过程，直到冲突图中所有节点已经压入栈中。
3. Assign：从栈顶依次弹出各个节点并重构冲突图。每弹出一个节点 v，要先从 $availr(v)$ 中删除 v 的相邻节点已经分配的颜色和与这些已分配颜色别名的颜色。如果 $availr(v)$ 没有变为空集，则从中选取一个颜色分配给 v，若 v 是上一阶段标记为“潜在溢出”的节点，则此种情形下不应再标记 v 为“潜在溢出”。如果 $availr(v)$ 变为空集，则 v 只可能是上一阶段标记为“潜在溢出”的节点，此时把 v 仍然标记为“潜在溢出”并加入到冲突图中（因为没有分配颜色，v 不会影响后续加入节点的 $availr$ 集合）。重复上述过程直到所有节点都已加入到冲突图中。
4. Post-assign：这一步的目标是针对冲突图中的“潜在溢出”节点，在存储空间为其寻找可以利用的存储碎片。这些存储碎片主要由伪寄存器的别名导致的。例如，图4.3中的存储器中，如果 $R_{32,0}$ 和 $R_{32,3}$ 已经被分配，则 $R_{64,0}$ 和

$R_{64,1}$ 无法再分配给别的节点。即便此时 $R_{32,1}$ 和 $R_{32,2}$ 仍然处于空闲状态，仍然可能出现 64 字节的“潜在溢出”节点。但是，$R_{32,1}$ 和 $R_{32,2}$ 中的空间足够存放一个 64 字节“潜在溢出”节点。这些空间即是“潜在溢出”节点可以利用的存储碎片，定义其为临时寄存器。算法4.5给出了为一个“潜在溢出”节点 v_i 寻找临时寄存器的过程。算法通过分析 v_i 的相邻节点占用的存储空间，寻找 SPM 中大小超过 v_i 的存储碎片并放入集合 $TRset$（5-24 步）。如果搜索过程结束后 $TRset$ 不为空，则将 $TRset$ 中最小的一块作为 v_i 的临时寄存器返回（25-27 步）。Post-assign 过程重复利用算法4.5 为每一个“潜在溢出”节点寻找临时寄存器，最终未能分配临时寄存器的则标记为“溢出”。

5. Rewrite：基于分配的结果修改程序代码。对于分配到 SPM 的变量（即被着色的变量），修改4.3.2节为其插装的数组拷贝和写回代码为片外存储和 SPM 之间的传输语句。对于标记为“溢出”的节点，则需要撤回4.3.2节为将其分割成独立的生命周期所做的操作，从而将该节点保留在片外存储中。

上述分配过程中，为了使 Simplify 阶段继续进行下去，需要基于启发式策略选择部分节点作为“潜在溢出”对象。由于本次图着色分配的目的是提高性能，所以首先应优先溢出性能收益最少的节点。由于 ECC 校验引入的访问延迟问题可以通过后台刷新等机制克服 [72, 121]，PPS 假设有 ECC 保护的 SPM 的访问速度和无保护的 SPM 相同。因此，同一个变量 v_i 放在有保护和无保护的 SPM 中可以获得的性能收益是相同的。性能收益来源于 SPM 和片外存储的访问速度差别，但是要扣除变量在 SPM 和片外存储之间传输需要的开销。v_i 的传输开销 TR_i 可用以下公式计算 [113]:

$$TR_i = c_s + c_t \times s_i \tag{4.3}$$

其中，c_s 为启动传输的延迟，c_t 为传输一个字节的延迟，s_i 为变量 v_i 的大小（字节为单位）。最终，v_i 的平均性能收益 $AvgPB_i$ 的计算公式为:

$$AvgPB_i = \frac{(l_{mem} - l_{spm}) \times f_i - TR_i \times fc_i}{s_i} \tag{4.4}$$

其中，l_{mem} 表示对片外存储进行一次访问需要的延迟，l_{spm} 表示对 SPM 进行一次访问需要的延迟，f_i 表示 v_i 被访问的频率，fc_i 表示 v_i 在片外存储和 SPM 之间传输的频率。另外，出于着色效率的考虑，图着色算法还应优先溢出对别的节点着色影响最大的节点。对于节点 v_i，其对相邻节点的影响可用 $\sum_{<v',v_i>\in E} alias(v_i, v')$ 来评估。最终，可以通过以下启发式函数评估节点 v_i 的溢出优先级:

$$PerfSpillPrior(v_i) = \frac{\sum_{<v',v_i>\in E} alias(v_i, v')}{AvgPB_i} = \frac{s_i \times \sum_{<v',v_i>\in E} alias(v_i, v')}{(l_{mem} - l_{spm}) \times f_i - TR_i \times fc_i} \tag{4.5}$$

算法 4.5 SearchTemporalReg (v_i, $nColor(v_i)$)

输入：“潜在溢出”节点 v_i，v_i 的相邻节点分配的颜色集合 $nColor(v_i)$

输出：分配给 v_i 的临时寄存器 TR

```
1: Put registers in nColor(v_i) into nColorArray and sort them in ascending order of address; //nColorArray 是伪寄存器类型数组
2: Let mStartAddr be the start address of SPM;
3: Let s_i be the size of v_i (in bytes);
4: Create temporal register temReg;
5: if nColorArray[0].startAddr ≠ mStartAddr then
6:     endFlagAddr = mStartAddr + s_i;
7:     if nColorArray[0].startAddr ≥ endFlagAddr then
8:         temReg.startAddr = mStartAddr;
9:         temReg.endAddr = nColorArray[0].startAddr;
10:        TRset = TRset ∪ {temReg}; //TRset 是大小可以存放 v_i 的临时寄存器的集合
11:    end if
12: end if
13: for each j from 0 to size of nColorArray do
14:    if nColorArray[j].endAddr = nColorArray[j + 1].startAddr then
15:        continue;
16:    else
17:        endFlagAddr = nColorArray[j].endAddr + s_i;
18:        if nColorArray[j + 1].startAddr ≥ endFlagAddr then
19:            temReg.startAddr = nColorArray[j].endAddr;
20:            temReg.endAddr = nColorArray[j + 1].startAddr;
21:            TRset = TRset ∪ {temReg};
22:        end if
23:    end if
24: end for
25: if TRset ≠ φ then
26:    TR = the temporal register with minimal size in TRset;
27:    return TR;
28: else
29:    return null;
30: end if
```

该函数表明节点占用的空间越大、获得的性能收益越少、对相邻节点可着色性影响越大，则该节点的溢出优先级越高。

4.3.6 变量脆弱性评估

面向性能需求的图着色分配过程没有考虑系统的可靠性需求。PPS 技术需要对已分配的变量进行脆弱性分析，然后基于分析结果将较为脆弱的变量调整到有 ECC 保护的 SPM 中，以达到充分利用有保护的 SPM、提高系统可靠性的目的。本小节先介绍变量脆弱性的评估方法。

对于 SPM 中存放的变量来说，其脆弱性首先和其在 SPM 中的访问模式有关。如图4.5所示，一个变量从分配到 SPM 开始，将会经历多次读写。最后一次访问后，如果变量已经被修改成“脏块”，则会被写回（writeback）到片外存储（如图4.5(a) 所示），否则变量会继续驻留相应的 SPM 空间直到被其它变量取代（如图4.5(b) 所示）。变量驻留存储期间，只有在 allocate—read、write—read、read—read、read—writeback 和 write—writeback 区间是活跃的，在其它区间则是死的。显然，变量只有在活跃区间发生的故障才有可能对系统可靠性产生影响。而且，变量的活跃区间越长，变量对系统可靠性的影响越大，即变量越脆弱。此外，变量的大小同样影响其脆弱性，变量占用的存储空间越大，则变量中出现故障的概率也越大。变量的活跃区间和变量的大小决定了系统其它部位从存储空间读取正确数据的概率，因此是评估变量脆弱性必须考虑的因素。

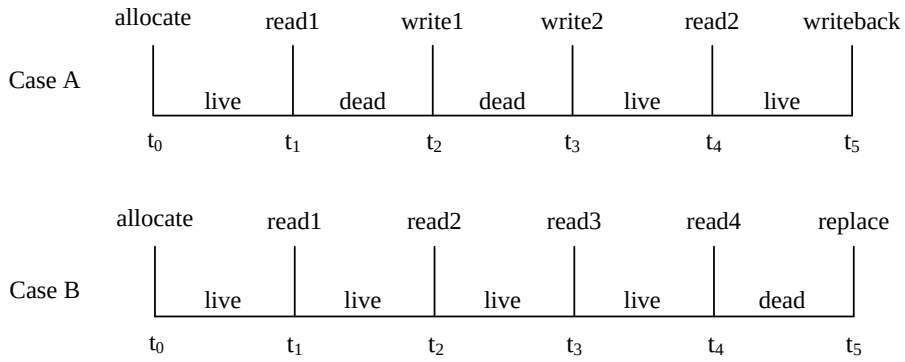


图 4.5 变量的存储访问模式

另一方面，由于在体系结构级和应用级还存在大量的故障屏蔽效应，即便存储空间的故障导致程序读取的数据是错误的，程序的最终结果也未必是错误的，所以应从系统级的角度评估变量的脆弱性。最终，变量 v_i 在一个活跃区间 (t_k, t_{k+1}) 的脆弱性可以用等式 (4.6) 计算:

$$V(t_k, t_{k+1}) = len_i \times (t_{k+1} - t_k) \times P_{sdc}(I_{k+1}) \tag{4.6}$$

其中，len_i 表示 v_i 的位数，$P_{sdc}(I_{k+1})$ 表示 t_{k+1} 时刻执行的变量访问操作 I_{k+1} 在获取错误数据后导致程序出现 SDC 的概率。

如果变量的活跃区间均是不连续的，则变量的脆弱性即各个区间脆弱性的总和。但是，如果存在多个连续的活跃区间，则先执行的活跃区间中的故障有可能传播到后续的活跃区间中。在这种情形下，需要考虑故障跨区间传播的影响。例如图4.5(a) 中变量 v_i 的脆弱性 Vul_i 可以基于以下等式计算：

$$\begin{aligned} Vul_i = &(t_1 - t_0) \times P_{sdc}(read1) \times len_i + (t_4 - t_3) \times P_{sdc}(read2) \times len_i + (t_5 - t_4) \\ &\times P_{sdc}(writeback) \times len_i + (t_4 - t_3) \times P_{mask}(read2) \times P_{sdc}(writeback) \times len_i \end{aligned} \tag{4.7}$$

其中，最后一项即为活跃区间 (t_3, t_4) 中的故障传播到活跃区间 (t_4, t_5) 中产生的脆弱性。$P_{mask}(read2)$ 表示 $read2$ 读取的数据中的错误被程序屏蔽的概率。变量读取操作的 P_{sdc} 和 P_{mask} 参数需要先通过小规模故障注入获取。

4.3.7 提高可靠性的再分配过程

基于对 SPM 中已分配变量的脆弱性评估结果，PPS 需要通过变量再分配调整这些变量在有保护和无保护的 SPM 中的分布，以达到兼顾系统性能和可靠性的目的。再分配过程需要进行两遍图着色分配，其总体流程如图 4.6所示。第一遍图着色针对 ECC 保护的 SPM 进行再分配，目标是基于脆弱性评估的结果，尽量将比较脆弱的变量分配到 ECC 保护的 SPM 中。第二遍图着色则针对第一遍中剩余的变量，将其中性能收益较大的变量分配到无保护 SPM 中。图中 Simplify、Assign 和 Post-assign 步骤的操作和提高性能的图着色过程类似，只是选择“潜在溢出”节点时的启发式策略不同。图中其它步骤的具体操作如下：

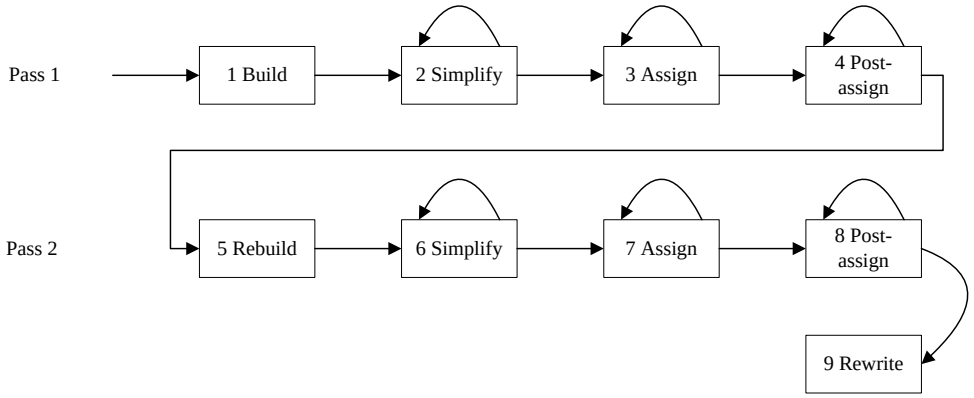


图 4.6　提高可靠性的图着色再分配过程

1. Build：为提高性能的图着色过程中已分配的变量构建冲突图 $IG(V, E)$，并将每个变量 v 的 $availr(v)$ 值赋为和其对齐大小相同的有 ECC 保护的伪寄存器集合。

2. Rebuild：将第一遍再分配中已分配颜色的变量及其邻接的边从冲突图中删除。冲突图中剩余的节点均是第一遍产生的“溢出”节点。将每个“溢出”节点 v 的 $availr(v)$ 值赋为和其对齐大小相同的、无保护 SPM 中的伪寄存器集合。
3. Rewrite：对于被重新分配 SPM 空间的变量，修改其在 SPM 中的地址。若第二遍再分配后出现新的“溢出”节点，则需要修改程序代码将该类节点保留在片外存储中。

