UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RAFAEL BALDIATI PARIZI

Implementação e Avaliação da Técnica ACCE para Detecção e Correção de Erros de Fluxo de Controle no LLVM

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Álvaro Freitas Moreira
Orientador

Prof. Dr. Luigi Carro
Co-orientador

Porto Alegre, maio de 2013

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Parizi, Rafael Baldiati

Implementação e Avaliação da Técnica ACCE para Detecção e Correção de Erros de Fluxo de Controle no LLVM / Rafael Baldiati Parizi. – Porto Alegre: PPGC da UFRGS, 2013.

105 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2013. Orientador: Álvaro Freitas Moreira; Co-orientador: Luigi Carro.

1. Tolerância a falhas. 2. Erros de fluxo de controle. 3. ACCE. 4. LLVM. I. Moreira, Álvaro Freitas. II. Carro, Luigi. III. Título.

“Ao ser produzido, o conhecimento novo supera outro
que antes foi novo e se fez velho
e se dispõe a ser ultrapassado por outro amanhã.”
— PAULO FREIRE

AGRADECIMENTOS

Agradeço a Deus por me disponibilizar uma vida repleta de desafios que, ao superá-los, encho-me de disposição e coragem para seguir crescendo pessoal e profissionalmente. Agradeço à minha mãe, que sempre se fez presente na minha vida, assumindo o papel de pai e mãe, mostrando-me o caminho e me ensinando a seguir em frente. Ao meu pai, que de algum lugar me protege e me dá forças para batalhar pelos meus objetivos. Aos demais familiares, que me ajudam das mais variadas formas, com ações de apoio e carinho. Aos meus amigos, pelo companheirismo, amizade e que entenderam minha ausência em diversas festas e confraternizações. Ao CNPQ, pelo auxílio, fundamental e imprescindível para a minha subsistência em Porto Alegre. Ao meu orientador, Professor Álvaro Moreira, pelo apoio e ensinamentos, acompanhando constantemente as atividades relacionadas ao trabalho. Ao professor Luigi Carro e ao Ronaldo Ferreira, que colaboraram de forma efetiva no desenvolvimento do trabalho. Aos colegas do laboratório 67-202 no Instituto de Informática da UFRGS, pelo companheirismo e amizade. Por fim, a todos que direta ou indiretamente colaboraram para minha formação profissional, pessoal e acadêmica, Obrigado!

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ABFT	Algorithm Based Fault Tolerance
ACCE	Automatic Correction of Control-flow Errors
A_sig	Assinatura Semelhante
CEDA	Control flow Errors Detection through Assertions
CFC	Control Flow Checking
CFE	Control Flow Error
CFG	Control Flow Graph
CFCSS	Control Flow Checking by Software Signatures
CRC32	32-bit Cyclic Redundancy Check
FEH	Function Error Handler
FFT	Fast fourier Transform
GCC	Gnu Compiler Collection
GDB	Gnu Debbuger
GEH	Global Error Handler
IDE	Integrated Development Environment
LLVM	Low Level Virtual Machine
LLVM-IR	Low Level Virtual Machine Intermediate Representation
NES	Node Exit Signature
Net	Network
NS	Node Signature
NT	Node Type
PC	Program Counter
S	Signature
SSA	Single Static Assignment
SIHFT	Software Implemented Hardware Fault Tolerance
YACCA	Yet Another Control flow Checking Approach

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Técnicas de prevenção de falhas como testes e verificação de software não são suficientes para prover dependabilidade a sistemas, visto que não são capazes de tratar falhas ocasionadas por eventos externos tais como falhas transientes. Nessas situações faz-se necessária a aplicação de técnicas capazes de tratar e tolerar falhas que ocorram durante a execução do software.

Grande parte das técnicas de tolerância a falhas transientes está focada na detecção e correção de erros de fluxo de controle, que podem corresponder a até 70% de erros causados por esse tipo de falha. Essas técnicas tratam as falhas em nível de software, alterando o programa com a inserção de novas instruções que devem capturar e corrigir desvios inesperados ocorridos durante a execução do software, sendo ACCE uma das técnicas mais conhecidas.

Neste trabalho foi feita uma implementação da técnica ACCE através da criação de um passo de transformação de programas para a infraestrutura de compilação LLVM. ACCE atua sobre a linguagem intermediária dos programas compilados com o LLVM, resultando em portabilidade de linguagem de programação e de arquitetura de máquina.

Além da implementação da técnica como um passo de transformação, o LLVM foi utilizado para a realização dos experimentos para avaliar o impacto na eficácia de ACCE quando aplicada em programas previamente otimizados por outras transformações. Esse tipo de avaliação é fundamental uma vez que dificilmente a compilação de programas é feita sem a ativação de otimizações, e, até onde sabemos, nunca havia sido feito anteriormente.

Os experimentos deste trabalho foram realizados através de baterias de injeção de falhas em programas da suíte de *benchmarks* Mibench, divididas em diferentes cenários, que avaliaram ACCE em termos de correção de falhas, quando aplicada em programas otimizados por transformações individuais e também por combinações de transformações.

Os resultados dos experimentos realizados mostram que a técnica ACCE é eficaz na correção de falhas, porém, para alguns programas otimizados por determinadas transformações, houve redução na correção de falhas. Esse trabalho analisa os experimentos nos quais houve redução da eficácia de ACCE e aponta possíveis causas.

Palavras-chave: Tolerância a falhas, Erros de fluxo de controle, ACCE, LLVM.

Implementation and Evaluation of the ACCE Technique to Detection and Correction of Control Flow Errors in the LLVM

ABSTRACT

Computer-based systems are used in several eletronic devices that are, in many cases, responsable by the execution of critical tasks. There are situations where techniques of prevention against faults such as software validation and verification, can not be sufficient for ensuring acceptable rates of confiability, because they are not capable of treating faults that occur in execution time, such as transient faults.

Most of the fault tolerance techniques for transient faults are focused in detection and correction of control flow errors, that can correspond to 70% errors caused by this kind of faults. These techniques treat the faults at software level, changing the program with the insertion of new instructions that must to capture and to correct illegal jumps occurred during the software execution, being ACCE the most known technique today.

In this work an implementation of the ACCE technique was developed as a program transformation pass in the LLVM compiler infraestructurre. ACCE acts over the intermediate language of LLVM, which results in both programming language and machine architecture language portabilities.

Besides the implemetation of the technique like a transformation pass, the LLVM was also used in the experiments for the avaliation of impact in the ACCE eficacy when it is applied into programs previously optimized by others compiler transformations. This evaluation is essential since hardly the compilation of programs is made without the activation of other optimizations. As far as we know this kind of evaluation has never beem made before.

The experimental results show that the ACCE techinque is effective in the fault correction, but for some programs optimized by specific transformations, there was a reduction in the correction rate. This work analyses these experiments and gives an explanation for what causes a reduction in the effectiveness of ACCE.

Keywords: Fault Tolerance, Control-flow Errors, ACCE, LLVM.

1 INTRODUÇÃO

1.1 Contexto e Motivação

Sistemas eletrônicos estão constantemente expostos a eventos externos, de ordem natural ou artificial, que podem provocar comportamentos inesperados devido à ocorrência de falhas. Em grande parte das vezes essas falhas não persistem durante toda a execução do sistema, sendo conhecidas como *falhas transientes*.

A redução no tamanho dos transistores é fundamental para a evolução tecnológica, e estima-se que nos próximos 10 anos os transistores terão um tamanho de 7,4 *nm* comparados com os atuais 22*nm* (JEONG; KAHNG, 2009). Essa redução resulta em ganhos de desempenho e economia de energia porém, compromete a confiabilidade dos sistemas pois os tornam suscetíveis a falhas transientes (WEE; MASTIPURAM, 2004).

Grande parte das falhas transientes atingem o hardware de um sistema computacional e podem levar ao surgimento de erros de fluxo de controle (ou CFEs para *Control Flow Errors*).

Para o tratamento de falhas transientes, foram desenvolvidas técnicas para a detecção e correção de CFEs, seja em nível de hardware ou de software. Técnicas de hardware requerem modificações em componentes físicos, como por exemplo, *Triple Modular Redundancy* (LYONS; VANDERKULK, 1962), técnica que triplica os módulos do sistema de forma a garantir que, caso uma falha ocorra em algum dos dispositivos, há outros que realizarão as operações de forma correta. Por outro lado, técnicas de software não realizam modificações em componentes físicos, apenas inserem código extra para a detecção e correção dos erros de fluxo de controle e, por isso, são conhecidas como *Software Implemented Hardware Fault Tolerance- SIHFT* ou ainda *Algorithm Based Fault Tolerance - ABFT* (OLGA et al., 2006).

Mecanismos de tolerância a CFEs tem sido objeto de estudo de diversos autores sendo que (VEMU; GURUMURTHY; ABRAHAM, 2007) e (OH; SHIRVANI; MCCLUSKEY, 2002) afirmam inclusive que 70% das falhas transientes levam a erros de fluxo de controle.

Técnicas de tolerância a falhas com base em alterações nos elementos físicos não são viáveis e em muitos casos infactíveis para sistemas embarcados, visto que não atendem as necessidades de tais sistemas tais como tamanho reduzido e baixo consumo energético, tornando as técnicas baseadas em software atrativas para esse tipo de sistema.

Uma técnica SIHFT que permite detecção e correção de erros de fluxo de controle com elevada taxa de cobertura de falhas é a técnica ACCE (*Automatic Correction of Control Flow Errors*) (VEMU; GURUMURTHY; ABRAHAM, 2007). De forma similar a outras técnicas de software, a técnica ACCE modifica programas durante o processo de compilação inserindo código extra. Vale salientar que, apesar de serem eficazes para o reestabelecimento do fluxo de controle, as técnicas SIHFT são insuficientes sem a aplicação

de algum mecanismo de correção do fluxo de dados.

A técnica ACCE foi originalmente proposta para o compilador GCC. No artigo (VEMU; GURUMURTHY; ABRAHAM, 2007) foram feitas análises da eficácia da técnica quanto à detecção e correção de falhas, mas nada foi dito sobre a relação entre ACCE e outras transformações/otimizações disponíveis em compiladores. Os artigos originais também não detalham aspectos da sua implementação tal como a forma de inserção de novas instruções e modificação de programas, sendo a implementação realizada neste trabalho, uma tentativa de reprodução dos conceitos apresentados.

1.2 Objetivos e Resultados Alcançados

O objetivo deste trabalho é disponibilizar a técnica ACCE no *framework* de compilação LLVM e também avaliar o impacto das transformações disponibilizadas por compiladores sobre as suas taxas de detecção e correção de erros de fluxo de controle.

A implementação da técnica foi realizada como um passo de transformação adicional para o compilador LLVM, que é uma infraestrutura de compilação *open source* que provê diversas transformações e análises de programas e é composto por diversos módulos que permitem a criação de novas transformações. O LLVM (*Low Level Virtual Machine*) (LATTNER; ADVE, 2002a), ganhou notoriedade nos meios industrial e acadêmico (LATTNER; ADVE, 2002b), (KOROBEYNIKOV, 2007), (ZHAO et al., 2012), sendo inclusive adotada em diversos projetos desenvolvidos pela Apple. Uma característica deste *framework* de compilação é a representação intermediária de programas, chamada de LLVM-IR, que além de prover portabilidade, facilita a implementação de diversas transformações/otimizações de programas.

Para os experimentos, um processo de injeção de falhas foi realizado em programas de *benchmark* da suíte Mibench, permitindo a obtenção dos índices de detecção e correção de CFEs.

Os dados resultantes dos experimentos realizados demonstram que a técnica ACCE é eficaz para grande parte dos programas compilados com as transformações disponibilizadas pelo LLVM, exceto para casos específicos que culminaram na redução da taxa de correção de falhas.

1.3 Organização do Texto

Este trabalho está organizado da seguinte forma: o Capítulo 2 caracteriza falhas transientes e erros de fluxo de controle causados por estas falhas.

No Capítulo 3 são apresentados detalhes do *framework* de compilação *Low Level Virtual Machine* e da sua representação intermediária de programas, a LLVM-IR.

No Capítulo 4, a técnica ACCE é descrita, em termos de obtenção das assinaturas e transformação de programas, além da sua implementação como um passo de transformação de programas para o compilador LLVM.

No Capítulo 5 são descritos os experimentos realizados nesse trabalho, além dos resultados alcançados, englobando detalhes de metodologia, os cenários de experimentação e detalhes da análise dos resultados obtidos, focando na cobertura de falhas, de programas otimizados pelas transformações fornecidas pelo LLVM.

Por fim, no Capítulo 6 são apresentadas conclusões, que mostram os fatores que provocaram a queda ou o aumento na cobertura de falhas quando da aplicação da técnica ACCE em programas transformados por otimizações disponíveis no LLVM. Além disso,

o capítulo 6 apresenta também alguns trabalhos futuros.

2 FALHAS TRANSIENTES E ERROS DE FLUXO DE CONTROLE

Este capítulo começa com uma breve discussão sobre o que são falhas transientes. A seguir, na Seção 2.2, são apresentados erros de fluxo controle causados por falhas transientes. Na Seção 2.3 são caraterizados os erros de fluxo de controle que são detectados e corrigidos pela técnica ACCE.

2.1 Falhas Transientes de Hardware

Sistemas computacionais estão constantemente expostos à eventos externos que podem comprometer o seu funcionamento, tal como a radiação, por exemplo. Partículas radioativas, ao atingirem o hardware dos sistemas computacionais, podem causar comportamentos inesperados. Esses comportamentos podem persistir por toda a vida útil do sistema ou podem ter a sua duração limitada. Neste último caso temos o que é chamado de *falhas transientes*

Esses comportamentos inesperados podem surtir efeito tanto na parte física do sistema computacional, com a inutilização da um componente devido a um alto aquecimento, por exemplo, como também podem resultar em alterações no software durante a sua execução.

Estudos mostram que cerca de 70% das falhas transientes que afetam o hardware se manifestam como erros de fluxo de controle de programas (CFEs) (VEMU; ABRAHAM, 2006).

Em última instância, CFEs levam o *program counter* (PC) a valores inesperados em tempo de execução. Exemplos de como o PC pode receber valores inesperados devido a ocorrência de falhas transientes são: um *bit flip* pode transformar temporariamente uma instrução que não é de desvio em uma instrução de desvio ou vice-versa, ou pode modificar o valor do operando de uma instrução de desvio.

2.2 Erros de Fluxo de Controle

O **fluxo de controle** de um programa pode ser representado por um *Control Flow Graph* (CFG). Em um CFG, os vértices representam os blocos básicos de um programa, e as arestas conectando blocos representam a possibilidade de transferência de controle de um bloco para outro. Um bloco básico é uma sequência de instruções onde a execução inicia na primeira instrução e termina na última, sem a presença de desvios, a não ser na última instrução. A Figura 2.1 apresenta um programa e o seu CFG correspondente.

A obtenção do conjunto de blocos predecessores e conjunto de blocos sucessores de um bloco é importante, pois com base nessas informações as técnicas de tolerância a

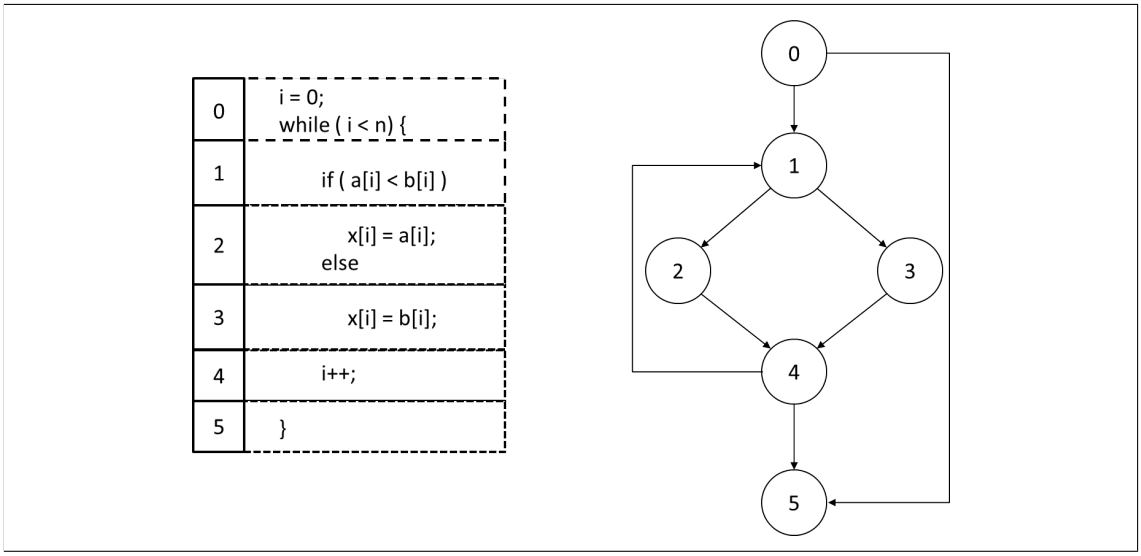

Figura 2.1: Exemplo de código fonte e seu grafo de fluxo de controle (OLGA et al., 2006)

falhas são capazes de verificar se o fluxo de execução do sistema está percorrendo um caminho esperado no grafo.

Um bloco A pertence ao conjunto de predecessores de um bloco B se e somente se houver uma aresta partindo de A com destino B. De forma similar, um bloco B é considerado sucessor de A se for o destino de uma aresta com origem em A. No CFG da Figura 2.1, usando como exemplo o bloco 4, os seus predecessores são os blocos 2 e 3, enquanto seus sucessores são os blocos 1 e 5.

Um **erro de fluxo de controle**, ou CFE (*Control Flow Error*) acontece quando ocorre um desvio não esperado na execução de um programa. CFEs podem ser classificados como *errados* ou *ilegais*:

- **CFEs errados:** acontecem quando, devido a uma falha, ao invés de executar o desvio de um bloco *A* para um bloco *B* sucessor de *A*, executa um desvio do bloco *A* para *C*, também sucessor de *A*, desvio esse não esperado para esse momento da execução. Usando como exemplo o CFG da Figura 2.1, um desvio errado acontece se o controle for desviado do bloco 0 para o bloco 5, quando o desvio esperado neste ponto do programa é do bloco 0 para o bloco 1.

- **CFEs ilegais:** ocorrem quando um desvio executado não existe no CFG, como por exemplo, caso ocorra um desvio do bloco 3 para o bloco 5, desvio este inexistente no CFG da Figura 2.1, ou caso ocorra um desvio para fora dos blocos do CFG original do programa. Um CFE também é do tipo ilegal quando é um desvio intra-bloco, ou seja, quando os blocos origem e destino do desvio são os mesmos (por exemplo, caso ocorra um desvio do bloco 3 para ele mesmo no CFG da Figura 2.1).

2.3 CFES detectados e corrigidos por ACCE

A técnica ACCE só consegue detectar CFEs ilegais, ou seja, desvios não existentes no CFG de programas. Isso porque a detecção de erros é baseada na verificação de assinaturas de blocos predecessores e sucessores (conforme será visto com mais detalhes no Capítulo 4). Um CFE errado não pode ser detectado por ACCE pois, nesse caso, o bloco destino de desvio errado faz parte do conjunto de blocos predecessores do bloco onde o

erro se manifestou.

Porém, nem todo CFE ilegal, é detectado por ACCE. ACCE é capaz de detectar somente CFEs *ilegais inter-blocos*, ou seja, desvios ilegais nos quais o bloco de destino é **diferente** do bloco de origem do desvio ilegal. ACCE não consegue, portanto, detectar CFEs ilegais nos quais o bloco destino é o mesmo bloco de origem do desvio (chamados de *CFEs ilegais intra-bloco*) ou quando o bloco destino não é bloco do programa.

ACCE consegue detectar todos os CFEs ilegais inter-blocos? A resposta é sim quando estamos falando de CFEs ilegais inter-blocos **não levando em conta** os blocos básicos contendo código que é adicionado, em tempo de compilação, pela transformação de programas correspondente a técnica ACCE. Blocos básicos "protetores" adicionados pela técnica ACCE **não são protegidos** pela técnica (mais detalhes no Capítulo 4).

A próxima pergunta que surge naturalmente é: ACCE consegue corrigir todas os CFEs que detecta? A resposta é não, ACCE, não consegue, por exemplo, corrigir CFEs quando uma falha corrompe instruções de verificação e atualização das assinaturas, impossibilitando a restauração do fluxo de controle do programa. Nesses casos, há um número limite definido para as tentativas de correção que, ao ser atingido, faz com que o programa seja encerrado para que não permaneça em um laço infinito de tentativas (mais detalhes no Capítulo 4).

Na Seção 2.4 é descrito como os CFEs são tratados com software e, além disso, são apresentados exemplos de técnicas que possibilitam esse tratamento. A leitura da Seção 2.4 pode ser omitida sem prejuízo para a compreensão do restante da dissertação.

2.4 Tratamento de CFEs com Software

O tratamento de erros de fluxo de controle com software é realizado com a transformação do programa através da inserção de instruções extras que capacitam-no contra os desvios ilegais ocorridos em tempo de execução. Essa transformação é realizada com base nos blocos básicos que compõem o programa, ou seja, conjunto de instruções executadas sequencialmente até a execução de uma instrução de desvio, que limita o final de um bloco.

Para cada bloco do programa são calculados e atribuídos, em tempo de compilação, valores que, durante a execução do programa, são avaliados quanto à correspondência em relação ao fluxo de controle. Tais valores são conhecidos como *assinaturas* e são calculados com base no CFG do programa, levando em consideração, para cada bloco, as relações com os predecessores e sucessores, de tal modo que, em corretas execuções do programas, as assinaturas sempre irão corresponder.

Um CFE pode ser detectado por meio de pontos de verificação (*checkpoints*) que comparam a assinatura de tempo de execução com uma pré-calculada em tempo de compilação do software. Em casos de divergência, há a ocorrência de um CFE.

Diversas técnicas de software permitem o tratamento de erros de fluxo de controle. Essas técnicas são conhecidas como (*Control-Flow Checking - CFC*) e utilizam as assinaturas e transformação de programas para tratar desse tipo de falhas transientes. Entre essas técnicas estão *Control Flow Checking by Software Signatures - CFCSS* (OH; SHIRVANI; MCCLUSKEY, 2002), *Yet Another Control flow Checking Approach - YACCA* (GOLOUBEVA et al., 2003) e *Control flow Errors Detection through Assertions - CEDA* (VEMU; ABRAHAM, 2011) .

Cada técnica possui suas características particulares, porém, as três capacitam programas para a detecção de erros de fluxo de controle, o que as diferencia da técnica ACCE

(detalhada no Capítulo 3), que além de detectar, possibilita que o fluxo de controle do programa seja corrigido.

Em termos de transformação de programas, as técnicas YACCA e CEDA são semelhantes, visto que inserem instruções no topo e no rodapé de cada bloco básico. Por outro lado, CFCSS se diferencia das demais já que realiza a inserção de instruções de checagem de assinaturas somente no topo de cada bloco básico.

O fator que diferencia CEDA de YACCA é a criação, por parte de CEDA, de parâmetros para a atualização das assinaturas em cada um dos blocos básicos do programa, em ambas as regiões, seja no topo ou no rodapé. Esses parâmetros possibilitam que a assinatura, em tempo de execução, seja sempre igual à esperada, em corretas execuções.

CEDA é a técnica de detecção de erros de fluxo de controle mais eficiente, se comparada com YACCA e CFCSS, comprovado com a realização de experimentos com programas do benchmark SPEC CPU 2000 (Spec Corporation, 2000), para os quais CEDA se configurou como 28% mais eficaz na detecção de falhas se comparada à CFCSS e 10% mais eficaz na detecção se comparada à YACCA (OLGA et al., 2006).

ACCE, apresentada em detalhes no Capítulo 4, é uma extensão da técnica CEDA, com a funcionalidade adicional de permitir a correção de desvios ilegais.

3 LLVM E LLVM-IR

Este capítulo apresenta em mais detalhes o framework de compilação LLVM e sua linguagem intermediária LLVM-IR. A Seção 3.1.1 mostra a arquitetura do *framework* destacando as etapas no processo de compilação e a Seção 3.1.2 da uma introdução a linguagem intermediária LLVM-IR.

3.1 LLVM

Low Level Virtual Machine (LLVM) é um compilador amplamente modular que favorece a implementação e a experimentação com diversas análises e transformações de programas. Ele é uma infraestrutura de compilação escrita em C++, criada na Universidade de Illinois em Urbana-Champaign por Chris Lattner, sob orientação de Vikram Adve, no início de 2000 (LATTNER; ADVE, 2002b).

A versão inicial do LLVM, lançada em 2003, foi focada em pesquisas acadêmicas em compiladores e otimização de código. A partir de 2005, o LLVM passou a ser usado na indústria, mais especificamente pela Apple, que contratou Lattner para incorporar o LLVM no desenvolvimento de vários produtos comercializados pela empresa, entre eles o XCode, uma IDE (*Integrated Development Environment*) para desenvolvimento de software, e o MAC OS X, sistema operacional dos sistemas computacionais comercializados pela Apple.

3.1.1 Arquitetura

O LLVM possui *front-ends* para diversas linguagens tais como C, C++, Haskell e Fortran. Programas escritos nessas linguagens são compilados para programas na linguagem intermediária de LLVM, chamada LLVM-IR. A infraestrutura de compilação LLVM possui também *back-ends* para diversos processadores, sendo capaz de gerar código para diversas arquiteturas distintas. Com LLVM pode-se, por exemplo, traduzir um programa C para LLVM-IR e após gerar código objeto para executar em um processador MIPS, X86, X86-64, além de outros.

A Figura 3.1 mostra as etapas do processo de compilação de programas no LLVM. As análises e transformações de programas são feitas em nível de representação intermediária, ou seja, sobre programas LLVM-IR.

Programas na linguagem intermediária LLVM-IR podem ser traduzidos para código objeto de uma arquitetura alvo através do gerador de código ou ainda, podem ser diretamente executados por um tradutor JIT, ou seja, um tradutor que converte, em tempo de execução do programa, as instruções de LLVM-IR para o código de máquina, rodando como se estivesse em uma máquina virtual, similar à linguagem de programação Java.

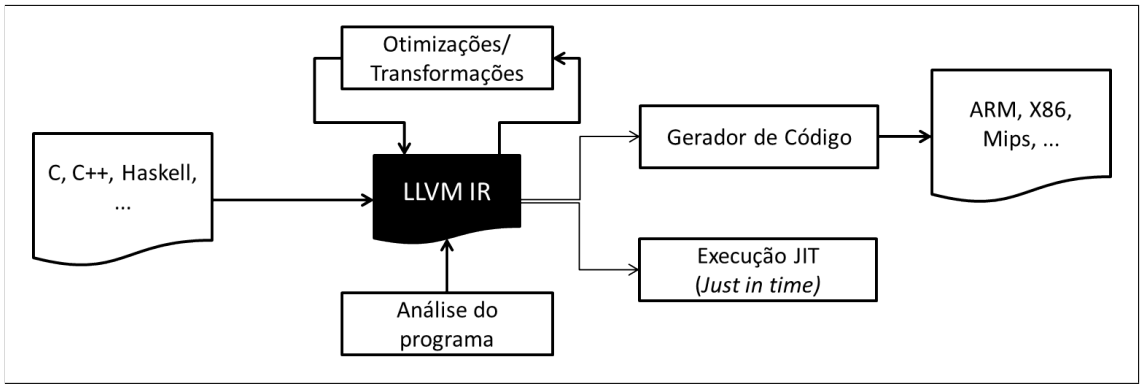

Figura 3.1: Etapas da compilação realizadas pelo LLVM.

A Figura 3.2 mostra a sequência de comandos utilizados para a compilação com o LLVM, partindo de um programa escrito na linguagem de programação C (no arquivo programa.c) até a geração do executável em linguagem de máquina do X86 (no arquivo programa.exe). Nessa compilação foi aplicada a transformação chamada de *adce*.

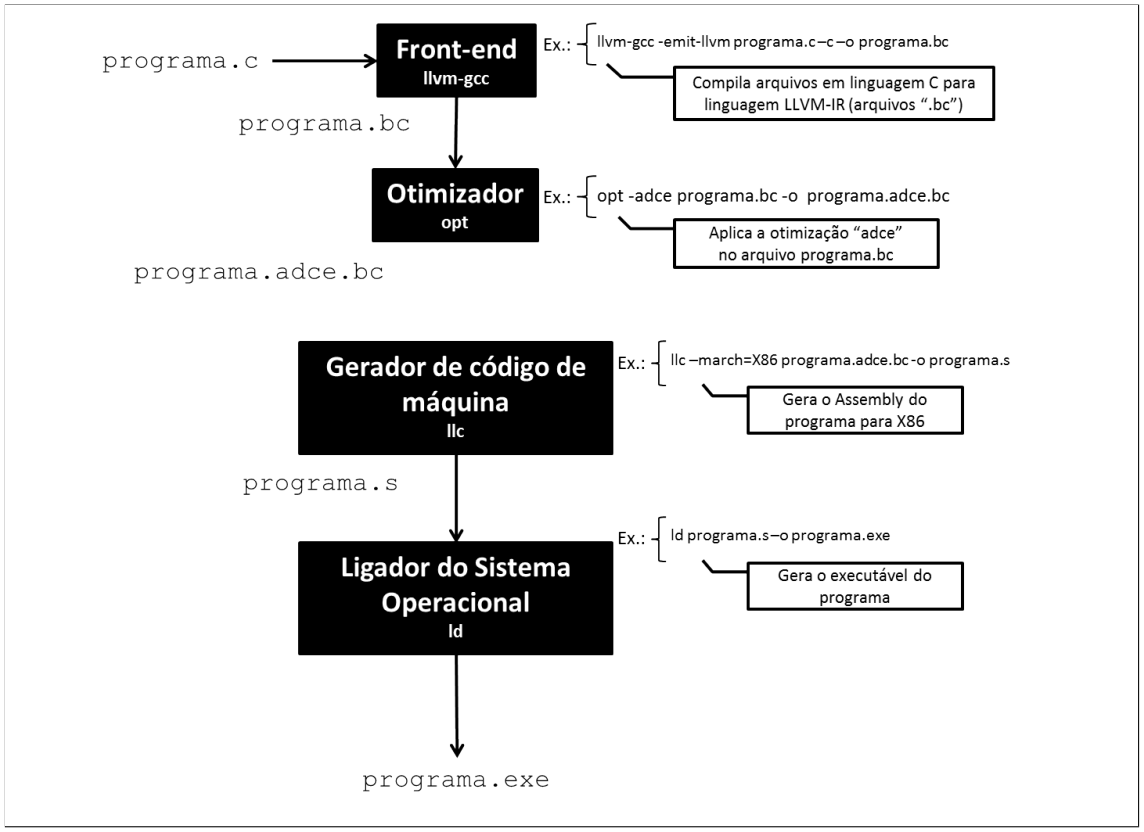

Figura 3.2: Comandos e processo de compilação do LLVM.

O LLVM provê dezenas de passos de transformação e análise que são feitas sobre a representação intermediária em LLVM-IR. Até dezembro de 2012 o site oficial de LLVM contava com sessenta (60) transformações e quarenta e quatro (44) análises de programas (LATTNER, 2013). Uma descrição de cada uma dessas transformações e análises pode ser encontrada em (REID; HENRIKSEN, 2012).

3.1.2 A Linguagem Intermediária LLVM-IR

Uma das características mais importantes do LLVM é a sua linguagem intermediária LLVM-IR, que é independente de linguagem de programação e também independente de arquitetura de máquina. LLVM-IR provê informação de alto nível sobre programas, como tipos, para suportar sofisticadas transformações e análises, e ao mesmo tempo é próxima ao *Assembly* contendo instruções no formato de 3 endereços, facilitando a tradução para código objeto.

A linguagem LLVM-IR é representada na forma *Single Static Assignement* (SSA) onde cada variável é atribuída apenas uma única vez. O formato SSA facilita a implementação de inúmeras otimizações no código de forma mais eficiente. Para alcançar a forma SSA, o LLVM possui um conjunto infinito de registradores, que são numerados sequencialmente em um programa.

A definição completa da sintaxe de LLVM-IR encontra-se em (REID; HENRIKSEN, 2012). Para facilitar a discussão e a apresentação da transformação ACCE, este trabalho considera um versão simplificada da sintaxe de LLVM-IR introduzida em (ZHAO et al., 2012) e disponível no Anexo I juntamente com alguns exemplos de programas. A implementação da técnica contudo considera a linguagem LLVM-IR completa.

Todo programa em LLVM-IR é estruturado como um módulo, formado por um *layout* (conjunto de informações que definem os tamanhos e alinhamentos de memória para os tipos), *named types* (representação de estruturas de dados) e uma lista de declarações ou definições de funções e variáveis globais.

Blocos básicos aparecem no corpo de definições de funções e sintaticamente são formados por um rótulo (*label*), instruções ϕ, comandos e instrução de terminação de bloco.

O rótulo é o identificador de bloco básico, composto por uma *string* de caracteres única em nível de função, ou seja, em uma mesma função dois blocos básicos não podem ter o mesmo rótulo.

As instruções denominadas *instruções* ϕ, são necessárias devido a forma SSA (*Single Static Assignment)* de representação de código. Elas são responsáveis pela correta atribuição de valores aos registradores de blocos básicos que contenham mais de um predecessor.

Instruções ϕ devem ser, obrigatoriamente, as primeiras instruções do bloco básico, ou seja, não pode haver instruções que não ϕ entre o *label* do bloco e a primeira instrução ϕ.

Para exemplificar o uso de instruções ϕ na linguagem de representação intermediária do LLVM, a Figura 3.3 apresenta um código fictício em LLVM-IR (a) e o seu respectivo CFG (b).

O trecho de código em LLVM-IR apresentado na Figura 3.3(a) define uma função com identificador *main*, composta de três blocos básicos cujos rótulos são: $entry$, bb_2 e bb_3. O uso da instrução ϕ ocorre na primeira instrução do bloco básico bb_2 pois ele possui mais de um bloco predecessor. Essa instrução ϕ é necessária aqui visto que o registrador %3 receberá o valor do registrador %2 caso o predecessor de bb_2 executado tenha sido o bloco básico $entry$, ou receberá o valor 0 caso o predecessor executado tenha sido o próprio bloco básico bb_2.

