寄存器软错误对程序可靠性影响的静态分析

徐建军 谭庆平
（国防科学技术大学计算机学院 长沙 410073）

摘 要 继性能和功耗问题之后，软错误导致的计算可信性已成为一个日益严峻的课题。其中，由于寄存器访问频繁却未能被良好保护，发生在其中的软错误成为影响程序可靠性的关键因素之一。基于程序汇编代码，提出一种针对寄存器软错误的程序可靠性静态分析方法。首先通过数据流分析技术提取所有可能影响程序执行的寄存器活跃区间，然后基于活跃区间的路径表达式分析其执行时间和出现频率，最后在此基础上计算在寄存器软错误影响下的程序可靠性。实验表明，该方法的分析结果与AVF分析法保持一致，同时其结果还指出相关的寄存器活跃区间的执行时间和出现频率，这为实现针对寄存器软错误的高效容错方法提供了依据。

关键词 软错误，寄存器，程序可靠性，程序分析

中图法分类号 TP302.7 **文献标识码** A

Static Analysis of Soft Errors Effect in Register Files for Program Reliability

XU Jian-jun TAN Qing-ping
(School of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract Subsequently to the wall of performance and power consumption, the dependability of computing caused by soft errors has become a growing concern. Since register files are accessed very frequently and can not be well protected, soft errors occurring in them are one of the top reasons for affecting the reliability of program. To access the effect of soft errors in register files, a static analysis approach for program reliability was presented based on the assembly codes. Firstly, all possible live interval of registers, which may degrade the program reliability, were sketched through the data flow analysis techniques; then the execution time and frequency of each live interval were analyzed according to the expression of execution path; finally the program reliability can be calculated under the occurrence of soft errors in register files. Experiments show that the analytical results are compatible with the AVF methods'. Moreover, the execution time and frequency of all involved interval have been presented, which are in favor of implementing the high efficient fault tolerance methods for soft errors in register files.

Keywords Soft error, Register file, Program reliability, Program analysis

软错误(Soft Error)，也称为单粒子翻转SEU(Single Event Upset)，是半导体电路中的一种瞬态故障现象，通常是由外部环境中的高能粒子辐照和电压扰动、电磁干扰等电子噪声诱发[1,2]。软错误一直都是航天计算面临的最主要的挑战之一，而近年来随着处理器逐步采用深亚微米制造工艺，在性能得到大幅提高的同时，处理器对于造成软错误的各种干扰却变得更加敏感，同时芯片集成晶体管数的指数级增长也使得整体的软错误率SER(Soft Error Rate)迅速增加。文献[3]预测在接下来的20年中，处理器的SER将会以每代8%的速度递增。在继性能和功耗问题之后，软错误导致的计算可信性已日益成为业界关注的热点。

当前，电路屏蔽和逻辑屏蔽等作用使得运算部件的SER相比存储部件要小很多[3]，而内存和Cache大多使用ECC编码或奇偶校验码等技术，并受到良好保护，但是寄存器由于访问非常频繁，发生在其中的错误会很快传播到系统其它区域，而保护寄存器在功耗和性能开销方面都存在困难，使得寄存器成为软错误影响系统可靠性的关键因素之一[4]。就发生在寄存器中的软错误对程序可靠性的影响进行分析并进一步采取合适的容错措施具有重要意义。

目前在评估软错误对系统可靠性影响方面，主要有故障注入法和故障分析法两类技术。故障注入作为一种评价系统可靠性的有效测试技术，已经在学术界和工业界得到广泛应用[5]。对于软错误来说，故障注入需要按照软错误特征在目标系统中人为产生单粒子翻转，再收集并统计分析系统的响应信息。但是实现时需要仔细权衡实现成本、抽象层次、模拟速度以及准确度之间的关系，而且为了取得有统计意义的结果，故障注入法往往要进行大量的实验。针对软错误的代表分析方法是Mukherjee等提出的AVF(Architectural Vulnerability Factors)分析法[6]。在该方法中，每个二进制位在某个周期发生的软错误如果会导致错误结果，则将其标识为ACE(Architectural Correct Execution)位。首先针对目标部件逐周期地分析ACE位的比例，然后推导出该部件的AVF值和

到稿日期：2010-02-05 返修日期：2010-04-23

徐建军(1980—)，男，博士生，主要研究方向为程序分析和容错编译，E-mail：jjun.xu@gmail.com；**谭庆平**(1965—)，男，教授，博士生导师，主要研究方向为高可信软件和分布式软件工程等。

失效率，处理器整体失效率则等于每个构成部件的失效率之和，但是这种分析需要基于具体的硬件性能模拟器进行。Lee等针对寄存器软错误提出了一种效果显著的静态分析方法[7]，该方法以程序基本块为单位，将寄存器生命周期分为基本块内部和基本块之间两部分，最后得出软错误对程序可靠性影响的方程组，但是分析结果还存在一定程度的不精确性。文献[8]提出一种在运行过程中动态预测寄存器软错误影响的方法，但是预测机制会带来程序运行时性能和功耗的开销。