上述两遍再分配过程中，需要基于启发式策略选择部分节点作为“潜在溢出”对象。第一遍再分配针对的是 ECC 保护的 SPM，启发式函数的设计基于以下两个方面的考虑：

- 出于可靠性收益考虑，应优先溢出脆弱性小的节点，因为这些节点分配到 ECC 保护的 SPM 中获得的可靠性收益较少。v_i 的平均可靠性收益 $AvgRB_i$ 的计算公式如下：

$$AvgRB_i = \frac{Vul_i}{s_i} \tag{4.8}$$

其中，s_i 表示变量 v_i 的大小。

- 出于着色效率的考虑，应优先溢出对别的节点着色影响最大的节点。对于节点 v_i，其对相邻节点的影响可用 $\sum_{<v',v_i>\in E} alias(v_i, v')$ 来评估。

最终，可以通过以下启发式函数评估节点 v_i 的溢出优先级：

$$ReliS\,pillPrior(v_i) = \frac{\sum_{<v',v_i>\in E} alias(v_i, v')}{AvgRB_i} = \frac{s_i \times \sum_{<v',v_i>\in E} alias(v_i, v')}{Vul_i} \tag{4.9}$$

该函数表明节点占用的空间越大、获得的可靠性收益越少、对相邻节点可着色性影响越大，则该节点的溢出优先级越高。第二遍再分配的目标是从第一遍再分配“溢出”的变量中，选取性能收益比较大的节点，分配到无保护的 SPM 中，优先级评估函数和提高性能的图着色过程中使用的启发式函数相同。

4.4 实验评估

本节在 SimpleScalar 平台 [105] 下通过模拟实验评估了 PPS 技术的效果。实验基于改造后的 SimpleScalar 模拟器获取数组变量的访问频率信息，并进行变量脆弱性分析。实验选取了五个测试程序作为测试用例，包括三个 Media benchmark[123] 测试程序 g721decode、toast 和 untoast，一个 Mibench benchmark[106] 测试程序 rijndael 和一个 WCET benchmark[124] 测试程序 adpcm。这些测试程序中包含较多的数组变量，方便验证 PPS 技术的效果。

性能收益评估需要用到四个模拟器参数：启动传输延迟 c_s，传输一个字节的延迟 c_t，对片外存储进行一次访问需要的延迟 l_{mem} 和对 SPM 进行一次访问需要的延迟 l_{spm}。实验中这些参数的配置为：$c_s = 20$，$c_t = 1$，$l_{mem} = 20$，$l_{spm} = 1$。实验分为两个部分，分别是性能评估和可靠性评估。为了和传统的 SPM 分配技术进行性能及可靠性方面的比较，实验也对4.2节所述的 SPM 分配技术 MC[113] 进行了评估。

4.4.1 性能评估

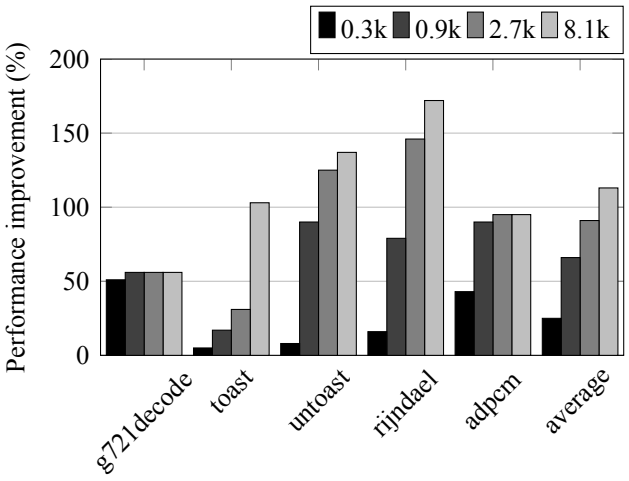


图 4.7　系统应用 SPM 后的性能提升

实验首先评估了应用 PPS 技术进行 SPM 分配后的系统性能，并将其和没有使用 SPM 的系统进行比较。实验分别配置了 0.3k、0.9k、2.7k 和 8.1k 四种 SPM 大小，无保护的 SPM 和 ECC 保护的 SPM 的大小比例固定为 2:1。性能比较的结果如图4.7所示。应用 SPM 的系统相比没有使用 SPM 的系统具有明显的性能优势，当 SPM 大小为 0.3k、0.9k、2.7k 和 8.1k 时，平均性能提升分别达到了 24.6%、66.4%、90.6% 和 112.6%。其中，数组占用空间较大、访问较为频繁的应用，性能提升更为明显。例如，rijndael 程序在系统引入 8.1k 的 SPM 空间后，性能提高了 172%。图中一些程序的性能没有随着 SPM 空间的增大而一直提高，例如 g721decode。出现这种情况的原因是，SPM 大小增加到特定大小后，待分配数据已经可以全部存放到 SPM，系统性能已经达到最优。从图中还可以发现，基于图着色实现的动态分配，可以有效地提高 SPM 的空间利用率。例如，如果采取静态分配，g721decode 需要 1.1k 的空间存放所有的待分配数组变量，而采取动态分配，则只需要 0.4k 的 SPM 空间，大大减少了存储空间开销。

实验还在 0.9k、2.7k 和 8.1k 三种 SPM 配置下，比较了 PPS 技术和 MC 技术的性能。由于 PPS 技术在图着色过程中对 SPM 分配后的碎片进行了搜索利用，

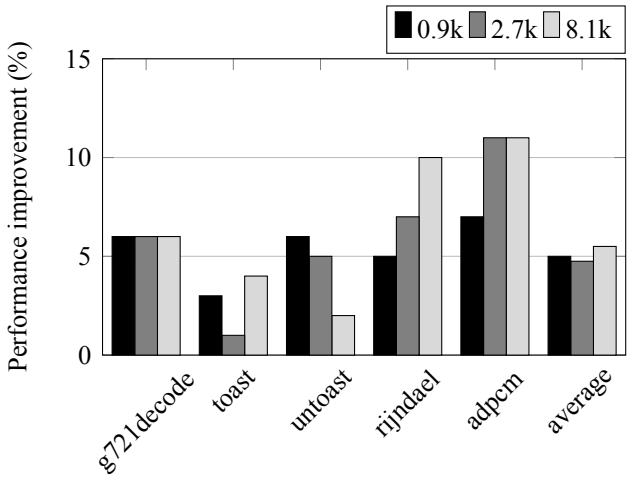


图 4.8　PPS 技术相对 MC 技术的性能提升

其 SPM 空间利用率比 MC 技术更高，因此 PPS 技术比 MC 技术具有更好的性能。图4.8给出了 PPS 技术相对 MC 技术的性能提升幅度。在三种 SPM 配置下，PPS 技术的性能平均比 MC 技术提高了 5%、4.75% 和 5.5%。不同大小的伪寄存器之间的别名关系是形成存储碎片的主要原因，因此，伪寄存器的种类越多，也就越容易出现存储碎片。但是，存储碎片能否利用，还要由待分配变量的大小决定。从这个角度考虑，如果程序中的待分配变量较小，则更容易利用存储碎片提高性能。一个典型的例子是 adpcm 程序，由于 PPS 技术在图着色过程中充分利用存储碎片保存“潜在溢出”的节点，程序性能比 MC 技术提高了 7%—11%。

4.4.2　可靠性评估

MC 技术仅面向系统的性能需求进行 SPM 分配，PPS 技术则兼顾系统的可靠性需求。实验基于脆弱性分析比较 PPS 技术和 MC 技术的系统可靠性。首先，基于公式 (1.5) 可以分别计算两种技术下 SPM 的 AVF：

$$AVF_{SPM}^{mc} = \frac{\sum vul_i^{mc}}{SPM\ size \times T_{mc}} \tag{4.10}$$

$$AVF_{SPM}^{pps} = \frac{\sum vul_j^{pps}}{SPM\ size \times T_{pps}} \tag{4.11}$$

其中，T_{mc} 和 T_{pps} 分别表示 MC 技术和 PPS 技术下程序的执行时间，$\sum vul_i^{mc}$ 表示被 MC 技术分配到 SPM 中的变量的脆弱性总和，$\sum vul_j^{pps}$ 表示被 PPS 技术分配到

无保护 SPM 中的变量的脆弱性总和。然后基于公式 (2.9) 可以计算出各自的可靠性：

$$MWTF_{mc} = \frac{1}{\lambda_{spm}} \times \frac{1}{AVF_{SPM}^{mc} \times T_{mc}} = \frac{1}{\lambda_{spm}} \times \frac{1}{T_{mc} \times \frac{\sum vul_i^{mc}}{SPM\ \ size \times T_{mc}}} = \frac{1}{\lambda_{spm}} \times \frac{SPM\ \ size}{\sum vul_i^{mc}} \tag{4.12}$$

$$MWTF_{pps} = \frac{1}{\lambda_{spm}} \times \frac{1}{AVF_{SPM}^{pps} \times T_{pps}} = \frac{1}{\lambda_{spm}} \times \frac{1}{T_{pps} \times \frac{\sum vul_j^{pps}}{SPM\ \ size \times T_{pps}}} = \frac{1}{\lambda_{spm}} \times \frac{SPM\ \ size}{\sum vul_j^{pps}} \tag{4.13}$$

基于可靠性评估，PPS 技术相比 MC 技术实现的可靠性提高可以采用以下公式计算：

$$ReliImprove = \frac{MWTF_{pps}}{MWTF_{mc}} - 1 = \frac{\frac{1}{\lambda_{spm}} \times \frac{SPM\ \ size}{\sum vul_j^{pps}}}{\frac{1}{\lambda_{spm}} \times \frac{SPM\ \ size}{\sum vul_i^{mc}}} - 1 = \frac{\sum vul_i^{mc}}{\sum vul_j^{pps}} - 1 \tag{4.14}$$

图4.9给出了 PPS 技术相对于 MC 技术实现的可靠性提高。可靠性评估实验将 SPM 的大小固定在 0.9k。PPS 技术将无保护 SPM 和 ECC 保护 SPM 的大小比例分别设定为 4:1、3:1 和 2:1，以比较不同的 SPM 保护力度下系统的可靠性。从图中可以看出，PPS 技术基于脆弱性分析进行 SPM 分配，可以充分地利用 ECC 保护的 SPM 提高系统可靠性。在设定的三种部分保护比例下，系统的可靠性相比 MC 技术平均分别提高了 45.8%、53.6% 和 61%。对于变量脆弱性差别较大的程序，部分保护 SPM 提供的可靠性收益更为明显。例如，rijndael 在三种保护比例下的可靠性提高分别达到了 55.3%、64.9% 和 72.1%。

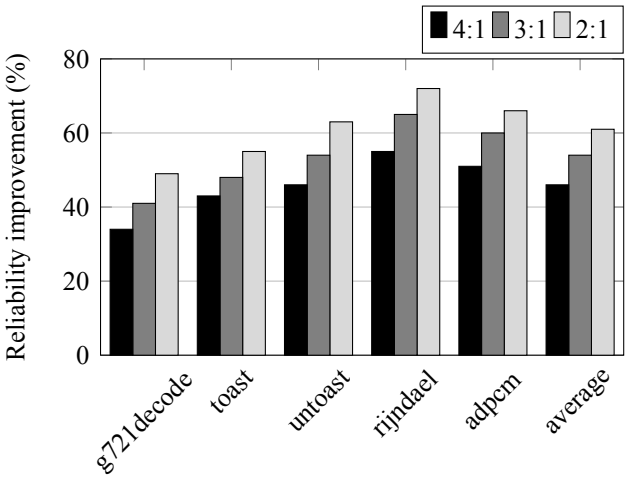


图 4.9　PPS 技术相比 MC 技术的可靠性提高

综合以上分析结果，基于部分保护的 SPM 存储结构和多次图着色分配算法，PPS 技术在确保系统性能的同时，有效降低了 SPM 中瞬时故障对系统可靠性的影

响。由于 PPS 技术的目标是提高系统性能的同时兼顾可靠性，本节没有对 ECC 保护 SPM 引入的功耗开销进行评估。但是，鉴于只有部分 SPM 空间应用 ECC 保护，相比采用 ECC 技术保护整个 SPM 空间，PPS 技术的功耗开销仍然是较低的。

4.5 本章小结

本章针对片上 SRAM 中瞬时故障引发的可靠性问题，提出了一种低代价的 SPM 保护技术 PPS。PPS 技术对 SPM 进行部分保护，然后根据待分配变量的大小将 SPM 划分成伪寄存器，并改造传统的寄存器图着色分配算法，最终通过多次图着色过程达到提高性能和可靠性的目的。和传统的 SPM 分配方法相比，PPS 技术在确保系统性能的同时，有效降低了 SPM 中瞬时故障对系统可靠性的影响。另外，由于采取的是部分保护方法，PPS 技术避免了完全 ECC 保护需要的大量开销。

第五章　基于程序分析的故障注入技术

5.1　引言

为了分析瞬时故障的影响和评估容错技术的效果，需要研究面向瞬时故障的可靠性分析技术。系统可靠性的研究已经有很长的历史，但是传统的可靠性研究主要针对系统硬件或软件自身固有的缺陷进行建模分析。由于瞬时故障的特征和这些系统缺陷不同，传统的可靠性评估技术无法直接用来评估瞬时故障的影响。针对瞬时故障对可靠性的影响，目前主流的分析方法是故障注入方法。故障注入技术采取模拟的方法人为地在系统中注入故障，然后根据系统运行结果分析故障对可靠性的影响。这种技术成本低廉，实施过程易于控制（可以方便地设置注入故障的位置、时机、故障率等），而且可以获得较为精确的分析结果。