Os comandos que compõem um bloco básico são as instruções do programa e, por fim, a instrução de terminação de bloco serve para a passagem do controle do programa para outro bloco para a continuação da execução das instruções. Caso seja o último bloco, a instrução de terminação será a instrução de retorno da função, que encerra a execução.

No Capítulo 4 veremos que as instruções ϕ devem ser removidas antes da aplicação da transformação ACCE. Essa remoção será feita através de um passo de transformação já

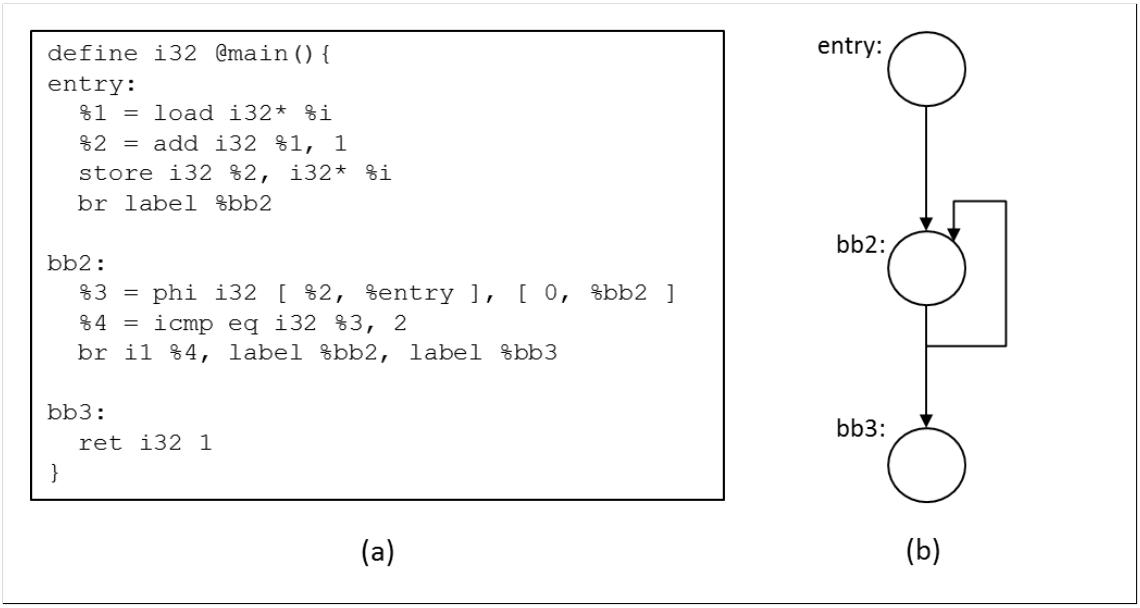

Figura 3.3: Exemplo de utilização de instruções ϕ

disponível em LLVM e consiste em usar a memória ao invés de registradores para guardar valores. Mais detalhes no Capítulo 4.

4 IMPLEMENTAÇÃO DE ACCE NO LLVM

Esse capítulo apresenta detalhes da técnica ACCE aplicada à linguagem LLVM-IR.

Os artigos originais que introduzem ACCE (VEMU; ABRAHAM, 2006), (VEMU; ABRAHAM, 2011) e (VEMU; GURUMURTHY; ABRAHAM, 2007), não explicitam detalhes das funcionalidades da técnica, tais como a maneira de transformar os blocos básicos para a inserção de instruções para detecção e correção de CFEs. As soluções adotadas para a transformação dos blocos básicos de programas LLVM-IR são, portanto, contribuições dessa dissertação.

A implementação de ACCE, como foi feita no *framework* de compilação LLVM, fica disponibilizada para diversas linguagens fonte e para diversas arquiteturas alvo para as quais LLVM disponibiliza *back-ends*. Além disso o uso de LLVM facilita a experimentação com a interação de ACCE com outras otimizações e/ou transformações tipicamente encontradas em compiladores.

O foco da técnica ACCE é a detecção de CFEs e também o reestabelecimento do fluxo de controle do programa para o ponto imediatamente anterior à ocorrência do desvio inesperado. Porém, a aplicação única e exclusivamente da técnica ACCE não garante a correção do fluxo de dados do programa, o qual pode ser corrigido com a aplicação, de forma combinada, de um outro mecanismo para reorganizar o fluxo de dados.

Nesse capítulo, a obtenção das assinaturas para os blocos básicos do programa é apresentada na Seção 4.1 com detalhes do cálculo da parte superior das assinaturas na Subseção 4.1.1 e da parte inferior das assinaturas na Subseção 4.1.2. Também nesse capítulo, na Seção 4.2, são dadas informações detalhadas sobre as alterações feitas pela técnica, como por exemplo, a adição de variáveis globais (Subseção 4.2.1) e as modificações nas funções do programa (Subseção 4.2.2).

Na Seção 4.3 é apresentada a transformação realizada por ACCE nos programas com exemplo de classes pertencentes à API do LLVM. Por fim, são apresentados o processo de detecção (Seção 4.4) e correção (4.5) de CFEs em tempo de execução.

4.1 Obtenção de Assinaturas para os Blocos Básicos

Esta seção descreve uma etapa de análise que precede a transformação de programas. Durante essa etapa são coletadas e armazenadas informações sobre o programa necessárias na etapa da transformação propriamente dita. As informações mais importantes a serem obtidas são as assinaturas dos blocos básicos do programa pois elas são fundamentais para a detecção de CFEs.

A técnica ACCE pertence ao conjunto de técnicas conhecidas como *Signature Checking*, ou verificação de assinaturas.

Assinaturas são valores determinados para cada bloco básico durante o processo de

compilação. Essa subseção tem como objetivo descrever como essas assinaturas são obtidas. O papel das assinaturas na detecção de CFEs será explicado mais adiante. Por hora é suficiente dizer que durante a execução, na "passagem" por um bloco, as suas assinaturas são comparadas com os valores legais esperados obtidos em tempo de execução. Quando há divergência entre esses valores tem-se um CFE.

Cada bloco básico possui duas assinaturas identificadas por *NS* e *NES* para *Node Signature* e *Node Exit Signature* respectivamente. E cada assinatura *NS* possui sua porção superior NS_u e sua porção inferior NS_l (para *upper* e *lower* respectivamente). O mesmo para as assinaturas *NES*.

Inicialmente o CFG é percorrido e, para cada bloco básico, é obtido o seu conjunto de blocos predecessores. Para o CFG da Figura 4.1, por exemplo, são obtidos os seguintes conjuntos de predecessores conforme a Tabela 4.1.

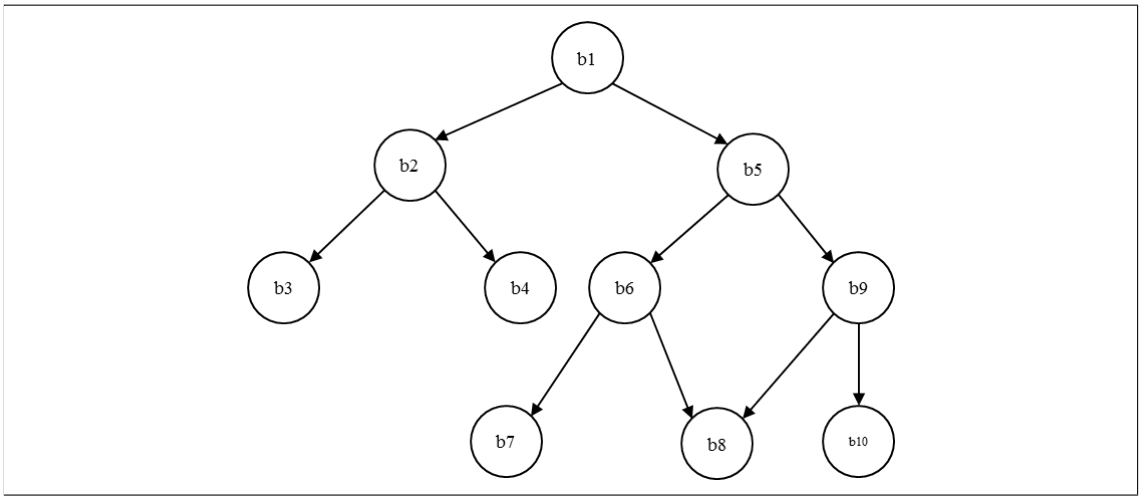

Figura 4.1: Exemplo de grafo de fluxo de controle.

Tabela 4.1: Blocos e seus predecessores obtidos a partir do CFG da Figura 4.1.

bloco	predecessores	NT(b_i)
b_1	$\{\ \}$	X
b_2	$\{b_1\}$	X
b_5	$\{b_1\}$	X
b_3	$\{b_2\}$	X
b_4	$\{b_2\}$	X
b_6	$\{b_5\}$	X
b_9	$\{b_5\}$	X
b_7	$\{b_6\}$	X
b_8	$\{b_6, b_9\}$	A
b_{10}	$\{b_9\}$	X

A partir do levantamento dos predecessores de cada bloco básico, parte-se para a classificação dos blocos em dois tipos (NT(b_i)), A ou X, de acordo com as seguintes regras:

- **Tipo A:** Um bloco é considerado do tipo A se e somente se possuir múltiplos predecessores e ao menos um desses predecessores possuir mais de um sucessor.

- **Tipo X:** Um bloco é do tipo X se e somente se não for do tipo A.

Dentre os blocos do CFG da Figura 4.1, o único que se configura como sendo um bloco do tipo A é o bloco b_8, uma vez que, possui mais de um predecessor (blocos b_6 e b_9) e ao menos um deles possui mais de um sucessor (bloco b_6, por exemplo).

Uma vez classificados os blocos básicos e obtidos os seus conjuntos de predecessores, a próxima etapa é organizar todos os blocos do CFG em conjuntos denominados de *Nets*. Cada *Net* é um conjunto não vazio que deve satisfazer a seguinte propriedade (VEMU; ABRAHAM, 2011):

$$b_i \in Net \rightarrow (\forall b_j.\ pred(b_i) \cap pred(b_j) \neq \emptyset \rightarrow b_j \in Net)$$

A Tabela 4.2 mostra todas as *Nets* obtidas a partir do CFG da Figura 4.1, juntamente com os seus respectivos conjuntos Net_Pred . Na primeira coluna da Tabela 4.2 vemos que, por exemplo, os blocos b2 e b5 fazem parte da mesma Net pois possuem predecessores em comum. Da mesma forma os bloco b8 deve fazer parte da mesma net que o bloco b7 pois possuem predecessor em comum. E o bloco b10 também fazer parte dessa mesma Net pois possui predecessor em comum com o bloco b8.

Cada conjunto Net possui um conjunto associado Net_pred que é união dos predecessores de cada bloco pertencente a Net:

$$Net\_Pred(Net) = \{pred(b_i) \mid b_i \in Net\}$$

O conjunto Net_Pred da Net{b1} é vazio pois b1 não possui predecessor, já o Net_Pred da Net{b7, b8, b10} é formado pelos blocos b6 (predecessor dos blocos b7 e b8) e b9 (predecessor dos blocos b8 e b10).

Nets e suas respectivas Net_Preds serão usadas para o cálculo das porções **superiores** das assinaturas NS e NES de blocos (assinaturas e assinaturas de saída dos blocos respectivamente).

Tabela 4.2: Nets e seus predecessores obtidos a partir do CFG da Figura 4.1.

Net	Net_Pred
$\{b_1\}$	$\{\ \}$
$\{b_2, b_5\}$	$\{b_1\}$
$\{b_3, b_4\}$	$\{b_2\}$
$\{b_6, b_9\}$	$\{b_5\}$
$\{b_7, b_8, b_{10}\}$	$\{b_6, b_9\}$

Um outro conjunto é definido, chamado de A_Sig. A_Sig contém informações sobre quais assinaturas de quais blocos devem ser iguais. No caso da Net{b7, b8, b10} da Tabela 4.3 por exemplo, a sua A_Sig associada está informando que os cálculos da NES de b6, da NES de b9 e da NS de b8 devem ser feitos em conjunto respeitando determinadas condições a serem vista nas subseções 4.1.1 e 4.1.2 a seguir. Este conjunto é composto por todas as assinaturas de saída (NES) dos blocos que pertencem a Net_Pred e NS dos blocos do tipo A pertencentes à Net, que ao serem criadas, receberão valores similares:

$$A\_sig(Net) = \{NES(b_i)\ :\ b_i\ \in\ Net\_Pred(Net)\} \bigcup \{\ NS(b_i)\ :b_i\ \in\ Net \wedge NT(b_i)\ =\ A\}$$

Os A_Sigs das linhas 1, 2, 3 e 4 da tabela 4.3 acima não tem interesse prático. Os A_sig obtidos só são relevantes se tiverem pelo menos dois elementos. O último conjunto

Tabela 4.3: Nets e respectivos A_Sig.

Net	A_Sig
$\{b_1\}$	$\{\ \}$
$\{b_2, b_5\}$	$\{NES(b_1)\}$
$\{b_3, b_4\}$	$\{NES(b_2)\}$
$\{b_6, b_9\}$	$\{NES(b_5)\}$
$\{b_7, b_8, b_{10}\}$	$\{NES(b_6), NES(b_9), NS(b_8)\}$

A_Sig da Tabela 4.3 indica que as assinaturas NES de b_6, NES de b_9 e NS do bloco b_8, mais especificamente as partes inferiores, deverão ser calculadas de forma semelhante (detalhes na seção 4.1.2).

4.1.1 Cálculo da parte superior das assinaturas

A partir dos conjuntos *Nets* e dos seus respectivos conjuntos *Net_pred* a construção das partes **superiores** das assinaturas NS_u e NES_u de cada bloco é simples e deve seguir as seguintes restrições:

- Para cada Net, a metade superior de todas as assinaturas presentes em A_Sig deve receber o mesmo valor. Esse valor deve ser diferente de qualquer outro valor usado em qualquer outra parte, superior ou inferior, de outras assinaturas.
- os NES_u e NS_u dos demais blocos podem assumir quaisquer valores desde que sejam únicos.

A Tabela 4.4 mostra os valores de NS_u e NES_u obtida a partir dos conjuntos *Net* e *Net_Pred* da Tabela 4.2. As letras maiúsculas representam valores obtidos que devem ser iguais seguindo a primeira restrição da regra acima e as letras minúsculas representam os demais valores.

Tabela 4.4: NS_u e NES_u de blocos.

bloco	NS_u	NS_l	NES_u	NES_l
b_1	a_2		j_2	
b_2	b_2		k_2	
b_3	c_2		l_2	
b_4	d_2		n_2	
b_5	e_2		o_2	
b_6	f_2		**M**	
b_7	g_2		p_2	
b_8	**M**		q_2	
b_9	h_2		**M**	
b_{10}	i_2		r_2	

4.1.2 Cálculo da parte inferior das assinaturas

O cálculo das partes **inferiores** de NS e NES dos blocos de cada Net, NS_l e NES_l, respectivamente, deve respeitar as seguintes condições (onde Zl(S) representa um conjunto cujos elementos são as posições com bit 0 da parte inferior da assinatura S. Por

exemplo se S=11010010 Zl(S) = {0,2,3} indicando que o bit zero aparece nas posições 0, 2 e 3 da parte inferior de S):

1. Duas assinaturas S_1 e $S_2 \in$ ***A_Sig(Net)*** devem ser diferentes ($Zl(S_1) \neq Zl(S_2)$), exceto para o caso onde ambas assinaturas são NES de nós que possuem um sucessor em comum do tipo X, pois este bloco X requer que todos os seus predecessores possuam assinaturas de saída (NES) idênticas, logo $Zl(S_1) = Zl(S_2)$. Essa regra garante que as assinaturas pertencentes ao conjunto A_Sig, que possuem metade superior idênticas, sejam distintas, tornando todas as assinaturas únicas.

2. A metade inferior de NES de um bloco b_j (NES_l) deve estar contida em NS de um bloco b_i, representada por $Zl(NES(b_j)) \subset Zl(NS(b_i))$, caso $b_i \in$ Net e seja do tipo A, além de b_j ser predecessor de b_i.

3. A metade inferior de NES de um bloco N_2 não deve estar contida na metade inferior de NS de um bloco b_i tal que b_i seja A, $b_j \in$ Net e não seja predecessor de b_i, logo ($Zl(NES(b_j) \subset Zl(NS(b_i))$). Essa regra garante que em caso de um salto ilegal de um nó não predecessor para um nó A resultará em assinaturas distintas em A.

As demais metades inferiores das assinaturas que não receberam nenhum valor, com base nas regras anteriores, devem receber um valor único.

Portanto, seguindo o exemplo apresentado anteriormente, a partir da Tabela 4.4, a Tabela 4.5 mostra as partes inferiores das assinaturas de cada bloco básico, tanto para NS quanto para NES. Vale ressaltar que, de todas as assinaturas calculadas, as únicas que se enquadraram em alguma das três regras definidas para a parte inferior foram as assinaturas $NS_l(b_8)$, $NES_l(b_6)$ e $NES_l(b_9)$ que, devido ao fato de que $NES_l(b_6)$ e $NES_l(b_9)$ são predecessores de $NS_l(b_8)$ que é do tipo A e, de acordo com a regra 2, estas assinaturas precisam ser contidas em $NS_l(b_8)$.

Novamente, de forma similar à atribuição das assinaturas superiores, as letras maiúsculas representam as assinaturas atribuídas de acordo com alguma regra, enquanto que as letras minúsculas representam as assinaturas distintas atribuídas. **CM1** e **CM2** representam que estas assinaturas devem estar contidas na assinatura M, conforme a regra 2. Portanto, unindo as duas Tabelas das assinaturas superiores e inferiores, obtém-se a

Tabela 4.5: NS_l e NES_l de blocos.

bloco	tipo	NS_u	NS_l	NES_u	NES_l
b_1	X		*a*		*b*
b_2	X		*c*		*d*
b_3	X		*e*		*f*
b_4	X		*g*		*h*
b_5	X		*i*		*j*
b_6	X		*k*		**CM1**
b_7	X		*l*		*m*
b_8	A		**M1**		*n*
b_9	X		*o*		**CM2**
b_{10}	X		*p*		*q*

Tabela 4.6 com as assinaturas para os blocos básicos.

Tabela 4.6: *NS* e *NES* de blocos.

bloco	tipo	NS_u	NS_l	NES_u	NES_l
b_1	X	a_2	*a*	j_2	*b*
b_2	X	b_2	*c*	k_2	*d*
b_3	X	c_2	*e*	l_2	*f*
b_4	X	d_2	*g*	n_2	*h*
b_5	X	e_2	*i*	o_2	*j*
b_6	X	f_2	*k*	**M**	**CM1**
b_7	X	g_2	*l*	p_2	*m*
b_8	A	**M**	**M1**	q_2	*n*
b_9	X	h_2	*o*	**M**	**CM2**
b_{10}	X	i_2	*p*	r_2	*q*

A Tabela 4.7 exemplifica o uso de bits para a representação de assinaturas com base no CFG da Figura 4.1 e nas informações extraídas e representadas na Tabela 4.6. De forma a deixar mais legível, foram usados apenas 16 bits em cada assinatura, porém podem ser usados até 64 bits.

Tabela 4.7: NS_u e NES_u de blocos em bits.

bloco	tipo	NS_u	NS_l	NES_u	NES_l
b_1	X	10000001	11000000	10001010	11000001
b_2	X	10000010	11000010	10001011	11000100
b_3	X	10000011	11000101	10001100	11000110
b_4	X	10000100	11000111	10001101	11001000
b_5	X	10000101	11001001	10001110	11010001
b_6	X	10000110	11001010	10000000	01100000
b_7	X	10000111	11001011	10001111	11001100
b_8	A	10000000	01111111	10010000	11001101
b_9	X	10001000	11001110	10000000	01000000
b_{10}	X	10001001	11001111	10010001	11010000

Finalmente, com a obtenção de NS e NES, são calculados dois parâmetros para cada nodo do CFG: d_1 e d_2, responsáveis pela atualização da assinatura em tempo de execução, deixando-a correspondente ao seu valor esperado em qualquer ponto do programa. Para os blocos do tipo X, d_1 é o resultado de uma operação de XOR entre NS do nó e NES de um nodo predecessor. Para os blocos do tipo A, todos os bits da metade superior devem ser o mesmo valor e a metade inferior deve ser igual à metade inferior de NS do próprio nodo. A atribuição de d_2 é dada por uma operação de XOR entre as duas assinaturas do nó, NS e NES.

A Tabela 4.8 mostra os valores de d_1 e d_2 obtidos com base nos valores de NS e NES de cada bloco, já apresentados na Tabela 4.7. O valor de d_1 do bloco b_1 é omitido por não ter predecessor, não sendo possível computar o seu valor.

Quanto à escalabilidade da técnica, ou seja, em relação ao número de blocos que podem ser comportados no sentido de geração de assinaturas diferentes tem-se que, considerando que todas as assinaturas a serem usadas no programa precisam ser distintas

Tabela 4.8: d_1 e d_2 dos blocos em bits.

bloco	tipo	d_1	d_2
b_1	X	omitido	0000101100000001
b_2	X	0000100000000010	0000100100000110
b_3	X	0000100000000001	0000111100000011
b_4	X	0000111100000011	0000100100001111
b_5	X	0000111100001000	0000101100011000
b_6	X	0000100000011011	0000011010101010
b_7	X	0000011110101011	0000100000000111
b_8	A	1111111101111111	0001000010110010
b_9	X	0000011000011111	0000100010001110
b_10	X	0000100110001111	0001100000011111

entre si, pode-se representar 2^{32} blocos básicos, usando assinaturas de 64 bits (dado que para cada bloco é definida uma assinatura para NS e NES).

4.2 Transformação de programas

A transformação propriamente dita, apresentada a seguir, faz uso de informações sobre o programa coletadas na etapa de análise, tais como assinaturas de blocos e identificação das funções do programa.

Um programa LLVM-IR é constituído de informações de *layout* dos tipos, seguido de declarações de nomes para estruturas (*namedt*), seguidos de declarações de variáveis globais, declarações e definições de funções:

$$P \quad ::= \overline{layout}\ \overline{namedt}\ \overline{glbvars}\ \overline{fdcls}\ \overline{fdfs}$$

A transformação ACCE é definida a seguir através de uma função de transformação TR que recebe programas LLVM-IR (e outras informações coletadas na etapa de análise) e devolve programas LLVM-IR.

TR mantém inalteradas as partes de layout de tipos, definição de estruturas e declaração de funções e transforma somente as partes de programas contendo **definição de variáveis globais** e **definição de funções**:

$$\mathsf{TR}(\overline{layout}\ \overline{namedt}\ \overline{glbvars}\ \overline{fdcls}\ \overline{fdfs}) = \overline{layout}\ \overline{namedt}\ \mathsf{TR}(\overline{glbvars})\ \overline{fdcls}\mathsf{TR}(\overline{fdfs})$$

A transformação $\mathsf{TR}(\overline{glbvrs})$ sobre variáveis globais é trivial e é apresentada na Seção 4.2.1. A Seção 4.2.2 detalha as transformações $\mathsf{TR}(\overline{fdfs})$ feitas sobre a parte contendo as funções de um programa (que consiste basicamente na modificação dos blocos básicos das funções já existentes e na adição de uma nova função).

4.2.1 Adição de novas variáveis globais

A transformação de variáveis globais é trivial: todas as variáveis globais do programa original são mantidas inalteradas (linha 2) e são acrescentadas sete (7) novas variáveis globais (linhas 3 a 9), conforme a Figura 4.2.

Os novos identificadores globais inseridos registram informações fundamentais para o funcionamento de ACCE, a saber:

1. $\mathsf{TR}(\overline{glbvars}) =$
2. $\quad \overline{glbvars}$
3. $\quad$ **@error_flag** = **global** *i1 false*
4. $\quad$ **@num_error_GEH** = **global** *i32 0*
5. $\quad$ **@num_error_FEH** = **global** *i32 0*
6. $\quad$ **@F** = **global** *i32 0*
7. $\quad$ **@SAtual** = **global** *i32 0*
8. $\quad$ **@SGEH** = **global** *i32 0*
9. $\quad$ **@.strErroNaoTratado** = **internal constant** *[19 x i8] c"Erro nao Tratado"*

Figura 4.2: Adição de novas variáveis globais por ACCE.

- **@error_flag:** variável booleana inicializada com *false* e que recebe *true* sempre que um CFE estiver sendo tratado na função GEH, descrita na subseção 4.2.2, e que é testada no bloco de entrada de cada função (subseção 4.2.2.1).

- **@num_error_GEH:** esta variável é usada para a contagem do número de iterações de GEH para a busca da função de origem do CFE;

- **@num_error_FEH:** a variável armazena o número de tentativas de transferência do controle para o bloco de origem do CFE.

- **@F:** armazena o identificador da função sendo executada. Essa variável global é atualizada sempre que o bloco de entrada de uma função é executado e também logo após o retorno da execução de uma instrução `call` de chamada de função, quando ela recebe a identificação da função que executou a chamada

- **@SAtual:** armazena o valor da assinatura corrente, ou seja, assinatura do bloco, das funções originais do programa que está sendo executado;

- **@SGEH:** armazena o valor da assinatura S do bloco que estava sendo executado imediatamente antes do início da execução de GEH. GEH é a sigla de *Global Error Handler* e é uma função, adicionada aos programas, que permite a localização da função de origem de um CFE caso este tenha como origem e destino instruções localizadas em blocos de diferentes funções.

- **@.strErroNaoTratado:** contem o string emitido quando o número de tentativas de correção de um CFE ultrapassa o limite máximo de tentativas. Isso é feito na função GEH e nos novos blocos FEH da função.

4.2.2 Transformação de Funções

A parte $\overline{fdefs}$ consiste de uma sequencia $fdef_1 \ldots fdef_n$ com as funções de um programa. A transformação TR modifica os blocos de cada uma das funções já existentes e introduz uma nova função global no programa chamada GEH (para *Global Error Handling*) responsável por lidar com CFEs entre blocos de funções diferentes. Na definição abaixo, a criação dessa nova função global GEH é feita por uma função de transformação auxiliar NewFunGEH:

$$\mathsf{TR}(fdef_1 \ldots fdef_n) \;=\; \mathsf{TR}(fdef_1) \ldots \mathsf{TR}(fdef_n)\;\; \mathsf{NewFunGEH}(\,)$$

A transformação $\mathsf{TR}(fdef)$ de uma função consiste basicamente (i) da transformação de cada um dos seus blocos básicos $b_{entry}, b_2, \ldots b_n$, onde b_{entry} é o bloco onde inicia a execução da função quando ela é chamada, e (ii) da adição de novos blocos ao final da função feita pela função auxiliar da transformação *NewBlksFEH*, conforme Figura 4.3.

1.	$\mathsf{TR}($**define** *typ fname*$(\overline{arg})$	$=$	**define** *typ fname*$(\overline{arg})$
2.	$\{b_{entry}$		$\{\mathsf{TR}(b_{entry})$
3.	b_2		$\mathsf{TR}(b_2)$
4	$\ldots$		$\ldots$
5.	$b_n\ \}$		$\mathsf{TR}(b_n)$
6)			$\mathsf{NewBlcksFEH}()\ \}$

Figura 4.3: Representação da transformação realizada por ACCE nas funções dos programas.

O bloco básico de entrada b_{entry} é transformado de forma diferente dos demais blocos básicos (detalhes nas subseções 4.2.2.1 e 4.2.2.2): para cada bloco regular são produzidos 2 novos bloco protetores adicionais, e para cada bloco de entrada de função são introduzidos 3 novos blocos. Os novos blocos básicos ao final, chamados de blocos FEH, são os responsáveis pela recuperação de CFEs entre blocos da mesma função.

A transformação deve garantir também que os rótulos de entrada de todos os novos blocos adicionados a cada função devem ser diferentes entre si e diferentes do rótulos dos blocos originais.

4.2.2.1 Transformação de bloco de entrada de função.

A transformação do bloco de entrada de uma função *fname* é definida na Figura 4.4. Por ser o primeiro a ser executado quando uma função é chamada, o bloco de entrada deve conter as instruções `alloca` de alocação de memória na pilha de execução da função e essas instruções sempre devem ser as primeiras do bloco, uma vez que, ao corrigir erros de fluxo de controle entre funções diferentes, o controle é inicialmente transferido para FEH e retorna para o bloco de origem do desvio inesperado dentro da função, que pode ter seus registradores não alocados.

A estrutura sintática dos comandos de um bloco de entrada pode portanto ser refinada para

$$\overline{c_{alloca}}\ \overline{c}$$

onde $\overline{c_{alloca}}$ são as instruções `alloca` do bloco e $\overline{c}$ são as demais instruções que as seguem.

A transformação de um bloco básico de entrada $l : \overline{c_{alloca}}\ \overline{c}\ tmn$ é definida como mostra a Figura 4.4. Na Figura, os elementos do bloco original estão em destaque dentro de "caixas retangulares".

Segue uma explicação dos blocos resultantes da transformação de um bloco de entrada de função:

- l_entry: bloco básico inserido por ACCE que será o primeiro a ser executado quando da chamada da função. Este bloco possui as seguintes instruções:
  - Primeiramente o $@F$ é inicializado com o identificador da função que está sendo executada (linha 3) e alocado o registrador %S (linha 4).

```
1.TR(l : c̄_alloca c̄ tmn) =
2.    l_entry :
3.      store FID(fname), @F
4.      %S = alloca i32
5.      [c̄_alloca]
6.      %tmp_error_flag = load @error_flag
7.      br %tmp_error_flag, %FEH_entry, [l]
8.
9.    [l] :
10.     store NS(fname, l), @SAtual
11.     store NS(fname, l), %S
12      %S_l = load %S
13.     %comp_1 = icmp ne %S_l, NS(fname, l)
14.     br %comp_1, %FEH_entry, lc
15.
16.   lc :
17.     %S_lc_1 = load %S
18.     %SxorD1_lc = xor %S_lc_1, D1(fname, l)
19.     store %SxorD1_lc, %S
20.     TR([c̄])
21.     %S_lc_2 = load %S
22.     %comp_2 = icmp ne %S_lc_2, NS(fname, l)
23.     br %comp_2, FEH_entry, ltmn
24.
25.   ltmn :
26.     %S_ltm = load %S
27.     %SxorD2_ltmn = xor %S_ltmn, D2(fname, l)
28.     store %SxorD2_ltmn, %S
29.     [tmn]
```

Figura 4.4: Transformação de bloco de entrada de função.

  - Todas as instruções de $alloca$ são movidas do bloco de entrada original, representadas por $\overline{c_{alloca}}$ (linha 5), para garantir que não hajam arestas no CFG que levem à execução de instrução que acesse um registrador sem a sua prévia alocação, uma vez que,
  - as duas últimas instruções do bloco (linhas 6 e 7) são usadas para verificar e transferir o controle para FEH, respectivamente, caso um CFE precise ser tratado, de acordo com o valor da variável **@error_flag**.

- l: bloco básico de proteção superior, composto por instruções que permitem a verificação da assinatura na entrada do bloco. Este bloco é composto pelas instruções descritas abaixo:

  - Nas instruções das linhas 10 e 11, o valor de S é armazenado nas variáveis **@SAtual** e %S para serem usadas posteriormente na comparação de assinaturas.
  - A instrução na linha 13 compara o valor de $\%S\_l$, obtido com a instrução de $load$ (linha 12) com o valor de NS do bloco, representado por $\mathsf{NS}(\mathit{fname}, l)$.

Nesse bloco em especial, o valor de S_l é comparado com o valor de $\mathsf{NS}(\textit{fname}, l)$ sendo que S_l é um $load$ de %S e não com $\mathsf{D1}(\textit{fname}, l)$, como nos demais blocos conforme mostrado nas subseções seguintes, pelo fato de que $\mathsf{D1}(\textit{fname}, l)$ é obtido de acordo com o predecessor do bloco, caso que este não possui. Porém, mesmo comparando os mesmos valores, pode ocorrer casos em que resulte em CFE, bastando apenas ocorrer um desvio inesperado com destino na instrução de comparação, o que resultará em um $\mathsf{NS}(\textit{fname}, l)$ diferente do %S.

- Na linha 14 a instrução de desvio é usada para transferir o controle para FEH caso haja divergência na comparação das assinaturas ou, caso contrário, transferir o controle para o próximo bloco na execução, o bloco lc.

- lc: bloco básico que contém as instruções originais do programa transformadas, e além delas, instruções inseridas por ACCE no início do bloco para a atualização do valor de %S e no final do bloco para a verificação de NS no interior do bloco. As instruções que compõem o bloco são:

  - Inicialmente, o valor de S é atualizado para conter o valor correspondente a NS dentro do bloco (linhas 17, 18 e 19).
  - A linha 20, $\mathsf{TR}(\boxed{\overline{c}})$, representa as instruções originais do programa.
  - Finalmente, nas linhas 21, 22 e 23 são inseridas instruções para a comparação de %S no final do bloco básico, para avaliar se um CFE ocorreu com destino o interior do bloco.

- ltm: bloco de proteção inferior composto por instruções que permitem a atualização do valor de %S por meio de uma operação de xor para que contenha o valor esperado para a saída do bloco. Este bloco é composto por:

  - O valor de %S é carregado da memória(linha 26) e atualizado com o parâmetro D2 através de uma instrução de XOR. Este valor corresponde ao NES do bloco calculado durante a obtenção das assinaturas.
  - Após a obtenção do valor, este é armazenado no registrado $\%S$ (linha 28) para então prosseguir com a execução do programa seguindo o fluxo de controle original, através da instrução de término do bloco original, movida para o final do bloco de proteção inferior, representada por $\boxed{tmn}$ (linha 29).

4.2.2.2 Transformação de blocos regulares de funções.

A transformação TR, dado um bloco básico com estrutura sintática $l\ \overline{c}\ tmn$ produz 3 blocos. Na definição da transformação dada pela Figura 4.5 assume-se que ela opera sobre um bloco de uma função de nome *fname* e sobre um bloco do tipo X.