本文从程序角度出发，提出一种面向寄存器软错误的程序可靠性静态分析方法 ASER(Analysis of Soft Errors in Register files)。基于程序汇编代码，ASER 首先提取出所有可能影响程序运行的寄存器活跃区间，然后具体计算每个活跃区间的执行时间和出现频率。ASER 的分析独立于具体硬件平台，可以在系统设计和部署的各个阶段进行，而且作为一种静态分析方法，ASER 不会对程序运行期行为产生影响。更重要的是，这种分析有助于理解程序自身结构与寄存器软错误影响之间的关系，明确其中影响程序可靠性的关键点，从而可以采取有针对性的错误检测和恢复技术。

1 总体思路

软错误直接影响二进制数据，在程序汇编层进行分析有针对意义，ASER 目前是在 MIPS32 指令集[9]的基础上进行分析。简化起见，本文不考虑处理器流水线等底层硬件，假设指令依次顺序执行，并且每条指令在一个指令周期(简称周期)内完成。

在程序运行过程中，一个寄存器 R 会被读取(称为引用)和写入(称为定值)多次。如图 1 所示，从某次定值 $Write_1$ 到再次定值 $Write_2$ 之间的区间称为 R 的一个生命周期，而从 $Write_1$ 到 $Write_2$ 之前的最后一次引用 $Read_{last}$ 之间的区间称为 R 的活跃区间(*live interval*)，从 $Read_{last}$ 到 $Write_2$ 之间的区间称为死亡区间(*dead interval*)。活跃区间发生的软错误会影响程序执行，在 AVF 分析法中，该区间中 R 的二进制位是 ACE 位。而死亡区间发生的软错误由于会被 $Write_2$ 写入的新数据所屏蔽，将不会影响程序可靠性，所以称这区间内 R 的二进制位为非 ACE 位。

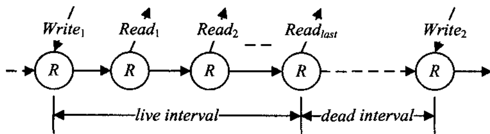

图 1 寄存器的生命周期

ASER 将寄存器 R 的生命周期定义为一个五元组(*def*, *end*, *dead*, *time*, *freq*)：

- *def*：定值点，向 R 定值的指令；
- *end*：终止点，再次向 R 定值之前最后一次引用 R 的指令；
- *dead*：死亡点，再次向 R 定值的指令，它也是下一个生命周期的定值点；
- *time*：活跃时间，即活跃区间的执行时间，在本文假设条件下，它对应于从 *def* 到 *end* 之间执行的指令数；
- *freq*：生命周期的出现频率，程序运行时同一个生命周期可能会出现多次。

通常认为一个二进制位的原始 SER 为常数，或者软错误发生概率随时间服从指数分布[10]。设原始 SER 为 λ，程序使用了 r 个寄存器，每个寄存器 $R_i(1\leqslant i\leqslant r)$ 包含的二进制位数是 $R_i.length$，有 k_i 个生命周期，可得下列等式：

$$ACE_{bit}=\sum_{i=1}^{r}(R_i.length\times\sum_{j=1}^{k_i}(time_j\times freq_j)) \tag{1}$$

$$RVF=ACE_{bit}/(T_{prog}\times\sum_{i=1}^{r}R_i.length) \tag{2}$$

$$P_{prog}=e^{-\lambda\times ACE_{bit}} \tag{3}$$

式中，ACE_{bit} 表示目标程序的寄存器 ACE 位的总数。RVF 与 AVF 分析法中寄存器的 AVF 指标有所不同，它等于 ACE_{bit} 与被程序使用的寄存器所有可能发生的软错误的比值。T_{prog} 和 P_{prog} 分别表示程序一次运行所需时间以及在寄存器软错误影响下的程序可靠性。所以，其中的关键是要确定寄存器各个生命周期的活跃区间，以及具体每个生命周期的 *time* 值和 *freq* 值。从图 1 可以发现，活跃区间反映了从定值点 *def* 到最后一次引用点 $Read_{last}$ 之间的定值引用关系，其困难在于这种分析是路径敏感的(path-sensitive)，必须考虑从 *def* 到 $Read_{last}$ 的所有可能执行路径。

ASER 采用一系列静态分析技术解决这些问题：①首先通过数据流分析方法针对每个寄存器确定所有的定值点，并从定值点后向分析获得之前的最后一次引用，然后提取出所有可能的生命周期及其活跃区间；②通过约简基本块邻接矩阵的方法获得关于活跃区间的执行路径表达式，然后基于分支概率计算活跃区间的执行时间(即 *time* 值)；③最后根据路径子图在程序控制流图中的入边和出边的执行频率计算生命周期的出现频率(即 *freq* 值)。