但是，故障注入技术面临如何权衡模拟速度与分析精度（或真实度）之间关系的挑战 [54]。由于故障在系统运行时发生的状态空间非常庞大，注入所有的故障将会使模拟时间变得无法控制。而直接减少注入故障的数量，则可能使分析的精度无法接受。鉴于已有的故障注入技术还不能有效地解决上述问题，本章提出了一种新的故障注入框架 SmartInjector，该框架可以在大幅度降低模拟时间开销的同时保持较高的分析精度。SmartInjector 首先基于程序分析从故障空间中删除等价类故障和结果确定型故障。等价类故障是指发生在相似的数据流或控制流上下文环境中的故障。这类故障往往会导致系统产生相同的反应，因此只需要从同一等价类中选取一个代表进行模拟注入即可，而同一等价类中其它指令中的故障则可以从故障空间中删除。结果确定型故障则是指那些可以通过程序分析（不需要实际模拟运行）就可以确定系统反应的故障。此外，为了获取每次故障模拟的结果，传统的技术往往需要将程序运行至结束，SmartInjector 则首次开发了一种故障结果预测技术，可以通过预测并在程序运行结束前提前判断故障模拟的结果，减少单次模拟的时间开销。基于上述故障删除技术和故障结果预测技术，SmartInjector 所需的模拟时间仅仅是模拟完整的故障空间所需时间的 0.15%，同时其可靠性分析精度达到了 96.3%。

本章以下内容组织如下：第5.2节介绍已有的故障注入方法，并和 SmartInjector 进行比较；第5.3节提出 SmartInjector 故障注入框架，并详细阐述其原理、方法；第5.4节介绍 SmartInjector 框架的实验评估方法；第5.5节给出实验评估的结果，并对 SmartInjector 的效果进行分析；最后，第5.6节对本章内容进行总结。

5.2 相关工作

针对如何权衡模拟速度与精度之间关系的问题，现有的故障注入研究进行了初步探索。CriticalFault[58] 技术利用 AVF 分析 [44] 从故障空间中删除在系统运行过程中将会被屏蔽的故障，以达到压缩故障空间、注入更多有效故障的目的。CriticalFault 不足之处在于，仅靠删除被屏蔽的故障所能获得的模拟加速是有限的，剩余的故障仍会导致大量的模拟开销。SimPLFIED[125] 采用一种强大的符号化执行方法抽象程序中变量的各种错误状态。具体方法是先在程序中的故障位置注入一个符号化错误来表示所有可能的故障，然后利用模型检验技术进行抽象执行，以分析符号化错误所有可能的传播路径，并在这些路径上分析程序受到的影响。SimPLFIED 的思路是减少在一个故障位置注入的故障值的数量，而 SmartInjector 框架则尽可能减少需要注入故障的位置，因此这两种技术是互补的关系。SimPLFIED 的缺点在于，这种基于模型检验的技术很难应用于大规模程序的分析。

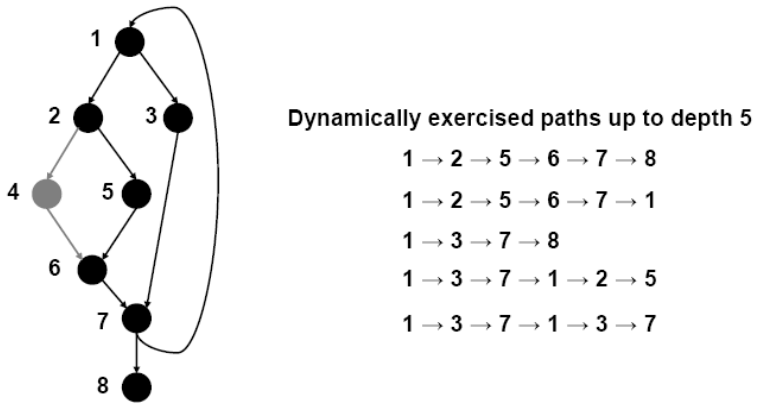


图 5.1 Relyzer 控制流等价示例

已有技术中，Relyzer[33] 技术和 SmartInjector 一样采取了故障删除方法压缩故障空间。Relyzer 主要依赖提出的控制流等价策略进行故障删除。图5.1 给出了一个控制流等价的例子 [33]。控制流等价策略首先列举出所有从基本块 1 开始的长度不超过 n（n 是可配置参数，此例子中取值为 5）的动态路径，然后在程序的执行轨迹（trace）中搜索这些路径是否重复出现。如果一条路径 p 有多个动态实例，则认为这些实例中具有相同 PC（Program Counter）地址的动态指令是控制流等价的。每个等价类中只需要随机选取一条动态指令进行故障注入模拟即可，其结果可以代表同类别的其它动态指令。通过这种策略可以大量删除需要故障注入的动态指令。SmartInjector 在 Relyzer 基础之上实现了更有效的故障删除，二者区

别体现在以下三个方面：（1）Relyzer 的控制流等价策略只考虑了指令的控制流上下文环境，忽略了数据流关系带来的影响，SmartInjector 则结合数据流分析提出了更有效的控制流等价删除策略；（2）Relyzer 等价类策略只开发了同一个静态指令的不同动态指令间的等价性，SmartInjector 则通过提出的数据流等价策略，开发了不同静态指令的动态指令间的等价性；（3）SmartInjector 提出了基于故障结果预测来降低单次故障模拟开销的方法，而 Relyzer 的故障模拟则需要运行至结束才能获取故障的结果信息。

5.3 SmartInjector 故障注入框架

5.3.1 SmartInjector 的总体结构

SmartInjector 的总体结构如图5.2所示。首先，SmartInjector 需要利用体系结构模拟器（SimpleScalar[105]）获取程序在给定输入下的一个执行轨迹（trace）。执行 trace 中的每一个动态指令被视为一个故障位置。在每个故障位置上，SmartInjector 考虑注入的故障包括寄存器故障和指令码故障。由于指令的源寄存器操作数中的故障最终会传播到目的寄存器中，SmartInjector 仅向目的寄存器中注入故障。程序执行 trace 中每个故障位置所包含的故障共同组成程序的初始故障空间（Fault Space）。

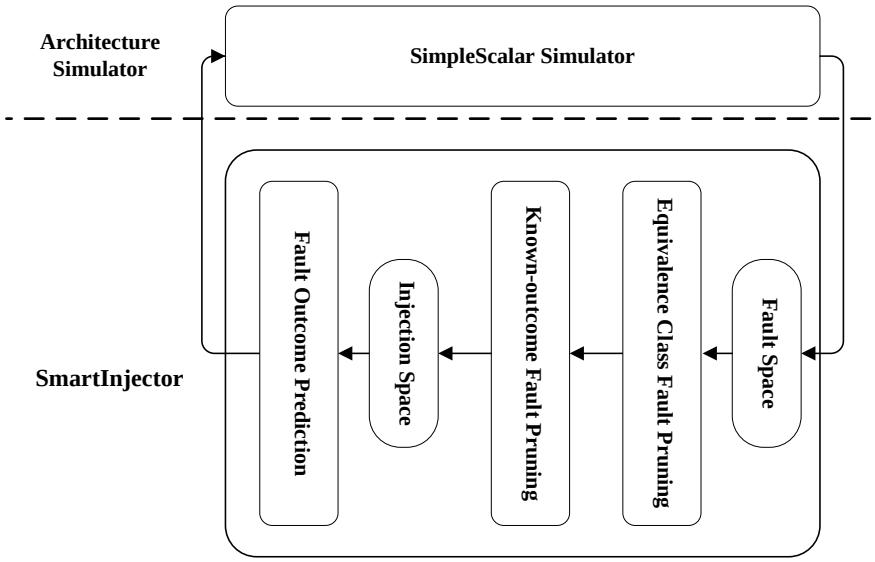


图 5.2 SmartInjector 的总体结构

通常情况下，程序的一个 trace 对应的初始故障空间是非常庞大的。假设每个寄存器为 32 位，每个指令为 64 位，则仅仅 1000 条动态指令需要注入的故障就将达到 $96 \times 1000 = 96000$ 个。庞大的故障空间将会使故障模拟的时间开销无法承受(或者需要大量的计算资源才能在合理时间内完成故障模拟)。为了减少故障注

入的开销，SmartInjector 先基于程序分析进行等价类故障删除（Equivalence Class Fault Pruning）和结果确定型故障删除（Known-outcome Fault Pruning），通过从初始故障空间中删除没有必要注入的故障获取较小的注入故障空间（Injection Space）。每次模拟运行只注入一个故障，注入故障后，SmartInjector 还利用提出的故障结果预测方法（Fault Outcome Prediction）减少单次故障模拟的运行时间。下面分别介绍 SmartInjector 的组成技术。

5.3.2 控制流等价故障删除

等价类故障删除策略包括控制流等价和数据流等价两个部分。控制流等价基于的原理是，对同一条静态指令处于相似控制流上下文环境中的不同动态指令注入故障后，这些动态指令对程序的影响很有可能是相同的，因此可以将这些动态指令划为一个等价类。故障注入模拟时只需要从中随机选取一条动态指令作为其它指令的引导指令（Pilot Instruction）进行实际的故障模拟即可。控制流等价策略通过开发同一静态指令的不同动态指令之间的等价性，可以大幅减少需要注入故障的动态指令。

SmartInjector 的控制流等价类划分过程如算法5.1所示。算法首先针对程序中的循环结构列举出循环内所有可能的执行路径（第 1-2 步），并在程序 trace 中搜索这些路径的动态实例（第 3-4 步），然后针对同一路径的不同动态实例，将出现在其中的具有相同 PC 的动态指令划为等价类（第 5 步）。上述过程是控制流等价划分的基本步骤 [33]，为了进一步减少不必要的指令类别，SmartInjector 需要基于数据流分析进行优化。优化策略的基本思路是，如果一条指令 *I* 和分支结构 *Br* 中的指令无数据流关系（即 *I* 的结果在 *Br* 结构的 *else* 和 *then* 部分均不会被使用或重新定值），则可以认为指令 *I* 对程序结果的影响和 *Br* 的跳转方向无关，因此不需要基于该分支确定的程序路径对指令 *I* 的动态实例进行分类。算法第 8-11 步先在 *Br* 所在的路径中寻找和 *Br* 存在数据流关系的指令，然后对除这部分指令以外的指令进行等价类合并。

控制流等价优化的例子如图5.3所示。在此控制流图中，分支结构确定了两条可能的程序执行路径 B1→B2 和 B1→B3。由于指令 I1 在 B1→B2 路径上被 I3 重新定值，在 B1→B3 路径上却不会被使用或定值，所以 I1 发生故障后对程序的影响与分支的跳转方向有关，因此需要将 I1 对应的动态指令基于该分支所确定的两条路径划分为两类。相反，由于指令 I2 和分支结构中的其它指令没有任何数据流关系，I2 注入故障后对程序的影响与该分支的跳转方向无关，因此不需要将 I2 的动态指令划分为两个等价类。

控制流等价优化可以基于静态数据流分析进行，其主要过程为：首先，分析分支结构 *Br* 内被读或写的变量，用集合 *AVset*(*Br*) 表示。然后，在 *Br* 之前的基

算法 5.1 CFEquivalAnalysis()

输入： 程序 *P* 的控制流图 *CFG* 和动态 trace *T*

输出： 动态指令的等价类划分结果

```
1:  for each loop L ∈ P do
2:      Enumerate all paths in L;
3:      for each path p ∈ L do
4:          Search the instances of p in trace T;
5:          Divide the dynamic instructions with the same PC from the instances of
            p into an equivalence class;
6:      end for
7:  end for
8:  for each branch Br ∈ L do
9:      Perform data-flow analysis to find the instructions which can affect Br;
10:     Merge the equivalence classes divided based on Br for the remaining instruc-
        tions;
11: end for
```

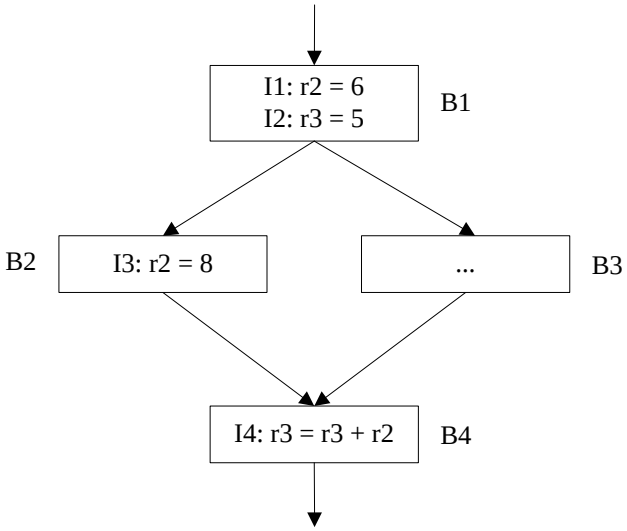


图 5.3　控制流等价优化示例

本块中分析提取 *AVset*(*Br*) 中变量的后向程序切片，即能够计算出 *AVset*(*Br*) 中变量值的指令序列。提取程序后向切片可以基于标准的数据流分析实现（即到达—定值分析）。定义 *RdIn*(*B*) 为到达基本块 B 入口处的定值集，*RdOut*(*B*) 为到达 B 出口处的定值集。定义 *Gen*(*B*) 为出现在基本块 B 中并到达 B 的出口处的定值。