A transformação para blocos do tipo A dá-se da mesma maneira exceto que a operação `xor`, que aparece nas linhas 4 e 11 da Figura, é substituída pela operação `and`. Na transformação da Figura 4.5, note que o label de entrada do primeiro bloco é o label l de entrada do bloco original, e os 2 blocos seguintes recebem labels de entrada novos lc e *ltmn* respectivamente. Assim, tem-se os seguintes blocos:

- l : chamado de *bloco de proteção superior*, recebe instruções protetoras, encarregadas de proteger a entrada do bloco básico original de CFEs que tenham como

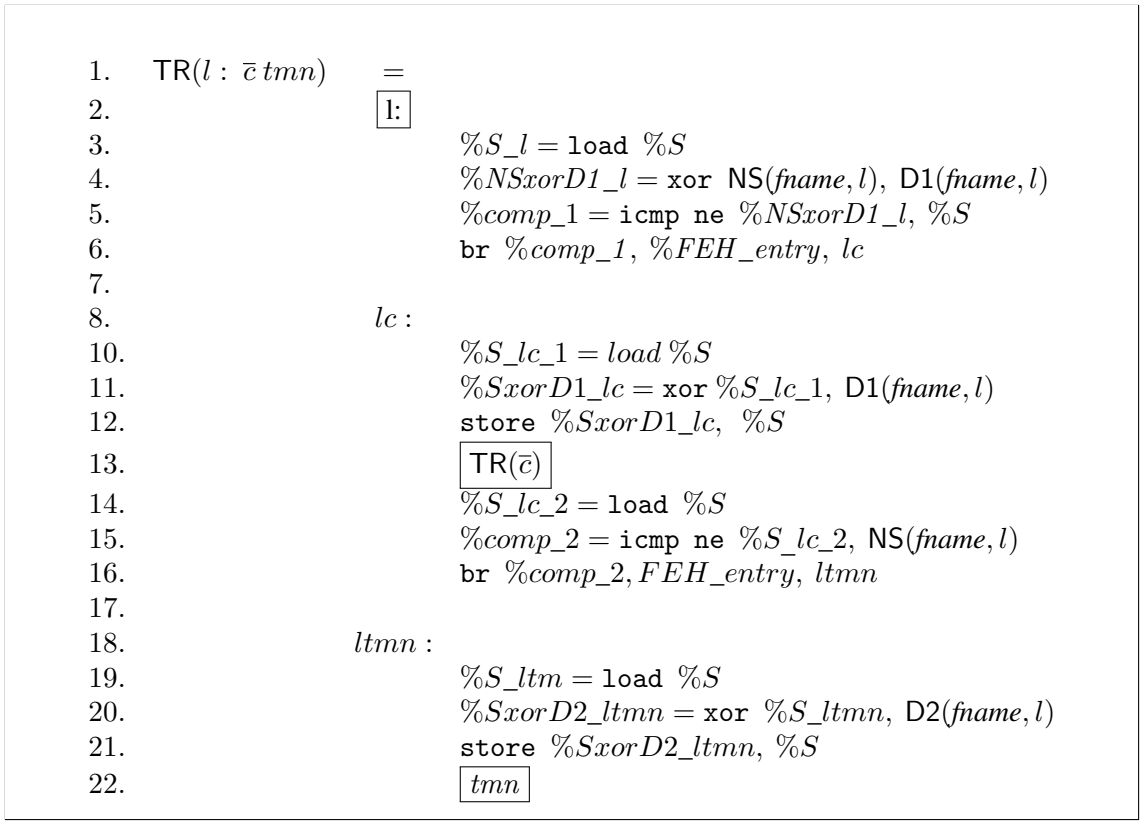

Figura 4.5: Representação da transformação realizada em blocos básicos regulares.

destino as instruções deste bloco. Algumas instruções que compõem o bloco l são descritas abaixo:

- No início do bloco são obtidos os valores de %S, via instrução $load$, e de NS `xor` D1(*fname*, l) , linhas 3 e 4, respectivamente, para uma posterior comparação.
- Na linha 5 é feita uma comparação entre os valores de %S e $\%NSxorD1\_l$. Em caso de divergência, o controle toma como destino os blocos básicos que representam FEH e, caso contrário, passa para a execução do próximo bloco básico, nesse caso o bloco lc (via instrução de desvio da linha 6).

- lc : contém as instruções $\bar{c}$ do bloco original transformadas e algumas instruções inseridas por ACCE. Estas instruções são inseridas antes e depois das instruções originais do programa para atualizar o valor de %S e detectar CFEs, respectivamente, conforme descrito abaixo:
  - Anteriormente às instruções originais do programa, são inseridas instruções para a atualização de %S após a verificação inicial feita no bloco anterior. Nesse sentido, o valor de $\%SxorD1\_lc$ (obtido com a operação de xor da linha 11) é armazenado em %S (linha 12), atualizando o seu valor.
  - No final deste bloco são inseridas as instruções para a comparação da assinatura no interior do bloco. Logo, as instruções das linhas 14,15 e 16 possuem funções similares às das linhas 4, 5 e 6, já descritas.

- $ltmn$: recebe instruções de proteção contra CFEs que tenham como destino instruções de dentro do bloco básico.

– De forma similar às instruções das linhas 10, 11 e 12, as instruções inseridas no bloco de label ltm servem para a atualização do valor de %S para prosseguir com o fluxo de execução, dado pela instrução $\boxed{tmn}$, movida do bloco original para este ponto, para preservar o fluxo de controle original do programa.

Vale salientar que as instruções das linhas 16 a 29 na Figura 4.4 das linhas 8 até 22 na Figura 4.5 são idênticas visto que representam a parte inferior dos blocos básicos, que são tratadas de forma similar para o bloco de entrada da função e para os demais blocos, sendo diferentes apenas a parte inicial destes blocos. Além disso, $\mathsf{TR}(\overline{c})$ retorna todos os comandos originas com a exceção de comandos do tipo *call* que são transformados para *call* seguido da atribuição para @F.

4.2.2.3 Novos blocos para tratar de CFEs locais a funções.

A inserção de blocos básicos extras em cada função (chamados de blocos FEH) para tratamento de erros de fluxo de controle é feita conforme a Figura 4.6. Os blocos de FEH são explicados abaixo:

- FEH_entry: Nos blocos básicos do programa, quando da verificação das assinaturas, há instruções que desviam para este bloco em caso de detecção de CFEs. Neste bloco são inseridas as instruções que verificam se o identificador **@error_flag** é verdadeiro, para saber se há necessidade de tratamento de desvio inesperado. Caso verdadeiro, ocorre um desvio para o bloco FEH_atual_S e, caso falso, o desvio se dá para o bloco FEH_vals, ambos descritos na sequência

- FEH_atual_S: bloco básico usado para a atualização do valor do identificador %S, necessário para a localização do ponto de ocorrência do desvio inesperado.

- FEH_vals : usado para obter e comparar os valores de %FID e @F, que representam, respectivamente, o valor da função em execução e o valor da função de origem do CFE. Se forem diferentes, o controle transferido para o bloco FEH_Call_GEH, que então transfere o controle para a função GEH, para que a mesma possa encontrar a função de origem do CFE.

- $FEH\_check\_n\_error$: Similar ao que ocorre em GEH, este bloco serve para garantir um número máximo de tentativas de correção de um CFE para que a execução do programa não entre em *loop* infinito. Por padrão são permitidas 3 tentativas e, caso este número seja alcançado, o controle é transferido para o bloco FEH_exit, que causa o término na execução do programa. Caso contrário o controle é desviado para o bloco FEH_busca_no.

- FEH_busca_no : bloco básico onde há a leitura do valor contido no identificador %S para a localização do bloco básico do programa em que houve a ocorrência de um CFE. Para tanto, uma instrução de *switch* usada para comparar o valor de %S com a assinatura de cada bloco da função (obtido pela função auxiliar NS(l, fname) que retorna a assinatura do bloco com rótulo de entrada l de função de nome *fname*).

- FEH_exit : Bloco usado para o término na execução do programa em casos onde o número de tentativas ultrapassou o limite estipulado. Nesse bloco há uma chamada para a função @*exit*, além da instrução *unreachable* que indica que é um ponto de código não alcançável.

```
1.  NewBlcksFEH()  =
2.                  FEH_entry :
3.                   %error_flag_2 = load @error_flag
4.                   br %error_flag_2, %FEH_atual_S, %FEH_vals

5.                  FEH_atual_S :
6.                   %SdeGEH = load @SGEH
7.                   store %SdeGEH, %S
8.                   br  %FEH_vals

9.                  FEH_vals :
10.                  %FIDFEH = FID(fname)
11.                  %FFEH = load @F
12.                  %compFID = icmp eq %FIDFEH, %FFEH
13.                  %br i1 %compFID, %FEH_check_n_error,  %FEH_Call_GEH

14.                 FEH_Call_GEH :
15.                  call void @GEH()
16.                  unreachable

17.                 FEH_check_n_error :
18.                  store false @error_flag
19.                  %num_error_FEH = load @num_error_FEH
20.                  %num_errorFEH_ADD = add 1, %num_error_FEH
21.                  store %num_errorFEH_ADD, @num_error_FEH
22.                  %comp_num_error_thresh = icmp sgt %num_errorFEH_ADD, 3
23.                  br %comp_num_error_thresh, %FEH_exit, %FEH_busca_no

24.                 FEH_busca_no :
25.                  %SFEH_do_teste = load %S
26.                  switch %SFEH_do_teste, %FEH_entry [
27.                  i32 NS(l_1, fname) %chamaNSB_1
28.                       ...
29.                  i32 NS(l_n, fname) %chamaNSB_n
30.                  i32 NES(l_1, fname) %chamaNESB_1
31.                       ...
32.                  i32 NES(l_n, fname) %chamaNESB_n
33.                  ]

34.                 FEH_exit :
35.                  call @printf(@.strErroNaoTratado)
36.                  call void @exit(0)
37.                  unreachable

38.                 chamaNSB_1 :
39.                  store NS(b_1, fname), %S
40.                  br %b_1
41.                       ...

42.                 chamaNSB_n :
43.                  store NS(b_n, fname), %S
44.                  br %b_n

45.                 chamaNESB_1 :
46.                  store NSE(b_1, fname), %S
47.                  br %ltmn_b_1
48.                       ...

49.                 chamaNESB_n :
50.                  store NES(b_n, fname), %S
51.                  br %ltmn_b_n
52.                 }
```

Figura 4.6: Blocos FEH.

- $chamaNSB_1, ..., chamaNSB_n$: Blocos criados para a atualização do valor de S com NS de acordo com o bloco a ser chamado caso a verificação de CFE tenha sido feita no bloco de proteção superior. Essa atualização é necessária visto que o valor de %S pode não ter sido inicializado no momento da ocorrência do CFE.

- $chamaNESB_1, ..., chamaNESB_n$: Blocos criados para a atualização do valor de S com NES de acordo com o bloco a ser chamado caso a verificação de CFE tenha sido feita no bloco de proteção inferior. Essa atualização é necessária visto que o valor de %S pode não ter sido inicializado no momento da ocorrência do CFE.

4.2.2.4 Função GEH *para Tratar de CFEs entre funções*

A detecção de CFEs é feita por código adicional incluído em cada função do programa. Todo erro detectado por esse código adicional, num primeiro momento, é tratado pelos blocos adicionais de código referidos por FEH (para *Function Error Handler*). Se o código FEH identifica que os CFEs na verdade tem origem e destino em funções diferentes do programa, a função GEH (para *Global Error Handler*) é chamada para tentar fazer com que o fluxo de controle seja corrigido. A correção nesse caso se dá pelo reinício da execução da função onde o CFE se originou.

Por questões de clareza, a função GEH definida na Figura 4.7 como uma função em um LLVM-IR é simplificada. A ênfase em deixar claros os pontos importantes da correção de CFEs com origem e destino em funções diferentes.

A função GEH composta pelos seguintes blocos básicos:

- *entry:* bloco básico de entrada da função, em que a variável global **@error_flag** é marcada como `true`, indicando que um CFE precisa ser tratado. Ainda neste bloco, é obtido o valor da variável global **@F** que guarda a identificação da função que estava sendo executada no momento da detecção do CFE. Essa função ser chamada novamente como forma de correção do fluxo de controle.

- *find_func*: bloco básico composto de uma instrução switch que permite a busca da função de origem do CFE, para então transferir o controle de volta à função para que o desvio inesperado seja tratado. Note que em tempo de compilação a transformação tem a sua disposição o nome de todas as funções do programa ($fname_1, \ldots fname_n$) e, dado o nome de uma função é possível obter também um valor único associado a ela dado por FID(*fname*). Caso não encontre o controle é desviado para o bloco com rótulo GEH_check_tresh.

- *GEH_chck_tresh*: bloco básico que verifica o número de tentativas de correção do CFE. Em ACCE, há a definição de um número limite de tentativas de correção de CFEs para que o programa não entre em loop infinito em casos em que um CFE não permitiu a inicialização de @F, não encontrando nenhuma função correspondente no programa.

- *GEH_exit*: bloco básico com instruções de término da execução do programa, chamado caso o número limite de tentativas de correção tenha sido atingido (regulado pela variável *thresh*).

- *f1, . . ., fn*: blocos básicos criados com instruções de chamada das funções para o retorno do controle para a função de origem do CFE. Em LLVM-IR não é possível a ocorrência de instrução `call` para a chamada de função dentro de uma instrução

```
NewFunGEH( ) =  define void @GEH () {
 1.               entry :
 2.                 store  true @error_flag
 3.                 %tmp_F = load  @F
 4.                 br %find_fun
 5.
 6.               find_fun :
 7.                 switch %tmp_F, %GEH_chck_thresh [
 8.                  FID(fname), %f1
 9.                   ...
10.                  FID(fname), %fn
11.                 ]
12.
13.               GEH_chck_thresh :
14.                 %n_error_GEH = load @num_error_GEH
15.                 %count_n_error = add 1, %n_error_GEH
16.                 store %count_n_error, @num_error_GEH
17.                 %thresh = icmp sgt %count_n_error, 3
18.                 br %thresh, %GEH_exit, %find_fun
19.
20.               %GEH_exit :
21.                 call @printf(@.strErroNaoTratado)
22.                 call void @exit(0)
23.                 unreachable
24.
25.               %f1 :
26.                 %1 = call @fname_1(args_f1)
27.                  ret void
28.                   ...
29.               %fn :
30.                 %n = call @fname_n(args_fn))
31.                  ret void
                }
```

Figura 4.7: Função GEH.

`switch`. Dessa forma, dentor do switch ocorrem desvios para blocos que executarão a instrução call (linhas 25 a 31), em GEH haver um bloco básico para cada função existente no programa.

4.3 Transformação de Programas em Tempo de Compilação: Implementação no LLVM

A técnica ACCE foi implementada no LLVM como uma transformação que modifica programas em nível de LLVM-IR. Para adicioná-la à estrutura do compilador e ativar essa transformação é necessário invocar o LLVM passando os parâmetros adequados conforme ilustra a Figura 4.8.

No quadro 1 da Figura 4.8 são descritos os passos/comandos para a compilação do programa *exemplo*, escrito em linguagem C, para a representação em linguagem intermediária. No quadro 2, são apresentados os comandos usados para transformar o programa LLVM-IR *exemplo.bc* com a técnica ACCE. O resultado desse processo é o arquivo *"exemplo.acce.bc"* com o programa LLVM-IR contendo as instruções, blocos e funções

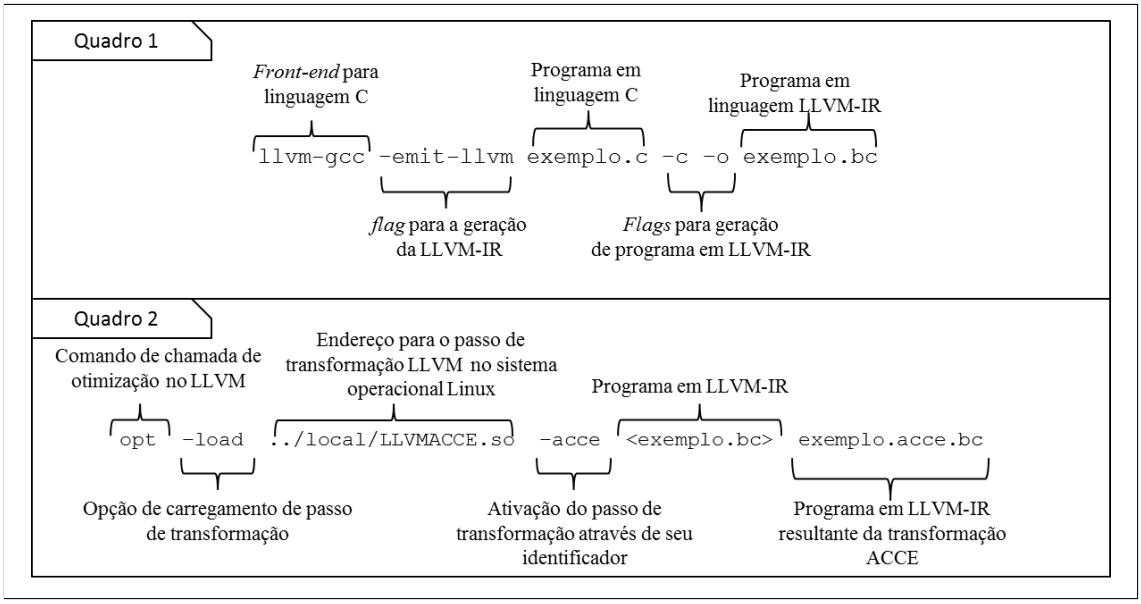

Figura 4.8: Comandos para a ativação de ACCE durante a compilação de programas com o LLVM

adicionais necessárias para a detecção e correção de erros de fluxo de controle.

O desenvolvimento de passos para transformação e análise na representação intermediária, a LLVM-IR, dos programas compilados com o LLVM, é facilitada com o uso de uma API específica para este fim. Essa API é composta de um conjunto de classes, amplamente documentadas, que dão suporte à inserção de novas instruções, blocos, entre outros, em programas. Por exemplo, para a criação de um bloco básico, pode-se usar a classe *BasicBlock* que também permite a extração/definição de várias informações a respeito da estrutura do programa, tal como os predecessores e sucessores do bloco, a instrução de término, as instruções no formato de lista, entre outras.

Outras classes que auxiliam no desenvolvimento em LLVM, por exemplo, são: *Instruction* (permite a criação de instruções), *Function* (permite o gerenciamento das funções pertencentes ao programa) e, *Module* (classe usada para armazenar todas as informações relacionadas a um módulo).

A documentação da API pode ser encontrada em (LATTNER, 2011), com a qual é possível ter um entendimento de como gerar um passo de análise e transformação de código.

No passo de transformação que implementa ACCE foram usadas diversas classes da API LLVM, conforme o trecho de código apresentado na Listagem 4.1, que ilustra a criação da função de tratamento de CFEs globais (GEH) e o bloco básico de entrada para esta função.

De forma similar aos outros passos de transformação existentes no LLVM, ACCE foi implementado em C++ com classes que derivam da classe *Pass*. O passo ACCE consiste de cinco (5) arquivos escritos em C++ que usam as classes *LLVMModule*, *LLVMContex*t, *PassManager* e outras. Todos os 9 arquivos devem estar localizados no diretório llvm/include/llvm/Transforms. Os arquivos são:

- *Assinatura.cpp*: classe usada para o cálculo das assinaturas referentes a cada bloco básico dos programas.

- *Bloco.cpp*: classe que representa os blocos básicos do programa. Nela são arma-

Listagem 4.1: Exemplo de uso da API LLVM

```
//Instrução que permite a criação de uma função (chamada de GEH).
%Function *GEH = cast<Function>(cteGEH);

/*Criação do bloco de entrada da função GEH. Para isto é usado o método CREATE da classe
    BasicBlock, definido o contexto de aplicação deste bloco, o nome (entry) e a função
    à qual fará parte.*/
%BasicBlock* entry = BasicBlock::Create(getGlobalContext(), "entry", GEH);

/*Classe que permite a criação de instruções e inserção delas em um bloco básico.*/
%IRBuilder<> builder(entry);

//Insere uma instrução de Store e insere no bloco básico criado.
%builder.CreateStore(ConstantInt::get(IntegerType::get(getGlobalContext(),1),true),
    error_flag);

/*Insere no bloco uma instrução de término de bloco. Neste caso uma instrução de desvio
    incondicional.*/
%builder.CreateBr(find_function);
```

zenados atributos como os predecessores e sucessores do bloco, a função a qual o bloco pertence, os valores para NS, NES, entre outros.

- *ceda_acce.cpp*: classe composta de métodos usados na transformação do programa através da inserção de novas instruções.

- *Net.cpp*: classe usada para a definição e cálculo das Nets formadas a partir dos blocos básicos. Importante para o cálculo das assinaturas.

- *Utils.cpp*: classe composta por diversos métodos de utilidade, que fornecem informações sobre o programa, necessárias para a aplicação da transformação. Exemplo de método: *carregaPreds*, que retorna os predecessores de um determinado bloco básico.

Como descrito na subseção 3.1.2, que apresenta LLVM-IR, as instruções ϕ, quando presentes, devem estar localizadas, obrigatoriamente, após o *label* de identificação dos blocos, ou seja, não pode haver qualquer outra instrução entre o *label* e a instrução ϕ. Isso impossibilitaria a inserção das instruções para verificação e atualização de assinaturas no topo do bloco. A solução para programas que possuam instruções ϕ é a aplicação do passo de transformação disponibilizado pelo LLVM chamado "reg2mem", que remove estas instruções, conforme informações disponíveis em (REID; HENRIKSEN, 2012).

4.4 Detecção de Erros de Fluxo de Controle em Tempo de Execução

A detecção de erros de fluxo de controle em programas realizada por ACCE, em tempo de execução, dá-se com a verificação das assinaturas no interior e no final da execução de cada bloco básico. Assim, durante a execução dos blocos de proteção superior e inferior dos blocos básicos, o valor do registrador de assinatura é comparado com as assinaturas calculadas em tempo de compilação. Durante essas verificações, caso os valores das assinaturas não correspondam, o controle é transferido para FEH, representando a detecção de um desvio ilegal.

A Figura 4.9 mostra um exemplo da verificação e atualização das assinaturas realizadas em tempo de execução em um bloco básico do programa.

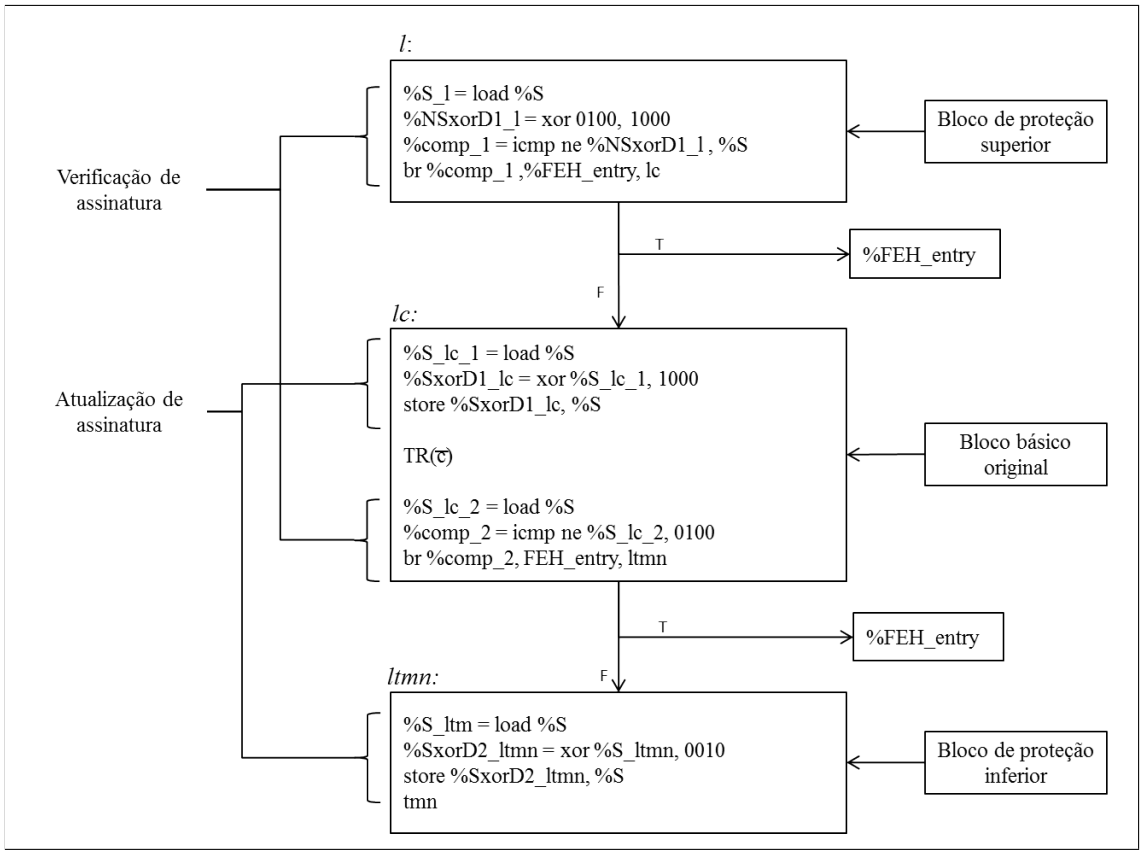

Figura 4.9: Exemplo de detecção de erro de fluxo de controle por ACCE

Nesse exemplo, na primeira verificação realizada no bloco *l*, caso o valor do registrador S, carregado da memória e armazenado no registrador %S_l, seja diferente da assinatura resultante da operação NS(l) xor D1(l), com os valores correspondentes (0100 xor 1000), o bloco de label FEH_entry é chamado, indicando um erro de fluxo de controle. Caso os valores correspondam, S é atualizado no início do bloco *lc* e passa a conter o valor de assinatura do interior do bloco. A instrução de atualização é dada com a operação de xor entre S e D1 do bloco.

No final do bloco *lc*, é realizada a segunda comparação, agora a assinatura de S sendo comparada com o valor interno no nodo (0100), para avaliar se o controle passou pelo bloco conforme esperado e, da mesma forma, em caso de divergência, o bloco de label %FEH_entry é chamado para indicar desvio inesperado.

Por fim, é realizada uma nova atualização de assinatura, agora dentro de *ltmn*, sendo armazenado no registrador %S o valor resultante da operação S xor D2 do bloco.

4.5 Correção do Controle do Programa em Tempo de Execução

O processo de correção inicia com a transferência do controle para FEH após a verificação da divergência nas assinaturas, com a chamada do label %FEH_entry, conforme o processo demonstrado na seção 4.4.

Uma vez em FEH, caso a origem e destino do CFE sejam blocos da mesma função, o controle é transferido para o bloco de origem do erro através de uma comparação no valor de assinaturas, restabelecendo o fluxo. Caso contrário, FEH transfere o controle para GEH, que deverá encontrar a função de origem do desvio ilegal, através do valor de

identificação das funções. Identificada a função onde o CFE teve origem, o controle é então transferido para esta função, que irá tratar o erro por meio da FEH adequada.

Para exemplificar a correção de um CFE, com base no CFG da Figura 4.1, vamos supor que um desvio inesperado ocorreu tendo como origem o bloco b1 e destino o bloco b4, ambos da mesma função, conforme mostram as Figuras 4.10 e 4.11, que representam o processo de detecção e correção de forma separada para facilitar o entendimento.

No exemplo, há um desvio inesperado partindo do bloco b1 com destino o inicio do bloco b4, representado pela seta pontilhada marcada por CFE, na Figura 4.10. O processo de correção dá-se com as seguintes etapas:

- Etapa 1: A verificação da assinatura em l4 detecta um CFE, pois há uma diferença entre as assinaturas, e transfere o controle para *FEH_Entry*. Esta etapa é representada em ambas as Figuras 4.10 e 4.11. Na Figura 4.10 o controle é transferido do bloco *l4* para os blocos que representam FEH e, na Figura 4.11, é mostrado a entrada do controle em FEH após a transferência feita por *l4*.

- Etapa 2: Em *FEH_entry* é detectado que a variável **@error_flag** é falsa (é verdadeira apenas quando os CFEs ocorrem entre funções diferentes) e então o controle é transferido para o bloco FEH_vals.

- Etapa 3: Durante a execução do bloco FEH_vals o valor de **@F** é comparado com o valor atribuído como FID da função. Essa comparação é realizada para verificar se o CFE teve como origem e destino instruções localizadas em blocos da mesma função. Assumindo essa premissa como verdadeira, o fluxo de execução é transferido para o bloco básico FEH_check_num_error, para a contagem do número de tentativas de correção, que não pode superar 3 tentativas.

- Etapa 4: Uma vez não tendo ultrapassado as 3 tentativas de correção, o controle é transferido para o bloco FEH_busca_no, para que, através dos valores de NS e NES o bloco de origem do CFE seja encontrado.

- Etapa 5: Se assinatura atual %S corresponder com alguma entrada em FEH_busca_no, o controle é transferido para o bloco correspondente para a chamada do bloco básico no programa. No caso do exemplo, como o CFE teve origem dentro do bloco *lc1*, %S corresponde ao NS do bloco, sendo então o controle transferido para chamaNSB1.

- Etapa 6: Finalmente, o controle retorna ao bloco de origem do CFE para continuar a execução do programa. Na Figura 4.10 a transferência do controle é representada com a saída do bloco *chamaNSB1* e na Figura 4.11 a correção do fluxo é mostrada com a entrada do controle no bloco *lc1*.

O processo de correção de CFEs que ocorram entre blocos pertencentes à funções diferentes é similar ao processo de correção para CFEs entre blocos da mesma função, diferenciando apenas nos seguintes aspectos:

- Ao executar o bloco básico FEH_vals será verificado que os valores de **@F** e de %FID não correspondem, o que indica que o CFE ocorreu entre funções distintas. Com isso, ao invés de ser transferido para o bloco FEH_check_num_error, conforme etapa 3 da Figura 4.10, o controle é transferido para o bloco FEH_call_GEH, responsável pela chamada à função GEH para encontrar a função de origem do CFE para que o mesmo seja tratado pelos blocos de FEH correspondente.

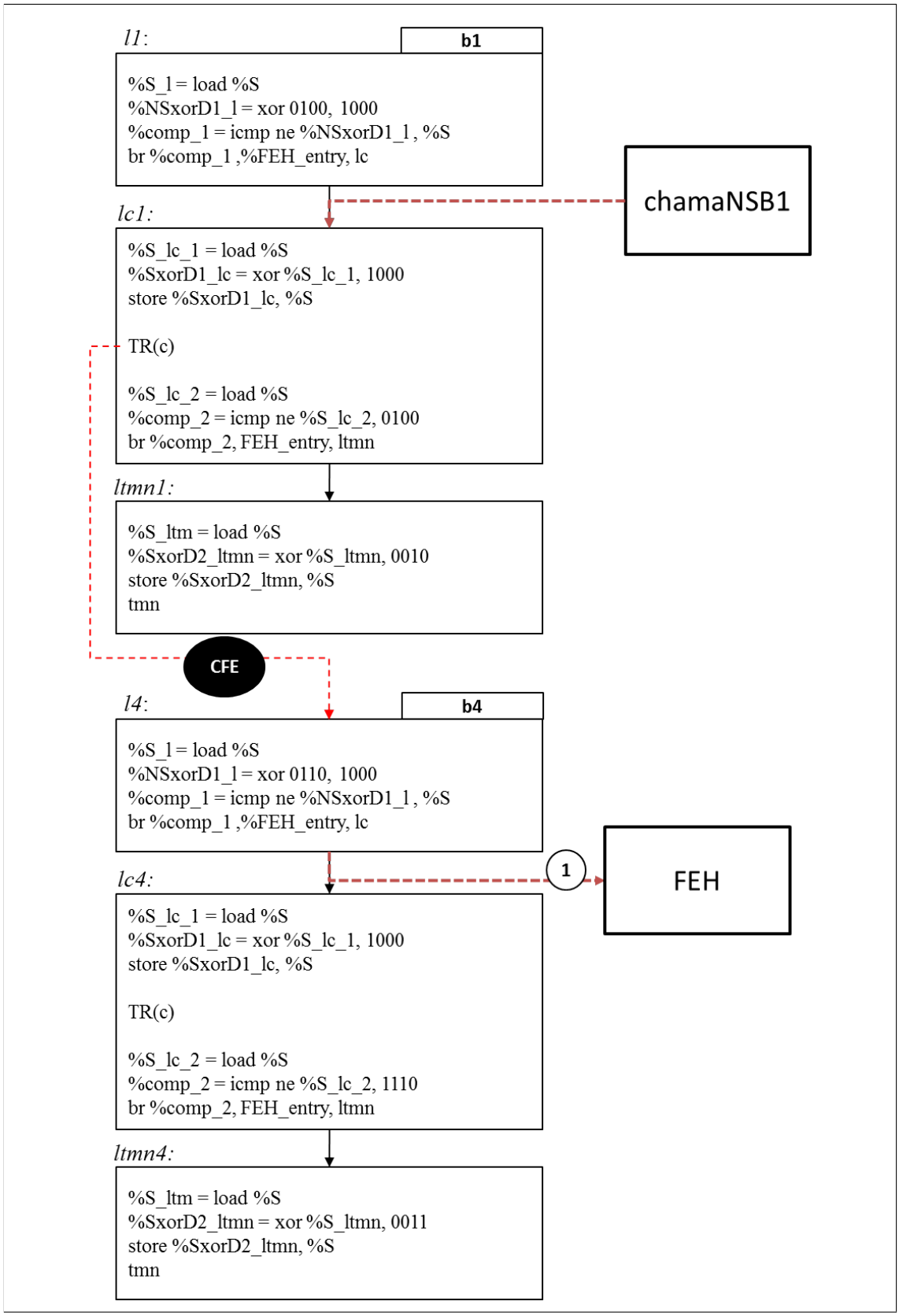

Figura 4.10: Exemplo de correção de CFE por ACCE em blocos da mesma função - Blocos básicos

- Uma vez executada GEH, o bloco de entrada da função de origem do CFE será

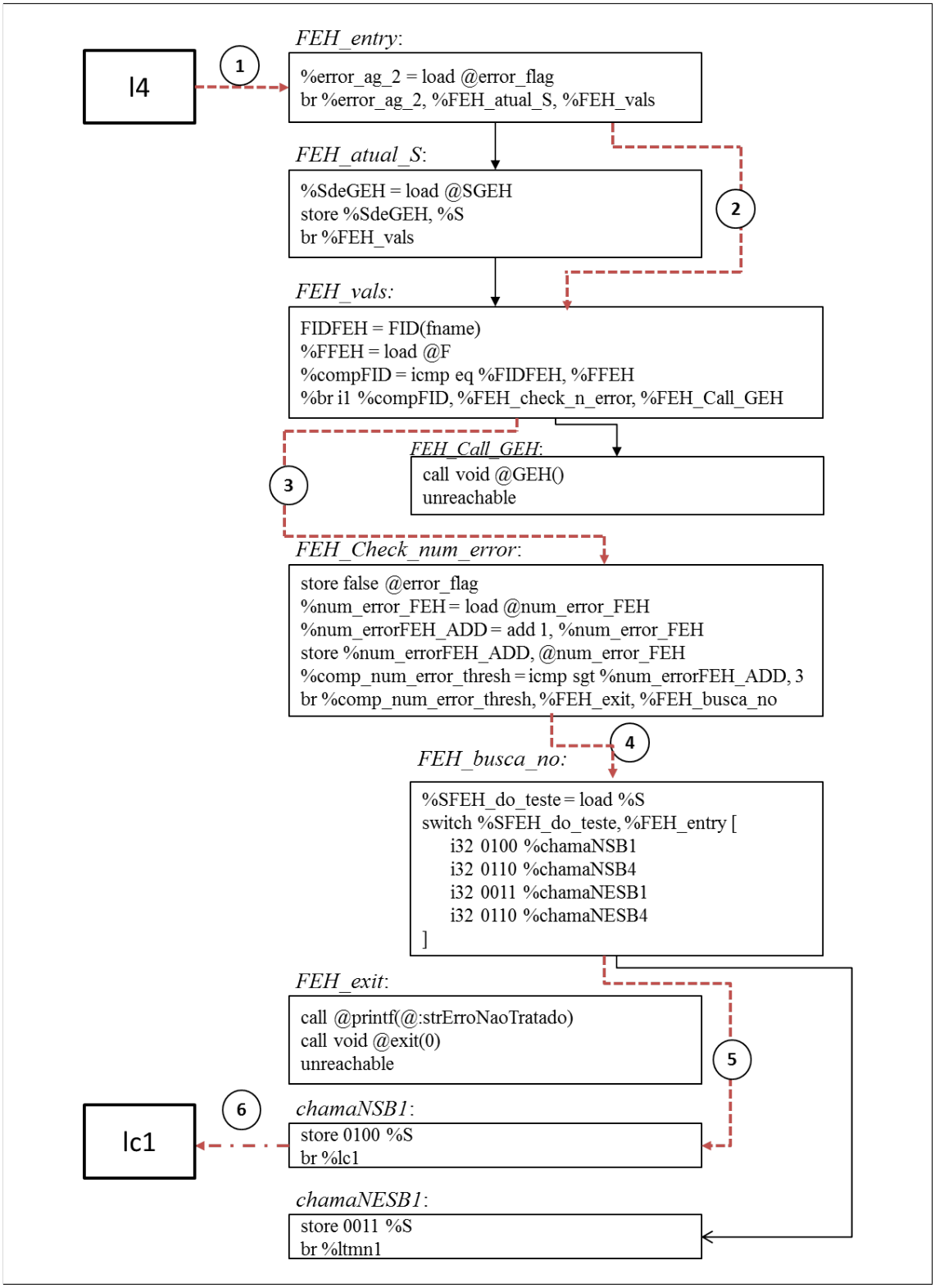

Figura 4.11: Exemplo de correção de CFE por ACCE em blocos da mesma função - FEH

executado (l_entry) e então, automaticamente, transfere o controle para FEH_entry da função, visto que o valor da variável @error_flag é setada para *true* em GEH.