2 ASER 方法

本文在 2.1—2.3 小节分别介绍上述 3 个问题的解决方案，在 2.4 小节给出具体实现算法和复杂度分析结果。

先引入以下概念：基本块是一个顺序执行的指令序列，$block=\langle inst_1, inst_2, \cdots, inst_n \mid n\geqslant 1\rangle$，其中只有 $inst_n$ 才能是分支或跳转等程序控制指令，只有 $inst_1$ 才能是程序控制指令的转移目标。基于基本块，程序可表示为一个由基本块和连接基本块之间的有向边构成的控制流图 $CFG=(BLOCK, EDGE)$。其中，$BLOCK$ 为基本块集合，$EDGE\in\mathscr{P}(BLOCK\times BLOCK)$ 为连接基本块之间的边的集合。对于 $\forall block\times BLOCK$，$PREV(block)$ 和 $SUCC(block)$ 分别表示其前驱基本块和后继基本块的集合，$F_B(block)$ 表示 *block* 在整个程序中的执行频率。对于 $\forall edge\in EDGE$，$source(edge)$ 和 $sink(edge)$ 分别表示其源基本块和目的基本块，并用 $F_E(edge)$ 和 $P_E(edge)$ 分别表示 *edge* 在整个程序中的执行频率和分支概率。基本块、边的执行频率以及边的分支概率既可以从程序的运行剖面(execution profile)中获取，也可以通过完全静态的启发式分支预测方法求得[11]。

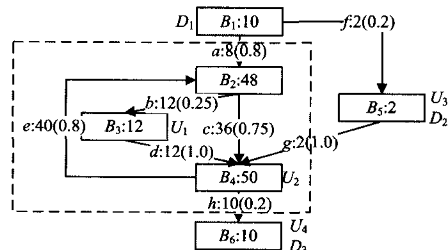

图 2 程序控制流图示例

程序控制流图示例如图 2 所示，基本块和边的冒号后为

对应执行频率，边的分支概率在对应括号中表示。D_i（$1\leqslant i\leqslant 3$）和 U_j（$1\leqslant j\leqslant 4$）分别表示对某个目标寄存器 R 的定值和引用。

2.1 确定生命周期

如果寄存器 R 邻接的两次定值都在一个基本块内部，那么可直接按照基本块的指令序列确定该生命周期及其活跃区间。但是如果 R 邻接的两次定值属于不同的基本块，就需要确定 R 在各个基本块入口和出口处的定值引用情况。这里针对基本块先引入下面 3 个函数：

$$GEN, KILL, DEF: BLOCK \rightarrow \mathcal{P}(REG)$$

其中，REG 为寄存器集合，对于一个基本块 $block$，$GEN(block)$ 表示 $block$ 中在重新定值之前会被引用的寄存器集合，$KILL(block)$ 表示 $block$ 中在引用之前会被重新定值的寄存器集合，$DEF(block)$ 表示所有被 $block$ 定值的寄存器集合，所以有 $\forall block \in BLOCK: KILL(block) \subseteq DEF(block)$。

在此基础上，针对一个具体寄存器 R，为每个基本块的入口和出口处的定值引用状态分别定义下面 4 个函数：

$$DEAD_{in}, DEAD_{out}: BLOCK \rightarrow \mathcal{P}(BLOCK \cup \{\bot\})$$

$$END_{in}, END_{out}: BLOCK \rightarrow \mathcal{P}(BLOCK \cup \{\bot\})$$

$DEAD_{in}(block)$ 和 $DEAD_{out}(block)$ 分别表示进入 $block$ 之前和离开 $block$ 之后在任何引用 R 之前会对 R 定值的基本块集合，$END_{in}(block)$ 和 $END_{out}(block)$ 分别表示进入 $block$ 之前和离开 $block$ 之后在任何对 R 定值之前会最后一次引用 R 的基本块集合，$\bot$ 表示程序执行结束的极值状态，详细的定义如表 1 所列。这里的分析属于后向数据流分析：每个基本块的出口状态是其后继基本块入口状态的并集，转化过程的起始状态是程序退出基本块（即 $SUCC(block)$ 为空集的基本块）的出口状态。对于一些特殊的功能寄存器（如 \$gp 等）来说，END_{out} 起始状态特殊处理，表示这些寄存器在程序结束时仍然活跃。为求解数据流转化等式，ASER 使用了基于工作列表的迭代算法[12]。

表 1 针对寄存器 R 的转化函数定义

$$DEAD_{out}(block)=\begin{cases}\{\bot\} & SUCC(block)=\phi \\ \bigcup_{block'\in SUCC(block)} DEAD_{in}(block') & \text{otherwise}\end{cases}$$

$$DEAD_{in}(block)=\begin{cases}\{block\} & R\in KILL(block) \\ \phi & R\in GEN(block) \\ DEAD_{out}(block) & \text{otherwise}\end{cases}$$

$$END_{out}(block)=\begin{cases}\{\bot\} & SUCC(block)=\phi \wedge R\in\{\$gp,\$sp,\$fp,\$ra\} \\ \phi & SUCC(block)=\phi \wedge R\notin\{\$gp,\$sp,\$fp,\$ra\} \\ \bigcup_{block'\in SUCC(block)} END_{in}(block') & \text{otherwise}\end{cases}$$

$$END_{in}(block)=\begin{cases}\phi & R\in KILL(block) \\ \{block\} & R\in GEN(block)\wedge(R\in DEF(block)\vee DEAD_{out}(block)\neq\phi) \\ END_{out}(block) & \text{otherwise}\end{cases}$$