对于基本块 B 中定值的每个变量 v，$Kill(B)$ 中包含除了出现在 $Gen(B)$ 中的定值点以外的其它所有 v 的定值点。基于上述定义，数据流分析方程为：

$$RdIn(B) = \bigcup_{B_i \in P(B)} RdOut(B_i) \tag{5.1}$$

$$RdOut(B) = (RdIn(B) - Kill(B)) \cup Gen(B) \tag{5.2}$$

其中 $P(B)$ 为 B 的前驱基本块集合。最后，将后向切片中的指令标记为需要基于 Br 确定的路径进行分类的指令，对于 Br 所在路径中的其余指令，则将其已经基于 Br 划分的类别重新合并。

5.3.3 数据流等价故障删除

在控制流等价删除后的剩余故障中，SmartInjector 需要进一步应用数据流等价故障删除策略。数据流等价策略主要基于指令间的数据流传播关系的相似性进行等价类划分。已有的等价类策略针对的都是同一静态指令的不同动态指令间的等价性，本节提出的数据流等价策略则首次开发了不同静态指令的动态指令间的等价性。数据流等价策略主要基于动态分析实现，下面先介绍分析时用到的程序模型及基本定义：

定义 5.1 动态数据流图（Dynamic Data-flow Graph，DDG）：基于收集的程序执行 trace 可以构建出一个有向图 $DDG(V, E)$, 其节点集合和边集分别为:

- 节点集合 V 中的每个元素表示程序 trace 中的一条动态指令;
- 边集 E 表示节点间的数据依赖关系。若指令 v 需要使用指令 u 的结果，则有边 $< u, v > \in E$。

定义 5.2 专门前驱节点（Special Precursor，SP）：如果有边 $< u, v > \in E$，则称 u 为 v 的前驱节点。特别地，当从 u 出发的边只有 $< u, v >$ 时, 即 u 的出度为 1，则称 u 为 v 的专门前驱节点。

定义 5.3 专门起源节点（Special Origin Node，SON）：若节点 u 和 v 满足下列条件，则称 u 是 v 的专门起源节点：

- u 是 v 的一个专门前驱，或
- u 是 v 的专门前驱的专门起源节点。

节点 v 的所有专门起源节点可以用指令链 $SO(v)$ 表示。在 DDG 中寻找节点 u 的专门前驱的过程很简单，只需要找到其出度为 1 的前驱节点即可。u 的专门起源节点则可以利用算法5.2进行分析。由于 u 和其专门起源节点构成的 DDG 子图实际上是一个倒置的树形结构，所以可以从 u 开始采取逆向深度优先的迭代搜索方法寻找 u 的专门起源节点。每次迭代对 u 的前驱节点进行逐个分析。如果前驱

节点 p 出度为 1，则先将 p 加入 $SO(u)$ 链表（第 2-4 步），然后分析 p 的专门起源节点并将结果添加到 $SO(u)$ 中（第 5-6 步）。

算法 5.2 SearchSON (u)

输入： 程序 *DDG* 中的节点 u
输出： u 的专门起源节点链表 $SO(u)$

```
1: SO(u) = φ;
2: for each precursor p of u in DDG do
3:         if outdegree(p) = 1 then
4:                 Add p into the instruction list SO(u);
5:                 SO(p) = SearchSON(p);
6:                 Add SO(p) into SO(u);
7:         end if
8: end for
9: return SO(u);
```

指令的数据流上下文特征是进行数据流等价划分的主要依据。目前，Smart-Injector 的数据流等价策略主要考虑以下三种情形：

（1）若一条动态指令 I 在 *DDG* 中存在专门前驱 P，并且 I 属于错误传播能力强的指令，则可以认为 I 和 P（对应的指令）是等价的，只需选择其中一个指令注入故障即可。错误传播能力强的指令是指屏蔽故障的概率较低的指令，例如数据传送、加减法、循环移位以及逻辑取反、异或、同或指令等。向指令 I 的专门前驱 P 的结果中注入故障后，一定会导致 I 的结果也出现错误，而且只会导致指令 I 的结果错误，因此可以近似认为二者的故障注入结果是相同的，可以将其划为等价类。

（2）若程序存在多个输出 o_1，o_2，$\cdots$，o_n，并且这些输出操作在 *DDG* 中的专门起源节点序列 $SO(o_1)$，$SO(o_2)$，$\cdots$，$SO(o_n)$ 遵循相同的数据传播模式，则可以认为不同输出操作的专门起源节点序列中对应位置的指令是等价的。两个指令序列的数据传播模式相同是指二者内部的指令操作相同，且相互间的依赖关系相同，唯一不同的是指令序列中的操作数不同。图5.4给出了两个数据流传播模式相同的指令序列对应的数据流图。显然，若这两个指令序列能够计算出两个不同的输出数据，则两个序列中对应的指令在注入故障后对程序结果的影响是相似的，因此可以把指令 9、10、11、12 分别与指令 15、16、17、18 划分为等价类。这种情形在数据处理类应用中是常见的，图5.5(a) 给出了一个实际的代码例子（图中数组 *blockA*、*blockB* 和 *blockC* 均为程序的输出变量）。

（3）为了提高系统性能，编译过程中通常会对循环进行展开，展开后的循环内通常包含多个和原循环相同的循环体。显然，其中一个循环体内的指令 I 与

其在其它循环体内对应的指令可以视为同一指令的多个动态实例。因此，只要这些动态指令在各自循环体内的控制流路径相同，这些指令发生故障后对程序的影响将是相似的，所以可以采取5.3.2节控制流等价的分析方法将这些指令划为等价类。图5.5(b) 给出了一个循环展开的例子，其中第一个循环实现了四次循环展开，第二个循环则完成剩余的循环遍。显然，由于原循环体内只有一条控制流路径，循环内的五条语句对应的动态指令都是等价的。

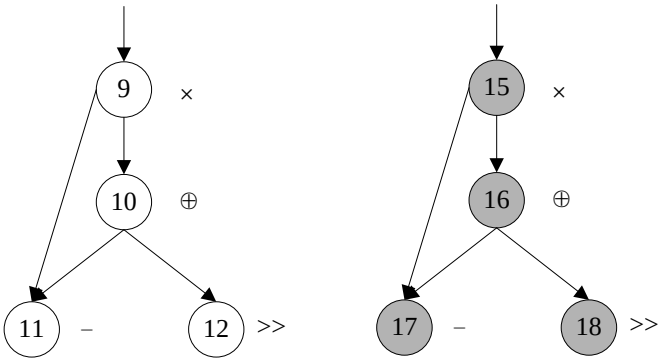


图 5.4　数据流传播模式示例

```
for (i = 0; i < 8; i++)
{
  blockA[i] = (z1 + z3) >> 3;
  blockB[i] = (z2 + z3) >> 3;
  blockC[i] = (z1 + z2) >> 3;
}
```

(a) 多输出示例

```
for (Inner = MAXSIZE; Inner > 3;
Inner -= 4)
{
  Array[Inner]   += Ntotal;
  Array[Inner-1] += Ntotal;
  Array[Inner-2] += Ntotal;
  Array[Inner-3] += Ntotal;
}
// clean-up code
for ( ; Inner > 0; Inner --)
{
  Array[Inner] += Ntotal;
}
```

(b) 循环展开示例

图 5.5　不同静态代码之间的等价性示例

5.3.4　结果确定型故障删除

部分故障对程序的影响可以基于程序分析提前确定，并不需要进行故障注入模拟。因此，SmartInjector 可以在等价类故障删除的基础上，进一步删除这些结

果确定型故障。下面给出基于程序分析确定故障类型的方法，具体分为以下几种情形：

非法指令码：指令码中的部分故障可以使指令变为非法指令，从而触发非法指令异常，该类故障可以认为是症状型故障。SmartInjector 可以通过分析从每条指令中识别出这类故障，并将其从故障空间中删除。

动态死指令：动态死指令可以分为两种类型：一种是初级动态死指令（First-level Dynamically Dead Instruction, FDD），这种指令的结果由于受动态控制流的影响没有被其它指令使用；另外一种是迁移动态死指令（Transitively Dynamically Dead Instruction, TDD），这种指令的结果仅被 FDD 指令或者其它 TDD 指令使用[44]。显然，向动态死指令的目的寄存器中注入的故障不会影响程序的运行，因此这些故障的类型确定为良性故障，可以将其从故障空间中删除。

未使用的位：程序中有部分逻辑位在执行过程中并不会被使用。例如，不少指令集格式中包含的保留位不会被使用，亚字长操作（例如单字节操作、半字长操作）中的部分位也不会被使用。这些未使用的位中的故障显然不会影响程序执行，因此可以认为是良性故障，并将其从故障空间中删除。

逻辑操作屏蔽位：若逻辑操作 u 在 DDG 中存在专门前驱 v，则 v 的目的寄存器中的故障可能会被 u 屏蔽。例如，逻辑与操作 $c = a \,\&\, 0\text{xff}$ 可以屏蔽操作数 a 中低 8 位以外出现的故障。这类被逻辑操作屏蔽的故障也是良性故障，因此可以从故障空间中移除。

访存操作：出现在访存操作的地址操作数中的故障可以导致程序发生地址越界异常（或存储保护异常）。和文献 [33] 一样，SmartInjector 先基于程序的执行剖面信息确定地址的有效范围，然后将可能导致访存越界的故障视为症状型故障，并将其从故障空间中删除。

Y 分支：已有的研究表明 [58, 126]，多达 40% 的动态分支是具有容错属性的，即程序输出的正确性不受某个分支是否跳转的影响，这种分支被定义为 Y 分支。图5.6给出了一个 Y 分支的示例。当编译器将这段代码编译成汇编代码后，if 结构将对应其分支条件变成三个分支指令。程序执行时，若第一个分支本该跳转却由于受瞬时故障的影响没有跳转，在后续两个分支有一个发生跳转的情形下，程序结果仍将是正确的。此时，第一个分支的动态实例就可以视为一个 Y 分支。显然，若条件分支 v 是 Y 分支，则 $SO(v)$ 中指令的目的寄存器中的故障均为良性故障，因而可以从故障空间删除。SmartInjector 先通过小规模的故障注入实验确定每个动态分支是否为 Y 分支。

上述所有情形中，非法指令码、未使用的位、访存操作中的结果确定型故障可以基于程序静态分析确定，其它几种情形则需要基于动态分析确定。

故障删除的相关信息都被存放在如表5.1所示的数据表中。其中，“Inst No”表示动态指令在 trace 中的编号，“PC”表示指令的地址，“Register”表示发生故障的寄存器编号（若该属性取值为“INST”，则说明故障类型是指令码故障），“Benign”、“SDC”和“Symptom”分别表示对应类型的结果确定型故障所发生的位段，“Injection”表示剩余需要实际故障注入的位段。此外，每个等价类中都含有一条引导指令，其编号被存放到表中该指令所代表的指令条目中（“Pilot Inst”列）。

```
if (val1 == et_symbol || val2 == et_symbol || val3 == et_symbol)
{
  eval_error = ERR_EXPR;
  return err_value;
}
```

图 5.6　Y 分支示例

表 5.1　故障删除信息表

Inst No	PC	Register	Benign	SDC	Symptom	Injection	Pilot Inst
1	0x4ff0	8	11-20	none	21-31	0-10	93
1	0x4ff0	INST	48-63	none	30-43,46,47	0-29,44,45	93
2	0x4ff8	5	16-31	0-15	none	none	none

5.3.5　故障结果预测

现有的故障注入研究只关注如何减少需要注入的故障数量，而没有研究减少单次故障模拟的运行时间。在已有的方案中，单次故障模拟必须运行至结束才能确定该次故障注入导致的结果（除非注入的是症状型故障）。直觉上来说，如果能够预测故障的结果并在程序结束前进行判断，将可以减少单次故障模拟的时间开销，进而降低整个故障注入的开销。SmartInjector 基于上述思路，提出了基于故障结果预测的故障注入优化方法。

由于一旦发生症状型故障，系统中的检测机制可以立即报告故障并停止程序运行，所以这类故障并不需要基于故障结果预测进行优化。SmartInjector 主要关注良性故障和 SDC 故障。首先，从 *DDG* 中选择具有屏蔽故障能力或可能导致 SDC 的指令，并将其定义为预测指令。然后，这些预测指令在 *DDG* 中的专门起源节点中的故障可被预测为良性故障或者 SDC 故障。例如，由于逻辑操作 I 可能屏蔽传播到 I 中的故障，发生在 $SO(I)$ 中指令的目的寄存器中的故障可被预测为良性故障。在故障模拟运行到预测指令时，SmartInjector 将会比较预测指令的结

果和其正确结果（通过程序的无故障运行获得）是否一致。如果比较结果一致且注入的故障被预测为良性故障，或者比较结果不一致但注入的故障被预测为 SDC 故障，则说明预测成功，可以提前结束此次故障模拟。否则说明故障结果预测失败，此次故障模拟需要运行至结束才能进行结果判定。故障结果预测需要基于动态分析进行，具体来说，SmartInjector 选取下列指令作为预测指令进行故障结果预测：