- Ao executar o FEH_entry, agora já na função de origem do CFE, o controle é transferido para o bloco FEH_atual_S, usado para atualizar o valor do registrador %S,

visto que por ter passado por GEH o valor de S pode ter sido trocado. Após esta atualização, o processo de correção se dá de forma semelhante ao de correção de CFEs ocorridos entre blocos da mesma função, tendo a sequência de apresentado anteriormente.

As Figuras 4.12, 4.13 e 4.14 mostram o processo de correção de CFEs com origem e destino instruções localizadas em blocos básicos pertencentes a funções diferentes.

Supondo agora que o bloco *b1* pertença à função *f1* e que o bloco *b4* pertença à função *f2*, o processo de correção de CFEs ocorre através das seguintes etapas:

- Etapa 1: No final de *lc4* a verificação de assinaturas detecta um CFE e transfere o controle paro bloco FEH_entry da sua função (f2).
- Etapa 2: Em FEH_Entry, o controle é transferido para FEH_vals após a verificação de que a variável **@error_flag** é falsa. Vale ressaltar que esta só será verdadeira após a execução da função GEH.
- Etapa 3: Em FEH_vals são comparados o valor atual de @F (1) no momento do desvio inesperado e o valor esperado de FID na função em execução de tratamento de CFE (2). Como são divergentes, o controle é transferido para o bloco FEH_Call_GEH.
- Etapa 4: A responsabilidade do bloco FEH_Call_GEH é chamar a função GEH, para encontrar a função de origem do CFE, e portanto, transfere o controle para GEH via instrução *call*.
- Etapa 5: No início da execução de GEH, no bloco GEH_entry, o valor da variável **@error_flag** é setado para *true*, indicando que um CFE precisa ser tratado e que a função de origem deve ser encontrada, para então transferir o controle para ela. Após, o controle é transferido para o bloco de label find_fun, para a busca da função de origem do CFE.
- Etapa 6: Após a busca pela função, o controle é transferido para o bloco que a chama, para que o controle do programa volte à função que estava em execução no momento imediatamente anterior à ocorrência do CFE. Aqui, já se conseguiu corrigir o fluxo em nível de função, faltando ainda corrigir em nível de bloco.
- Etapa 7: O controle é transferido para o início da função originária do CFE, sendo o primeiro bloco executado o *l_entry*, isto é, o bloco de entrada da função modificado pela técnica ACCE.
- Etapa 8: Como já explicado anteriormente, o bloco de entrada da função analisa a variável **@error_flag**, que nesse caso será verdadeira, transferindo o controle para o FEH_entry de *f1*, visto que o fluxo de controle ainda não foi corrigido.
- Etapa 9: Uma vez em FEH_entry, da função f1, o controle é transferido para o bloco FEH_atual_S, visto que é necessário atualizar o valor de %S já que o controle passou pela função GEH.
- Etapa 10: Após a atualização do valor de %S o controle é transferido para o bloco FEH_vals para então reestabelecer o fluxo de controle. Daí em diante, o processo de correção é similar ao apresentado quando da ocorrência de CFEs entre blocos da mesma função, repetindo as etapas 3→4→5→6.

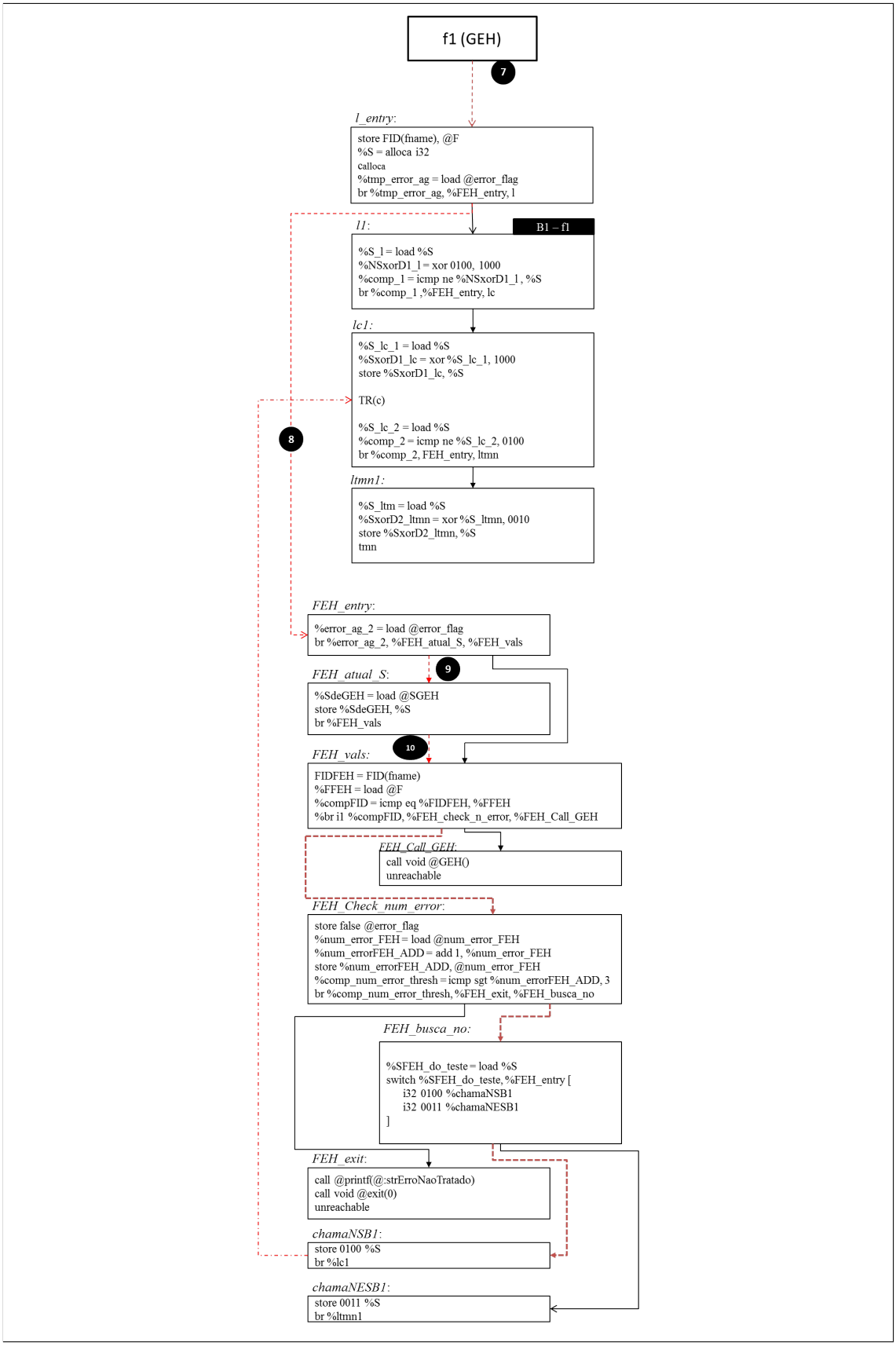

Figura 4.12: Exemplo de correção de CFE por ACCE em blocos de funções diferentes - representação da função 1.

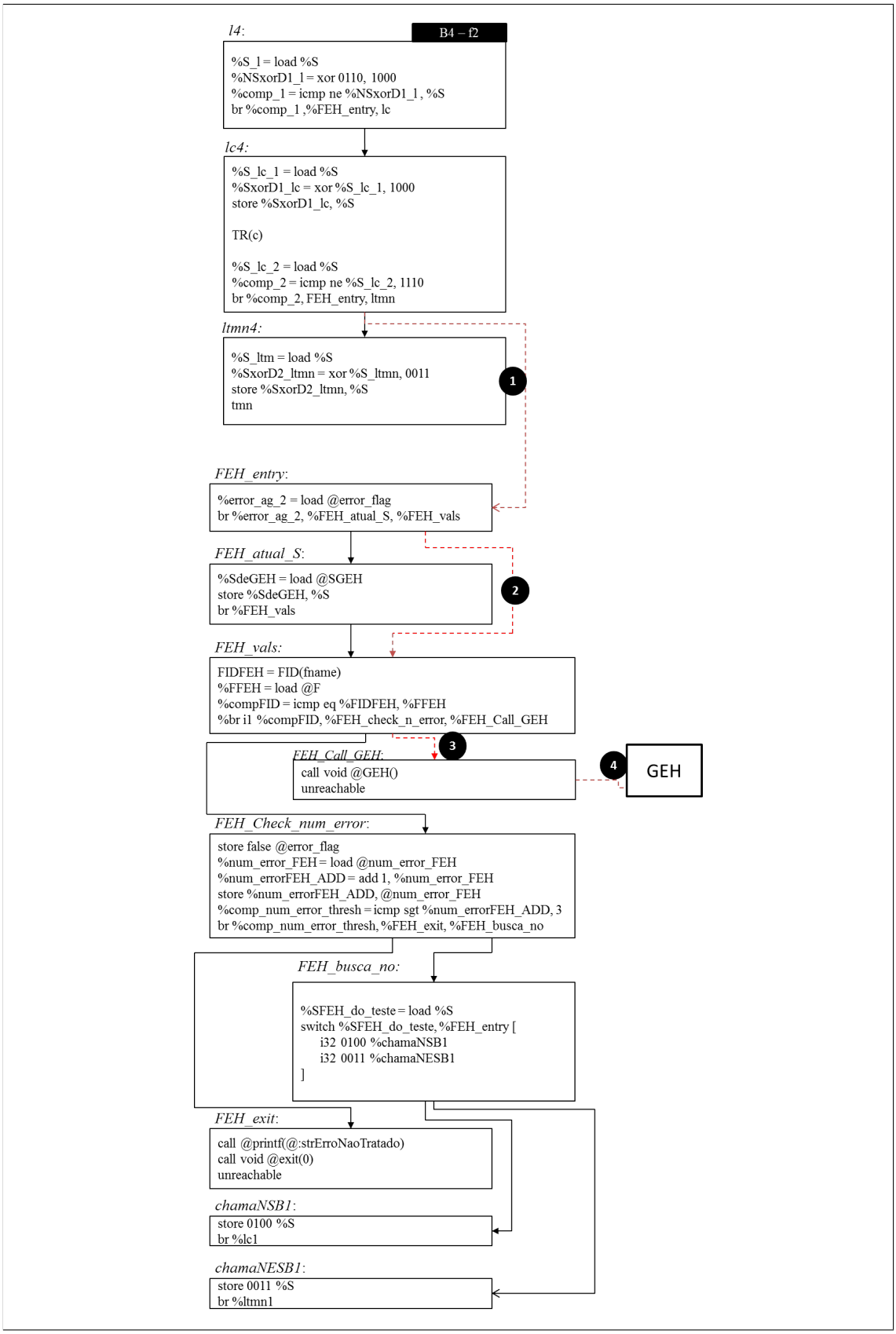

Figura 4.13: Exemplo de correção de CFE por ACCE em blocos de funções diferentes - representação da função 2.

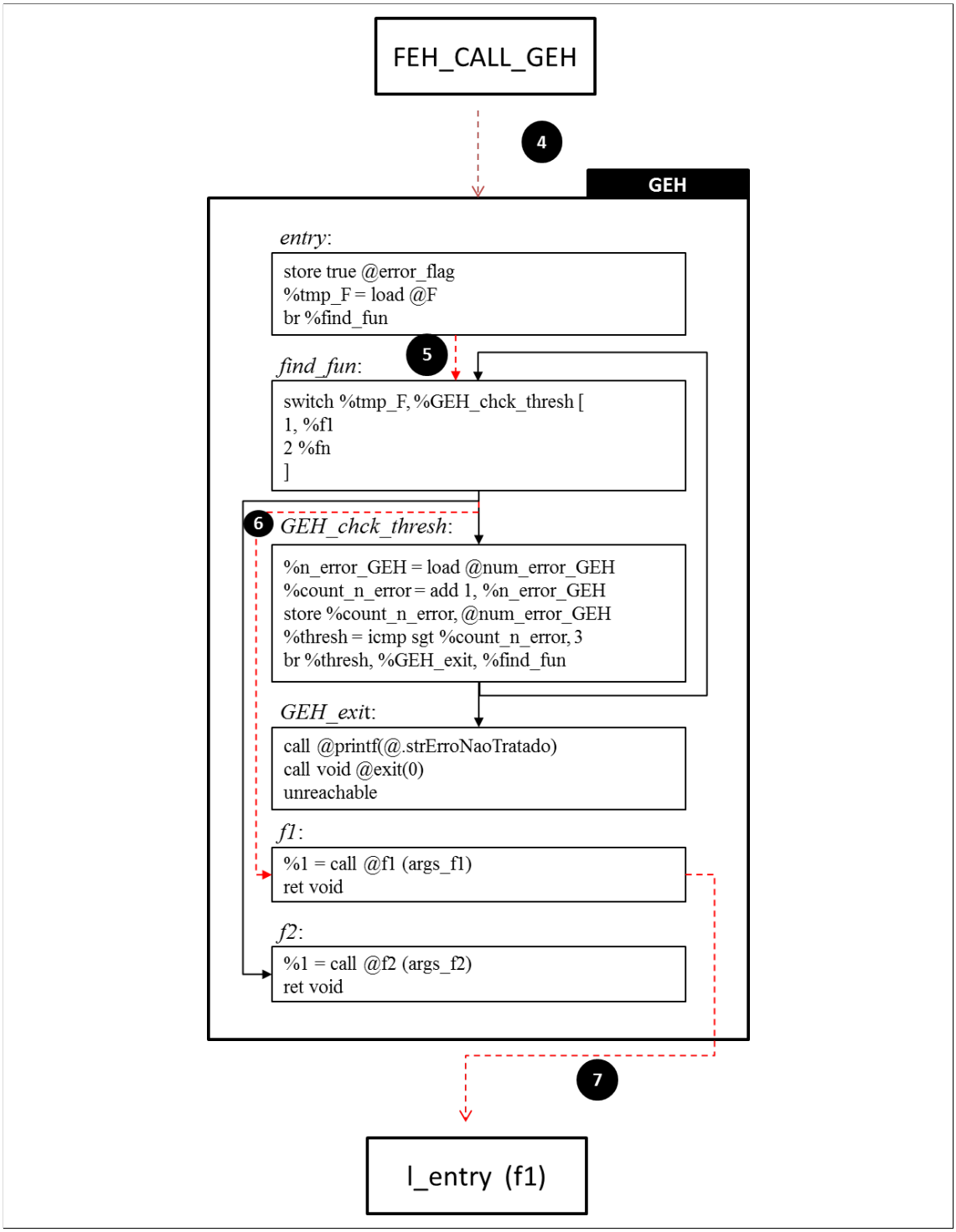

Figura 4.14: Exemplo de correção de CFE por ACCE em blocos de funções diferentes - representação de GEH.

5 EXPERIMENTOS E RESULTADOS

Os experimentos realizados neste trabalho têm por objetivo avaliar o impacto na correção de falhas transientes da técnica ACCE quando aplicada em conjunto com outras transformações de código disponibilizadas em compiladores.

Esta é portanto uma contribuição uma vez que a técnica ACCE, desenvolvida por (VEMU; GURUMURTHY; ABRAHAM, 2007), foi originalmente avaliada de forma isolada. A realização de experimentos que avaliem a interação de técnicas de tolerância a falhas tais como ACCE, com outras transformações de códigos providas por compiladores é fundamental visto que dificilmente programas são compilados sem a ativação de uma ou mais dessas transformações/otimizações.

A Seção 5.1 apresenta algumas conclusões obtidas com os experimentos sobre o impacto de transformações sobre ACCE. A Seção 5.2 apresenta a metodologia aplicada aos experimentos (*benchmarks*, forma de injeção de falhas e cenários de experimentação utilizados). A Seção 5.3 detalha os cenários de experimentação e os seus resultados e retoma a análise já discutidas brevemente na Seção 5.1.

5.1 Objetivos e Conclusões dos Experimentos

A Figura 5.1 esquematiza a aplicação das transformações de código disponíveis no LLVM e da técnica ACCE. Inicialmente, o código fonte em linguagem de programação é traduzido para a linguagem intermediária do LLVM. O programa LLVM-IR resultante passa então por transformações disponibilizadas pelo compilador, gerando um programa transformado, ainda em linguagem LLVM-IR. Na sequência, sobre o programa já transformado, a técnica ACCE é ativada e então são inseridas novas instruções para o tratamento dos CFEs. Finalmente, é gerado o código objeto para a arquitetura de máquina desejada.

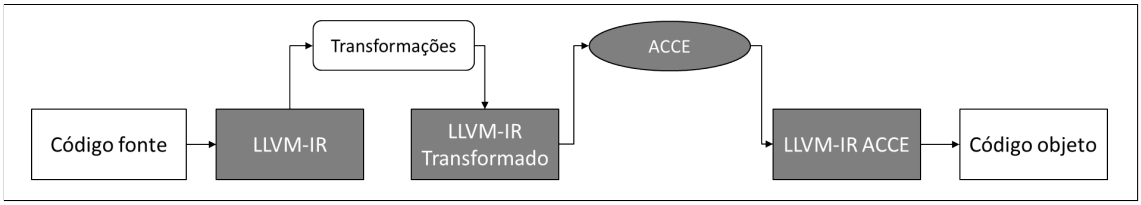

Figura 5.1: Processo de aplicação da técnica ACCE no LLVM.

Neste trabalho os experimentos foram feitos em cima de código gerado para o X86 e buscaram identificar *corner cases* em relação a eficácia na detecção/correção de erros. Uma vez identificados, os *corner cases* foram analisados visando compreender a influência de transformações sobre ACCE.

Uma primeira constatação importante em relação ao uso de ACCE em conjunção com outras transformações é que ela deve ser sempre a última a ser aplicada nos programas durante a compilação, uma vez que a maioria das transformações removem instruções, o que poderia resultar na exclusão das instruções inseridas por ACCE, quebrando o efeito da técnica.

Os resultados dos experimentos mostraram que algumas transformações disponibilizadas pelo LLVM podem ser benéficas para alguns *benchmarks* e prejudiciais para outros. Ou seja, para dizer se uma determinada transformação é benéfica ou prejudicial para a eficácia de ACCE é preciso analisá-la em conjunto com a estrutura do programa original sobre o qual a transformação será aplicada.

Com nossos experimentos, descobriu-se também que o aumento da quantidade de blocos pode ser **prejudicial** a cobertura de falhas. Isso contradiz o artigo original (VEMU; GURUMURTHY; ABRAHAM, 2007) que diz o seguinte sobre a eficácia de ACCE em relação ao número de blocos de um programa:

> *"In this paper, we deal with detecting and correcting only inter-node CFEs. If more coverage needs to be obtained, a node can be divided into sub-nodes and the proposed technique applied, thus detecting intra-node CFEs which now become inter-node CFEs. This will lead to increased performance and memory overhead along with increased coverage".*

No trecho acima, extraído do artigo (VEMU; GURUMURTHY; ABRAHAM, 2007), os autores informam que a técnica detecta e corrige apenas CFEs que ocorram inter-blocos, o que foi constatado nesse trabalho. Porém, na sequência, informam que, para aumentar a eficácia da técnica, em termos de correção de falhas, deve-se dividir os blocos básicos em sub-blocos, dado que com isso haveriam menos desvios intra-blocos, o que não foi confirmado no presente trabalho, uma vez que os "novos" sub-blocos terão poucas instruções, o que resultará em falhas sobre as instruções inseridas pela técnica, o que será mostrado que ocasionam falhas não corrigidas.

Nos experimentos pode-se verificar que algumas transformações introduzem blocos básicos muito pequenos compostos apenas pelo *label* de identificação e uma instrução de desvio para o bloco sucessor. Estes blocos, chamados aqui de blocos *label-br*, são criados por algumas transformações para separar o fluxo de controle do programa de modo a melhorar a organização do programa.

Muitos blocos protegidos muito pequenos impactam negativamernte na eficácia de ACCE. Isso acontece pois ACCE insere mais instruções protetoras (não protegidas) do que as que irá proteger, aumentando assim a probabilidade de que mais falhas ocorrerão sobre tais instruções nesses blocos pequenos não protegidas. A situação é ainda mais desfavorável a ACCE quando tais blocos ocorrem dentro de laços aninhados.

Essa constatação leva a conclusão de que é preciso preocupar-se com as transformações que são aplicadas em programas com laços aninhados quando se pretende obter alta eficácia de tolerância a falhas com a técnica ACCE.

O detalhamento dos experimentos e dos seus resultados são explicados na Seção 5.2.3.

5.2 Metodologia dos Experimentos

5.2.1 *Benchmarks* Utilizados

Um conjunto de 10 programas da suíte de *benchmarks* Mibench (GUTHAUS et al., 2001) foi utilizado nos experimentos. Mibench é formada por 35 programas escritos em

C, que são agrupados em seis categorias que simulam diferentes áreas de aplicação de sistemas embarcados:

- *Controle industrial e automotivo:* compreende programas que representam o uso de processadores em sistemas de controle embarcados voltados a operações matemáticas como manipulação de *bits*, de matrizes, entre outros.
- *Dispositivos de clientes:* categoria composta por programas usados por equipamentos de multimídia tais como *scanners*, impressoras, entre outros. Os programas são baseados em codificação/decodificação de imagens, conversão de cores, arquivos de áudio e páginas *html*.
- *Automação de escritório:* algoritmos de manipulação de texto como pesquisa por palavras em frases e outros, que simulam equipamentos usados em escritórios, como por exemplo, editores de texto.
- *Rede: benchmarks* que representam processadores embarcados em dispositivos de rede, como roteadores. Estes programas desempenham funções de busca pelo menor caminho para o trafego de dados, tratamento de pacotes de dados em operações de entrada e saída, etc.
- *Segurança:* Categoria que compreende os algoritmos que permitem aplicar segurança aos dados. Estão contidos nessa categoria programas que realizam criptografia de dados.
- *Telecomunicações:* Algoritmos que simulam operações realizadas em redes de comunicação como codificação e decodificação de voz e análise de frequência.

A Tabela 5.1 mostra os programas de *benchmark* disponíveis no Mibench, agrupados de acordo com suas categorias, destacando, em negrito, os programas utilizados durante os experimentos:

Tabela 5.1: Programas de *benchmark* disponíveis na suíte Mibench

Aut. industrial	Clientes	Escritório	Rede	Segurança	Telecom.
basicmath	jpeg	ghostscript	**dijkstra**	blowfish	**crc32**
bitcount	lame	ispell	**patricia**	pgp sign	**fft**
qsort	mad	rsynth	crc32	pgp verify	ifft
susan	tiff2bw	sphinx	sha	**rijndael**	adpcm
	tiff2rgba	**stringsearch**	blowfish	sha	cgsm
	tiffdither				
	tiffmedian				
	typeset				

Os 10 programas do *benchmark* usados nos experimentos estão descritos abaixo:

- *Basicmath:* Executa cálculos matemáticos simples que não necessitam de hardware específico, como solução de funções cúbicas, raízes quadradas, conversões de unidades, entre outras.
- *Bitcount:* Realiza a contagem do número de bits de vetores compostos por valores inteiros, para avaliar a capacidade do processador em tratar bits.

- *32-bit Cyclic Redundancy Check (crc32):* Algoritmo de verificação utilizado para identificação de erros durante transmissões de dados.
- *Dijkstra:* Algoritmo que calcula caminhos possíveis entre pares de vértices em um grafo e seleciona o menor.
- *Fast Fourier Transform (fft):* representa a transformada de *Fourier*, utilizada no processamento de sinais digitais.
- *Patricia:* Programa usado para representar tabelas de roteamentos utilizados para prestação de serviços nas áreas de redes de computadores e conexões.
- *Qsort:* Programa utilizado para a ordenação de dados.
- *Rijndael:* Executa operações de encriptação e decriptação de arquivos para garantir segurança.
- *Stringsearch:* realiza a busca de palavras em sentenças, operação bastante usada no processamento de textos.
- *Susan:* Permite o processamento de imagens como conversão de cores e tons.

5.2.2 Injeção de Falhas

Os experimentos consistiram na realização de uma campanha de injeção de falhas em 3 cenários experimentais diferentes (veja Seção 5.2.3). As falhas foram injetadas sobre código *Assembly* do X86 gerado a partir do programa LLVM-IR, com o auxílio do *framework* de depuração do X86, GDB (*Gnu Debugger*) disponível para Linux. O processo de injeção de falhas, similar ao apresentado em ((VEMU; ABRAHAM, 2006), (VEMU; ABRAHAM, 2011) (VEMU; GURUMURTHY; ABRAHAM, 2007)), insere uma única falha em cada execução. A falha é inserida mudando o valor do programa counter (PC) e essa mudança de valor é feita seguindo uma dentre as possibilidades abaixo:

- falha em uma instrução *regular* ou seja, um instrução que não é de desvio. Nesse caso o PC, ao invés de ser incrementado para apontar para a próxima instrução, receberá um outro valor qualquer. Esse modelo de falhas foi chamado de Criação de Desvio em (VEMU; ABRAHAM, 2006).
- falha em uma instrução de desvio. O PC, ao invés de receber o valor determinado um dos operandos da instrução de desvio, é incrementado e passa a apontar para a próxima instrução. Esse modelo de falhas foi chamado de Remoção de Desvio em (VEMU; ABRAHAM, 2006).
- falha em uma instrução de desvio na qual o PC, ao invés de receber o valor de um dos operandos da instrução, recebe um outro valor qualquer. Esse modelo de falhas foi chamado de Alteração de Operando de Desvio em (VEMU; ABRAHAM, 2006).

O processo de injeção de falhas é feito de acordo com as seguintes etapas:

1. O *trace* de execução do programa em linguagem *Assembly* é extraída com o GDB.
2. Um modelo de falha (criação, remoção ou alteração de operando de desvio) é aleatoriamente selecionado.

3. Uma das instruções do *trace* obtida na etapa 1 é randomicamente selecionada para a injeção da falha. A instrução é escolhida randomicamente mas deve ser adequada ao modelo de falhas escolhido no passo 2.

4. Uma variável de contagem é usada para armazenar o número de execuções da instrução selecionada na etapa 3 e, randomicamente, uma dessas execuções é selecionada. Esse procedimento é realizado pois, por exemplo, as instruções internas a um laço serão executadas várias vezes, e em apenas em uma dessas execuções é que a falha será injetada.

5. Com o GDB novamente, um *breakpoint* é inserido na execução selecionada na etapa 4 na instrução selecionada na etapa 3.

6. Durante a execução do programa, ao alcançar o *breakpoint* inserido na etapa 5, o contador do programa (PC) é alterado para injetar um específico modelo de falha selecionado na etapa 2.

7. O programa retoma a execução com o novo valor do PC e continua a execução até o fim.

O universo de falhas injetadas com os passos descritos contém falhas de todos os tipos apresentados no Capítulo 2, com falhas que representam desvios ilegais, ou seja, desvios não previstos no CFG do programa, e falhas que representam desvios errados que, apesar de existir no CFG do programa, não é esperado para aquele momento do programa. Além disso, com o processo de falhas apresentado, as falhas são injetadas em instruções escolhidas de forma aleatória, o que permite que possam ser injetadas falhas tanto em instruções originais do programa quanto em instruções inseridas pela técnica ACCE. Portanto, os valores que representam as taxas de correção de falhas, apresentados na Seção 5.3, são percentuais de correção sobre o universo completo de falhas.

5.2.3 Cenários de Experimentação

Os cenários para a realização dos experimentos são os seguintes:

- Cenário 1 - Cada um dos 10 programas de *benchmark* foi compilado submetido a transformação ACCE somente. Esse cenário estabelece *baselines* de comparação com os resultados dos demais cenários descritos abaixo.

- Cenário 2 - Cada um dos programas (*Basicmath*, *Bitcount*, *CRC32*, *Dijkstra*, *FFT*, *Patricia*, *Qsort*, *Rijndael*, *Stringsearch* e *Susan*) foi compilado 58 vezes. Em cada compilação de cada programa foi aplicada somente uma das 58 transformações disponíveis no LLVM seguida da aplicação de ACCE. A Figura 5.2 ilustra esse cenário que resultou em 58 versões diferentes de cada programa. Esse cenário teve como objetivo avaliar a influência de cada passo de transformação LLVM na tolerância a falhas com ACCE.

- Cenário 3 - Este cenário é baseado em metodologia aplicada em (FURSIN et al., 2008) que mostra que a aplicação aleatória de otimizações pode resultar em aumento no desempenho de programas. Neste cenário, cada *benchmark* foi compilado 5 vezes. Para cada uma dessas 5 compilações foram selecionadas e aplicadas 10 transformações seguidas de ACCE. As 10 transformações foram aleatoriamente selecionadas e ordenadas. Todo esse processo foi então repetido com 5 compilações

com 20, com 30, 40, 50 e com 58 transformações. Na Figura 5.3 $Comb_1Transf_{10}$ representa a primeira das 5 sequências compostas por 10 transformações aleatoriamente selecionadas aplicadas sobre o programa, $Comb_2Transf_{10}$ representa a segunda sequência composta por outras 10 transformações e assim por diante.

- Cenário 4 - Neste cenário, três *benchmarks* (*crc32, quicksort, e string search*) foram compilados 58 vezes. Em cada compilação de cada programa foi aplicada somente uma das 58 transformações disponíveis no LLVM seguida da aplicação de ACCE, com o objetivo de avaliar o efeito das transformações, juntamente com ACCE, no consumo de energia e tempo de execução, conforme a Figura 5.4.

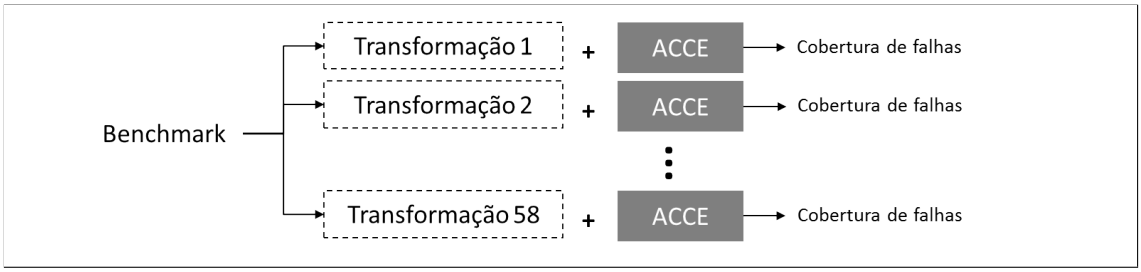

Figura 5.2: Esquematização do cenário 2 dos experimentos.