在上述分析的基础上，ASER 采用如下步骤求得关于寄存器 R 的所有生命周期：1）选择一个还没有被分析且对 R 有定值的基本块 $block$，首先根据 $block$ 的指令序列确定位于 $block$ 内部 R 的各个生命周期，然后把最后一条定值的指令记为 def；2）逐个遍历 $END_{out}(block)$ 中的元素 $block'$，通过 $block'$ 的指令序列确定首次对 R 定值之前最后引用 R 的指令，记为 end，并把 $block'$ 首次对 R 定值指令记为 $dead$，即构造得 R 的一个生命周期；3）如果 $block'$ 中没有对 R 的定值指令，则把 $block'$ 中最后引用 R 的指令记为 end，并逐个遍历 $DEAD_{out}(block')$ 中的每个元素 $block''$，再通过 $block''$ 的指令序列确定在 $block''$ 中首次定值 R 的指令，记为 $dead$，即构造得 R 的一个生命周期；4）如果 $block'$ 中没有对 R 的定值指令且 $END_{out}(block')$ 不为空，则把 $block'$ 记为 $block$，循环执行步骤 2）到 4）；5）把 $block$ 标记为已分析，如果还有未分析且对 R 有定值的基本块，则转向 1），否则结束。由于基本块总数有限，而且生命周期定义使得一个基本块不会在步骤 2）到 4）的循环中重复出现，因此上述过程确定会结束，最后获得关于 R 的所有生命周期。

2.2 计算活跃时间

如果活跃区间在一个基本块内部，根据基本块指令序列，从定值点 def 到结束点 end 之间执行的指令数即等于该活跃区间的执行时间。如果活跃区间跨越基本块，需要首先明确从 def 所在基本块（记为 B_{def}）到 end 所在基本块（记为 B_{end}）的执行路径 $PATH_{live}$，然后再计算 $PATH_{live}$ 所需的执行时间。

两个基本块之间的执行路径可以用由控制流图中的边、'+'、'·'、'*'和括号构成的表达式表示，'+'、'·'和'*'分别表示路径合并、路径连接和路径循环，括号用来确定优先级，在无括号限制的情况下，从'+'、'·'到'*'的优先级依次递增。例如图 2 中，从 B_2 到 B_4 的可能路径有 $b\cdot d$ 和 c，所以 B_2 到 B_4 路径表达式为 $b\cdot d+c$，存在从 B_4，B_2，B_3 再到 B_4 的循环，所以有 B_4 到 B_4 的路径循环 $(e\cdot(b\cdot d+c))*$，从 B_4 到 B_6 的路径可以表示为 $(e\cdot(b\cdot d+c))*\cdot h$。

程序控制流图可以用基本块邻接矩阵来表示，矩阵的行和列分别表示基本块，矩阵的具体元素用从行对应块到列对应块在控制流图中的边表示。然后，ASER 使用约简该邻接矩阵的方法[13]求 B_{def} 到 B_{end} 的路径表达式：逐个选择除 B_{def} 和 B_{end} 之外的基本块 $block$，根据 $block$ 在原矩阵中所在的行和列求所有经过 $block$ 的执行路径表达式，然后与原矩阵中不经过 $block$ 的执行路径合并，并且删除 $block$ 所在的行和列。如此迭代，直到剩下一个由 B_{def} 和 B_{end} 构成的二维矩阵，$PATH_{live}$ 的表达式由该二维矩阵求得。根据活跃区间的定义，除 B_{def} 和 B_{end} 外不能再有对目标寄存器有定值的基本块，所以 ASER 在约简矩阵之前，首先删除 B_{def} 和 B_{end} 之外对目标寄存器有定值的基本块邻接边，由此可显著提高约简效率。

例如，为了求示例程序从 B_1 到 B_6 的活跃区间的路径表达式，由于 B_5 会对 R 定值，因此首先删除 B_5 的邻接边 f 和 g，最后得到的约简结果如下所示，即寄存器 R 从 B_1 到 B_6 的活跃区间的路径表达式为 $a\cdot(b\cdot d+c)\cdot(e\cdot(b\cdot d+c))*\cdot h$。

$$\begin{array}{c|cc|} & B_1 & B_6 \\ B_1 & & a\cdot(b\cdot d+c)\cdot(e\cdot(b\cdot d+c))*\cdot h \\ B_6 & & \end{array}$$

一条路径的执行时间等于所包含基本块的指令数之和，但分支和循环的存在使得各个基本块在同一路径中的执行概率并不相同。基于路径表达式，ASER 先根据运算符'·'、'+'和'*'确定各个子路径在整个路径中的执行概率，然后根据所连接基本块的指令数确定整个路径的执行时间。根据运算符的定义和文献[11]的分析，ASER 按如下规则计算边和子路径的执行概率 $prob: PATH \rightarrow [0,1]$。

• $prob(\alpha)=P_E(edge)$，如果路径 α 只包含边 $edge$；

• $prob(\alpha\cdot\beta)=prob(\alpha)\times prob(\beta)$；

• $prob(\alpha+\beta)=prob(\alpha)+prob(\beta)$；

• $prob((\alpha)*)=1/(1-prob(\alpha))$，$prob((\alpha)*)$是循环入口处的执行概率，实际循环体 α 在整个循环中的执行概率为 $prob(\alpha)\times prob((\alpha)*)=prob(\alpha)/(1-prob(\alpha))$。

然后，ASER 按如下规则进行计算路径的执行时间 $time$：$PATH\rightarrow Real$，其中 $COUNT(block)$表示基本块 $block$ 包含的指令计数。