故障屏蔽操作：故障屏蔽操作 I_{mask}（例如逻辑操作、亚字长操作等）的专门起源节点中的故障传播到 I_{mask} 后，很有可能被 I_{mask} 操作屏蔽。由于这些故障被屏蔽后将不会再影响程序执行，所以故障屏蔽操作的专门起源节点中的故障可以被预测为良性故障。

输出操作：本文假设的是存储映射 I/O 模型（memory-mapped I/O），所以输出操作是指与程序输出相关的存储操作。由于输出操作的专门起源节点中的故障最终只传播到输出操作，而且很有可能直接破坏程序输出，所以可以将这些故障预测为 SDC 故障，提前进行结果判断。

分支操作：由于分支操作的专门起源节点中的故障常常不会改变分支方向，所以这些故障可以被预测为良性故障。故障注入运行时可以在分支执行后提前进行结果判断。

表 5.2　故障结果预测信息表

Inst No	SDC		Benign	
	PI No	Correct Result	PI No	Correct Result
1	129	66	7	80
2	780	51	0	0

目前，故障结果预测仅基于上述策略进行，并且只考虑寄存器故障。虽然开发更复杂的结果预测策略可以获得更大的收益，但是往往会需要更复杂的程序分析，带来较大的分析开销。而且，从5.5节的评估结果来看，上述策略已经可以覆盖大量的注入故障，并明显降低故障模拟的时间开销。表5.2给出了故障结果预测时需要使用的信息表。其中，“Inst No”表示被预测结果的动态指令编号。针对每种预测的结果（SDC 或 Benign），需要记录相应的预测指令编号“PI No”和预测指令的正确结果“Correct Result”。分支操作的正确方向用 0 或 1 表示（1 表示跳转，0 表示不跳转）。此外，由于一个指令中的故障可能传播到不同的预测指令，所以可能会有不止一个预测结果。

5.4 SmartInjector 的实现及评估

由于 SmartInjector 需要利用 SimpleScalar[105] 模拟器的功能，我们基于改造 SimpleScalar 模拟器设计、实现了 SmartInjector 框架。SmartInjector 的设计采用了模块化的设计方法，其模块设计如图5.7所示。其中，程序初步分析模块负责对输入的程序 trace 进行初步分析，包括指令解析、控制流分析、数据流分析等；故障删除模块和故障结果预测模块分别在输入程序的初始故障空间基础上，进行故障空间压缩和故障结果预测；故障注入模块针对注入故障空间中的故障进行模拟分析，包含的步骤为初始化故障注入（选择故障、设置故障注入参数等)、注入故障和结果分析。SmartInjector 主要的数据结构设计如图5.8所示，关键的数据结构定义主要包括动态指令、故障、故障删除记录等。

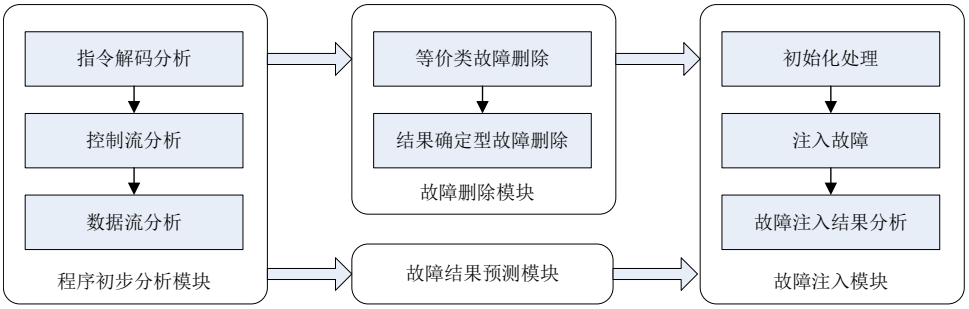


图 5.7 SmartInjector 模块设计图

SmartInjector 的评估实验采用了 8 个 Mibench[106] 程序作为测试用例，这些测试程序均被编译为 PISA 指令集。SmartInjector 采用的故障模型是发生在指令目的寄存器或指令码中的瞬时故障。由于模拟器中的寄存器为 32 位，指令码为 64 位（其中包含 16 位保留位)，所以每条动态指令最多需要 80 次故障注入。程序的 SDC 率是评估系统可靠性的关键指标。基于 SmartInjector 故障模拟的结果，SDC 率可以通过以下等式计算：

$$P(SDC) = \frac{N_{pruned}^{sdc} + N_{injected}^{sdc}}{F_{init}} \tag{5.3}$$

其中 N_{pruned}^{sdc} 表示被删除的 SDC 故障数量，$N_{injected}^{sdc}$ 表示注入故障中导致 SDC 的数量，F_{init} 表示初始故障空间的大小。

等价类故障删除技术采用启发式的策略删除故障，基于其获得的分析结果的精度需要评估。定义一条引导指令的分析精度为该指令的故障注入结果和其所代表的指令的故障注入结果的匹配程度。例如，对一条引导指令注入的一个故障导致了 SDC，而同样的故障注入到该指令的所有等价指令后，有 95% 的比例导致了 SDC，则可以认为引导指令的分析精度为 95%。为了减少评估需要的计算

开销，我们在程序中随机选取 100 条引导指令，并将这些指令的平均分析精度（针对 SDC 故障）作为等价类故障删除方法的精度。等价类故障删除技术会影响 SmartInjector 的总体分析精度。为了评估 SmartInjector 的分析精度，我们随机选取 1000 条指令进行完全的故障注入，以此结果为基准判断 SmartInjector 的分析结果的精度。

```
struct dynamicInst{                                  //动态指令
    unsigned int opcode;                             //指令操作码位段
    unsigned int operand;                            //操作数位段
    unsigned long int instPC;                        //指令PC地址
    int instNo;                                      //动态指令编号
    struct dynamicInst *pilotInst;                   //关联的pilot指令
    struct dynamicInst *dependencyInst;              //依赖的指令
    struct dynamicInst *nextInst;                    //所在trace中下一条指令
    };
struct dynamicInst *programTraceHead;                //程序trace
struct fault{
    int type;                                        //故障类型
    int bitNo;                                       //故障位
    int regNo;                                       //故障所在的寄存器号
    struct dynamicInst *inst;                        //故障发生的动态指令
    struct fault *nextFault;
    };
struct fault *injectionFaultSpaceHead;               //注入故障空间
unsigned int GPR[32];                                //通用寄存器
struct basic_block{                                  //基本块
    int block_number;                                //基本块号
    struct dynamicInst *headInst;                    //块头
    struct dynamicInst *tailInst;                    //块尾
    struct basic_block *predBlock;                   //程序流图中的前驱块
    struct basic_block *succBlock;                   //程序流图中的后继块
    };
struct faultPredictionInfo{                          //故障结果预测信息
    struct dynamicInst *inst;                        //目标指令
    struct dynamicInst *predictionInst;              //预测指令
    int predictionType;                              //预测结果类型
    int goldResult;                                  //正确结果
    struct faultPredictionInfo *nextInst;
    };
 struct faultPruningInfo{                            //故障删除信息
    int type;                                        //故障类型
    int regNo;                                       //故障所在的寄存器号
    int benignBits;                                  //benign故障的位段
    int sdcBits;                                     //sdc故障的位段
    int symptomBits;                                 //symptom故障的位段
    int injectionBits;                               //注入故障的位段
    struct dynamicInst *inst;                        //故障所在的动态指令
    struct faultPruningInfo *nextPruningInfo;
    };
```

图 5.8　SmartInjector 的关键数据结构设计图

5.5 实验结果

SmartInjector 的目标是保持可靠性分析精度的前提下尽可能减少故障模拟的时间开销，因此实验首先从模拟时间的角度评估了 SmartInjector 的整体效果，并和已有的技术进行比较，然后具体分析了构成 SmartInjector 的各个组成技术的效果，最后对 SmartInjector 的分析精度进行了评估。

5.5.1 SmartInjector 的整体效果

SmartInjector 的整体效果如表5.3所示。基于提出的故障删除和故障结果预测技术，SmartInjector 将故障模拟时间平均减少了 99.85%（相比注入整个故障空间中的故障需要的模拟时间)。其中，62.53% 的模拟开销优化来自于指令码中的故障，37.23% 的优化则与寄存器故障有关。尽管表5.3中寄存器故障的优化效果（Register 列）同时来自故障删除技术和故障结果预测技术，指令码故障的优化效果仅来自故障删除技术，但是指令码故障的优化对故障模拟的加速效果更好，主要原因是每条指令有 64 个指令码故障，却只有 32 个寄存器故障，前者在故障空间中占的比例高于后者。

表 5.3 SmartInjector 的整体优化效果

Application	Inst Code (%)	Register (%)	Total (%)
susan	60.84	38.42	99.26
dijkstra	55.41	44.58	99.99
adpcm	58.65	41.31	99.96
rijndael	67.90	32.02	99.92
basicmath	62.76	37.23	99.99
blowfish	65.23	34.72	99.95
patricia	59.87	40.12	99.99
sha	69.55	30.18	99.73
average (%)	62.53	37.32	99.85

相比已有的故障注入框架 Relyzer[33]，SmartInjector 开发了更为有效的故障删除策略，并提出了基于结果预测降低单次模拟开销的方法，因此，SmartInjector 可以更好地对故障模拟进行加速。图5.9给出了两个框架的模拟优化效果比较。评估结果表明，SmartInjector 将故障模拟时间在 Relyzer 基础上进一步减少了 36.5%。

5.5.2 SmartInjector 的单个组成技术的效果

为了最大限度地减少故障模拟需要的时间开销，SmartInjector 分别应用了等价类故障删除、结果确定型故障删除和故障结果预测技术。图5.10给出了故

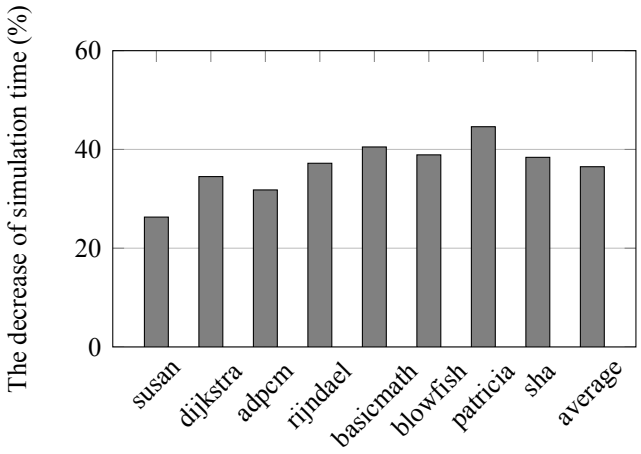


图 5.9　相比 Relyzer 的模拟加速

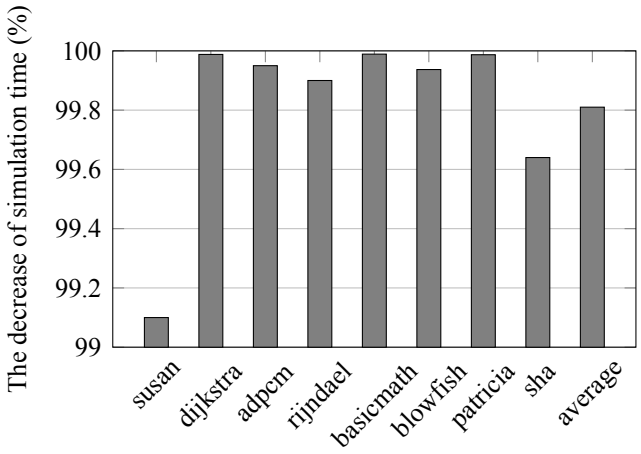


图 5.10　故障删除技术的总体优化效果

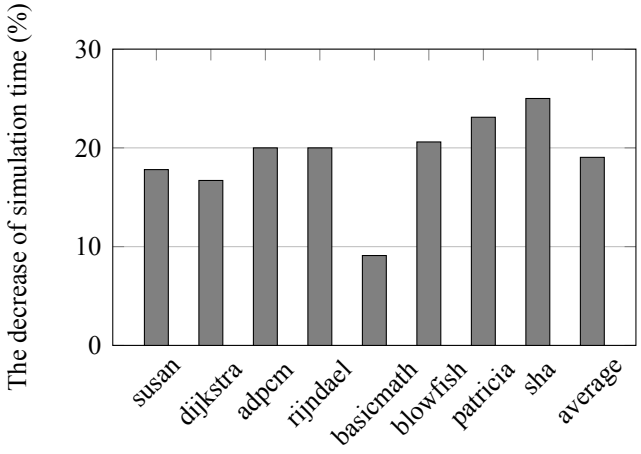


图 5.11　故障结果预测技术提供的模拟加速

障删除技术的总体效果。结果显示，故障删除技术将故障模拟时间平均减少了99.81%。对于剩余需要模拟的故障，故障结果预测技术可以提供额外的模拟加

速。如图5.11 所示，故障结果预测技术在故障删除技术取得效果的基础上，进一步将故障模拟开销减少了 19%。实验也分别评估了结果确定型故障删除、控制流等价故障删除和数据流等价故障删除技术的效果，具体结果如图5.12所示。被删除的故障中，平均 35.4% 的故障是结果确定型故障，控制流等价和数据流等价技术删除的故障则分别占 46% 和 18.6%。