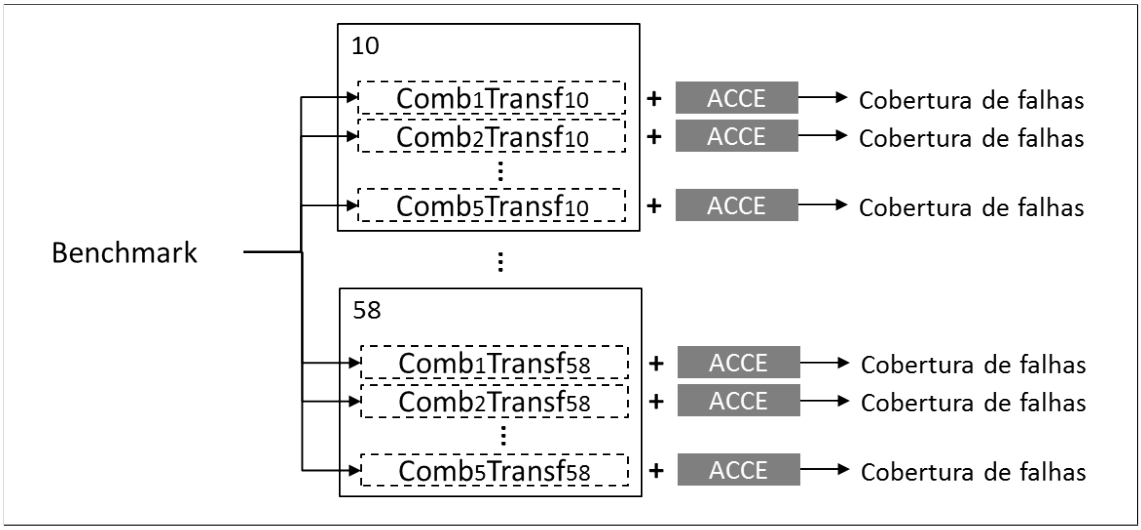

Figura 5.3: Esquematização do cenário 3 dos experimentos.

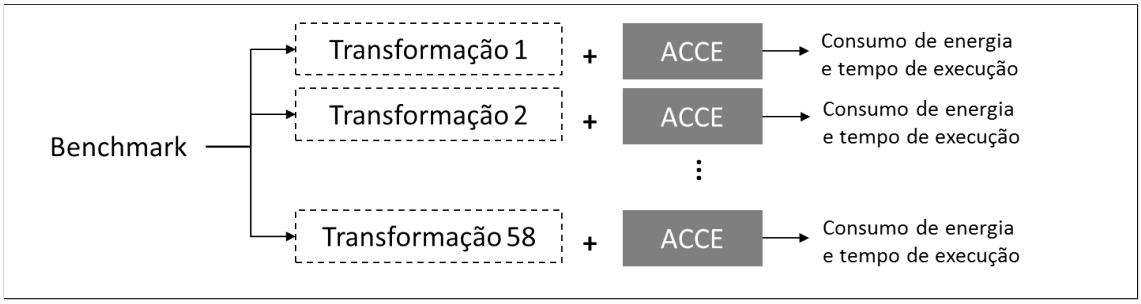

Figura 5.4: Esquematização do cenário 4 dos experimentos.

5.3 Experimentos

Para a realização dos experimentos de avaliação da eficácia de ACCE, foi utilizado o compilador LLVM versão 2.9, executando em um computador Intel Core i5 2.4 GHz, arquitetura X86-64 com 4 *Gigabytes* de memória RAM. As falhas foram injetadas em código gerado para o processador X86 a partir de programa LLVM-IR.

5.3.1 Aplicação Individual de Transformações seguidas por ACCE

Nesse cenário de experimentos, as 58 transformações providas pelo LLVM foram aplicadas individualmente sobre dez *benchmarks* da suíte Mibench, descritos na Subseção 5.2.1. A última transformação aplicada na compilação de cada *benchmark* foi a técnica ACCE, conforme processo ilustrado pela Figura 5.2. Assim, nesse cenário foram obtidas 58 versões distintas para cada um dos *benchmarks*, resultando em um total de 580 versões de programas para a avaliação da técnica ACCE. Para cada versão obtida dos programas foi realizada a injeção de uma falha e esse processo foi repetido 1000 vezes para cada programa, resultando em 580.000 falhas injetadas.

Uma primeira constatação é que, na média, ACCE continua eficaz na correção de falhas mesmo quando combinada com transformações. A Figura 5.5, mostra que a técnica ACCE restabeleceu o fluxo de controle em média para cerca de 68,4% das falhas injetadas, confirmando a eficácia da técnica no tratamento de falhas transientes que se manifestam como erros de fluxo de controle. Para o cenário baseline essa média de correção foi de 67,45%. A Figura 5.6 mostra o desvio padrão para os experimentos no cenário 2.

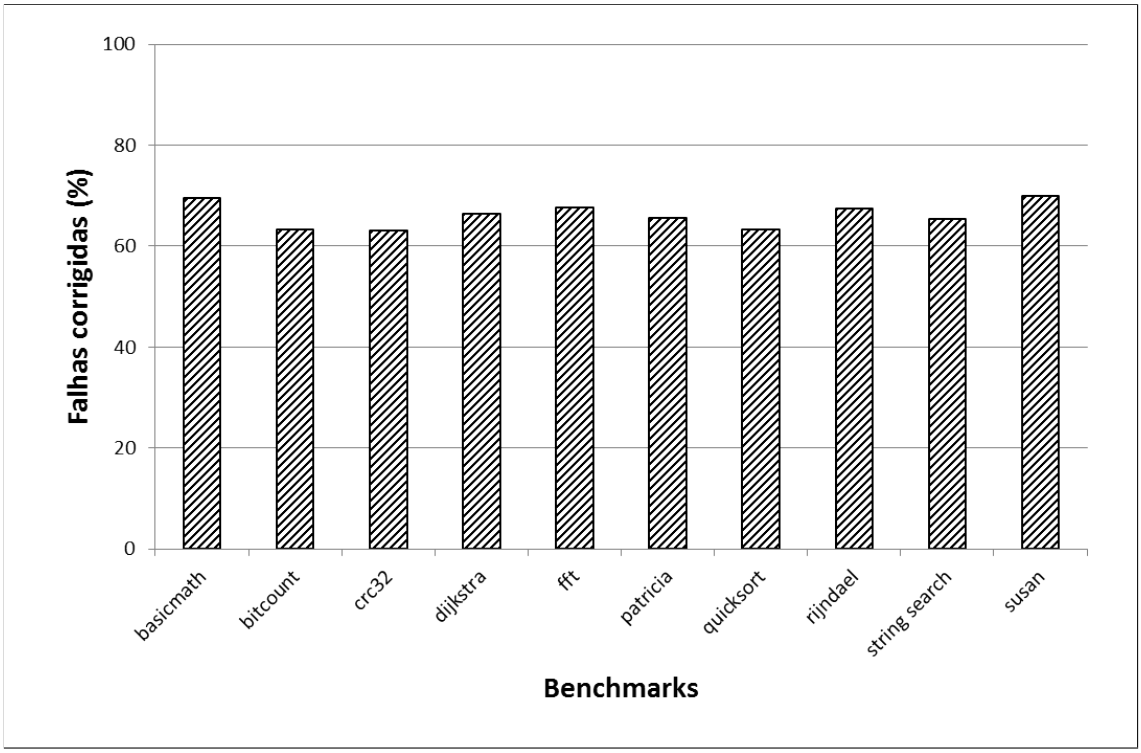

Figura 5.5: Taxas de correção de ACCE quando combinada com 1 transformação.

Embora os experimentos nesse cenário tenham mostrado que ACCE continua, na média, eficaz quando combinada com transformações, a análise continua em busca de *corner cases* buscando explicar o que influencia positivamente/negativamente a taxa de cobertura de determinados casos.

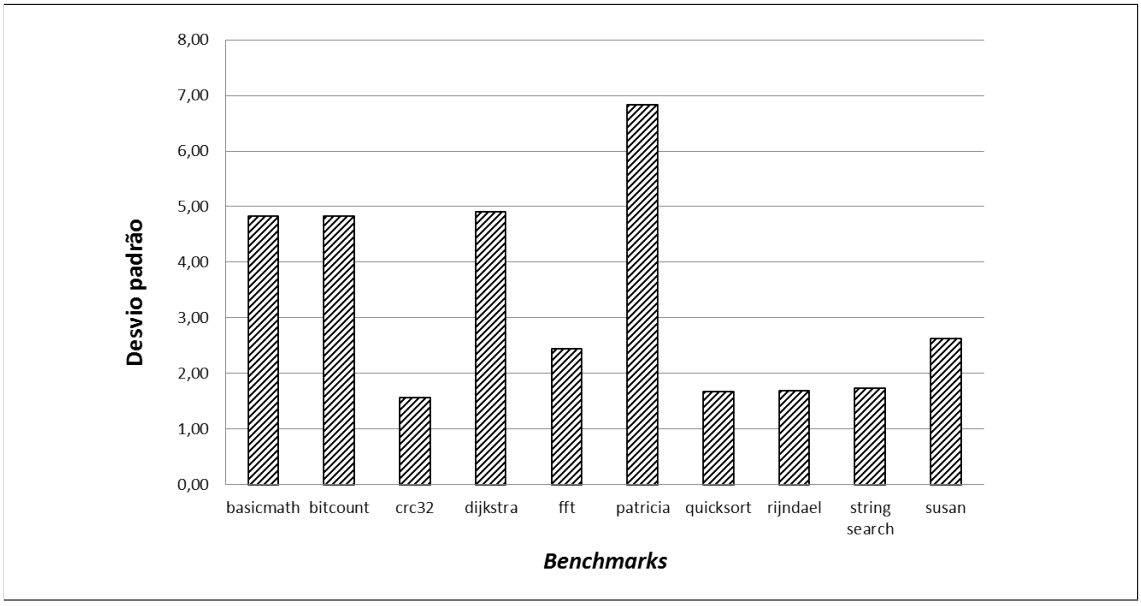

Figura 5.6: Desvio padrão da correção de ACCE quando combinada com 1 transformação.

A Tabela 5.2 apresenta as taxas de correção de ACCE neste cenário, normalizadas com base nos resultados extraídos dos programas compilados apenas com a aplicação da técnica ACCE, sem qualquer outra transformação ativada. Em destaque na Tabela 5.2 estão os casos que resultaram em aumento/redução na taxa de correção de falhas superior a 10% quando confrontadas ao *baseline*, casos chamados de *corner cases*.

Os resultados demonstrados pela Tabela 5.2, mostram, dentre outras coisas, que uma mesma transformação pode ser benéfica para alguns e prejudicial para outros *benchmarks*, no que diz respeito à correção de falhas. A Figura 5.7 dá um exemplo desse fato apresentando o efeito sobre a correção de CFEs de *loop-reduce* para os programas do *benchmark* quando comparado ao baseline.

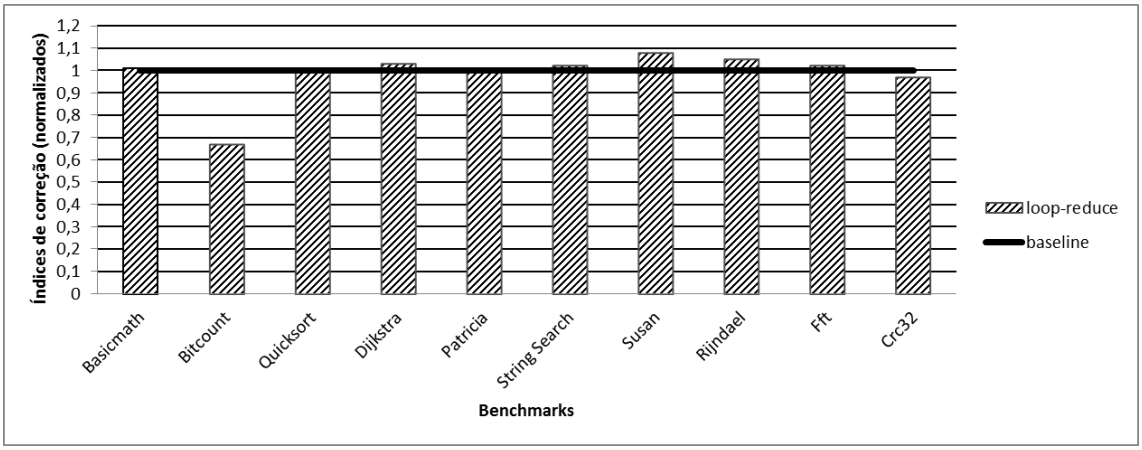

Figura 5.7: Efeito de *loop-reduce* sobre a correção de CFEs nos *benchmarks*.

De acordo com a Figura 5.7, a aplicação da transformação *loop-reduce* nos programas *basicmath, dijkstra, patricia, string search, susan, rijndael e fft*, colaborou positivamente para a taxa de correção de ACCE, tendo o maior aumento ocorrido em *susan* (8%). Apesar disto, para os programas *bitcount* e *crc32* o efeito da aplicação de *loop-reduce* foi oposto, resultando em uma redução na correção das falhas de até 33% a menos do que o *baseline*,

Tabela 5.2: Resultados da correção de CFEs com ACCE para cada otimização do LLVM.

	Benchmarks									
	basicmath	*bitcount*	*quicksort*	*dijkstra*	*patricia*	*string search*	*susan*	*rijndael*	*fft*	*crc32*
adce	0,96	1,06	0,98	1,03	1,05	1,02	1,04	1,06	1	1,01
always-inline	0,95	1,03	1	1,01	1,02	1	1,06	1,09	1,02	1
argpromotion	0,99	1,06	0,98	1	1,07	1,04	1,04	1,06	0,97	1,04
block-placement	0,98	1,06	0,99	1,12	1,07	1,03	1,02	1,01	0,86	1,03
break-crit-edges	0,99	1,04	1,05	1,01	0,75	1,02	1,01	1,04	1,07	1
codegenprepare	0,97	1,09	1,02	1,03	1,07	1,02	1,06	1,03	1	1,01
constmerge	0,96	1,06	0,99	1,02	1,01	1	1,05	1,02	1,05	0,99
constprop	1	1,01	0,96	0,98	1,03	1,02	1,05	1,02	1,05	0,99
dce	0,98	1,08	0,98	1,02	1,02	1,04	1,06	1,09	1,05	1,01
deadargelim	0,97	1,01	0,99	0,97	1,03	1,01	1,05	1,04	1,01	1,03
deadtypeelim	0,96	1,06	0,98	0,94	1,04	1,02	1,07	1,05	1,02	1,03
die	0,96	0,94	0,97	1,02	1,05	1,04	1,06	1,01	0,99	0,96
dse	0,96	1,08	0,97	1,01	1,05	0,99	1,03	1,06	1,01	1,03
functionattrs	1,04	1,05	1,01	1,02	1,04	1,04	1,03	1,08	1,02	1,02
globaldce	0,99	1,02	1,01	1,03	1,05	1,02	1,03	1,04	0,97	0,98
globalopt	0,96	1,1	1,02	1,01	1,02	1,01	1,05	1,06	0,96	1,04
gvn	1	1,02	1,07	1,06	0,84	1,06	1,03	1,04	1,01	1,06
indivars	1	0,73	0,96	1	1,03	1,01	1,03	1,08	0,96	1,06
inline	0,98	1,01	0,96	1	1,07	0,98	1,04	1,01	1	1,02
instcombine	1,02	1,02	1,02	1,03	0,78	1,03	1,05	1,01	1,13	1,03
internalize	0,97	1,08	1	1,03	1,03	1,05	1,06	1,08	1,01	1,03
ipconstprop	1	1,11	0,96	1,03	1,02	1,04	1,01	1,07	0,99	1,05
ipsccp	0,98	1,05	1	1	1,07	0,98	1,08	1,06	0,98	1,04
jump-threading	0,99	1,03	1,02	0,99	0,82	1,01	1,02	0,99	1,04	1,04
lcssa	1	1,07	1	1,06	1,04	0,99	1,03	1,08	1	0,98
licm	0,97	0,83	0,96	0,97	1,02	1,03	1,06	1,03	1,09	1,01
loop-deletion	0,96	0,99	0,99	0,96	1,05	1,04	1,07	1,05	0,97	0,99
loop-extract	0,98	0,99	0,99	0,99	1,06	1,13	1,03	1,06	0,96	1,01
loop-extract-single	0,96	1,03	1,01	1	0,75	1,05	0,98	1,04	1,01	1,02
loop-reduce	1,01	0,67	1	1,03	1,01	1,02	1,08	1,05	1,02	0,97
loop-rotate	0,53	1	0,96	0,49	1,01	1,01	1,01	1,06	1,01	0,98
loop-simplify	0,99	1,02	0,98	1,01	1,05	1,02	1,08	1,09	0,98	0,99
loop-unroll	0,96	1,01	0,95	0,97	1,05	0,98	1,01	1,04	1,01	1,02
loop-unswitch	0,94	1,03	0,98	1,01	1,05	1,02	1,05	1,05	1,01	1,01
loweratomic	0,98	0,99	1,01	1,06	1,03	1,04	1,04	1,05	1,01	1,03
lowerinvoke	1,01	1	1	0,97	1,05	1	1,03	1,05	1	0,99
lowerswitch	0,97	1,06	0,99	1,04	1,04	0,97	0,92	1,06	0,97	1
mem2reg	1,02	1,04	1,03	1,05	0,77	1,02	0,9	1,08	1,04	0,99
memcpyopt	1	1,02	0,97	1,03	1,03	1,03	1,08	1,1	1	1
mergefunc	0,96	1,01	0,98	0,99	1,06	1	1,08	1,1	1,02	1,03
mergereturn	1	0,98	0,97	1,01	1,06	1	1,07	1,08	1,03	1,03
partial-inliner	0,97	1,03	0,95	1,02	1,02	1,04	1,06	1,04	1,03	1,01
prune-eh	1	0,99	0,99	1,01	1,05	1,01	1,08	1,08	0,99	1,01
reassociate	0,95	1,06	0,97	1,02	1,06	1,03	1,06	1,06	1,03	1,02
reg2mem	1	1	1,02	1,05	0,8	1	1,04	1,08	1	1
scalarrepl	1,01	1,04	1	1	0,87	1,02	0,94	1,08	0,97	1,02
sccp	1	1,04	0,99	1,01	1,01	0,98	0,97	1,01	0,99	1,07
simplifycfg	0,97	1,01	0,97	0,98	0,82	1,07	1,01	1,03	1,03	1,03
simplify-libcalls	0,95	1,04	1,02	1,04	1,02	1,01	1,03	1,07	1	0,99
sink	0,98	0,97	0,94	1	0,73	1,04	1,03	1,07	0,99	1,04
sretpromotion	0,99	1	0,97	1,03	1,02	1,01	1,03	1,06	1	1,03
strip	0,96	1,02	0,99	0,97	1,04	1,02	1,06	1,06	1,01	0,95
strip-dead-debug-info	0,97	1,02	1	1	1,03	0,99	1,06	1,07	1	1,03
strip-dead-prototypes	1,02	0,98	0,98	1,01	1,02	1	1,03	1,05	0,99	1,03
strip-debug-declare	0,99	1,04	1,04	1,03	1,03	1,03	1,05	1,05	1	0,98
strip-non-debug	0,97	1,04	1	1,04	1,05	1	1,07	1,08	1,01	0,98
tailcallelim	0,93	1,08	1	1	1,04	0,98	1,04	1,04	1,01	1,01
tailduplicate	0,99	1,02	0,97	1	1,05	1,06	1,06	1,07	1	1,03

índice registrado para o *benchmark bitcount*.

Os valores da Tabela 5.2 também demonstram que cada transformação pode impactar um mesmo programa de forma específica na correção de falhas realizada por ACCE. Essa característica era esperada visto que um mesmo programa é afetado de formas diferentes por diferentes transformações.

Um exemplo desta situação é ilustrado gráfico da Figura 5.8, que apresenta os diferentes resultados extraídos de algumas transformações aplicadas sobre o *benchmark Dijkstra*. A taxa de correção de falhas aumentou em 12% para *Dijkstra* transformado por *block-placement*, enquanto em outros casos a taxa de falhas corrigidas reduziu, como por

exemplo, 51% para a versão de *Dijkstra* compilada com a transformação *loop-rotate*.

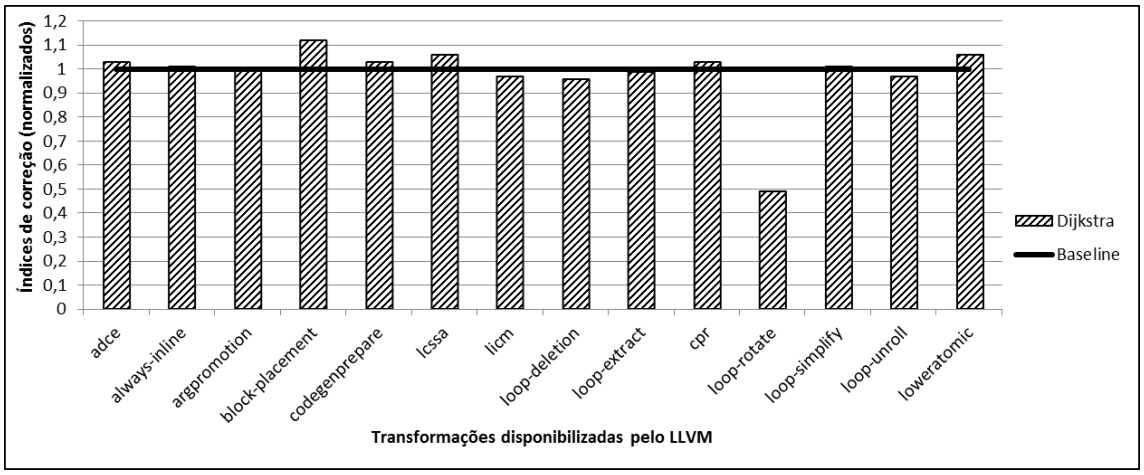

Figura 5.8: Taxa de correção de falhas de algumas transformações aplicadas em Dijkstra.

A Tabela 5.3 contém a quantidade de blocos básicos contidos em cada programa após as transformações serem aplicadas. Em destaque nessa tabela aparecem os valores correspondentes aos *corner cases* de correção de falhas.

No artigo original sobre ACCE (VEMU; GURUMURTHY; ABRAHAM, 2007) é dito que o aumento do número de blocos levaria a um aumento na correção de falhas. Entretanto os dados da Tabela 5.2 e da Tabela 5.3 (juntamente com informações de número de blocos do cenário baseline) mostra que não há indícios que comprovem essa afirmação. Por exemplo, o programa *basicmath* possui 449 blocos no baseline e 513 após a aplicação de loop rotate seguida de ACCE. Houve portanto um aumento do número de blocos mas uma redução 47% de correção em relação ao baseline. Já a versão de *basicmath* compilada com a transformação *loop-extract*, resultou em 665 blocos básicos, porém com uma taxa de correção de falhas muito próxima ao valor base, com uma pequena redução de 2%. E *loop-extract-single* quando aplicada sobre *Susan* aumentou consideravelmente o número de blocos básicos, de 4159 no baseline para 6228 e, mesmo assim, a correção de falhas teve um aumento ínfimo de 3%.

Para elucidar, de forma genérica, os fatores que impactaram na correção de falhas realizada por ACCE, o *corner case basicmath* transformado com *loop-rotate* que apresentou redução de 47% é analisado em detalhes na próxima seção.

5.3.1.1 Basicmath transformado por loop-rotate

Loop-rotate é uma transformação que atua sobre os laços de iteração do programa, inserindo cópia da expressão condicional do laço ao final ao do mesmo para verificar a necessidade de reexecutá-lo. Ela transforma laços "for"e "while"em "do-while", ou seja, os laços de pré-teste passam a assumir também a forma de pós-teste.

Os efeitos de *loop-rotate* em um laço de iteração são exemplificados através do trecho de código que representa um laço *for*, mostrado na Listagem 5.1, cujo CFG é dado pela Figura 5.9.

Após a transformação *loop-rotate* ter sido aplicada, o CFG resultante apresentado na Figura 5.10, mostra que houve a inserção de blocos básicos extras e de uma nova instrução de teste para a condição do laço. Os símbolos "[A]"e "[B]"representam trechos de códigos anteriores e posteriores à ocorrência do laço *for* no programa, respectivamente, úteis para caracterizar o início e final do laço junto ao grafo de fluxo de controle.

Tabela 5.3: Número de blocos básicos correspondente a cada *benchmark* para cada uma das transformações disponibilizadas pelo LLVM

	benchmarks									
	Basicmath	*Bitcount*	*Quicksort*	*Dijkstra*	*Patricia*	*String search*	*Susan*	*Rijndael*	*Fft*	*Crc32*
adce	449	603	144	378	1032	945	4159	9623	568	178
always-inline	449	603	144	378	1032	945	4159	9630	568	178
argpromotion	449	603	144	378	1032	945	4159	9630	568	178
block-placement	449	603	144	378	1032	945	4158	9630	568	178
break-crit-edges	449	709	157	428	1176	1087	5477	10703	593	184
codegenprepare	384	596	144	358	1032	897	4018	9528	554	172
constmerge	449	603	144	378	1032	945	4159	9630	568	178
constprop	449	603	144	378	1032	945	4159	9630	568	178
dce	449	603	144	378	1032	945	4159	9623	568	178
deadargelim	449	603	144	378	1032	945	4159	9630	568	178
deadtypeelim	449	603	144	378	1032	945	4159	9630	568	178
die	449	603	144	378	1032	945	4159	9623	568	178
dse	449	603	144	378	1032	945	4159	9630	568	178
functionattrs	449	603	144	378	1032	945	4159	9630	568	178
globaldce	449	603	144	378	992	945	4159	9630	568	178
globalopt	449	603	144	378	1032	945	4159	9631	568	178
gvn	449	604	143	393	1134	1017	5351	10795	551	156
indvars	449	624	144	384	1039	969	4179	9707	575	178
inline	449	603	144	392	1012	945	4159	9630	661	178
instcombine	449	709	157	428	1176	1087	5477	9570	593	184
internalize	449	603	144	378	1032	945	4159	9630	568	178
ipconstprop	449	603	144	378	1032	945	4159	9630	568	178
ipsccp	449	603	144	373	1000	945	4154	9623	563	178
jump-threading	449	604	143	388	1129	1017	5311	10629	546	156
lcssa	449	603	144	378	1032	945	4159	9630	568	178
licm	449	624	144	384	1039	969	5484	9719	593	184
loop-deletion	449	624	144	384	1039	969	4179	9707	575	178
loop-extract	665	896	217	563	1488	1589	6228	12001	827	265
loop-extract-single	476	743	184	455	1209	1142	5522	10829	619	211
loop-reduce	449	624	144	384	1039	969	4179	9707	575	178
loop-rotate	513	656	152	440	1055	1065	4499	9928	671	202
loop-simplify	449	624	144	384	1039	969	4179	9707	575	178
loop-unroll	449	624	144	384	1039	969	4179	9707	575	178
loop-unswitch	449	624	144	384	1039	969	4179	9707	575	178
loweratomic	449	603	144	378	1032	945	4159	9630	568	178
lowerinvoke	449	603	144	378	1032	945	4159	9630	568	178
lowerswitch	449	603	144	378	1032	945	5725	11358	568	178
mem2reg	449	709	157	428	1176	1087	5477	10557	593	184
memcpyopt	449	603	144	378	1032	945	4159	9630	568	178
mergefunc	449	603	144	378	1032	945	4159	9630	568	178
mergereturn	449	603	144	378	1039	945	4187	9630	575	178
partial-inliner	449	603	144	378	1032	945	4159	9630	568	178
prune-eh	449	603	144	378	1032	945	4159	9630	568	178
reassociate	449	603	144	378	1032	945	4159	9631	568	178
reg2mem	449	709	157	428	1176	1087	5477	10710	593	184
scalarrepl	449	709	157	428	1176	1087	5477	9641	593	184
sccp	449	603	144	378	1032	945	4159	9623	568	178
simplifycfg	449	591	143	388	1129	1017	5243	10240	546	156
simplify-libcalls	449	603	144	378	1032	945	4159	9630	568	178
sink	449	709	157	428	1176	1087	5477	11589	593	184
sretpromotion	449	603	144	378	1032	945	4159	9630	568	178
strip	449	603	144	378	1032	945	4159	9630	568	178
strip-dead-debug-info	449	603	144	378	1032	945	4159	9630	568	178
strip-dead-prototypes	449	603	144	378	1032	945	4159	9630	568	178
strip-debug-declare	449	603	144	378	1032	945	4159	9630	568	178
strip-nondebug	449	603	144	378	1032	945	4159	9630	568	178
tailcallelim	449	603	144	378	1032	945	4159	9630	568	178
tailduplicate	449	603	144	378	1032	945	4146	9630	593	178
Baseline	449	603	144	378	1032	945	4159	9630	568	178

Os blocos extras inseridos pela transformação, segundo a documentação do LLVM, servem para separar as instruções que são só utilizadas em blocos sucessores, em um processo de retardamento da execução das instruções cujos resultados são importantes apenas posteriormente. Tais blocos contém apenas o label de identificação e a instrução de desvio, que chamamos de *label-br*.

Listagem 5.1: Exemplo de um laço for em linguagem C

```
[A]
for (i=1;i<=10;i++){
    ...
}
[B]
```

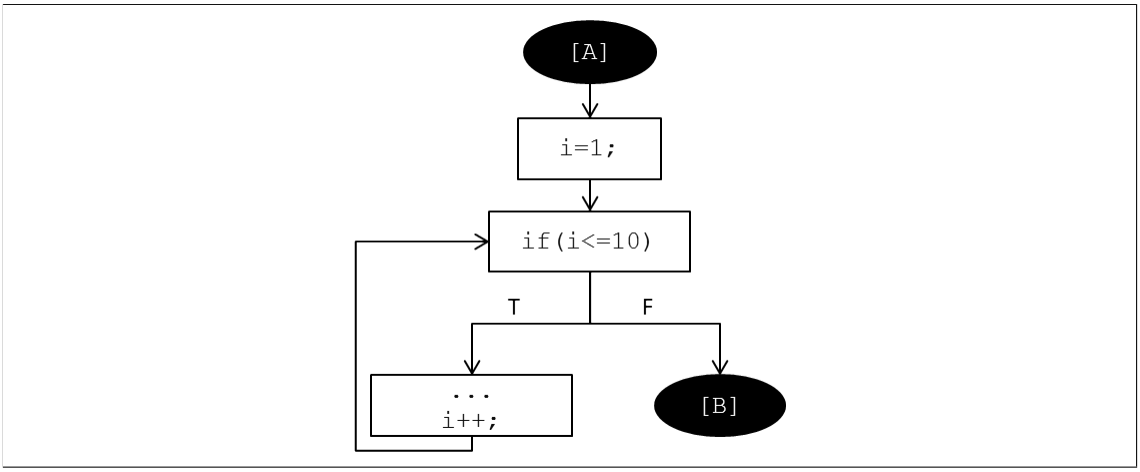

Figura 5.9: CFG correspondente ao código da Listagem 5.1

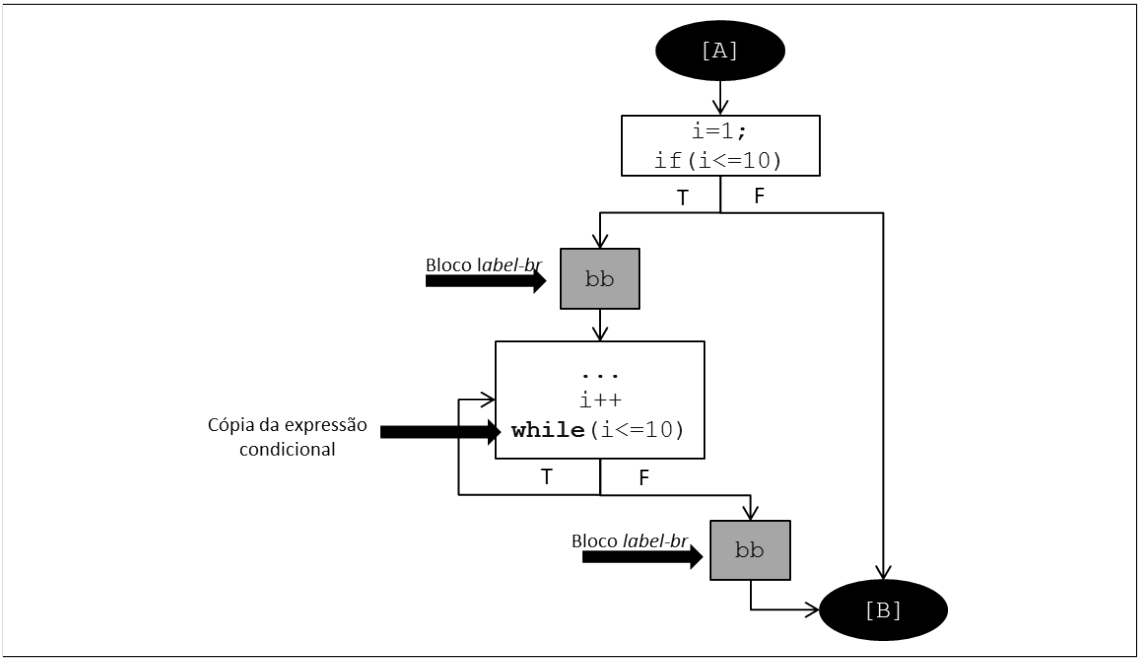

Figura 5.10: CFG correspondente ao código da Listagem 5.1 transformado com *Loop-rotate*

A Listagem 5.2 mostra um trecho do programa *basicmath* com laço aninhado. No caso do programa *basicmath*, as transformações realizadas por *loop-rotate* são exemplificadas através das Figuras 5.11 e 5.12, que mostram o CFG da função *main* antes e após à aplicação da transformação *loop-rotate*, respectivamente.

Listagem 5.2: trecho do programa basicmath com laçco aninhado.

```
for(a1=1;a1<10;a1++) {
  for(b1=10;b1>0;b1--) {
    for(c1=5;c1<15;c1+=0.5) {
      for(d1=-1;d1>-11;d1--) {
        SolveCubic(a1, b1, c1, d1, &solutions, x);
          printf("Solutions:");
        for(i=0;i<solutions;i++)
              printf(" %f",x[i]);
           printf("\n");
      }
    }
  }
}
```

O bloco básico *bb15* do CFG da Figura 5.11 está em destaque por representar blocos básicos em laços aninhados pois, a partir dele, os demais blocos básicos pertencem aos laços, como por exemplo, bb16, bb28.