• $time(\alpha)=COUNT(source(edge))$，如果路径 α 只包含边 $edge$；

• $time(\alpha\cdot\beta)=time(\alpha)+time(\beta)$；

• $time(\alpha+\beta)=time(\alpha)\times prob(\alpha)/prob(\alpha+\beta)+time(\beta)\times prob(\beta)/prob(\alpha+\beta)$；

• $time((\alpha)*)=time(\alpha)\times prob(\alpha)\times prob((\alpha)*)$。

最后，根据定值点 def 和结束点 end 在 B_{def} 和 B_{end} 中的位置对 $time(PATH_{live})$进行调整，计算得生命周期的活跃时间。

在图 2 示例程序中，边 b,c,d 和 e 的执行概率分别为 0.25，0.75，1 和 0.8，所以子路径 $b\cdot d$，c 和 $e\cdot(b\cdot d+c)$的执行概率分别为 0.25，0.75 和 0.8，使得循环$(e\cdot(b\cdot d+c))*$的循环体执行概率为 4。假设每个基本块只有一条指令，那么计算得 $a\cdot(b\cdot d+c)\cdot(e\cdot(b\cdot d+c))*\cdot h$ 的 $time$ 值为 12.25 个周期。

2.3 计算出现频率

如果生命周期在一个基本块内部，那么其出现频率 $freq$ 值等于该基本块在整个程序中的执行频率。如果生命周期的结束点 end 和死亡点 $dead$ 属于不同的基本块，则出现频率 $freq$ 值不仅与活跃区间相关，而且与死亡区间相关。如图 3 左子图所示，B_i 只有对于 B_n 来说才是活跃区间的结束点，对于 B_m 来说其结束点是 B_j。所以对于结束点为 B_i、死亡点为 B_n 的生命周期来说，在计算出现频率时要求其生命周期一定是在 B_n 终止的。反之，如果仅仅考虑 B_i 和起始定值点之间的情况，当 B_j 的起始定值点和 B_i 相同时，那么会重复计算部分结束点为 B_j、死亡点为 B_m 的生命周期。

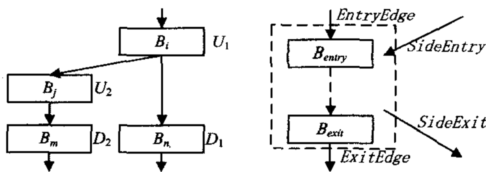

图 3 计算生命周期出现频率

同样，用约简邻接矩阵的方法求得从 end 到 $dead$ 的路径 $PATH_{dead}$，最后得关于生命周期的完整路径$(PATH_{live})\cdot(PATH_{dead})$。在求 $PATH_{dead}$ 之前，也可通过删除 end 和 $dead$ 所在基本块外所有对目标寄存器有定值和引用的基本块邻接边，以提高分析效率。

$(PATH_{live})\cdot(PATH_{dead})$中的边和连接的基本块构成原程序控制流图的一个子图，例如图 2 中的虚线框就表示了 $a\cdot(b\cdot d+c)\cdot(e\cdot(b\cdot d+c))*\cdot h$ 划分成的子图。对于整个程序控制流图来说，$(PATH_{live})\cdot(PATH_{dead})$构成的子图除了 $PATH_{live}$ 的入边(记为 $EntryEdge$)和 $PATH_{dead}$ 的出边(记为 $ExitEdge$)，还有其它旁入边(记为 $SideEntry$)和旁出边(记为 $SideExit$)，如图 3 右子图所示。执行$(PATH_{live})\cdot(PATH_{dead})$的概率应该等于从 $EntryEdge$ 进入的概率乘以从 $ExitEdge$ 退出的概率。由于子图的执行频率等于 $F_E(EntryEdge)+F_E(SideEntry)$，因此$(PATH_{live})\cdot(PATH_{dead})$的执行频率(即 $freq$ 值)应该等于：

$$\frac{F_E(EntryEdge)\times F_E(ExitEdge)}{(F_E(EntryEdge)+F_E(SideEntry))}$$

图 2 关于寄存器 R 的示例生命周期中，结束点和死亡点都在 B_6 内，所以完整路径仍为 $a\cdot(b\cdot d+c)\cdot(e\cdot(b\cdot d+c))*\cdot h$。在程序控制流图中，该路径构成子图有入边 a、出边 h，有旁入边 g，所以执行频率为 $F_E(a)\times F_E(h)/(F_E(a)+F_E(g))=8\times10/(8+2)=8$，即该生命周期的 $freq$ 值为 8。

2.4 实现算法及复杂度分析

在上述分析的基础上，实现算法如表 2 所列。首先在第 6、7 步确定每个基本块的 GEN，$KILL$ 和 DEF 函数。然后针对每个寄存器 R，先分析各个基本块的 $dead$ 和 end 状态(第 9、10 步)，在此基础上求得所有关于 R 的生命周期，然后针对 R 的每个生命周期，通过 def 到 end、end 到 $dead$ 的执行路径表达式求活跃时间和出现频率(13－16 步)。