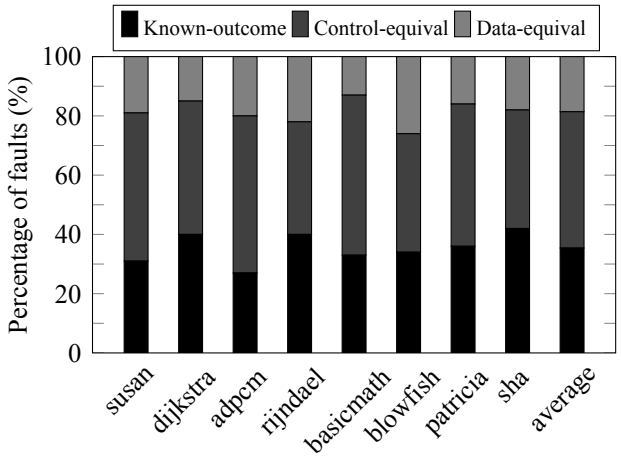


图 5.12　单个故障删除技术的效果

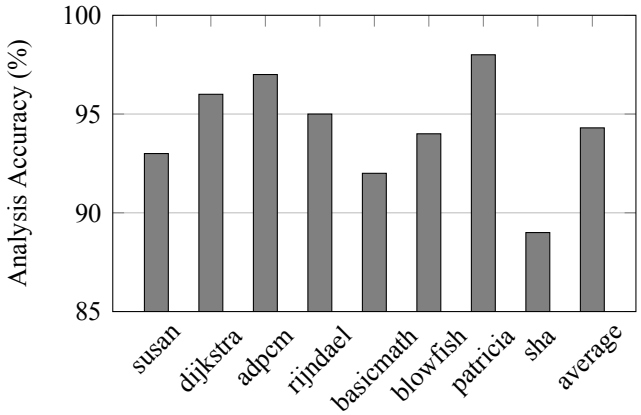


图 5.13　等价类故障删除技术的分析精度

5.5.3　SmartInjector 的精度分析

下面首先评估等价类故障删除技术的分析精度。图5.13给出了等价类故障删除技术的精度评估结果。结果显示，等价类故障删除技术的平均分析精度达到了 94.3%。考虑到等价类故障删除技术只是 SmartInjector 的组成部分，需要进一步分析 SmartInjector 的整体精度，结果如图5.14所示。除了被删除的等价类故障，其它故障的结果均是通过实际的模拟运行获得或者通过程序分析确定，所以基于

剩余故障获得的分析精度接近 100%。因此，SmartInjector 的总体分析精度应当高于等价类故障删除技术的分析精度。图5.14中的结果证明了上述结论，针对每个测试用例的总体分析精度均比等价类故障删除技术的分析精度有所提高，最终 SmartInjector 的平均分析精度达到了 96.3%，可以满足绝大多数情况下的需求。

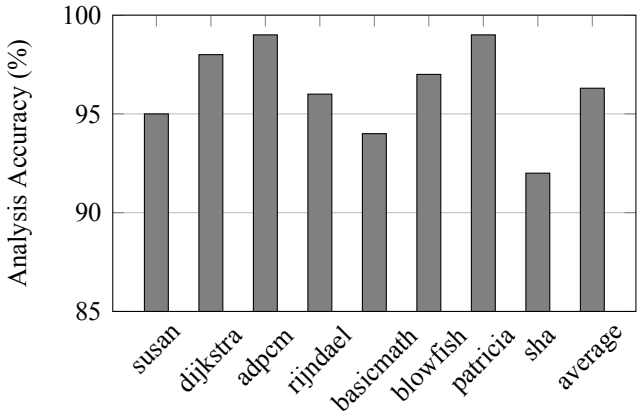


图 5.14　SmartInjector 的分析精度

5.6　本章小结

故障注入作为一种评估系统可靠性的有效方法已经被广泛应用，但是故障注入技术面临着如何平衡模拟速度和精度的问题。本章提出了 SmartInjector 故障注入框架，该框架可以在大幅度降低故障模拟开销的同时，保持较高的可靠性分析精度。SmartInjector 首先基于程序分析技术从故障空间中删除大量不需要注入的故障。然后，在对剩余的故障进行模拟运行时，SmartInjector 采取故障结果预测方法进一步减少单次故障模拟运行的时间。实验结果表明，综合采取以上方法后，SmartInjector 所需要的模拟运行时间仅仅是模拟完整的故障空间所需时间的 0.15%，同时可靠性分析精度可以达到 96.3%。

第六章　结束语

随着集成电路设计朝向更高的频率、更高的晶体管密度和更低的工作电压发展，瞬时故障引发的可靠性问题已经扩散到整个计算市场。由于不同领域的用户对系统可靠性、成本、性能、功耗等指标的要求不同，传统的高可靠性、高成本的容错技术无法满足整个计算市场的需求。为了应对这种挑战，本文着重研究了可配置、低代价的容错技术。本章先对本文的主要贡献进行总结，然后展望下一步需要完成的工作。

6.1　本文主要贡献

本文工作大致可分为容错保护和容错评估两个方面，取得的具体研究成果和创新点如下:

1. 提出了一种可配置的数据流检测技术

   数据流错误检测通常基于冗余计算的方法进行。如何降低冗余计算的开销(性能、硬件开销等）是数据流检测的难点问题。本文结合软硬件容错技术的优势，提出了一种混合软、硬件实现的数据流检测技术 Epipe。Epipe 首先通过改造现有的超标量流水线处理器，以较低的硬件开销提供了一个能够对指令进行冗余保护的硬件平台。为了减少冗余保护产生的性能开销，Epipe 基于程序分析方法评估每个指令的重要性。程序运行时，Epipe 可以根据用户的性能和可靠性要求，配置保护最重要的一部分指令。Epipe 的创新点在于，Epipe 只冗余保护对 SDC 故障最敏感的部分指令（即最重要的指令)，对于导致系统异常或超时的故障则直接利用系统中的异常检测机制加以处理，而剩余的将会被屏蔽的故障则不需要任何处理。这种分类处理故障的方法有效减少了需要冗余保护的指令，再结合时空开销较低的硬件指令保护技术，使得 Epipe 技术可以更低的开销保护程序数据流。

2. 提出了一种可配置的控制流检测技术

   实现控制流检测的一种有效方法是基于软件实现的标签分析方法。遗憾的是，现有的标签分析技术除了在时空开销和可靠性方面存在不足外，对可配置性和容错机制的自我保护也缺乏研究。为了克服上述不足，本文提出了一种新的标签分析控制流检测算法 CFCES。CFCES 可以较少的开销有效地克服已有算法存在的检测盲点。而且，CFCES 在设计检测机制时引入了一种被称为“对等性”的不变量，通过对这种不变量进行检测，CFCES 可以以

极低的代价实现检错机制的自容错保护。此外，CFCES 还提供了可配置的优化方法，可以灵活配置不同代码段的保护强度，以满足用户不同的时空开销和可靠性约束。CFCES 优化方法的特点在于其可以提高 CFCES 的容错效率，且可以用于优化其它基于标签分析的控制流检测算法。

3. 提出了一种基于部分保护的存储单元容错技术

瞬时故障不仅可能发生在处理器组合逻辑网络中，也有可能出现在存储系统。鉴于利用 ECC 技术保护片上 SRAM 结构会带来大量的面积、性能和功耗开销，本文针对一种特殊的片上 SRAM 结构 SPM 提出了低代价的 PPS 保护技术。尽管用 ECC 对 SPM 进行完全保护的开销很高，但是对部分 SPM 存储进行 ECC 保护并进行合理分配仍是非常有价值的。PPS 技术首先设计了基于部分 ECC 保护 SPM 的存储体系结构（ECC 保护的比例可以根据可靠性、性能、功耗等需求决定)，然后对程序中的待分配变量进行脆弱性分析，并根据变量大小将 SPM 划分为伪寄存器，最后采取基于优先级的图着色方法将较为脆弱的变量优先分配到 ECC 保护的伪寄存器中。基于上述方法，PPS 能够以较低的开销获得较高的可靠性。

4. 提出了一种基于程序分析的故障注入框架

故障注入是一种有效且广为应用的可靠性分析方法。故障注入技术面临的困难是如何平衡故障模拟速度与精度的关系。本文提出了一种基于程序分析的故障注入框架 SmartInjector。SmartInjector 首先基于程序分析从故障空间中删除等价类故障和结果确定型故障。等价类故障是指发生在相似的数据流或控制流上下文环境中的故障。结果确定型故障则是指那些通过程序分析就可以确定系统反应的故障。SmartInjector 还首次开发了一种故障结果预测技术，该技术通过在程序运行结束前提前判断故障模拟的结果，可以减少单次模拟的时间开销。结合提出的故障删除技术和故障结果预测技术，SmartInjector 以少量的精度损失极大地减少了故障注入模拟所需的时间开销。

6.2 研究展望

为了应对当前和今后一段时期内计算市场面临的可靠性挑战，除了本文工作研究的内容，还可以在以下几个方面展开进一步的研究：

1. 研究系统级的可靠性分析技术

计算机中的瞬时故障一旦产生，将会从底层电路级向微体系结构级、体系结构级、应用级不断传播，并且直到应用级才能确定故障导致的最终后果。由于在上述各个抽象层次中，故障均有可能被屏蔽，可靠性分析应当覆盖上述整个过程，即进行系统级的可靠性分析，才能得到最精确的结果。现有的可靠性分析技术通常只在部分抽象层次上进行，其分析结果与真实的情况存在明显的偏差，因此有必要进一步研究系统级的可靠性分析技术。一般说来，可靠性分析覆盖的抽象层次越多，往往需要考虑的细节也越多，可能会因此导致无法承受的分析开销。所以，系统级可靠性分析的关键是构建出精确高效的分析模型。

2. 研究面向特定计算平台的容错技术

随着计算机体系结构的快速发展，计算平台正呈现出越来越明显的多样性。不同的计算平台通常拥有不同的硬件资源，同时也具有不同的计算特征。传统的针对通用 CPU 开发的容错技术往往不适合在新型计算平台中直接应用，或者无法有效发挥新型平台的优势。针对这些新的计算平台，开发与之相适应的容错技术具有重要的意义。例如，多核技术是当前体系结构设计发展的潮流之一，多核平台提供的大量并行计算资源，自然地为容错技术提供了新的实现途径。已有研究证明，面向多核平台开发软件实现的冗余多线程、冗余多进程等容错技术可以以较低的代价提供很高的可靠性。

3. 研究面向特定应用的容错技术

利用特定应用的特征，往往可以开发出比通用容错方法更高效的技术。典型的例子就是早期开发的面向特定算法的容错技术（Algorithm Based Fault Tolerance，ABFT)。开发此类容错技术对容错设计者的要求较高。但是，随着嵌入式平台的广泛应用，开发这类技术的意义愈发明显。嵌入式平台中的应用往往功能单一、规模较小，开发专门的容错技术的难度相对较低，而且只要针对这些应用开发出专门的容错技术，即可有效地解决整个系统的可靠性问题。

4. 研究低代价的故障检测器

容错技术的一种基本方法是冗余计算。冗余计算的缺点是很容易导致较高的时空开销。为了减少冗余计算的开销，一种有效的方法是针对程序开发一些低代价的故障检测器，用以替代部分冗余计算。故障检测器可以基于程序的一些不变量信息，利用特定的约束断言实现故障检测。和冗余计算技术相比，故障检测器引入的额外代码很少，因此将其和冗余计算技术结合在一起

可以降低容错开销。开发故障检测器的关键是寻找程序中的不变量，这些不变量可以表现为变量值的范围、变量间的特定关系、指令间的控制流关系等多种形式。

致　谢

“日月忽其不淹兮，春与秋其代序”。五年的攻博生涯转眼即将过去，掩卷沉思，这一刻是值得铭记的时刻，更是用心感恩的时刻！

首先衷心感谢我的导师谭庆平教授！从硕士到博士，我在谭老师指导下度过了七年快乐、充实和进取的学习时光，能够跟随恩师学习这么多年是我的幸运。在读研的这些年里，我的哪怕一点微小的进步都跟谭老师的教导分不开关系。感谢谭老师在学业上对我的严格要求和悉心指导，无论是平时的课程学习，还是在我论文的开题、研究、撰写、修改等过程中，谭老师都倾注了大量的心血和汗水。谭老师治学严谨，学识渊博，不管是在传道授业的讲台上，还是在竭智攻关的实验室里，谭老师都赢得了广泛的赞誉，令我深深地敬佩，也给了我学习的目标和榜样！更为宝贵的是，谭老师高标准、严要求的工作作风和求是创新的科研精神不断地熏陶、感染着我，是令我受益终生的精神财富！总之，谭老师在学术上对我的谆谆教诲、生活上对我的殷切关怀，以及精神上对我的熏陶，我都将永生铭记！

另外要特别感谢的是多年来朝夕相处、一起拼搏学习过的徐建军师兄和谭兰芳同学。你们不但在我学习、研究的过程中给予了很多热情无私的帮助，你们身上很多优秀的品质也让我感触深刻、获益匪浅。记不清曾经多少次一起调试系统、一起加班写稿、一起讨论难题，同甘共苦的情谊将令我永远无法忘怀！谢谢你们！

感谢我在新南威尔士大学联合培养期间的导师薛京灵教授。薛京灵教授不但在我申请出国的过程中给予了大量协助，更在我留学澳洲期间给予了细致耐心的指导，对我的课题研究起了重要的推动作用。十分感谢薛京灵教授的无私帮助！感谢同一课题组的宁洪老师、李暾老师、周会平老师、姜晶菲老师、窦勇老师、周海芳老师、罗宇老师、邓胜兰老师等，在项目攻关、课题研究的过程中，我从你们那里学到了很多宝贵的经验，受到了很多有益的启发。感谢 602 教研室的毛晓光老师、毛新军老师、董威老师、王挺老师、熊岳山老师、徐锡山老师、文艳军老师、李梦君老师、刘万伟老师、尹俊文老师、叶常春老师等，你们不但在课堂上教会了我很多知识，还在课题开题、论文评阅以及课题研究过程中给予了我宝贵的指导和帮助！还要感谢网络所的李爱平老师、计算机所的黄峰老师，以及系办和教务办的翟英、薛源、潘晓辉、钱程东、赵歆、刘峰等老师。