Na Figura 5.12 são mostrados apenas os blocos básicos que correspondem aos laços aninhados da função *main* após a aplicação de *loop-rotate* e, de maneira similar ao CFG original da função *main*, o bloco básico *bb15* está em destaque, com o objetivo de apontar o começo dos laços.

Como pode ser percebido, houve a criação de blocos básicos extras do tipo *label-br* com a aplicação da transformação *loop-rotate* em *basicmath*. Os blocos *label-br* são os blocos com rótulos cujos nomes terminam por “label-br”.

O fato das transformações disponíveis no LLVM gerarem novos blocos básicos com a estrutura de *label-br* irá resultar em programas que, ao serem transformados pela técnica ACCE, terão blocos para a proteção de erros de fluxo de controle, os chamados *blocos protetores*, com um número maior de instruções do que os blocos originais do programa, nominados de *blocos protegidos*. Esse fator aumenta a probabilidade de instruções adicionadas pela técnica ACCE serem afetadas pelas falhas, resultando em falhas não corrigidas, ou seja, falhas em que o fluxo de controle não pode ser reestabelecido. E se estes blocos *label-br* estiverem localizados em laços com vários níveis de aninhamento, fará com que as instruções de ACCE, em maior número que as originais do programa, tenham ainda mais chance de serem afetadas pelas falhas, visto que estas instruções serão executadas várias vezes enquanto o programa estiver rodando.

Se analisarmos a aplicação de *loop-rotate* sobre outro *benchmark*, tal como o *bitcount*, representado pelo CFG na Figura 5.13, podemos perceber que, apesar da criação de *blocos label-br*, estes **não** estão localizados em laços, o que reduz a quantidade de instruções de ACCE executadas, reduzindo a probabilidade das instruções da técnica serem as afetadas pelas falhas, justificando os índices de correção de falhas próximos ao *baseline*.

Por outro lado, para os casos em que houve aumento na correção de falhas (ver Tabela 5.2), oriundos da aplicação das transformações *ipconstprop*, *instcombine* e *loop-extract* para os programas *bitcount*, *fft* e *string search*, respectivamente, identificou-se que houve redução no número de blocos *label-br* ou a sua ocorrência fora dos laços de repetição,

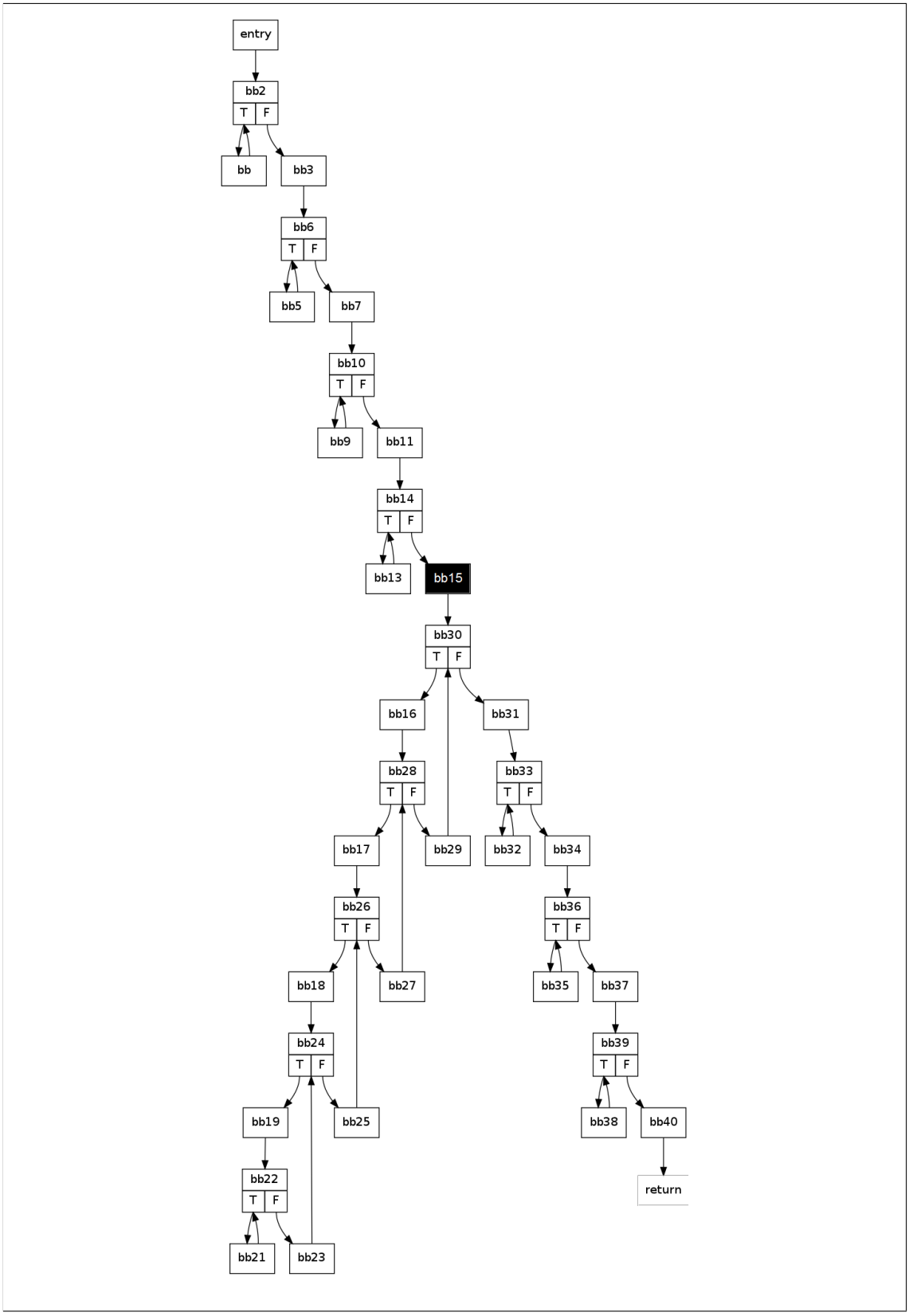

Figura 5.11: CFG da função *main* do programa *basicmath* antes da aplicação de *loop-rotate*.

fatores que aumentam a probabilidade de falhas acontecerem em instruções originais do programa, ou seja, instruções consideradas protegidas pela técnica, o que resulta em mais

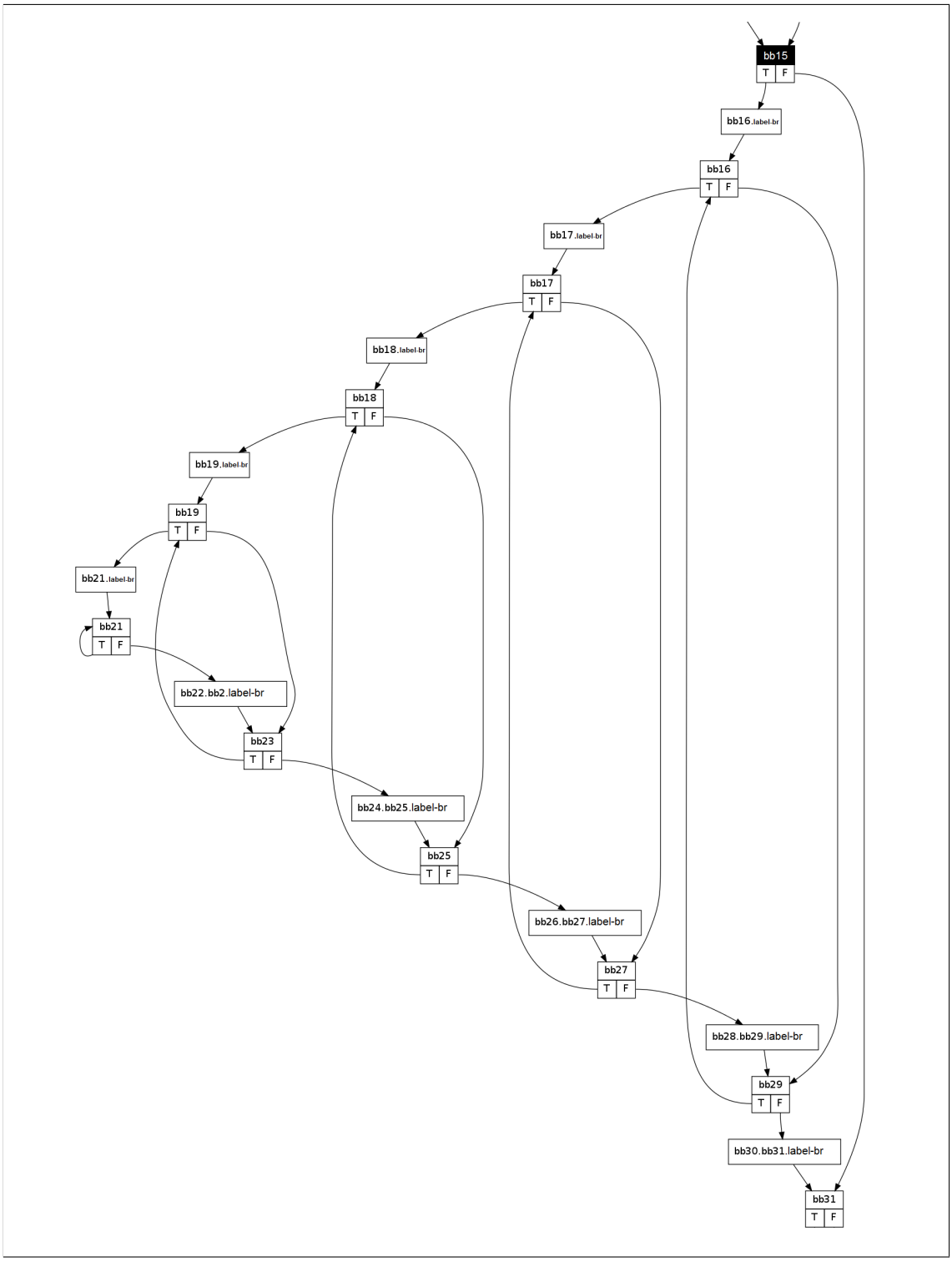

Figura 5.12: CFG da função *main* após a aplicação da transformação *loop-rotate* em *basicmath*

falhas corrigidas.

Para o *corner case fft* compilado com a transformação *instcombine*, o número de blocos básicos reduziu *em relação às transformações que resultam na redução da taxa de correção de falhas*. Esse fato é exemplificado pela Figura 5.14, que mostra o CFG da função *main* de *fft* depois da aplicação da transformação *instcombine*. Como pode-se perceber pela Figura 5.14, o bloco *label-br* presente está localizado fora do laço de repetição,

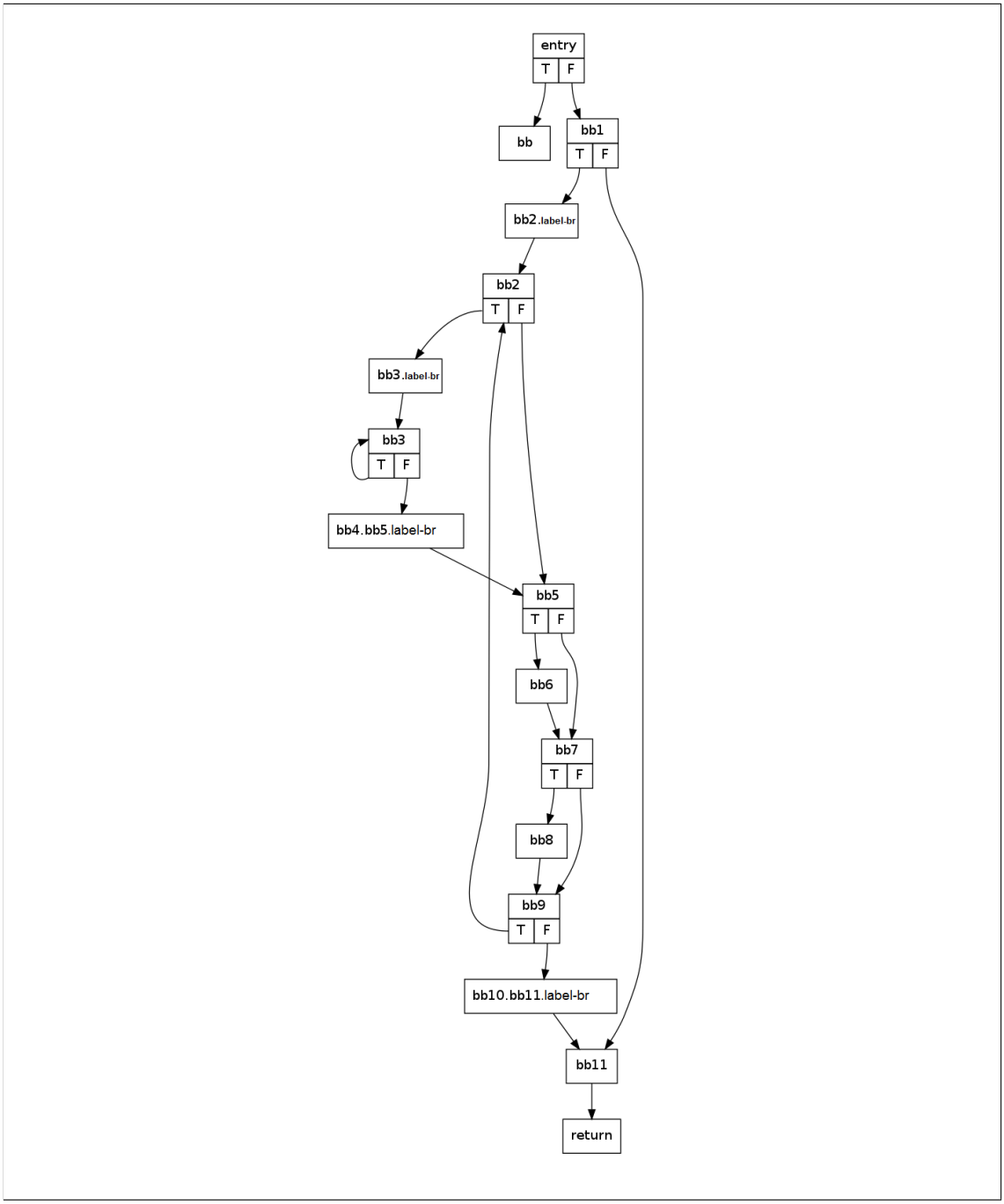

Figura 5.13: CFG da função *main* de *bitcount* após a aplicação da transformação *loop-rotate*

característica que reduz a chance de instruções ACCE serem atingidas por falhas transientes, aumentando a eficácia de correção, que antes era de 68,2% e passou a ser de 77% com aplicação de *instcombine*.

5.3.2 Aplicação de Combinações de Transformações seguidas por ACCE

No gráfico da Figura 5.15, são mostrados os valores resultantes da aplicação das combinações de transformações. Observou-se que as taxas de correção resultaram em taxas de correção de falhas próximos às obtidas para os programas compilados sem nenhuma transformação (ver Figura 5.5). São apresentados os valores correspondentes às médias

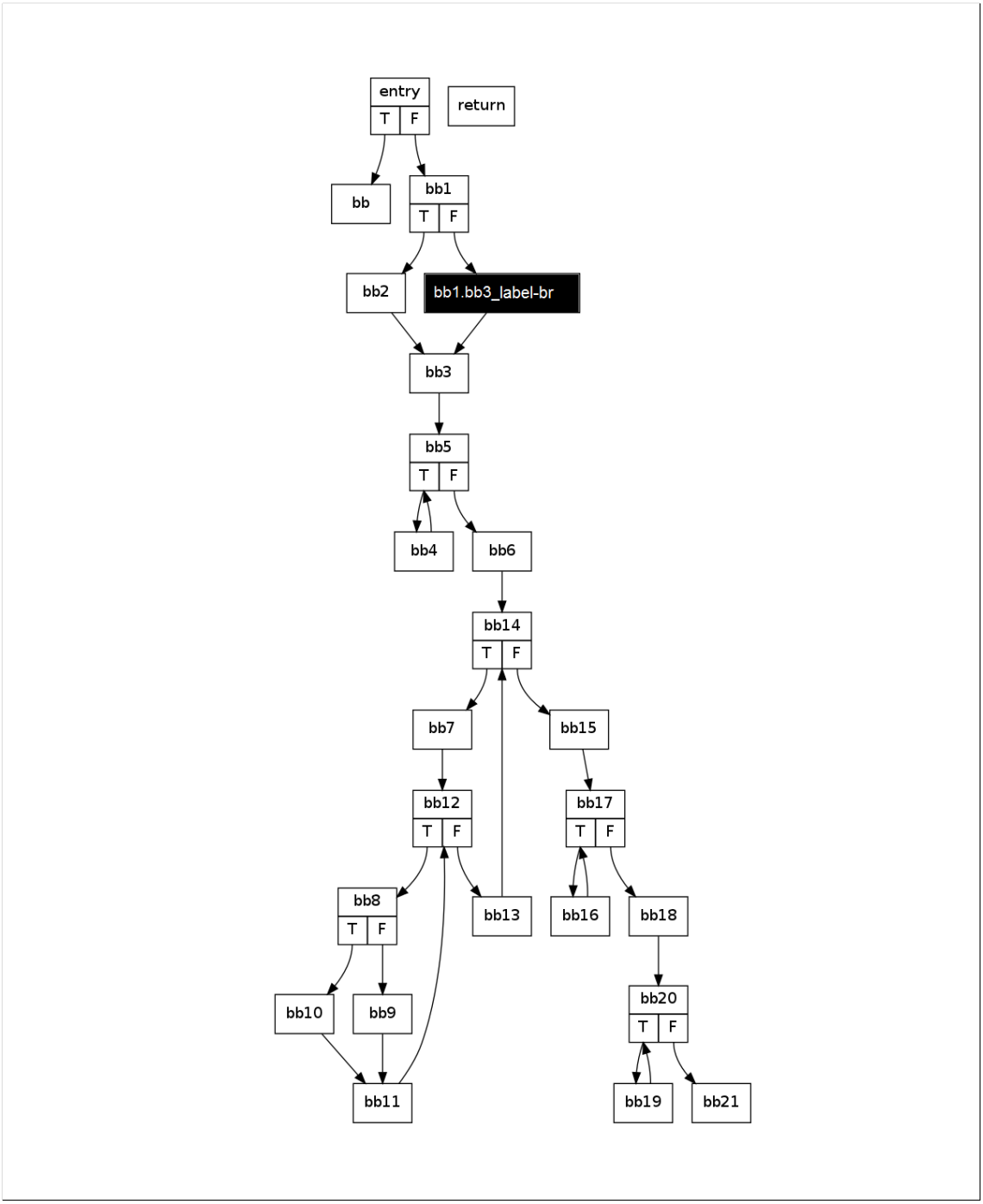

Figura 5.14: CFG da função *main* de *fft* compilado com *instcombine*

das combinações de transformações das sequências 10, 20, 30, 40, 50 e 58 transformações de cada *benchmark* testado. Na Figura 5.16 é apresentado o desvio padrão deste cenário dos experimentos.

A Tabela 5.4 contém a média das correções resultantes das combinações para cada benchmark, o valor do baseline e o valor normalizado a partir do baseline, mostrando que as diferenças são mínimas, sendo o maior valor 0,07% para o *basicmath*.

De acordo com os dados extraídos neste cenário dos experimentos, pode-se considerar a técnica ACCE eficaz quando aplicada em conjunto com diversas outras transformações de programas. Portanto, as transformações que levaram a *corner cases* no cenário ante-

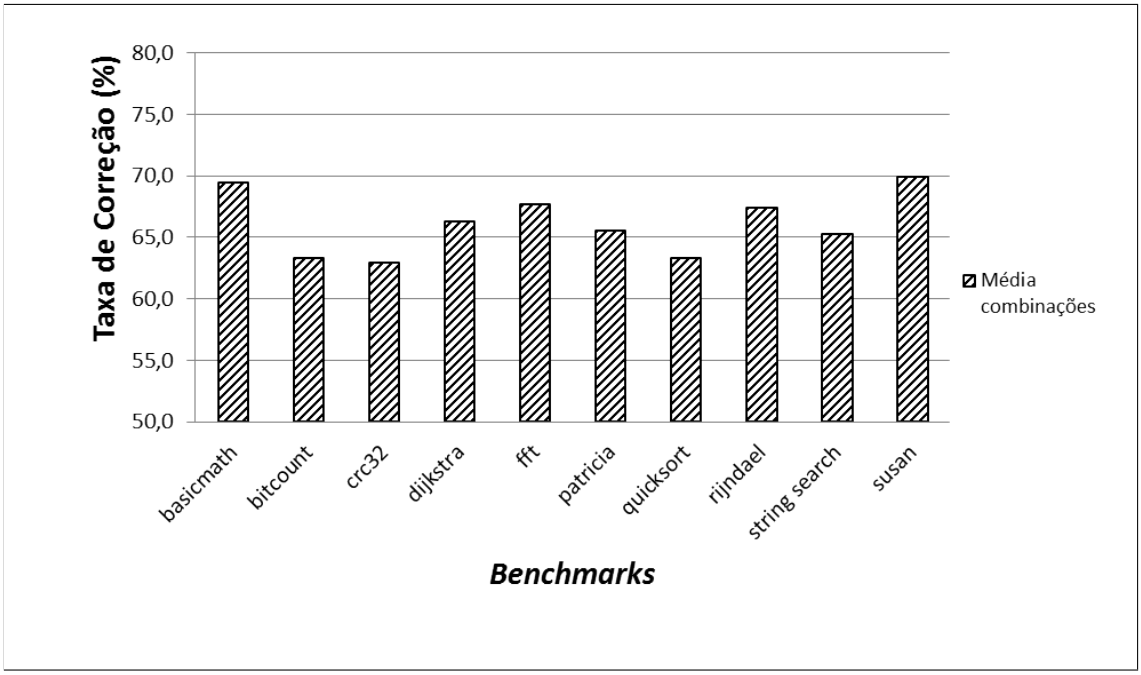

Figura 5.15: Taxas de correção de ACCE no cenário de transformações combinadas.

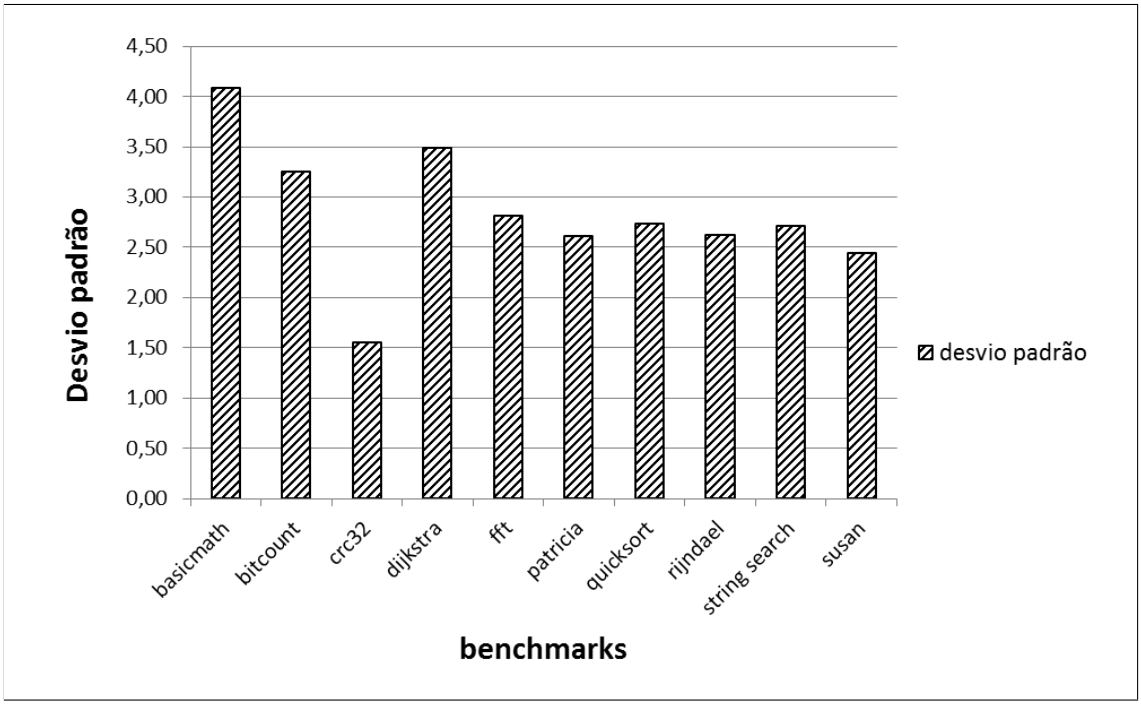

Figura 5.16: Desvio padrão no cenário de transformações combinadas.

rior dos experimentos, quando usadas em conjunto com outras, não impactaram negativamente em ACCE.

As análises das transformações e possibilidades de combinações que resultem em melhores resultados em termos de correção de falhas são trabalhos futuros a serem realizados.

Tabela 5.4: Valores da correção de falhas resultante da aplicação das combinações de transformações nos *benchmarks* comparados com o *baseline*

	correção média das combinações (%)	Baseline (%)	Normalizado
basicmath	69,5	74,8	0,93
bitcount	63,4	65,1	0,97
crc32	63,0	63,3	1,00
dijkstra	66,3	65,9	1,01
fft	67,7	68,2	0,99
patricia	65,6	70,2	0,93
quicksort	63,3	64,1	0,99
rijndael	67,5	67,1	1,01
string search	65,3	64,9	1,01
susan	70,0	70,9	0,99

5.3.3 Avaliação de ACCE em termos de Consumo Energético e Desempenho

Neste cenário dos experimentos foram avaliados o consumo de energia e o *overhead* de desempenho dos programas transformados individualmente com as otimizações disponibilizadas pelo LLVM, seguidas por ACCE. Para obter os dados nesse cenário, foi utilizado um simulador de energia (RUTZIG; BECK; CARRO, 2009), que trabalha com base na plataforma de simulação Simics (MAGNUSSON, et al). Simics fornece virtualização de hardware que permite a simulação e extração de consumo de energia e desempenho de aplicações de software.

Nesse cenário foram utilizados três *benchmarks*: *crc32, quicksort, string search* compilados com cada uma das 58 otimizações do LLVM. A Tabela 5.5 mostra os resultados normalizados para consumo de energia, onde o valor base da normalização é a aplicação compilada somente com ACCE. Nesse caso, os valores menores do que 1 representam uma economia de energia em relação ao valor base, enquanto os valores maiores que 1 representam um consumo acima do consumo base. Apenas são apresentados as transformações que apresentaram aumento ou diminuição no consumo de energia, sendo que os demais mantiveram os mesmos índices.

Com base nos experimentos e resultados, a transformação de programa provida pelo LLVM que mais influenciou em economia de energia foi *functionattrs* para o programa *crc32*, obtendo uma redução de 67% no consumo energético. Quanto ao aumento de consumo, *dce* para o programa *string search*, foi a otimização que apresentou um maior gasto de energia, com um aumento de 33%.

Na Tabela 5.6 são mostrados os resultados quanto ao tempo de execução para as otimizações, normalizados em relação ao resultado obtido somente com a aplicação de ACCE. Esses valores, assim como as medições de energia, foram obtidos com o Simulador Simics. Os valores menores que 1 representam ganho de desempenho, enquanto os valores maiores de 1 indicam a ocorrência de um *overhead* de desempenho. Similarmente à Tabela 5.5, apenas são apresentados os valores com impactos significativos no tempo de execução.

Os valores da Tabela 5.6 mostram que algumas otimizações proporcionaram uma grande redução no tempo necessário para a execução dos programas, chegando a 72% no caso de *functionattrs* e 59% no caso de *dse*, ambas otimizações aplicadas no programa *crc32*. Quanto ao *overhead* de tempo de execução, nenhuma transformação de programa realizada pelo LLVM resultou em grandes alterações, sendo que o maior *overhead* foi de

Tabela 5.5: Avaliação de energia de três programas transformados com otimizações fornecidas pelo LLVM, seguidas por ACCE.

Benchmark	Transformação	Comparação com o valor base
crc32	*Block-placement*	1,01
	dse	0,46
	functionattrs	0,53
	gvn	0,87
	Loop-rotate	0,83
	mem2reg	1,06
	reg2mem	0,96
	scalarrepl	1,06
	sink	1,03
quicksort	*codegenprepare*	0,81
	gvn	0,99
	Jump-threading	0,99
	simplifycfg	0,99
string search	*dce*	1,33
	strip-dead-debug-info	1,02
	Strip-dead-prototypes	1,02

Tabela 5.6: Avaliação do tempo de execução de três programas otimizados pelas transformações fornecidas pelo LLVM e ACCE.

Benchmark	Transformação	Comparação com o valor base
crc32	*dse*	0,41
	functionattrs	0,68
	gvn	0,85
	loop-reduce	0,81
	lowerswitch	0,87
	reg2mem	0,97
	scalarrepl	0,87
	sink	1,03
quicksort	*codegenprepare*	0,81
	gvn	0,99
	Jump-threading	0,99
	simplifycfg	0,99
string search	*strip-dead-debug-info*	1,02
	Strip-dead-prototypes	1,02

3% para *sink* aplicada sobre *crc32*.

Para concluir o capítulo que descreve os experimentos realizados, vale mencionar que, embora as milhares de falhas tenham sido injetadas de forma randômica, isso não dá garantias sobre a qualidade amostral. Os percentuais de detecção e correção de CFEs obtidos seriam mais significativos se as falhas injetadas fossem classificadas como erradas, ilegais inter-blocos e intra-blocos, trabalho futuro a ser realizado. Para isso, pretende-se armazenar o local de injeção da falha para realizar, após a injeção, a contabilização dos locais de ocorrência, para estimar quantas das falhas que afetaram instruções da técnica e instruções do programa, foram corrigidas.

6 CONCLUSÕES E TRABALHOS FUTUROS

Com a evolução dos sistemas computacionais com base nos avanços tecnológicos obtidos recentemente, aumenta cada vez mais a necessidade de garantias de confiabilidade e de funcionamento correto desses sistemas. Porém, devido ao fato da exposição dos computadores à radiação e também pela diminuição no tamanho dos componentes de hardware que os compõem, estes sistemas estão se tornando ainda mais susceptíveis à ocorrência de falhas, tal como *bit-flip*, que podem causar danos gigantescos dependendo dos efeitos produzidos e local da ocorrência das falhas.

Nesse cenário, diversas são as técnicas que realizam tolerância a falhas, sendo que algumas são baseadas em hardware, enquanto outras combatem as falhas de hardware em nível de software. Assim sendo, devido ao fato da popularização dos sistemas embarcados, as técnicas de software, que não agridem os componentes físicos, ao contrário das de hardware, tornam-se atrativas.

Com base nesse cenário, o objetivo deste trabalho é avaliar o impacto de transformações de programas na eficácia de técnica de software ACCE para a tolerância a falhas que se manifestam como erros de fluxo de controle.

A técnica ACCE foi descrita em maiores detalhes e escolhida como objeto de pesquisa por detectar e corrigir desvios ilegais ocorridos no fluxo de controle durante a execução do software. Ainda no capítulo de apresentação de ACCE, foram descritos detalhes de como a técnica foi implementada no compilador LLVM como um passo de transformação de programas, que insere instruções em nível de linguagem intermediária.

A avaliação da influência da aplicação de ACCE em programas transformados com otimizações disponibilizadas pelo LLVM focaram nos índices de correção de erros de fluxo de controle com base em dez *benchmarks* da suíte Mibench (*Basicmath*, *Bitcount*, *CRC32*, *Dijkstra*, *FFT*, *Patricia*, *Qsort*, *Rijndael*, *Stringsearch* e *Susan*).

Os *benchmarks* foram submetidos a um processo de injeção de 880 K falhas para simular criação de desvios ilegais, remoção de desvios e alteração no destino de desvios. As falhas foram injetadas através da ferramenta de depuração de programas disponível para ambiente Linux, o GDB, que possibilitou a extração do rastro de execução do programa e a alteração do PC de acordo com o modelo de falha a ser injetado.

Os experimentos se deram com a implementação da técnica ACCE como um passo de transformação no LLVM, infraestrutura de compilação amplamente modular que têm conquistado espaço em pesquisas acadêmicas e também na indústria. O LLVM foi utilizado como ferramenta de apoio nesse trabalho por disponibilizar 60 passos de transformação de programas.

Metodologicamente, os experimentos foram divididos em cenários, entre eles: aplicação individual de cada transformação disponível no LLVM seguida pela aplicação da técnica ACCE para avaliar a cobertura de falhas, e aplicação de sequências de trans-

formações aleatoriamente selecionadas, sequências variando de 10 a 58 transformações, metodologia similar a (FURSIN et al., 2008), com ACCE aplicada na sequência, com o objetivo de avaliar o comportamento da técnica em termos de correção de falhas em programas amplamente otimizados.

Através dos experimentos realizados descobriu-se que a técnica ACCE é não aplicável em programas que possuem em sua representação intermediária (LLVM-IR) instruções conhecidas como ϕ, que são usadas em linguagens representadas na forma SSA para a atribuição correta de valores aos registrados de acordo com o fluxo de controle tomado pelo programa. Instruções ϕ, quando presentes em programas LLVM-IR, devem estar posicionadas, obrigatoriamente, no início dos blocos básicos, isto é, não pode haver instruções não-ϕ entre o *label* do bloco e a primeira instrução ϕ. Essa característica da linguagem intermediária do LLVM incapacita a aplicação de ACCE pois ela necessita inserir instruções no topo do bloco básico. Portanto, para programas em que foram geradas instruções do tipo ϕ, o passo de transformação $reg2mem$ foi aplicado como transformação precedente a ACCE, transformação que desfaz a forma SSA removendo as ϕ, permitindo a inserção das instruções por ACCE.

Por meio de várias baterias de injeção de falhas, foram obtidos resultados que mostraram que os índices de correção de falhas quando da transformação do programa, seja de forma individual, ou com a combinação de diferentes sequências de passos de otimização, não são relacionados com a quantidade de blocos básicos presentes no programa após o processo de compilação. Além disso, os resultados mostraram que uma mesma transformação compromete a eficácia de ACCE de forma distinta para cada *benchmark*, como também, para um mesmo *benchmark*, cada transformação acarreta em uma taxa particular de cobertura de falhas, determinando que a correção de falhas é altamente relacionada com a estrutura do programa, composta por características que vão além do número de blocos básicos do programa, tais como o número de instruções de um bloco básico e a localização dos blocos com poucas instruções em laços aninhados.