表 2 ASER 算法

```
1. PROCEDUCE ASER(CFG,REG)
2. INPUT:CFG the control flow graph,REG registers list
3. OUPUT:ACE_bit the architectural correct execution bits of registers in REG
         for the target program
4. BEGIN
5.    ACE_bit ←0
6.    FOR(each block in BLOCK) DO
7.       KILL,GEN and DEF functions analysis for block
8.    FOR(each R in REG) DO
9.       analyzing the DEAD state for each block through fixpoint analysis
10.       analyzing the END state for each block through fixpoint analysis
11.       LIFESET←calculating all possible life intervals for R
12.       FOR each interval in LIFESET DO
13.          PATH_live←finding the path from def to end
14.          PATH_dead←finding the path from end to dead
15.          time←calculating execution time for PATH_live
16.          freq←calculating execution frequency for(PATH_live)·(PATH_dead)
17.          ACE_bit = ACE_bit + time×freq×R.length
18.       ENDFOR
19.    ENDFOR
20.    RETURN ACE_bit
21. END
```

设程序中有 b 个基本块、e 条边，使用了 r 个寄存器。已知第 9、10 步不动点分析算法的时间复杂度为 $O(b\cdot e)$[12]，第 11 步得到的生命周期量级为 $O(b^3)$，第 13、14 步约简邻接矩阵得到的执行路径表达式中边的最高量级为 $O(b^3\cdot e)$，所以第 15 步导致内层循环的时间复杂度是 $O(b^6\cdot e)$。程序总的最高时间复杂度是 $O(b^6\cdot e\cdot r)$，仍是多项式时间。实际上由于寄存器访问频繁，在分析具体 $PATH_{live}$ 和 $PATH_{dead}$ 时根据读写限制可以删除大量无关边，使得分析效率得到显著提升。在我们针对一组小规模基准程序的分析实验中，平均所需时间只有 0.3s。

程序控制流图和基本块邻接矩阵的使用空间都是 $O(b^2)$ 量级，GEN，$KILL$ 和 DEF 函数所需空间是 $O(b\cdot r)$，而 $dead$ 和 end 状态的空间复杂度是 $O(b^2)$，第 11 步得到的生命周期和约简邻接矩阵使用的临时矩阵所需的空间都为 O

(b^3)，所以算法的空间复杂度为 O(b^3)。

3 实验与分析

我们用 Java 语言具体实现了 ASER 方法的原型，并选择 6 个小规模的基准程序(程序说明见表 2 第 2 列)进行了分析实验。先用 GCC 3.3.3 把程序交叉编译成 MIPS32 格式的汇编代码，并基于程序对在一个 MIPS32 模拟器 SPIM[14] 运行得到的运行剖面信息进行分析。ASER 的分析目标是实验程序用到的通用寄存器，分析结果如表 3 所列。

表 3 分析实验结果

program	description	T_{prog} [cycles]	ACE_{bit} [bit-cycles]	RVF [%]	$1-P_{prog}$ [$\times 10^{-14}$]
bsort	冒泡排序	6,892	937,430	60.52	5.17
qsort	快速排序	3,053	402,974	41.04	2.22
gauss	高斯消元法	7,237	1,085,454	57.94	6.00
mm	矩阵乘法	466,253	80,632,518	60.04	445.09
shuf	随机数测试	1,269,524	183,942,016	50.30	1015.35
monte	蒙特卡罗法求 π	1,314,121	191,463,584	50.59	1056.88

表 3 第 3 列是程序一次运行平均所需时间，第 4 列是分析得到的寄存器 ACE 的位数，即在程序运行过程中发生在这些 bit-cycle 的寄存器软错误都有可能影响程序执行。第 5 列 *RVF* 是 ACE_{bit} 与程序使用到的寄存器所有可能发生的软错误的比值，值越小则表示受寄存器软错误的影响越小，如‘qsort’。为计算程序可靠性，软错误率 λ 取 5.52×10^{-20} errors/bit-cycle，它相当于在主频为 25MHz 的处理器中每兆数据一天内发生一次软错误。第 6 列给出程序平均一次执行的失效概率 $1-P_{prog}$，由于程序规模和 λ 取值较小，实验程序一次执行的可靠性都非常高。一个基本规律是：程序的时空复杂度越高，软错误产生和传播的机会就越大，可靠性则越低，如‘monte’。

图 4 给出具体寄存器在这 6 个程序中的平均 RVF 值，AVG 组是所有寄存器的平均值。由于寄存器很多时候处于空闲状态，因此很多发生在寄存器中的软错误不会影响到程序运行(AVG 组值为 48.26%)。而且各个寄存器之间存在明显差异，$gp 作为全局指针寄存器，在所有程序的运行过程中都没有被重新修改过。而 $fp 作为帧指针寄存器在进入子程序后被写入新值，直到子程序退出后才会被更改，所以 $gp 和 $fp 的 RVF 值几乎等于 100%。$t9 是在子程序调用时才会被使用的临时寄存器，活跃时间非常短，导致 RVF 值最小。所以，各个寄存器对程序可靠性的影响程度并不相同，这也是文献[7,8]等采取部分保护(Partial Protection)寄存器工作的出发点，即优先保护那些对可靠性影响大的寄存器(如 $gp)。