感谢一起在师门学习过的兄弟姐妹们：曹国荣、邵则铭、刘锋、熊磊、叶俊、吴浩、吴大愚、诸利勇、熊荫乔、李磊、解金刚、李林虎、李剑明、项俊、张南、

徐毅、邓锦州、孟宪凯、赵昕琳、钟泊晨等，尤其感谢张南、徐毅、邵则铭、赵昕琳等师弟、师妹们在繁重的答辩准备过程中提供的高效协助！师门的同学团结友爱，相互鼓励帮助，良好的风气给我提供了绝佳的学习氛围，感谢你们！

感谢好友胡安波、李聪、刘金鹏、刘敏、何明、邹丹、罗准辰、陆华彪、徐新海、侯松、谢欣伟、沈剑良、张羽丰、徐晓陆、赵文华、辛凯、张雪萌、胡翠云、屈婷婷、张硕、马文琪、王文珍、邓瑶、胡忠明、王飞、杨惠等。你们的友谊和支持是我前进的动力，感激不尽！感谢新南威尔士大学的万青、狄鹏、叶森、晏高云、Brad、Adrian、Tim、Jenny 等，和你们一起度过的日子，给我留下了美好的回忆。

感谢本科、硕士、博士期间的学员大队和学员队郑光辉、王新国、杨红运、朱涛、林红锦、鲍资富、王艳丰、欧阳登轶、孙友佳、何磊、贺毅、李晗等领导的关怀和教育。

最后，深深地感谢我的家人。父爱如山，母爱如水。父、母亲的辛劳和付出常人难以企及，非文字能尽述。我的哥哥嫂嫂们在我攻博期间代替我一肩挑起了赡养老人、照顾家庭的重担，还对我关怀备至，手足之情同样伟大！还有我活泼可爱的小侄儿嘉明、锐梵和一冉，你们给我带来了很多欢乐和欣喜。我爱你们，我的家人！

参考文献

[1] Firouzi F, Salehi M E, Wang F, et al. An accurate model for soft error rate estimation considering dynamic voltage and frequency scaling effects [J]. Microelectronics Reliability. 2011, 51 (2): 460 – 467.

[2] Eaton P, Benedetto J, Mavis D, et al. Single event transient pulsewidth measurements using a variable temporal latch technique [J]. IEEE Transactions on Nuclear Science. 2004, 51 (6): 3365–3368.

[3] Shivakumar P, Kistler M, Keckler S, et al. Modeling the effect of technology trends on the soft error rate of combinational logic [C]. In Proceedings of the 2002 International Conference on Dependable Systems and Networks. 2002: 389–398.

[4] Nguyen H, Yagil Y. A systematic approach to SER estimation and solutions [C]. In Proceedings of the 41st Annual IEEE International Symposium on Reliability Physics. 2003: 60–70.

[5] Ramakrishnan K, Rajaramant R, Suresh S, et al. Variation Impact on SER of Combinational Circuits [C]. In Proceedings of the 8th International Symposium on Quality Electronic Design. 2007: 911–916.

[6] Buchner S, Baze M, Brown D, et al. Comparison of error rates in combinational and sequential logic [J]. IEEE Transactions on Nuclear Science. 1997, 44 (6): 2209–2216.

[7] 徐建军．面向寄存器软错误的容错编译技术研究 [D]．长沙：国防科技大学，2010．

[8] Vemu R. Low-cost Assertion-based Fault Tolerance in Hardware and Software [D]. [S. l.]: The University of Texas at Austin, 2008.

[9] http://en.wikipedia.org/wiki/Single_event_upset. 2009.

[10] http://news.nationalgeographic.com/news/2003/08/0827_030827_kyotoprizepark\er.html. 2003.

[11] http://www.nasa.gov/mission_pages/swift/main/index.html. 2004.

[12] 庄奕琪．微电子器件应用可靠性技术 [M]．电子工业出版社，1996．

[13] Johnson R T, Thome F V, Craft C M, et al. A Survey of aging of electronics with application to nuclear power plant instrumentation [J]. IEEE Transactions on Nuclear Science. 1983, 30 (6): 4358–4362.

[14] 吴艳霞．基于汇编语言的控制流错误检测算法研究 [D]．哈尔滨：哈尔滨工业大学，2008．

[15] Mukherjee S. Architecture Design for Soft Errors [M]. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[16] 谭兰芳．面向软错误的故障恢复和验证技术研究 [D]．长沙：国防科技大学，2013．

[17] 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．

[18] 邢克飞．星载信号处理平台单粒子效应检测与加固技术研究 [D]．长沙：国防科技大学，2007．

[19] Ziegler J, Puchner H. SER – History, Trends and Challenges: A guide for designing with Memory ICs [M]. Cypress Semiconductor Corp., 2004.

[20] Michalak S, Harris K, Hengartner N, et al. Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer [J]. IEEE Transactions on Device and Materials Reliability. 2005, 5 (3): 329–335.

[21] Bronevetsky G, de Supinski B R, Schulz M. A foundation for the accurate prediction of the soft error vulnerability of scientific applications [C]. In Proceedings of the 5th IEEE Workshop on Silicon Errors in Logic System Effects. 2009.

[22] ZHANG Y. Runtime Speculative Software-only Fault Tolerance [D]. [S. l.]: Princeton University, 2012.

[23] Govindavajhala S, Appel A W. Using Memory Errors to Attack a Virtual Machine [C]. In Proceedings of the 2003 IEEE Symposium on Security and Privacy. 2003: 154–154.

[24] Boneh D, DeMillo R A, Lipton R J. On the Importance of Checking Cryptographic Protocols for Faults [C]. In Proceedings of the 1997 International Conference on the Theory and Application of Cryptographic Techniques. 1997: 37–51.

[25] `http://www.intel.com/cd/corporate/home/apac/zho/346894.htm`. 2009.

[26] Ding Q, Wang Y, Wang H, et al. Output remapping technique for critical paths soft-error rate reduction [J]. IET Computers & Digital Techniques. 2010, 4 (4): 325–333.

[27] Feng S, Gupta S, Ansari A, et al. Shoestring: probabilistic soft error reliability on the cheap [C]. In Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems. USA, 2010: 385–396.

[28] Li M-L, Ramachandran P, Sahoo S K, et al. Understanding the propagation of hard errors to software and implications for resilient system design [C]. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems. Seattle, WA, USA, 2008: 265–276.

[29] Hari S K S, Adve S V, Naeimi H. Low-cost program-level detectors for reducing silent data corruptions [C]. In Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2012: 1–12.

[30] 胡谋．计算机容错技术 [M]．中国铁道出版社，1995．

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

[32] Reis G, Chang J, Vachharajani N, et al. SWIFT: software implemented fault tolerance [C]. In Proceedings of the 2005 International Symposium on Code Generation and Optimization. San Jose, California, USA, 2005: 243–254.

[33] Hari S K S, Adve S V, Naeimi H, et al. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults [C]. In Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems. 2012: 123–134.

[34] Ramabadran T. A coding scheme for m-out-of-n codes [J]. IEEE Transactions on Communications. 1990, 38 (8): 1156–1163.

[35] Stone J, Greenwald M, Partridge C, et al. Performance of checksums and CRC's over real data [J]. IEEE/ACM Transations on Networking. 1998, 6 (5): 529–543.

[36] Kumar U K, Umashankar B S. Improved Hamming Code for Error Detection and Correction [C]. In Proceedings of the 2nd International Symposium on Wireless Pervasive Computing. 2007: 498–500.

[37] Lo J-C, Thanawastien S. The design of fast totally self-checking Berger code checkers based on Berger code partitioning [C]. In Proceedings of the 18th International Symposium on Fault-Tolerant Computing. 1988: 226–231.

[38] Paulitsch M, Morris J, Hall B, et al. Coverage and the use of cyclic redundancy codes in ultra-dependable systems [C]. In Proceedings of the 2005 International Conference on Dependable Systems and Networks. 2005: 346–355.

[39] Khudia D S, Wright G, Mahlke S. Efficient soft error protection for commodity embedded microprocessors using profile information [C]. In Proceedings of the 2012 International Conference on Languages, Compilers, Tools and Theory for Embedded Systems: 99–108.

[40] Kim J, Hardavellas N, Mai K, et al. Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding [C]. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. Chicago, Illinois, USA, 2007: 197–209.

[41] Rao T R N, Fujiwara E. Error-control coding for computer systems [M]. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[42] Shirvani P, Saxena N, McCluskey E. Software-implemented EDAC protection against SEUs [J]. IEEE Transactions on Reliability. 2000, 49 (3): 273–284.

[43] Chen C L. Error-correcting codes for semiconductor memories [J]. ACM SIGARCH Computer Architecture News. 1984, 12 (3): 245–247.

[44] Mukherjee S S, Weaver C, Emer J, et al. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor [C]. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture. 2003: 29–40.

[45] Xu J, Shen R, Tan Q. PRASE: An Approach for Program Reliability Analysis with Soft Errors [C]. In Proceedings of the 14th IEEE Pacific Rim International Symposium on Dependable Computing. 2008: 240–247.

[46] Li X, Adve S, Bose P, et al. SoftArch: an architecture-level tool for modeling and analyzing soft errors [C]. In Proceedings of the 2005 International Conference on Dependable Systems and Networks. 2005: 496–505.

[47] Li X. Soft Error Modeling and Analysis for Microprocessors [D]. USA: Illinois University, 2008.

[48] Sridharan V, Kaeli D R. Eliminating microarchitectural dependency from Architectural Vulnerability [C]. In Proceedings of the 2009 International Symposium on High-Performance Computer Architecture. 2009: 117–128.

[49] Biswas A, Racunas P, Cheveresan R, et al. Computing Architectural Vulnerability Factors for Address-Based Structures [C]. In Proceedings of the 2005 Annual International Symposium on Computer Architecture. 2005: 532–543.

[50] 熊磊．面向程序级的软错误容错研究 [D]．长沙：国防科技大学，2012．

[51] Hsueh M-C, Tsai T, Iyer R. Fault injection techniques and tools [J]. IEEE Computer. 1997, 30 (4): 75–82.

[52] Arlat J, Aguera M, Amat L, et al. Fault injection for dependability validation: a methodology and some applications [J]. IEEE Transactions on Software Engineering. 1990, 16 (2): 166–182.

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

[54] Arlat J, Crouzet Y, Karlsson J, et al. Comparison of physical and software-implemented fault injection techniques [J]. IEEE Transactions on Computers. 2003, 52 (9): 1115–1133.

[55] Yu J, Garzarán M J. A Detector for Harmful Errors [C]. In Proceedings of the 5th IEEE Workshop on Silicon Errors in Logic System Effects. 2009.

[56] Cook J, Zilles C. A characterization of instruction-level error derating and its implications for error detection [C]. In Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. Anchorage, AK, USA, 2008: 482–491.

[57] L Xiong Z S, Q Tan. Exploration of the effects of soft errors from dynamic software behaviours [J]. IET Software. 2012, 6: 514–523(9).

[58] Xu X, Li M-L. Understanding soft error propagation using Efficient vulnerability-driven fault injection [C]. In Proceedings of the 2012 International Conference on Dependable Systems and Networks. 2012: 1–12.

[59] Bernick D, Bruckert B, Vigna P D, et al. NonStop Advanced Architecture [C]. In Proceedings of the 2005 International Conference on Dependable Systems and Networks. Yokohama, Japan, 2005: 12–21.

[60] Spainhower L, Gregg T A. IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective [J]. IBM Journal of Research and Development. 1999, 43 (5): 863–873.

[61] Yeh Y. Triple-triple redundant 777 primary flight computer [C]. In Proceedings of the 1996 IEEE Aerospace Applications Conference: 293–307.

[62] Rotenberg E. AR-SMT: a microarchitectural approach to fault tolerance in microprocessors [C]. In Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing. Madison, Wisconsin, USA, 1999: 84–91.

[63] Reinhardt S K, Mukherjee S S. Transient Fault Detection via Simultaneous Multithreading [C]. In Proceedings of the 27th Annual International Symposium on Computer Architecture. Vancouver, British Columbia, Canada, 2000: 25–36.

[64] Vijaykumar T N, Pomeranz I, Cheng K. Transient-Fault Recovery Using Simultaneous Multithreading [C]. In Proceedings of the 29th International Symposium on Computer Architecture. 2002: 87–98.

[65] Austin T M. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design [C]. In Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture. 1999: 196–207.

[66] Weaver C, Austin T M. A Fault Tolerant Approach to Microprocessor Design [C]. In Proceedings of the 2001 International Conference on Dependable Systems and Networks. 2001: 411–420.

[67] Bower F A, Sorin D J, Ozev S. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors [C]. In Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture. 2005: 197–208.

[68] Shyam S, Constantinides K, Phadke S, et al. Ultra low-cost defect protection for microprocessor pipelines [C]. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. 2006: 73–82.

[69] Ray J, Hoe J C, Falsafi B. Dual use of superscalar datapath for transient-fault detection and recovery [C]. In Proceedings of the 34th International Symposium on Microarchitecture. 2001: 214–224.