Em busca das razões que levaram ACCE a apresentar redução/aumento na cobertura de falhas, análises mais detalhadas foram realizadas em um subconjunto de casos considerados *corner cases* pois exibiram uma taxa de redução/aumento de cobertura com uma diferença considerável em relação aos casos baseline. Para explicar os motivos das divergências, tais *corner cases* foram submetidos a comparações com outros experimentos que tiveram resultados semelhantes aos do *baseline*, para apontar as diferenças e os fatores da perda/ganho de eficácia de ACCE.

Dessa forma constatou-se que a causa de algumas reduções significativas nos valores de correção de falhas não está relacionada com o número de blocos básicos que formam o programa transformado, e sim com a sua estrutura. Descobrimos que muitas das transformações disponíveis no LLVM criam novos blocos básicos contendo apenas o label de identificação do bloco e uma instrução de desvio.

Após a aplicação da transformação ACCE para cada um desses blocos, chamados de bloco *label-br*, são criados 2 novos blocos básicos protetores. Quando há muitos desses blocos (os *label-br* e seus blocos protetores) dentro de laços aninhados, aumenta a probabilidade de que falhas possam atingir áreas não protegidas.

Programadores devem tomar cuidado, portanto, com programas que possuam laços aninhados, pois a aplicação de determinadas transformações sobre esses programas seguida da aplicação de ACCE pode reduzir a cobertura de falhas.

Apesar das diversas análises e experimentos realizados, outros testes e avaliações seriam interessantes, no sentido de identificar os fatores e/ou alternativas que possibilitem

a obtenção de melhores índices de cobertura de falhas. Neste ambiente, alguns trabalhos futuros a serem desenvolvidos são:

- Estudo e análise de estrutura de programas que garanta aumento na taxa de correção de falhas;
- Avaliação do número mínimo de instruções necessárias em um bloco básico para que este não afete a correção de falhas, aspecto similar aos *label-br*.
- Busca de mecanismos que permitam a criação de sequências de transformações que possam aumentar a correção de falhas e ainda melhorar os indicadores de desempenho e consumo energético.
- Realização de experimentos com injeção dirigida de falhas somente em instruções originais do programa.

REFERÊNCIAS

ABADI; BUDIU; ERLINGSSON; LIGATTI. **Control-flow integrity principles, implementations, and applications.** ACM TRANSACTIONS INFORMATION SYSTEMS SECRITY, v.13, New York, USA, 2009, p.4:1–4:40, 2009.

AVIZIENIS; LAPRIE; RANDELL; LANDWEHR. **Basic Concepts and Taxonomy of Dependable and Secure Computing**. IEEE TRANSACTIONS ON DEPENSABLE SECURITY COMPUTING. V.1, Los Alamitos, CA, USA, 2004, p.11–33, 2004.

BERGAOUI, S.; LEVEUGLE, R. **Impact of Software Optimization on Variable Lifetimes in a Microprocessor-Based System.** In: VI IEEE INTERNATIONAL SYMPOSIUM ON ELETRONIC DESIGN, TEST AND APPLICATION,6., DELTA 2011, Queenstown, New Zealand, 2011. Proceedings..., Washington, USA: IEEE, p.56?61, 2011.

BORIN; WANG; WU; ARAUJO. **Dynamic binary control-flow errors detection.** SIGARCH COMPUTING ARCHITECT NEWS, 33., New York, NY, USA, 2005, ACM, p.15–20, 2005.

DUBACH; JONES; BONILLA; FURSIN; O'BOYLE. **Portable compiler optimisation across embedded programs and microarchitectures using machine learning.** In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 42., MICRO 42, NY, USA. Proceedings... ACM, 2009. p.78–88, 2009.

DUBACH, C.; JONES, T. M.; O'BOYLE, M. F. **Exploring and predicting the architecture/optimising compiler co-design space.** IN: COMPILERS, ARCCOMPILERS, ARCHITECTURES AND SYNTHESIS FOR EMBEDDED SYSTEMSL. (CASES '08), New York, NY, USA, 2008.,**Proceedings...** ACM, p.31–40. 2008.

FARAZMAND, N.; FAZELI, M.; MIREMADI, S. G. **FEDC: control flow error detection and correction for embedded systems without program interruption.** IN: THIRD INTERNATIONAL CONFERENCE ON AVAILABILITY, RELIABILITY AND SECURITY. (ARES '08), Washington, DC, USA. **Proceedings...** IEEE Computer Society, p.33?38, 2008.

FENG; GUPTA; ANSARI; MAHLKE. **Shoestring: probabilistic soft error reliability on the cheap**. IN: ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS.(ASPLOS '10), New York, NY, USA. **Proceedings...** ACM,p.385?396, 2010.

FURSIN; MIRANDA; TEMAM; NAMOLARU; YOM-TOV; ZAKS; MENDELSON; BONILLA; THOMSON; LEATHER; WILLIAMS; O'BOYLE; BARNARD; ASHTON;

COURTOIS; BODIN. **MILEPOST GCC: Machine Learning Based Research Compiler**. In: GCC DEVELOPERS' SUMMIT. 2008, **Proceedings...** [S.l.: s.n.], 2008.

GOLOUBEVA; REBAUDENGO; REORDA; VIOLANTE. **Soft-Error Detection Using Control Flow Assertions.** IN: DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS. **Anais...** [S.l.: s.n.], p.581?588, 2003.

GUTHAUS; RINGENBERG; ERNST; AUSTIN; MUDGE; BROWN. **MiBench : a free, commercially representative embedded benchmark suite** *IN:* PROCEEDINGS OF THE WORKLOAD CHARACTERIZATION, 2001. WWC-4. IEEE International Workshop, pages 3–14, Washington, DC, USA. IEEE Computer Society, 2001.

HOSTE, K.; EECKHOUT, L. COLE: Compiler Optimization Level Exploration. IN: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION, 6., (CGO '08), New York, NY, USA. **Proceedings...** ACM, 2008. p.165–174., 2008.

JEONG, K.; KAHNG, A. B. **A Power-Constrained MPU Roadmap for the International Technology Roadmap for Semiconductors (ITRS)**. 2009.

KOROBEYNIKOV, A. **Improving Switch Lowering for The LLVM Compiler System**. PROC. OF THE 2007 SPRING YOUNG RESEARCHERS COLLOQUIUM ON SOFTWARE ENGINEERING, SYRCONSE?07,2007.

LATTNER, C. **The LLVM Compiler Infrastructure**. Online. Disponível em <www.llvm.org>. Acesso em 22/12/2012.

LATTNER, C.**LLVM Programmer's Manual**. Online. Disponível em <http://llvm.org/docs/ProgrammersManual.html>. Acesso em 21/02/2013.

LATTNER, C.; ADVE, V. **The LLVM Instruction Set and Compilation Strategy**. [S.l.: s.n.], 2002.

LATTNER, C.; ADVE, V. **LLVM: An Infrastructure for Multi-stage Optimization**. 2002. 68p. Tese (Doutorado em Ciência da Computação) — University of Illinois. 2002.

LEE, J.; SHRIVASTAVA, A. **Static analysis to mitigate soft errors in register files**. In: Conference on Design, Automation and Test in Europe, 3001 Leuven, Belgium. **Proceedings...** European Design and Automation Association, 2009. p.1367–1372. (DATE '09).

LYONS, R. E.; VANDERKULK, W. **The Use of Triple-Modular Redundancy to Improve Computer Reliability**. **IBM JOURNAL**, [S.l.], n.April, 1962.

MOHANRAM, K.; TOUBA, N. **Cost-effective approach for reducing soft error failure rate in logic circuits**. IN: INTERNATIONAL TEST CONFERENCE. PROCEEDINGS. ITC 2003, [S.l.], v.1, p.893–901, 2003.

OH, N.; SHIRVANI, P.; MCCLUSKEY, E. **Control-flow Checking by Software Signatures**. In: IEEE Transactions on Reliability, [S.l.], v.51, n.2, p.111–122, 2002.

OLGA, M.; GOLOUBEVA, S.; MAURIZIA, R.; REBAUDENGO, M.. **SOFTWARE-IMPLEMENTED HARDWARE FAULT TOLERANCE**. 1.ed. Torino, Italy: Springler, 238p., 2006.

PAN, Z.; EIGENMANN, R. **Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning.** IN: IEEE INTERNATIONAL SYMPOSIUM ON CODE GENEREATION AND OPTIMIZATION, Washington, DC, USA. (CGO '06), Proceedings on IEEE Computer Society, p.319–332., 2006.

MAGNUSSON et al., **Simics: A full system simulation platform**, *IN:* COMPUTER SOCIETY, vol.35, no.2, pp.50-58, 2002.

REBAUDENGO; REORDA; TORCHIANO; VIOLANTE. **Soft-error Detection through Software Fault-Tolerance techniques.** IN: IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS. [S.l.: s.n.], p.210?218, 1999.

REBAUDENGO; REORDA; TORCHIANO; VIOLANTE. **A Source-to-source Compiler for Generating Dependable Software**. IN: FIRST IEEE INTERNATIONAL WORKSHOP ON SOURCE CODE ANALYSIS AND MANIPULATION. 1th, Florence, Italy, Proceedings, 33 - 42, 2001.

SPENCER, R; HENRIKSEN, G. **LLVM's Analysis and Transform Passes**. Data de acesso: 12 abr. 2012. Disponível em : *llvm.org/docs/Passes.html*

RUTZIG, M. B; BECK, A.C.; CARRO, L. **Dynamically adapted low power ASIPs**. *IN*: RECONFIGURABLE COMPUTING: ARCHITECTURES, TOOLS AND APPLICATIONS, LNCS, vol. 5453, pp. 110–122. Springer–Verlag, 2009.

SINGER, J.; TJORTJIS, C.; WARD, M. **Using Software Metrics to Evaluate Static Single Assignment Form in GCC**.*IN:* PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON GCC RESEARCH OPPORTUNITIES (GROW), 2010.

Spec Corporation. **Standard Performance Evaluation Corporation (SPEC) website**. 2000.

VEMU, R.; ABRAHAM, J. A **CEDA: Control-flow error detection through assertions.***IN:* PROCEEDINGS OF THE 12TH IEEE INTERNATIONAL SYMPOSIUM ON ON-LINE TESTING, pages 151?158, 2006.

VEMU, R.; ABRAHAM, J. A. **CEDA: Control-flow Error Detection using Assertions.** IEEE TRANSACTIONS ON COMPUTERS, [S.l.], v.60, p.1233–1245, 2011.

VEMU, R.; GURUMURTHY, S.; ABRAHAM, J. A. **ACCE: Automatic Correction of Control-flow Errors.** IEEE INTERNATIONAL TEST CONFERENCE, ITC2007, [S.l.], p.1–10, 2007.

WAGNER, C.; MARGARIA, T.; PAGENDARM, H.-G. **Analysis and Code Model Extraction for C/C++ Source Code.** IN: IEEE INTERNATIONAL CONFERENCE ON ENGINEERING OF COMPLEX COMPUTER SYSTEMS. (ICECCS '09), Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p.110?119, 2009.

WEE, E.; MASTIPURAM, R. **Soft errors' impact on system reliability**. Cypress Semiconductor - September 30. 2004.

WOLFE, M. **Data Dependence and Program Restructuring.** JOURNAL OF SUPERCOMPUTERS, Hingham, MA, USA, v.4, n.4, p.321–344, Jan. 1991.

ZHAO; NAGARAKATTE; MARTIN; ZDANCEWIC. **Formalizing the LLVM intermediate representation for verified program transformations.** IN: ACM SIGPLAN-SIGACT SIMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 39., (POPL '12), New York, NY, USA. **Proceedings...** ACM, 2012. p.427–440. 2012.

ZHAO, M.; CHILDERS, B.; SOFFA, M. L. **Predicting the Impact of Optimizations for Embedded Systems**. IN: ACM SIGPLAN CONFERENCE ON LANGUAGES, COMPILERS AND TOOLS FOR EMBEDDED SYSTEMS, 2003. **Anais...** [S.l.: s.n.],. p.1?11, 2003.

ANEXO I - SINTAXE DA LINGUAGEM LLVM-IR

LLVM-IR é uma linguagem tipada, sofisticada e com diversas construções e recursos, representada na forma *Single Static Assignment*, formada por instruções com características de linguagens de baixo nível como código de três endereços. Este anexo tem como objetivo apresentar um subconjunto da sua sintaxe e algumas das suas características com destaque para aquelas consideradas mais relevantes para este trabalho. Para uma descrição completa da linguagem ver (LATTNER; ADVE, 2002a).

A Figura 6.1 contém a sintaxe do subconjunto de LLVM-IR conforme definido em (ZHAO et al., 2012).

Modules		::=	$\overline{layout}\ \overline{namedt}\ \overline{prod}$
Layouts	*layout*	::=	**bigendian** | **littleendian** | **ptr** $sz\ align_0\ align_1$ | **int** $sz\ align_0\ align_1$
		|	**float** $sz\ align_0\ align_1$ | **aggr** $sz\ align_0\ align_1$ | **stack** $sz\ align_0\ align_1$
Namedt	*namedt*	::=	%id = **type** { $\overline{arg}$}
Products	*prod*	::=	*id* = **global** *typ cnst align* | **define** *typ id*($\overline{arg}$) {$\overline{b}$} | **declare** *typ id*($\overline{arg}$)
Floats	*fp*	::=	**float** | **double**
Tipos	*typ*	::=	isz | *fp* | **void** | *typ** | [sz *x typ*] | {$\overline{typ_j}^j$} | *typ* $\overline{typ_j}^j$ | *id*
Valores	*val*	::=	*id* | *cnst*
Op. Binárias	*bop*	::=	**add** | **sub** | **mul** | **udiv** | **sdiv** | **urem** | **srem** | **shl** | **lshr** | **ashr** | and | **or** | **xor**
Op. Float	*fbop*	::=	**fadd** | **fsub** | **fmul** | **fdiv** | **frem**
Blocos	*b*	::=	$l\ \overline{\phi}\ \overline{c}\ tmn$
Blocos ϕ	ϕ	::=	i = **phi** *typ* $\overline{[val_j, l_j]}^j$
Tmns	*tmn*	::=	**br** *val* $l_1\ l_2$ | **br** l | **ret** *typ val* | **ret void** | **unreachable**
Comandos	*c*	::=	*id* = *bop*(**int** *sz*) $val_1\ val_2$ | *id* = *fbop fp* $val_1\ val_2$ | *id*=**load**(*typ**) $val_1\ align$
		|	**store** *typ* $val_1\ val_2\ align$ | *id* = **malloc** *typ* $val_1\ align$ | **free** *(typ*) val*
		|	*id*= **alloc** *typ* $val_1\ align$ | *id* = *trop* typ_1 *val* **to** typ_2 | *id*= *cop* typ_1 *val* **to** typ_2
		|	*id* = **icmp** *cond typ* $val_1\ val_2$ | *id* = **select** val_0 *typ* $val_1\ val_1$
		|	*id* = **fcmp** *fcond fp* $val_1\ val_2$ | *id* = **call** $typ_0\ val_0\ \overline{param}$
		|	*id* = **getelementptr** *(typ*)val* $\overline{val_j}^j$

Figura 6.1: Sintaxe de um subcojunto de LLVM-IR

Todo arquivo fonte compilado com o LLVM é considerado um **módulo**. Um módulo é composto de um ou mais *layouts*, tipos nomeados *namedt* e uma ou mais produções *prod*:

- ***Layout*** permite especificar determinadas informações sobre determinados tipos tais como seu tamanho em bits (*sz*) e o seu alinhamento na memória (*align*).

  Bigendian e **littleendian** referem-se à ordem em que os bits são armazenados na memória, representando, respectivamente, a ordem crescente de peso numérico (caso dos dispositivos atuais) e ordem decrescente de peso numérico, isto é, os bits mais significantes tem endereços mais baixos na memória.

Por exemplo, o layout int 32 32 32 indica que inteiros de 32 bits terão 32 bits alinhados em ambos os casos $align_0$ e $align_1$.

- **Tipos nomeados (*namedt*)** permitem a definição de novos tipos de dados tal como *struct* em C. Por exemplo, a Listagem 6.1 mostra um trecho de código em C que cria uma nova estrutura, chamada *teste*, contendo dois campos *f1* e *f2*, ambos do tipo inteiro:

Listagem 6.1: Exemplo de *struct* em C.

```
struct teste {
    int f1;
    int f2;
};
```

Na Listagem 6.2 é apresentado o tipo nomeado correspondente à estrutura teste da Listagem 6.1. Os campos f1 e f2 são definidos como inteiros contendo 32 bits, representados por *i32*.

Listagem 6.2: Representação de *teste* na LLVM-IR com tipos nomeados.

```
%teste = type {i32, i32}
```

- **Produções (*prod*)** são elementos que representam a definição ou a declaração de funções, ou ainda podem representar variáveis globais.
  - id = **global** *typ const align* - declara uma variável global de nome id, informando também o seu tipo *typ*, seu valor inicial *const* e seu alinhamento *align*. Exemplo: *@X* = **global** *i32 17 align 1* representa a definição de uma constante global de tipo inteiro de 32 bits (i32) com o valor 17.
  - **define** *typ id(*$\overline{arg}$*) {*$\overline{b}$*}* - define uma função de nome id que retorna valor do tipo *typ*. A função pode receber 0 ou mais argumentos $\overline{arg}$. O corpo de uma função é composto por uma sequência de blocos básicos $\overline{b}$. Exemplo: *define i32 @main(i32 3){bbs}* representa a definição de uma função chamada "main"com tipo de retorno inteiro de 32 bits e com o argumento do mesmo tipo com valor 3, composta por diversos blocos básicos, representada por "bbs".
  - **declare** *typ id(*$\overline{arg}$*)* semelhante à definição de funções exceto que é usado para introduzir em um programa funções que foram definidas externamente.

LLVM-IR provê suporte a tipos. A Tabela 6.1 explica os tipos de LLVM-IR presentes na gramática da Figura 6.1.

Tabela 6.1: Tipos de LLVM-IR

i*sz*	inteiro de tamanho *sz* (número de bits do inteiro) (i1, i8, i16, i32, i64,...)
fp	ponto flutuante (**float** para 32 bits e **double** para 64 bits)
void	tipo void
*typ**	ponteiros
[sz x typ]	arrays compostos por um número inteiro de elementos (sz) de um determinado tipo (typ)
typ $\overline{typ_j}^j$	tipos de funções com tipo de retorno *typ* e uma lista de tipos de seus argumentos

As operações em LLVM-IR realizam cálculos e computações sobre valores *val*, que podem ser: constantes *cnst* e identificadores *id*. Identificadores locais iniciam com "%"como em %X, %a, etc. Já os valores globais e nomes de funções possuem identificadores iniciados com o símbolo "@".

LLVM-IR dispõe de diversas operações binários *bop* usuais em linguagens intermediárias para operações aritméticas, lógicas e de deslocamento.

Em LLVM-IR são possíveis operações binárias de ponto flutuante *(fbop)*, com identificadores similares às operações inteiras, apenas com a adição da letra 'f', como por exemplo *fadd* que representa a adição em ponto flutuante.

As funções são formadas por blocos de instruções, chamados *blocos básicos*. Cada bloco básico contém apenas um ponto de entrada para o fluxo de execução (a primeira instrução) e apenas um ponto de saída do fluxo de execução (a última instrução). A sintaxe de um bloco básico *b* em LLVM-IR é dada por: $\overline{\phi}\ \overline{c}\ tmn$

Um bloco básico é representado por um label, seguido por zero ou mais instruções ϕ. Cada bloco possui um conjunto de instruções $\overline{c}$ e uma instrução terminadora *tmn*. O *label* é o identificador atribuído a cada bloco básico de forma única.

Exemplos e mais detalhes de instruções ϕ são apresentados na Seção 3.1 no capítulo 3, quando da descrição da arquitetura do compilador LLVM.

A Tabela 6.2 explica de maneira informa a semântica das demais instruções/comandos presentes na gramática da Figura 6.1.

Tabela 6.2: Comandos em LLVM-IR

Sintaxe	Descrição
id = *bop* (**int** *sz*) val_1 val_2	usado para representar uma instrução binária inteira entre dois valores
id = *fbop fp* val_1 val_2	usado para representar uma operação binária de ponto flutuante entre dois valores
id = **load** (*typ** val_1 *align*)	usado para ler valores da memória
store *typ* val_1 val_2 *align*	usado para armazenar o valor val1 no item val2
free (*typ**) *val*	desaloca a região da memória associada ao ponteiro informado em *val* do tipo *typ*
id = **alloc/malloc** *typ val align*	Alocam regiões da memória. Tomam como base um tipo e uma valor, além do alinhamento dos dados na região alocada.
id = *trop* typ_1 *val* **to** typ_2	"trubca"o valor de um tipo para outro tipo informado. Exemplo: i32 10 -> i1 1
id = *cop* typ_1 *val* **to** typ_2	converte o valor de um tipo para outro sem alterar os bits
id = **icmp** *cond typ* val_1 val_2	retorna um valor booleano da comparação entre dois ou mais valores
id = **select** val_0 *typ* val_1 val_2	Usado para a escolha de um valor em um conjunto, sem desvio
id = **fcmp** *fcond fp* val_1 val_2	representa uma instrução de comparação de valores de ponto flutuante.
option id = **call** typ_0 val_0 $\overline{param}$	usada para a chamada de funções
id = **getelementptr** (*typ**) *val* $\overline{val_j}^j$	Usada para capturar o endereço de um subelemento de uma estrutura de dados agregados

Instruções de término devem ser necessariamente as últimas instruções de cada bloco básico, sendo que todo bloco básico deve conter uma instrução de término. A Tabela 6.3 mostra as instruções de término em LLVM-IR.

Tabela 6.3: Instruções de término

Sintaxe	Descrição
br *val* l_1 l_2	Se *val* é verdadeiro, o desvio ocorre para o label l_1, caso contrário, desvia para o label l_2.
br l	desvio incondicional para o bloco básico de label l.
ret *typ val*	retorno de função com valor *val* do tipo *typ*
ret void	retorno de função de tipo *void*.
unreachable	informa ao otimizador que um determinado código não é alcançável, ou seja, em termos de CFG, não há uma aresta que faça a ligação para este código.

A Listagem 6.3 apresenta um exemplo de programa em C, seguida pela Listagem 6.4 que mostra o código em linguagem intermediária correspondente ao programa (neste exemplo, por uma questão de simplicidade são omitidas as informações de layout que são produzidas pelo compilador LLVM)

Listagem 6.3: Exemplo de programa em C.

```
int main() {
   int a=3;
   int b=6;
   int c = a * b;
   if(a==5){
      c = a * c;
   }
   return 0;
}
```

Listagem 6.4: Função main da Listagem 6.3 em LLVM-IR.

```
1  define i32 @main() {
2  entry:
3     %retval = alloca i32 align 4
4     %a = alloca i32 align 4
5     %b = alloca i32 align 4
6     %c = alloca i32 align 4
7     store i32 3, i32* %a, align 4
8     store i32 6, i32* %b, align 4
9     %0 = load i32* %a, align 4
10    %1 = load i32* %b, align 4
11    %2 = mul i32 %0, %1
12    store i32 %2, i32* %c, align 4
13    %3 = load i32* %a, align 4
14    %4 = icmp eq i32 %3, 5
15    br i1 %4, label %bb, label %bb1
16
17 bb:                ; preds = %entry
```

```
18    %5 = load i32* %a, align 4
19    %6 = load i32* %c, align 4
20    %7 = mul i32 %5, %6
21    store i32 %7, i32* %c, align 4
22    br label %bb1
23
24 bb1:                ; preds = %bb, %entry
25    br label %return
26
27 return:            ; preds = %bb1
28    %retval2 = load i32* %retval
29    ret i32 %retval2
30
31 }
```

A função principal do programa (*main*), retorna um inteiro de 32 bits. Ela possui quatro blocos básicos (*entry*, *bb*, *bb1* e *return*). O bloco *entry* é definido por padrão pela linguagem intermediária de LLVM como sendo o primeiro bloco de cada função, não possuindo nenhum predecessor. Este bloco é composto por diversas instruções, sendo que o fluxo de execução segue sequencialmente até alcançar a instrução terminadora (linha 21), que transfere o controle para o bloco *bb* caso o valor do registrador $\%4$ seja verdadeiro ou transfere para o bloco $bb1$ caso o valor armazenado em $\%4$ seja falso.

Como LLVM-IR é construída respeitando o formado SSA, os registradores são numerados sequencialmente, de forma automática, sendo cada um atribuído uma única vez.

A execução do programa é concluída quando o fluxo de controle alcança a instrução terminadora de bloco (linha 35), instrução que representa a instrução de retorno localizada no final da função *main*. A Figura 6.2 apresenta o grafo de fluxo de controle.

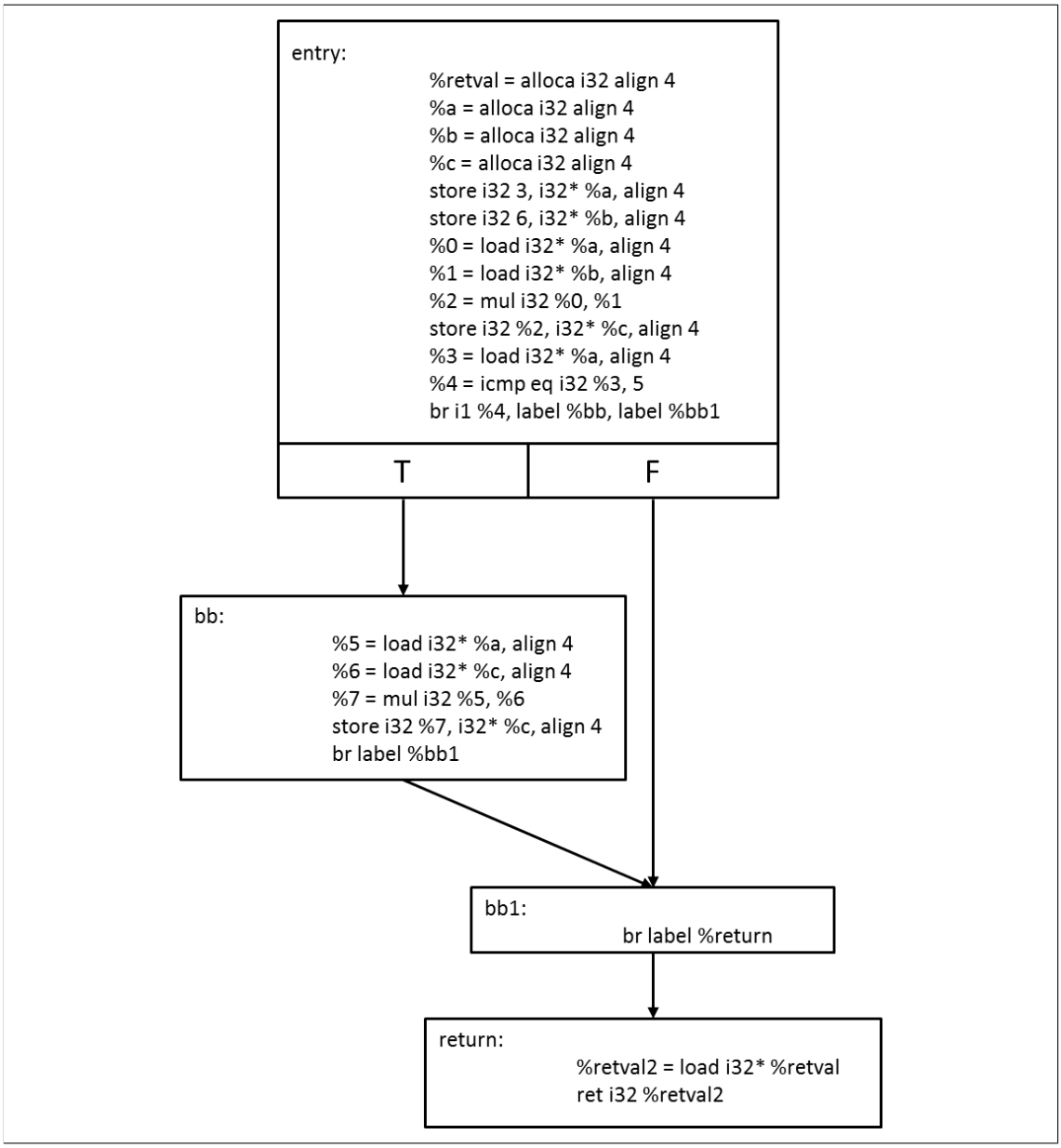

Figura 6.2: Exemplo de programa em Linguagem LLVM-IR

ANEXO II - ARTIGOS PUBLICADOS

Três artigos, relacionados ao tema da dissertação, são apresentados neste anexo. Inicialmente, é mostrado o artigo intitulado *Impact on Reliability in the Control-Flow of programs under Compiler Optimizations*, foi publicado nos anais do III Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC), em novembro de 2012, ocorrido em Natal - Rio Grande do Norte.

O segundo artigo relacionado à dissertação, intitulado *Compiler Optimizations Do Impact the Reliability of the Control-Flow of Radiation Hardened Software*, foi aceito para publicação no IV Workshop Sobre os Efeitos da Radiação Ionizante em Componentes Eletrônicos e Fotônicos de Uso Aeroespacial (WERICE), evento ocorrido em outubro de 2012 na cidade de São José dos Campos - São Paulo. A versão estendida deste artigo foi aceita para publicação no Journal of Aerospace Management & Technology (JATM).

Finalmente, o artigo intitulado *Compiler Optimizations Do Impact the Reliability of Control-Flow Radiation Hardened Embedded Software* foi aceito no International Embedded Systems Symposium (IESS:13), a ser realizado em Paderborn, Alemanha, em Junho de 2013.

Impact on Reliability in the Control-Flow of Programs under Compiler Optimizations

Rafael Parizi, Ronaldo Ferreira, Álvaro Freitas and Luigi Carro
Instituto de Informática
Federal University of Rio Grande do Sul
Porto Alegre, Brazil
{rbparizi, rrferreira, afmoreira, carro}@inf.ufrgs.br

***Abstract*—This paper evaluates the impact on reliability in the control-flow of programs that compiler optimizations incur in terms of fault coverage for the Automatic Correction of Control-flow Errors technique. This technique was implemented in the LLVM framework, enabling the automated analysis of programs. In order to evaluate the efficiency of the technique of fault tolerance we performed a series of fault injection experiments using the MiBench benchmark suite as case study, measuring how individual and combined optimizations impact reliability.**

I. Introdução

O tamanho do transistor tem diminuído com o passar dos anos de acordo com a Lei de Moore, resultando em aumento no desempenho e em redução no consumo de energia dos microprocessadores. Porém, essa diminuição torna os transistores mais vulneráveis e compromete a sua confiabilidade [1]. Assim sendo, faz-se necessária a implantação de mecanismos de proteção contra falhas, mecanismos estes que, ao mesmo tempo em que aumentam a dependabilidade dos sistemas, devem respeitar as restrições do domínio em que serão inseridos, como, por exemplo, consumo de energia, desempenho e tamanho de programa, no caso dos sistemas embarcados. Nesse contexto, técnicas que não requerem alterações no hardware são atrativas para a tolerância a falhas. A tolerância a falhas em software é uma alternativa às técnicas que alteram o hardware para garantir confiabilidade aos sistemas computacionais.

Técnicas baseadas em software para a tolerância a falhas modificam o programa inserindo instruções adicionais, usadas para a detecção e também correção de falhas ocorridas no hardware em tempo de execução. As falhas transitórias de hardware, isto é, falhas de duração limitada que afetam componentes físicos de sistemas computacionais, se manifestam em sua maioria como erros de fluxo de controle (CFE) podendo atingir índices de até 70% do total de falhas [9]. Um CFE é um desvio ilegal de uma instrução a outra, sendo esse desvio não esperado quando da não ocorrência de falhas.

ACCE é uma técnica de software que provê tolerância a falhas permitindo a detecção e inclusive a correção de falhas transitórias de hardware manifestadas como CFEs. Conforme experimentos realizados com a técnica [10], ela se mostrou capaz alcançar até 80% de correção de falhas. Originalmente, ACCE foi implementada como uma modificação no compilador GCC (*Gnu Compiler Collection*), sendo sua eficácia avaliada com a injeção de falhas para o benchmark SPEC2000. Na ocasião, não foi levado em consideração o impacto de otimizações providas pelo compilador na sua eficácia de correção de CFEs.

Nesse trabalho, ACCE foi implementada como um passo de transformação adicional ao compilador LLVM [4], uma infraestrutura de compilação que fornece um amplo número de otimizações e análises possíveis de serem aplicadas em programas. As transformações realizadas nos programas compilados no LLVM com a aplicação de ACCE são feitas sobre a linguagem intermediária LLVM (LLVM-IR) [5], resultando em portabilidade de linguagem de programação e portabilidade de arquitetura de máquina.

O objetivo deste trabalho é ir além das análises originais realizadas em ACCE [10]. O objetivo deste artigo é avaliar a interação da técnica com as demais transformações do LLVM, justificado pelo fato de que dificilmente outras transformações disponíveis em compiladores não são utilizadas durante a compilação de programas. Para tanto, 10 programas da suíte Mibench [3] foram compilados de duas formas distintas: inicialmente, foi realizada a aplicação individual de cada passo de transformação disponível no LLVM seguida de ACCE. Na segunda forma, foram aplicadas combinações de transformações randomicamente selecionadas, com ACCE aplicada por conseguinte. Em ambas as formas, foram avaliadas as taxas de detecção e correção de erros de fluxo de controle e a influência das transformações sobre estas taxas.

As principais contribuições desse trabalho são portanto:

- implementação da técnica ACCE no *framework* LLVM, permitindo que ACCE possa ser usada na proteção de programas escritos nas diversas linguagens de programação suportadas pelo LLVM, e executar nas diversas arquiteturas para as quais o LLVM gera código;
- avaliação de ACCE quanto à eficácia de detecção e correção de erros de fluxo de controle quando usada em combinação com as transformações disponibilizadas pelo LLVM.

Esse artigo está organizado da seguinte forma: a Seção II apresenta os trabalhos relacionados. A Seção III detalha a técnica ACCE e as alterações que ela realiza em um programa para a inserção de tolerância a falhas. A Seção IV descreve os experimentos realizados e os resultados obtidos quanto à

cobertura de falhas. Por fim, a Seção V relata as conclusões e trabalhos futuros.