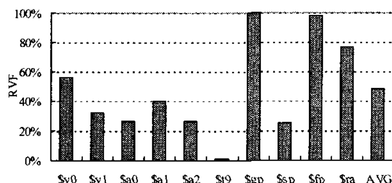

图 4 寄存器的 RVF 值

ASER 方法的分析结果进一步指出哪些可能影响程序正确执行的寄存器的具体生命周期的活跃时间和出现频率。图 5 选择了 $v1，$a0 和 $sp 这 3 个寄存器，给出了它们不同活跃时间的生命周期在各自 RVF 中所占的比例，最后一组是平均值。可以发现，虽然这 3 个寄存器的平均 RVF 值相当，但是各个程序之间却有所区别，而且具体生命周期的活跃时间和出现频率之间存在很大差异：$v1 的大部分活跃时间都小于 5 个周期，$a0 的大部分活跃时间都大于 5 个周期而小于 20 个周期，但是 $sp 很多时候活跃时间在 200 个周期以上。所以，可以根据 ASER 的分析结果对那些影响大的寄存器活跃区间优先采取加固措施，例如 $sp 大于 200 个周期的活跃区间。而如果仅仅从 RVF 指标出发先对 $v1 进行保护，由于 $v1 会被频繁写入新的数据，采用 ECC 编码等具体容错实现时会带来较大的功耗和性能开销，因此 ASER 的分析结果有利于实现更加有效的容错策略。

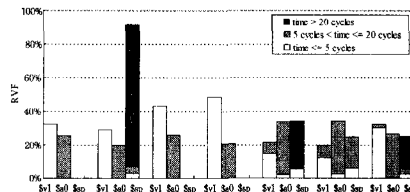

图 5 不同时长活跃区间的分布

为验证分析结果的有效性，我们还在 SPIM 模拟器上进行了 AVF 分析实验。AVF 分析实验结果与 ASER 分析对比如图 6 所示，AVG 组是平均值。可以发现，两种分析结果的一致性非常好，ASER 和 AVF 方法分析得到的 RVF 平均值分别是 53.41%和 53.39%。因此，在程序执行剖面信息的基础上，ASER 方法对实验程序的分析精度是很高的。

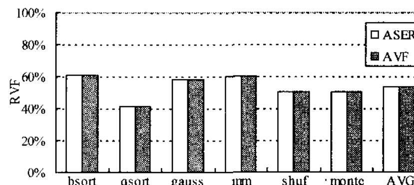

图 6 ASER 与 AVF 分析结果比较

结束语 在现代微处理器中，发生在寄存器中的软错误是影响系统可靠性的关键因素之一。本文提出的 ASER 方法通过一系列静态分析技术，可以定量分析寄存器软错误对程序可靠性的影响。实验表明，ASER 方法的分析结果与 AVF 方法保持一致，并且 ASRE 的分析结果可以指出具体影响程序可靠性的寄存器活跃区间的执行时间和出现频率，有利于实现更加高效的容错策略。

目前 ASER 方法还没有考虑上下文敏感性(context-sensitive)。下一步将重点解决这个问题，并且把 ASER 方法应用于大规模程序，以验证 ASER 的分析能力。同时计划在 ASRE 方法的基础上，研究针对寄存器软错误的高效容错技术。

参 考 文 献

[1] Baumann R C. Radiation-induced soft errors in advanced semiconductor technologies [J]. IEEE Trans on Device and Materials Reliability, 2005, 5(3): 305-316

[2] International technology roadmap for semiconductors 2007 executive summary [EB/OL]. http://www.itrs.net/

[3] Shivakumar P, Kistler M, Keckler S W, et al. Modeling the effect of technology trends on the soft error rate of combinational logic [C]//Proc. of the Int'l Conf. on Dependable Systems and Networks. 2002: 389-399

(下转封三)

并且还可获得可重构资源占用信息。协同函数的软硬件执行时间使用专门的时间测量函数获得。

4.3 目标应用程序运行时支持

协同函数调度器生成的目标应用的运行时约束文件如表1所列。通过约束文件对目标应用程序进行控制，使程序的执行更加灵活高效，方便控制。应用设计人员可手动更改某个协同函数的执行方式。若将硬件执行方式的协同函数指定为软件方式执行，需将执行方式列的H改为S，并将位置布局约束对应列清空（设置为“无”）；若将软件执行方式改为硬件执行方式，相对复杂一些，不仅要将执行方式列的S改为H，还要修改所占用的位置约束列，不能使硬件实现的协同函数位置重叠。协同函数约束文件在目标应用运行前被加载，用来约束目标应用中被调用的协同函数。具体加载过程由专门设计的 loadConstraintFile(char * path)函数实现，并在运行环境中建立约束信息链表。

表1 约束文件所包含信息

协同函数名	占用列数	执行方式	布局约束	硬件配置文件存储路径
hamming_encode	3	H	1	/cofun/hamming/encode. bit
hamming_decode	3	H	4	/cofun/hamming/decode. bit
tri_des_encrypt	5	S	无	/cofun/3des/encrypt. bit
tri_des_decrypt	5	H	1	/cofun/3des/decrypt. bit
draw_line	2	S	无	/cofun/shape/line. bit
draw_circle	2	H	6	/cofun/shape/circle. bit