[70] Banakar R, Steinke S, Lee B-S, et al. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems [C]. In Proceedings of the 10th International Symposium on Hardware/Software Codesign. 2002: 73–78.

[71] Montesinos P, Liu W, Torrellas J. Shield: Cost-Effective Soft-Error Protection for Register Files [C]. In Proceedings of the 3rd IBM TJ Watson Conference on Interaction between Architecture, Circuits and Compilers. 2006.

[72] Lee K, Shrivastava A, Issenin I, et al. Partially protected caches to reduce failures due to soft errors in multimedia applications [J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 2009, 17 (9): 1343–1347.

[73] Farbeh H, Fazeli M, Khosravi F, et al. Memory Mapped SPM: Protecting Instruction Scratchpad Memory in Embedded Systems against Soft Errors [C]. In Proceedings of the 9th European Dependable Computing Conference. 2012: 218–226.

[74] Oh N, Mitra S, McCluskey E J. ED^{4}I: Error Detection by Diverse Data and Duplicated Instructions [J]. IEEE Transactions on Computers. 2002, 51: 180–199.

[75] 李建立．空间辐射环境下软件实现的硬件故障检测技术研究 [D]．长沙：国防科技大学，2008．

[76] Yu J, Garzaran M J, Snir M. ESoftCheck: Removal of Non-vital Checks for Fault Tolerance [C]. In Proceedings of the 7th International Symposium on Code Generation and Optimization. Seattle, WA, USA, 2009: 35–46.

[77] Nicolescu B, Velazco R. Detecting soft errors by a purely software approach: method, tools and experimental results [C]. In Proceedings of the 2003 Design Automation and Testing in Europe. 2003: 57–62.

[78] Rebaudengo M, Reorda M, Violante M, et al. A source-to-source compiler for generating dependable software [C]. In Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation. 2001: 33–42.

[79] Brown D T. Error Detecting and Correcting Binary Codes for Arithmetic Operations [J]. IRE Transactions on Electronic Computers. 1960, EC-9 (3): 333–337.

[80] Schiffel U. Hardware Error Detection Using AN-Codes [D]. Germany: Technische Universität Dresden, 2011.

[81] Engel H. Data flow transformations to detect results which are corrupted by hardware faults [C]. In Proceedings of the 1996 IEEE High-Assurance Systems Engineering Workshop. 1996: 279–285.

[82] Zhang Y, Ghosh S, Huang J, et al. Runtime Asynchronous Fault Tolerance via Speculation [C]. In Proceedings of the 10th International Symposium on Code Generation and Optimization. San Jose, California, USA, 2012: 145–154.

[83] Wang C, Kim H, Wu Y, et al. Compiler-Managed Software-Based Redundant Multi-Threading for Transient Fault Detection [C]. In Proceedings of the 2007 International Symposium on Code Generation and Optimization. 2007: 244–258.

[84] Zhang Y, Lee J W, Johnson N P, et al. DAFT: decoupled acyclic fault tolerance [C]. In Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques. Vienna, Austria, 2010: 87–98.

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

[86] Li A, Hong B. On-line control flow error detection using relationship signatures among basic blocks [J]. Computers and Electrical Engineering. 2010, 36 (1): 132–141.

[87] Alkhalifa Z, Nair V, Krishnamurthy N, et al. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection [J]. IEEE Transactions on Parallel and Distributed Systems. 1999, 10 (6): 627–641.

[88] Nicolescu B, Savaria Y, Velazco R. Software detection mechanisms providing full coverage against single bit-flip faults [J]. IEEE Transactions on Nuclear Science. 2004, 51 (6): 3510 – 3518.

[89] Mohamed A, Zulkernine M. A Connection-Based Signature Approach for Control Flow Error Detection [C]. In Proceedings of the 9th International Conference on Dependable, Autonomic and Secure Computing. 2011: 129 –136.

[90] Borin E, Wang C, Wu Y, et al. Software-based transparent and comprehensive control-flow error detection [C]. In Proceedings of the 2006 International Symposium on Code Generation and Optimization. 2006.

[91] Vemu R, Abraham J. CEDA: Control-Flow Error Detection Using Assertions [J]. IEEE Transactions on Computers. 2011, 60 (9): 1233 –1245.

[92] Meixner A, Bauer M, Sorin D. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores [C]. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. Chicago, Illinois, USA, 2007: 210–222.

[93] Wang N J, Patel S J. ReStore: Symptom Based Soft Error Detection in Microprocessors [C]. In Proceedings of the 2005 International Conference on Dependable Systems and Networks. Yokohama, Japan, 2005: 30–39.

[94] Sahoo S K, Li M-L, Ramachandran P, et al. Using likely program invariants to detect hardware errors [C]. In Proceedings of the 2008 International Conference on Dependable Systems and Networks. 2008: 70–79.

[95] Yalcin G, Unsal O S, Cristal A, et al. SymptomTM: Symptom-Based Error Detection and Recovery Using Hardware Transactional Memory [C]. In Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques. 2011: 199–200.

[96] Nakka N, Pattabiraman K, Iyer R. Processor-Level Selective Replication [C]. In Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. Edinburgh, UK, 2007: 544–553.

[97] Yan J, Zhang W. Compiler-guided register reliability improvement against soft errors [C]. In Proceedings of the 5th ACM international conference on Embedded software. 2005: 203–209.

[98] Slegel T, Averill I, RM, Check M, et al. IBM's S/390 G5 microprocessor design [J]. IEEE Micro. 1999, 19 (2): 12–23.

[99] Sorin D J, Martin M M K, Hill M D, et al. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery [C]. In Proceedings of the 29th International Symposium on Computer Architecture. 2002: 123–134.

[100] Yu P, Mitra T. Satisfying real-time constraints with custom instructions [C]. In Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis. 2005: 166–171.

[101] Crandall J R, Chong F T. Minos: Control Data Attack Prevention Orthogonal to Memory Model [C]. In Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture. Oregon, Portland, 2004: 221–232.

[102] Weaver C, Emer J, Mukherjee S S, et al. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor [C]. In Proceedings of the 31st Annual International Symposium on Computer Architecture. München, Germany, 2004: 264–264.

[103] Qureshi M K, Mutlu O, Patt Y N. Microarchitecture-Based Introspection: A Technique for Transient-Fault Tolerance in Microprocessors [C]. In Proceedings of the 2005 International Conference on Dependable Systems and Networks. Yokohama, Japan, 2005: 434–443.

[104] Reis G A. Software Modulated Fault Tolerance [D]. USA: Princeton University, 2008.

[105] Burger D, Austin T M. The SimpleScalar tool set version 2.0 [R]. 1997.

[106] Guthaus M R, Ringenberg J S, Ernst D, et al. MiBench: A free, commercially representative embedded benchmark suite [C]. In Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization. Austin, Texas, USA, 2001: 3–14.

[107] Ohlsson J, Rimen M, Gunneflo U. A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog [C]. In Proceedings of the 1992 Annual International Symposium on Fault-Tolerant Computing. 1992: 316 –325.

[108] Schuette M, Shen J. Processor Control Flow Monitoring Using Signatured Instruction Streams [J]. IEEE Transactions on Computers. 1987, 36 (3): 264–276.

[109] Mahmood A, McCluskey E J. Concurrent Error Detection Using Watchdog Processors-A Survey [J]. IEEE Transactions on Computers. 1988, 37 (2): 160–174.

[110] Tomas S P, Tomas S P, Daniel, et al. A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems [J]. Microprocessing and Microprogramming. 1987, 20 (4): 249–269.

[111] Lu D. Watchdog Processors and Structural Integrity Checking [J]. IEEE Transactions on Computers. 1982 (7): 681 –685.

[112] Wang M, Wang Y, Liu D, et al. Improving the reliability of embedded systems with cache and SPM [C]. In Proceedings of the 6th IEEE International Conference on Mobile Adhoc and Sensor Systems. 2009: 825–830.

[113] Li L, Feng H, Xue J. Compiler-directed scratchpad memory management via graph coloring [J]. ACM Transactions on Architecture and Code Optimization. 2009, 6 (3): 1–17.

[114] Damavandpeyma M, Stuijk S, Basten T, et al. Thermal-aware scratchpad memory design and allocation [C]. In Proceedings of the 2010 IEEE International Conference on Computer Design. 2010: 118–124.

[115] 汪黎．面向软件管理片上存储器的编译优化技术研究 [D]．长沙：国防科技大学，2009．

[116] Jeyapaul R, Shrivastava A. Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors [C]. In Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems. 2011: 105–114.

[117] Quach N. High availability and reliability in the itanium processor [J]. IEEE Micro. 2000, 20 (5): 61–69.

[118] Li J-F, Huang Y-J. An Error Detection and Correction Scheme for RAMs with Partial-Write Function [C]. In Proceedings of the 2005 IEEE International Workshop on Memory Technology, Design, and Testing. 2005: 115–120.

[119] Phelan R. Addressing soft errors in ARM core-based designs [R]. 2003.

[120] Zhang W. Replication Cache: A Small Fully Associative Cache to Improve Data Cache Reliability [J]. IEEE Transactions on Computers. 2005, 54 (12): 1547–1555.

[121] Zhang W, Gurumurthi S, Kandemir M, et al. ICR: in-cache replication for enhancing data cache reliability [C]. In Proceedings of the 2003 International Conference on Dependable Systems and Networks. 2003: 291–300.

[122] Briggs P, Cooper K D, Torczon L. Improvements to Graph Coloring Register Allocation [J]. ACM Transactions on Programming Languages and Systems. 1994, 16: 428–455.

[123] http://euler.slu.edu/~fritts/mediabench/. 1997.

[124] Gustafsson J, Betts A, Ermedahl A, et al. The Mälardalen WCET Benchmarks – Past, Present and Future [C]. In Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis. 2010: 137–147.

[125] Pattabiraman K, Nakka N, Kalbarczyk Z, et al. SymPLFIED: Symbolic program-level fault injection and error detection framework [C]. In Proceedings of the 2008 International Conference on Dependable Systems and Networks. 2008: 472–481.

[126] Michael N W, Fertig M, Patel S. Y-Branches: When You Come to a Fork in the Road, Take It [C]. In Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques. New Orleans, LA, USA, 2003: 56–67.

作者在学期间取得的学术成果

[1] Jianli Li, Qingping Tan, Lanfang Tan. Implementing Low-Cost Fault-Tolerance via Hybrid Synchronous/Asynchronous Checks, Journal of Circuits, Systems, and Computers, Vol. 22, No. 7 (SCI: 000323354000008)

[2] Jianli Li, Jingling Xue, Xinwei Xie, Qing Wan, Qingping Tan, Lanfang Tan. Epipe: A low-cost fault-tolerance technique considering WCET constraints, Journal of Systems Architecture, 2013, DOI: 10.1016/j.sysarc.2013.06.003 (SCI 检索)

[3] 李建立, 谭庆平, 谭兰芳, 徐建军. 一种基于虚拟基本块和格式化标签的控制流检测方法, 计算机学报 (一级学报)

[4] Jianli Li, Qingping Tan. SmartInjector: Exploiting Intelligent Fault Injection for SDC Rate Analysis, The 26th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT 2013), New York, USA (EI 检索)

[5] Jianli Li, Qingping Tan, Lanfang Tan, Tongchuan Xin. REE: Exploiting Idempotent Property of Applications for Fault Detection and Recovery, The 9th International Conference on Natural Computation (ICNC 2013), Shenyang, China (EI 检索)

[6] Jianli Li, Qingping Tan, Lanfang Tan. EnHTM: Exploiting Hardware Transaction Memory for Achieving Low-cost Fault Tolerance, 2013 International Conference on Digital Manufacturing and Automation, Qingdao, China (EI 检索)

[7] Jianli Li, Qingping Tan, Jianjun Xu. A Software-implemented Configurable Control Flow Checking Method, 2010 International Symposium on Parallel Architectures, Algorithms and Programming, Dalian, China (EI: 20111113752999)

[8] Jianli Li, Qingping Tan, Jianjun Xu. Reconstructing control flow graph for control flow checking, 2010 IEEE International Conference on Progress in Informatics and Computing，Shanghai, China (EI: 20110713666460)

[9] 谭庆平, 李建立, 宁洪等. 一种基于还原程序的硬件故障检测方法, 国家发明专利, 专利号: 200910226770.7

[10] 谭庆平, 李建立, 宁洪等. 一种基于重构控制流图的控制流错误检测优化方法, 国家发明专利, 专利号: 201010504386.1

[11] 李建立, 谭庆平, 徐建军. 一种辐射环境下瞬时故障的软件检测方法，计算机工程与科学，第 32 卷, 第 3 期

[12] 徐建军, 谭庆平, 李建立, 李剑明. 一种基于格式化标签的可扩展控制流检测方法, 计算机研究与发展，第 48 卷, 第 4 期 (EI 检索)

[13] Lanfang Tan, Qingping Tan, Jianjun Xu, Jianli Li. A Note on “On the Use of Model Checking for the Verification of a Dynamic Signature Monitoring Approach”, IEEE Transaction on Nuclear Science, Vol. 58, No.1 (SCI 检索)

[14] Lanfang Tan, Qingping Tan, Jianjun Xu, Jianli Li. Transient-Error Detection and Recovery via Reverse Computation and Checkpointing, 2012 IEEE Cluster Computing’s Workshop on Power and QoS Aware Computing, Beijing, China (EI 检索)