协同函数的不同实现方式的切换是通过修改的动态链接器（Linux 的动态链接器为 ld-linux. so）实现的。动态链接器截取未定位的协同函数名，查询软件函数名对应的地址及硬件接口函数名对应的地址，根据约束文件中的信息让应用程序跳到正确的函数地址处执行。修改的动态链接器可以通过设置的环境变量来区别普通共享库函数和软硬件协同库函数。若是普通共享库函数，保持加载解析过程不变。可重构资源管理器基于可重构片上系统的内部配置访问端口（ICAP）模块实现，系统将 ICAP 作为一个字符设备集成到 Linux 内核中，依据约束信息链表中的位置约束，配置动态可重构区域。有了上述各模块的支持，目标应用程序就可以顺利执行。

结束语 以协同函数库为基础的过程级软硬件协同设计方法的目标是解决可重构片上系统存在的设计及编程困难问题。本文分别从目标应用系统的描述、综合和运行时环境3个阶段展开，研究了过程级软硬件协同设计方法流程。我们给出了软硬件协同函数的结构和设计方法；专门设计了协同函数调度器工作模型，并对目标应用程序的运行环境作了必要的修改，以支持调度器所产生的目标应用约束文件的运行时注册，为目标应用程序提供了完善的执行环境；验证了协同设计方法的关键部分设计及可行性。本文方法对应用设计人员屏蔽了可重构片上系统的软硬件实现细节，降低了可重构片上系统的设计与编程复杂度。另外，目标应用程序的执行方式由综合阶段生成的约束文件控制，既增加了程序的灵活性，又降低了系统运行时额外的开销。

在目标应用约束文件生成流程中，构造高效的调度器搜索算法，是下一步要深入研究的问题。

参 考 文 献

[1] Wayne W. A Decade of Hardware/Software Co-design[J]. Computer, 2003, 36(4): 38-43

[2] Stitt G, Vahid F. A Decompilation Approach to Par-titioning Software for Microprocessor/FPGA Platfor-ms[C]// Proceedings of the Conference on Design, Automation and Test in Europe. 2005: 396-397

[3] Stitt G, Guo Z, Vahid F, et al. Techniques for Synt-hesizing Binaries to an Advanced Register/Memory Structure[C]// Proc. of ACM Symp. on FPGA. 2005: 118-124

[4] Hayden K S, Brodersen R. A Unified Hardware/So-ftware Runtime Environment for FPGA-based Rec-onfigurable Computers using BORPH[J]. ACM Tra-nsactions on Embedded Computing Systems, 2008, 7(2): 59-64

[5] Wiangtong T, Cheung P, Luk W. Hardware/software co-design: a systematic approach targeting dataintensive applications[J]. IEEE Signal Processing Magazine, 2005, 22(3): 14-22

[6] Huang Wei, Edward J. Column-based Precompiled Configuration Techniques for FPGA[C]// Proceedings of the the 9th Annual IEEE Symposium on Field Programmable Custom Computing Machines. 2001: 137-146

[7] 吴强，边计年，薛宏熙. 基于抽象体系结构模板的多路软硬件划分算法[J]. 计算机辅助设计与图形学学报，2004，16(11)：1562-1567

[8] Xilinx Inc. Application Note: Virtex Series Con? gu-ration Architecture User Guide[EB/OL]. http://www. xi-linx. com/support/documentation/application_notes/xap-p151. pdf, 2004

[9] Xilinx Inc. Virtex-II Pro Data Sheets[EB/OL]. http://ch-ina. xilinx. com/support/documentation/virtex-ii_pro_da ta_sheets. htm, 2007

（上接第 294 页）

[4] Blome J A, Gupta S, Feng S, et al. Cost-efficient Soft Error Protection for Embedded Microprocessors [C]// Proc of the Int'l Conf on Compilers, Architecture and Synthesis for Embedded Systems. 2006: 421-431

[5] Hsueh M C, Tsai T K, Iyer I K. Fault Injection Techniques and Tools [J]. IEEE Computer, 1997, 30(4): 75-82

[6] Mukherjee S S, Weaver C, Emer J, et al. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-performance Microprocessor [C]// Proc of the 36th Int'l Symp on Microarchitecture. 2003: 29-40

[7] Lee J, Shrivastava A. Static Analysis to Mitigate Soft Errors in Register Files [C]// Proc of the Design, Automation, and Test in Europe. 2009: 1367-1372

[8] Montesinos P, Liu W, Torrellas J. Using Register Lifetime Predictions to Protect Register Files Against Soft Errors [C]// Proc. of the 37th Int'l Conf. on Dependable Systems and Networks. 2007: 286-296

[9] MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set [R]. MIPS Technologies Corp, 2002

[10] Ziegler J F. IBM Experiments in Soft Fails in Computer Electronics(1978-1994) [J]. IBM Journal of Research Development, 1996, 40(1): 3-18

[11] Wu Y, Larus J. Static Branch Frequency and Program Profile Analysis [C]// Proc of the 27th Int'l Symp on Microarchitecture. 1994: 1-11

[12] Nielson F, Nielson H R, Hankin C. Principles of Program Analysis(2nd)[M]. New York: Springer-Verlag, 2005

[13] Lunts A G. A Method of Analysis of Finite Automata [J]. Soviet Physics, 1965(10): 102-103

[14] SPIM: A MIPS32 Simulator [EB/OL]. http://pages. cs. wisc. edu/～larus/spim. html