II. Trabalhos Relacionados

Um conjunto de técnicas de tolerância a falhas, conhecidas como SIHFT [2] (*Software Implemented Hardware Fault Tolerance*), permite o tratamento em nível de software de falhas ocorridas no hardware, sem a necessidade de modificações em componentes físicos. Técnicas SIHFT se baseiam na verificação de assinaturas e inserção de instruções, em tempo de compilação, nos blocos básicos do programa para que, em tempo de execução, essas assinaturas possam ser verificadas e, em caso de divergência entre a assinatura atual de execução e as assinaturas calculadas em tempo de compilação, o erro seja detectado. O cálculo das assinaturas de cada um dos blocos básicos é realizado em tempo de compilação através da representação do programa como um grafo de fluxo de controle (CFG), sendo os vértices os blocos básicos e as arestas o fluxo de controle do programa.

Algumas técnicas SIHFT recentemente desenvolvidas para a detecção de CFEs são: *Control-Flow Checking By Software Signatures* (CFCSS) [8], *Yet Another Control-Flow Cheking Using Assertions* (YACCA) [2] e *Control-Flow Error Detection through Assertions* (CEDA) [9]. Ainda para a detecção e correção de CFEs, uma técnica recente e eficaz é ACCE, apresentada em maiores detalhes na seção III.

As técnicas CFCSS, YACCA, CEDA e ACCE realizam transformações similares nos programas. Porém, uma diferença a ser apontada entre elas é o fato de CFCSS inserir apenas um ponto de verificação da assinatura para cada bloco básico, enquanto que YACCA, CEDA e ACCE verificam as assinaturas em dois pontos distintos de cada bloco.

Quantitativamente, no que diz respeito à detecção de CFEs, CEDA se mostra ser a técnica mais eficaz, com resultados que se aproximam dos 70% de detecção, enquanto CFCSS resulta em 38,5% e YACCA 36%. Quanto ao *overhead* de desempenho, CEDA também se mostra mais atrativa que as demais, visto que seu *overhead* de desempenho é de 20%, enquanto CFCSS e YACCA possuem *overheads* de 50% e 45%, respectivamente.

Quanto à ACCE, seus índices de detecção são similares aos obtidos com CEDA, uma vez que ACCE se baseia em CEDA, tendo como adendo funcionalidades que inserem em programas a capacidade de restabelecer o controle do programa em casos de desvios ilegais oriundos de falhas transientes.

Nenhuma das técnicas citadas acima têm sua implementação disponível e também não foram avaliadas quando empregadas em conjunto com outras transformações de programas disponíveis em compiladores.

III. Detecção e Correção Automática de Erros de Fluxo de Controle

ACCE é uma técnica de software para tolerância a falhas que transforma programas, em tempo de compilação, modificando os blocos básicos com a inserção de instruções para atualização e comparação de assinaturas, permitindo a detecção e correção de erros de fluxo de controle. Dessa forma, para fins de clareza, a transformação feita por ACCE em programas, bem como as fases de detecção e correção de CFEs serão explicadas separadamente nas subseções III-A, III-B e III-C, respectivamente.

A. Transformação de Programas em Tempo de Compilação

ACCE modifica os programas para o tratamento de falhas, iniciando durante a compilação, com a representação do programa como um grafo de fluxo de controle, identificando os blocos básicos, seus respectivos sucessores e predecessores. Com os blocos básicos identificados, ACCE realiza o cálculo das assinaturas esperadas para o interior e final de cada bloco durante a execução do programa. Na sequência, para a detecção de CFEs, ACCE representa cada bloco em 3 regiões: cabeçalho (*header*), interior e, rodapé (*footer*). No *header*, ACCE insere duas instruções, necessárias para verificar se o fluxo de controle está sendo desviado de um predecessor existente, ou seja, se o controle do programa está percorrendo um caminho existente no CFG. No interior do bloco, ficam contidas as instruções originais de cada bloco, instruções estas que serão executadas caso a verificação realizada no *header* não detecte a ocorrência de um CFE. Por fim, no *footer*, ACCE adiciona instruções para verificar se a execução percorreu o interior do bloco e não foi desviado de outro ponto qualquer do programa, caso que também configura um CFE. Em termos de implementação, as regiões *header* e *footer* representam a criação de dois blocos adicionais, chamados de *blocos de proteção*.

A Figura 1 mostra um exemplo de bloco básico transformado após a aplicação da técnica ACCE. Na Figura 1(a) é mostrado um bloco original, enquanto na Figura 1(b) é apresentado o bloco transformado, representado com os blocos de proteção e com indicação do local de inserção das instruções extras para verificação e atualização das assinaturas.

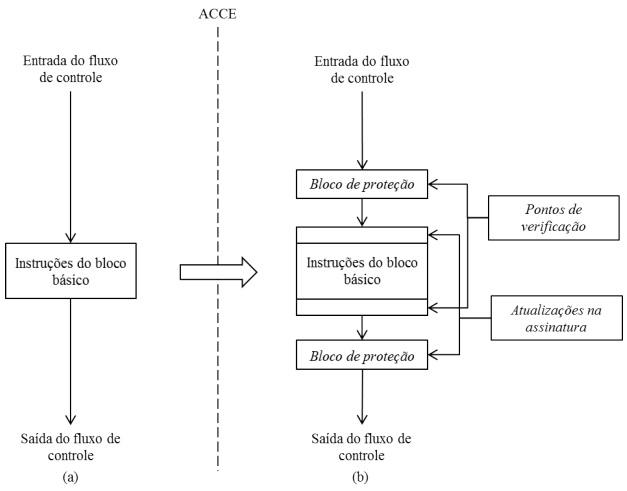

Fig. 1. Bloco básico modificado por ACCE para a detecção e correção de CFEs

ACCE continua com a inserção, ainda em tempo de compilação, de trechos de códigos usados para a correção de

CFEs. Nesse ponto, ACCE define um identificador único para cada uma das funções do programa, usado para a identificação da função de origem do desvio ilegal. Além disso, em cada função do programa, um trecho de código conhecido como manipulador de erros da função (*Function Error Handler* - FEH), é inserido para o tratamento dos desvios ilegais ocorridos na função, sendo somente executado quando a verificação das assinaturas resultar em uma diferença, ou ainda quando um desvio ilegal acontecer diretamente para o FEH.

A última etapa do processo de transformação de programas feita por ACCE é a inserção de uma função para a manipulação global de erros (*Global Error Handler* - GEH), a qual será executada caso o desvio ilegal tenha como origem e destino funções distintas do programa. Dessa forma, em tempo de execução, caso uma FEH constate que o erro não ocorreu na função de sua abrangência, ACCE transfere o controle para a GEH e esta, por sua vez, transfere o controle para a função em que de fato o erro ocorreu para que o mesmo seja corrigido.

B. Detecção de Erros de Fluxo de Controle em Tempo de Execução

A detecção de CFEs realizada por ACCE, em tempo de execução, dá-se com a verificação das assinaturas no interior e no final de cada bloco básico. Um registrador de assinatura é mantido pela técnica, constantemente atualizado para conter em cada ponto do programa o valor atual da assinatura. Assim, durante a execução do *header* e *footer* dos blocos básicos, o valor do registrador de assinatura é comparado com as assinaturas calculadas em tempo de compilação. Durante essas verificações, caso os valores das assinaturas não correspondam, o controle é transferido para FEH, representando a detecção de um desvio ilegal.

A Figura 2 mostra um exemplo da verificação e atualização das assinaturas realizadas em tempo de execução em um bloco básico do programa, onde a transferência do controle para FEH em caso de detecção de CFE é representada por *error()*.

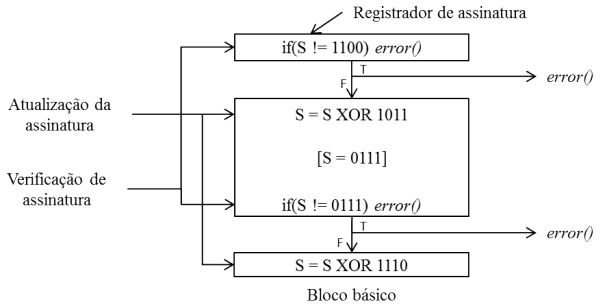

Fig. 2. Exemplo de detecção de erro de fluxo de controle por ACCE

C. Correção de Erros de Fluxo de Controle em Tempo de Execução

A correção de CFEs realizada por ACCE durante a execução do programa é automática, isto é, não há a necessidade da interrupção na execução para que o erro seja corrigido. O processo de correção inicia após a transferência do controle para FEH por uma das instruções de verificação de algum dos blocos básicos do programa. Uma vez em FEH, caso a falha tenha sido interna à função de abrangência da FEH em execução, o controle é transferido para o bloco de origem do erro, restabelecendo o fluxo. Caso contrário, FEH transfere o controle para GEH, que busca pela função de origem do desvio ilegal através do valor de identificação das funções que, uma vez identificada, o controle é transferido para esta função, que irá tratar o erro por meio da FEH adequada.

Um exemplo de correção de CFE é demonstrado na Figura 3 que, para ser didático, os blocos protetores não foram representados. No exemplo, há um desvio ilegal partindo do bloco N2 da função F1 com destino ao interior do bloco N6 da função F2. A verificação da assinatura ao final de N6 detecta um CFE, pois há uma diferença entre a assinatura esperada (0110) e a assinatura atual (0111), e transfere o controle para *FEH_2*(etapa 1). Em *FEH_2* é detectado que o valor de identificação da função de origem do CFE ($F = 1$) difere do valor da função atualmente em execução ($F = 2$) e, então, o controle é transferido para GEH (etapa 2). Em GEH, a variável *err_flag* recebe o valor 1 indicando que um erro deve ser corrigido e o controle é transferido para F1, uma vez que o identificador da função onde o desvio ilegal teve origem (F) é igual a 1 (etapa 3). No início de F1, dado que *err_flag* é igual a 1, o controle é transferido direto para *FEH_1* (etapa 4). Finalmente, em *FEH_1* é verificado que o valor da assinatura equivale ao interior de N2, logo, o controle é restabelecido para o interior do bloco N2 e a execução continua.

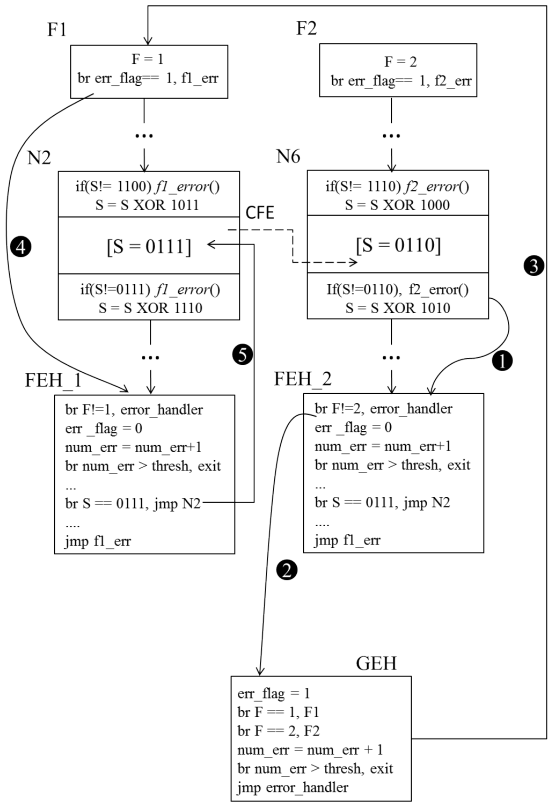

Fig. 3. Exemplo de correção de erro de fluxo de controle por ACCE

```
bb:
  %1 = load i32* %i, align 4
  %2 = add nsw i32 %1, 1
  store i32 %2, i32* %i, align 4
  br label %bb2

bb2:
  %3 = phi i32 [ %2, %bb ], [ 0, %entry ]
  %4 = icmp sle i32 %3, 2
  br i1 %4, label %bb, label %bb3
```

Fig. 4. Nodo ϕ em LLVM-IR

Alguns CFEs não são corrigidos por ACCE, pois em alguns casos as falhas podem ocorrer em instruções inseridas pela técnica, impossibilitando a restauração do fluxo de controle para a instrução correta imediatamente anterior à ocorrência da falha.

A técnica ACCE possui uma restrição de aplicação em programas na linguagem intermediária do LLVM que possuam instruções ϕ, constatação não descrita pelos autores da técnica [10]. ϕ é uma instrução especial usada em blocos que possuem mais de um predecessor e compartilham com eles alguma variável, permitindo que o valor correto seja atribuído à variável dependendo por qual predecessor o fluxo de execução passou. Essas instruções sao características de linguagens representadas na forma SSA (*Single Static Assignment*), onde cada registrador é atribuído uma única vez, caso da LLVM-IR.

Blocos básicos que possuem instruções ϕ são chamados de *Nodos* ϕ, sendo que estas instruções devem estar obrigatoriamente, segundo restrições da linguagem intermediária do LLVM , posicionadas no início do bloco básico, não podendo haver nenhuma outra instrução não-ϕ entre o label do bloco e a primeira instrução ϕ. Essa característica de LLVM-IR impossibilita a aplicação da técnica ACCE em Nodos ϕ, uma vez que ACCE requer a inserção de instruções no início do bloco. Nesses casos é necessário aplicar o passo de transformação "*reg2mem*", em tempo de compilação, que remove os nodos ϕ do programa através de desconstrução da forma SSA.

A Figura 4 mostra um exemplo fictício de código em LLVM-IR, contendo dois blocos básicos (bb e bb_2). A instrução ϕ aparece destacada no início de bb_2, logo após o label de identificação do bloco, composta por pares de argumentos, sendo que cada par representa o valor da variável compartilhada seguido do bloco predecessor no CFG. No exemplo, o registrador $\%3$ em bb_2 receberá o conteúdo do registrador $\%2$ caso o controle tenha passado por bb antes de bb_2, ou o valor 0 se o controle foi transferido pelo bloco básico $entry$.

IV. Experimentos & Resultados

A. Metodologia & Ferramentas

A avaliação do impacto das otimizações fornecidas por compiladores na cobertura de falhas em programas transformados por ACCE foi realizada por meio da implementação da técnica na infraestrutura de compilação LLVM [6], escolhida por prover diversos passos de transformação e análise de programas.

ACCE foi implementada como um passo de transformação similar aos fornecidos pelo LLVM, que atua sobre a linguagem intermediária de programa do LLVM com a inserção das instruções, blocos básicos e funções para a detecção/correção automática de CFEs. A linguagem intermediária do LLVM é composta de instruções de 3 endereços, similar à linguagem *Assembly*, com características de linguagem de alto nível e estruturada na forma SSA.

Um conjunto de 10 programas da suíte de benchmarks Mibench foi utilizado nos experimentos: *basicmath, bitcount, 32-bit Cyclic Redundancy Check (crc32), dijkstra, Fast Fourier Transform (fft), patricia, quicksort, rijndael, string search e susan*. *Basicmath* realiza operações matemáticas, *bitcount* executa funções de manipulação de bits, *crc32* realiza detecção de erros em transmissão de dados, *dijkstra* calcula caminhos entre pares de vértices em um grafo escolhendo o menor caminho, e *fft* executa uma transformada rápida de Fourier usado no processamento de sinais digitais. *Patricia* é usado para representar tabelas de roteamento no domínio de redes, *quicksort* realiza ordenação de dados, *rijndael* executa operações de criptografia, *string seacrh* busca por palavras em sentenças e *susan* é usado no processamento de imagens.

A fase de experimentação foi dividida em 2 cenários distintos, avaliando ACCE das seguintes formas:

1) Cada programa foi compilado com a aplicação individual de cada um dos 58 passos de transformação do LLVM seguidos pela aplicação da técnica ACCE, resultando em 58 versões diferentes de cada programa, a fim de avaliar a influência de cada passo de transformação LLVM na tolerância a falhas com ACCE;
2) Para cada programa foram criadas 5 combinações de otimizações constituídas de seis sequências de tamanhos distintos (10, 20, 30, 40, 50, 58 otimizações randomicamente selecionadas), para avaliação de como a combinação de passos de transformação distintos influencia na cobertura de CFEs, metodologia similar a [7] que mostra que com a aplicação aleatória de otimizações é possível aumentar o desempenho de programas.

Para a realização dos experimentos foi utilizado o compilador LLVM versão 2.9, executando em um computador Intel Core i5 2.4 GHz, arquitetura x86-64 bits com 4 GB de memória RAM. Foram injetadas 1000 falhas em cada uma das versões dos programas compilados com os passos de transformação do LLVM, faltas dos modelos: *criação de desvio, remoção de desvio* e *alteração do destino de desvio*. Os experimentos compreenderam um processo de injeção de mais de 800 mil falhas, metodologicamente similar a [9].

B. Impacto Individual das Transformações de Programas na Taxa de Correção de Erros

Nesse cenário, os programas foram compilados com a aplicação individual de cada transformação fornecida pelo LLVM, com a aplicação de ACCE na sequência. Os resultados

obtidos com os experimentos mostraram que a maioria das transformações para os diversos benchmarks apresentaram valores próximos de redução ou aumento de correção de CFEs em relação ao programa compilado sem transformações, valor assumido como base para as comparações. Para exemplificar, a Figura 5 apresenta um subconjunto de transformações aplicadas sobre o benchmark Dijkstra, mostrando que vários dos passos de transformação impactaram na eficácia da técnica em menos de 10%. No eixo x encontram-se as transformações do LLVM e no eixo y os valores normalizados em relação ao valor base, cujos valores maiores que 1 representam um aumento de correção de falhas, enquanto os valores menores que 1 indicam diminuição da correção.

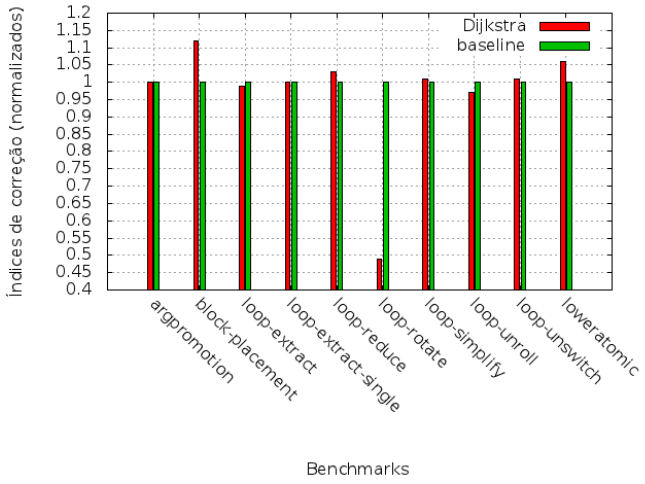

Fig. 5. Impacto de transformações providas pelo LLVM no Dijkstra

Como é possível perceber na Figura 5, para o programa Dijkstra, o uso da transformação *Block-placement* provocou melhoria na taxa de correção de falhas em 12%, enquanto que o uso da transformação *loop-rotate* prejudicou a correção de CFEs em 51%, permanecendo as demais transformações muito próximas ao valor base. Também se pode constatar nesse cenário dos experimentos que o impacto de uma transformação sobre a eficácia da técnica ACCE está diretamente relacionada com a estrutura do programa em que a transformação foi aplicada e o quanto esse programa foi transformado, uma vez que os resultados apresentaram que uma mesma transformação se mostrou benéfica para alguns benchmarks, aumentando a correção de falhas, enquanto para outros benchmarks esta transformação foi prejudicial, causando redução na correção das falhas.

Um exemplo de que uma mesma transformação pode afetar de formas distintas os programas, dependendo de sua estrutura, é apresentado pela Figura 6, onde os programas foram compilados com a transformação *loop-reduce*, sendo que para o benchmark *Bitcount* colaborou negativamente, reduzindo em 33% a taxa de correção de CFEs, enquanto que para o benchmark *susan*, *loop-reduce* aumentou a correção de CFEs em 8%. Novamente, o valor base assumido para as comparações é o resultado do programa compilado sem qualquer transformação.

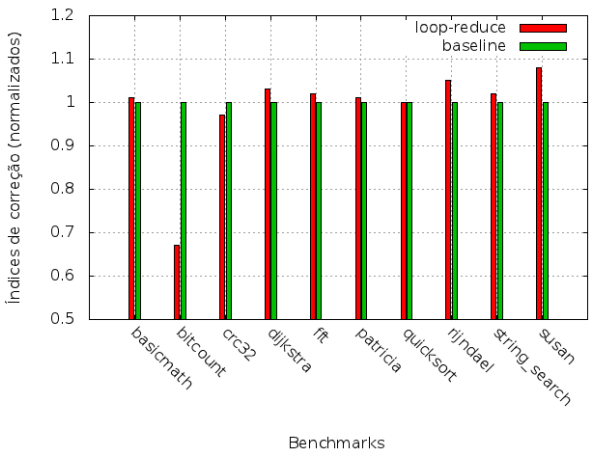

Fig. 6. Tranformação loop-reduce afetou positiva e negativamente os benchmarks

A explicação dos diferentes resultados apresentados pelas transformações de programas disponibilizadas no LLVM em relação à cobertura de falhas está na criação de blocos com apenas uma instrução de desvio incondicional pelas transformações, blocos que nomeamos de *blocos label-br*. Como a técnica ACCE insere instruções e cria blocos básicos novos e, devido ao fato dos blocos *label-br* possuírem um baixo número de instruções, aumenta a chance de uma falha afetar as instruções, seja de verificação ou atualização das assinaturas, reduzindo assim o número de falhas corrigidas.

Ainda no cenário 1 dos experimentos pode-se perceber que a transformação *functionattrs* foi a única que acarretou em aumento da taxa de correção falhas em todos os 10 benchmarks em que as falhas foram injetadas, o que leva a crer que esta transformação pode ser utilizada nos programas sem a possibilidade de causar prejuízos no que diz respeito à funcionalidade de tolerar falhas dos programas. A Figura 7 mostra taxas de correção resultantes da aplicação da transformação *functionattrs* nos diversos benchmarks do cenário 1 dos experimentos.

C. Impacto das Transformações de Programas Combinadas na Taxa de Correção de Erros

A Tabela I apresenta os resultados do cenário 2 dos experimentos, cenário em que os programas foram compilados e otimizados por sequências variadas, randomicamente selecionadas, de passos de transformação fornecidos pelo LLVM, seguidos pela técnica ACCE. A taxa de correção de falhas nesse cenário foi obtida para sequências de 10, 20, 30, 40, 50, 58 passos de transformações. Os valores apresentados na tabela foram normalizados com base na taxa de correção resultante da injeção de falhas no programa compilado e transformado somente por ACCE, sem nenhuma outra transformação.

Com base nos valores da apresentados na Tabela I, as sequências de transformações produziram resultados bastante similares às transformações aplicadas de forma individual.

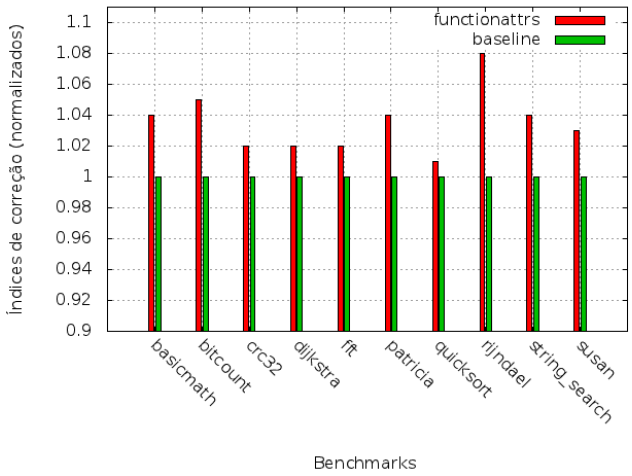

Fig. 7. Índices de ganho de correção resultantes da aplicação da transformação *functionattrs*

TABLE I
AUMENTO/DIMINUIÇÃO DA TAXA DE CORREÇÃO PARA SEQUÊNCIAS DE PASSOS DE TRANSFORMAÇÃO FORNECIDOS PELO LLVM.

	# de passos de transformações LLVM					
	10	20	30	40	50	58
basicmath	0,92	0,87	0,91	0,91	0,92	0,9
bitcount	0,99	0,86	0,99	1,01	0,99	1,01
quicksort	1,05	1	0,94	1,01	0,93	0,99
dijkstra	0,97	1	1,05	1,08	0,96	0,98
patricia	0,97	0,88	0,96	0,91	0,97	0,92
string search	1	1,04	0,9	1,03	1	1,06
susan	1,02	1,01	1	0,95	1	0,95
rijndael	1,01	1,02	0,99	1,03	0,97	1,02
fft	1,03	1,01	0,99	0,97	0,97	0,98
crc32	0,99	1,01	0,99	0,99	0,99	1

Esses resultados não permitem uma garantia de quanto o número de otimizações aplicadas sobre um programa pode influenciar ACCE. Os valores da Tabela I mostram que a taxa de correção não teve grande variação entre as sequências de otimização, não excedendo os 8% de ganho (*Dijkstra* com 40 transformações) e 12% de perda (*Patricia* com 20 passos de transformação) na correção de CFEs.

V. CONCLUSÕES E TRABALHOS FUTUROS

Esse artigo avaliou a influência dos passos de transformação de programas fornecidos pelo LLVM na correção de falhas obtida com a técnica ACCE, implementada como um passo de transformação do compilador LLVM. Os experimentos foram realizados com uma campanha de injeção de cerca de 880.000 falhas em 10 diferentes programas da suíte de *benchmarks* Mibench.

Através da implementação de ACCE no LLVM, descobriu-se a não aplicabilidade da técnica em programas que em sua representação intermediária possuem blocos ϕ, visto que ACCE necessita inserir instruções no início do bloco básico e, no caso do LLVM, os blocos ϕ necessariamente devem conter como primeira instrução uma instrução ϕ. Portanto, para as otimizações que geraram instruções ϕ, foi usada de forma combinada o passo *reg2mem*, que remove as instruções ϕ permitindo a aplicação de ACCE.

Com os experimentos de injeção de falhas, foi possível identificar apenas um passo de transformação de programas vantajoso na taxa de correção para todos os benchmarks avaliados (*functionattrs)*. Notou-se, ainda, que a taxa de correção de uma transformação é altamente dependente da aplicação e da quantidade de blocos *label-br* que a transformação aplicada cria, uma vez que a mesma transformação resultou em taxas de cobertura de falhas distintas para diferentes benchmarks, diminuindo para um benchmark a correção em 33% e aumentando para outro em 8%. Outra constatação foi que a maioria das combinações de transformações resultou em valores aproximados de correção em comparação ao valor base, programa compilado apenas como ACCE.

Como trabalhos futuros, será avaliado como a estrutura do programa influencia na técnica ACCE, a fim de identificar o comportamento de ACCE em determinadas estruturas de programas e como escolher a transformação adequada para obter maiores índices de cobertura de falhas.

VI. AGRADECIMENTOS

Este trabalho é financiado pela Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS), pelo CNPq e pela CAPES.

REFERENCES

[1] Baumann, R. (2005). Soft errors in advanced computer systems. *IEEE Des. Test*, 22:258–266.

[2] Goloubeva, O. and Rebaudengo, M. and Sonza Reorda, M. and Violante, M. (2003). Soft-error detection using control flow assertions. In *Proc. of the 18th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotech. Sys.*, DFT '03, pages 581–588. IEEE.

[3] Guthaus, M. R. and Ringenberg, J. S. and Ernst, D., Austin, T. M. and Mudge, T. and Brown, R. B. (2001). Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA. IEEE Computer Society.

[4] Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. of the int. symp. on Code generation and optimization*, CGO '04, pages 75–, Washington, DC, USA. IEEE Computer Society.

[5] LLVM (2012a). LLVM's analysis and transform passes.

[6] LLVM (2012b). LLVM's programming documentation.

[7] Fursin, G. and Kashnikov, Y. and Wahid, A. and Chamski, Z. and Temam, O. and Namolaru, M. and Bilha, E. and Ayal, M. and Eric, Z. and Bodin, F. and Barnard, P. and Ashton, E. and Bonilla, E. and Thomson, J. and Williams, C. and Boyle, M. O. (2008). Milepost GCC : Machine Learning Enabled Self-tuning Compiler. *In Proceedings of the GCC Developers? Summit*, June 2008. pages 296–327.

[8] Oh, N., Mitra, S., and McCluskey, E. J. (2002). Ed4i: Error detection by diverse data and duplicated instructions. *IEEE Trans. Computers*, 51(2):180–199.

[9] Vemu, R. and Abraham, J. A. (2006). CEDA: Control-flow error detection through assertions. In *Proc. of the 12th IEEE Inter. Symp. on On-Line Testing*, pages 151–158. IEEE.

[10] Vemu, R. and Gurumurthy, S., and Abraham, J. (2007). ACCE: Automatic correction of control-flow errors. In *ITC '07. IEEE Int. Test Conf.*, pages 1–10.

Compiler Optimizations Do Impact the Reliability of the Control-Flow of Radiation Hardened Software

Rafael Parizi, Ronaldo Ferreira, Luigi Carro, Álvaro Moreira
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Porto Alegre, Brazil
{rbparizi, rrferreira, carro, afmoreira}@inf.ufrgs.br

***Abstract*—This paper discusses how compiler optimizations influence software reliability when the optimized application is compiled with a technique to enable the software itself to detect and correct radiation induced control-flow errors. Supported by a comprehensive fault injection campaign using an established benchmark suite in the embedded systems domain, we show that the compiler is a non-negligible source of noise when hardening the software against radiation induced soft errors.**

Keywords-component; compiler optimization; control-flow errors; LLVM ; radiation; reliability; soft errors;

I. INTRODUCTION

The Moore's Law is still far to come to an end with industry being capable of sustaining it along the years. Industry already offers microprocessors built with 22 nm transistors, with a prediction that transistor's size will reach 7.4 nm by 2014 [1]. This aggressive technology scaling creates a big challenge concerning the reliability of microprocessors using newest technologies. Smaller transistors are more likely to be disrupted by transient sources of errors caused by radiation, known as *soft-errors* [2]. Radiation particles originated from the space when striking a circuit provoke bit flips during software execution, and since transistors are becoming smaller there is a higher probability that transistors will be disrupted by a single radiation particle, because smaller transistors require a smaller amount of charge to disrupt their stored logical value. The newest technologies are so sensitive to radiation that their usage will be compromised even at the sea level, as predicted in the literature [3]. For instance, considering that 22 nm computing devices are already widespread (e.g., GPU cards), reliability is a serious concern. In [4] it is shown that modern 22 nm GPU cards are susceptible to such an error rate that makes their usage unfeasible in critical embedded systems.

Compiler optimizations are essential in modern software development, enabling applications to execute more efficiently in the target hardware architecture. Modern architectures have complex inner structures that were designed to boost performance, and if the software developer were to be aware of all those details, performance optimization would be a burden to the development processes which ultimately would jeopardize time to market. Those optimizations are transparent to the developer, which picks them based on a performance threshold, or, even letting this task to the compiler itself by flagging if it should be less or more aggressive.

In aerospace systems software certification is one of the most important tasks of software engineering. Software certification requires that several steps in the software engineering process to be predictable, and, as such, unexpected noises can jeopardize the certification process. The deep understanding and control of the compilation process when hardening software against radiation s mandatory, because, as shown in this paper, those optimizations can be unpredictable in terms of reliability of radiation hardened software.

In this paper, we evaluate the impact that compiler optimizations have on reliability when the software is hardened against control-flow errors caused by radiation. Supported by a comprehensive fault injection campaign, we injected faults in applications of the MiBench embedded benchmark suite. Those applications were compiled with optimizations provided by the LLVM compiler, and they had their control-flow segment hardened with the state-of-the-art Automatic Correction of Control-flow Errors (ACCE) [5] software hardening technique. This paper shows that compiler optimizations impact in a non-negligible fashion the software reliability. The understanding of this impact is of major interest when developing radiation hardened software, since the overall software reliability can be jeopardized depending on the optimizations that were chosen, usually aiming to increase software's performance.

This paper is organized as follows: Section II briefly reviews the ACCE software hardening technique; Section III describes the fault model and the methodology used to conduct the experiments; Section IV presents the fault injection experiments using the LLVM compiler with the distinct compiler optimizations; and, finally, Section V concludes the paper, drawing conclusions and pointing possible future work.

II. AUTOMATIC CORRECTION OF CONTROL-FLOW ERRORS

ACCE [5] is a software technique for reliability that detects and corrects control-flow errors (CFE) due to random and arbitrary bit flips that might occur during software execution. The hardening of an application with ACCE is done at compilation, since it is implemented as a transformation pass in the compiler. ACCE modifies the applications' basic blocks with the insertion of extra instructions that perform the error detection and correction during software execution. In this section we briefly explain how ACCE works in two separate subsections, one dedicated to error detection and the other to error correction in the subsections II.A and II.B, respectively.

The reader should refer to the ACCE article for a detailed presentation and experimental evaluation [5].

A. Control-Flow Error Detection

ACCE performs online detection of CFEs by checking signatures in the beginning and in the end of each basic block of the control-flow graph. The basic block signatures are computed and generated during compilation; the signature generation is critical because it needs to compute non-aliased signatures between the basic block, i.e. each block must be unambiguously identified. For each basic block found in the CFG two additional code regions are added, the *header* and the *footer*. The signature checking during execution takes place inside these code regions. Fig. 1 shows two basic blocks (labeled as **N2** and **N6**) with the additional code regions. The top region corresponds to the header and the bottom to the footer. Still at compilation ACCE creates for each function in the application two additional blocks, the function entry block and the *Function Error Handler* (FEH). Fig. 1 depicts two functions, *f1* and *f2*, both owning *entry blocks* labeled as **F1** and **F2**, and function error handlers, labeled as **FEH_1** and **FEH_2**, respectively. Finally, ACCE creates a last extra block, the *Global Error Handler* (GEH), which can only be reached from a FEH block.

At runtime ACCE maintains a global *signature register* (represented as **S**), which is constantly updated to contain the signature of the basic block that the execution has reached. Therefore, during the execution of the *header* and *footer* code regions of each basic block, the value of the signature register is compared with the signatures generated during compilation for those code regions and, if those values do not match, a control flow error has just been detected and the control should be transferred to the corresponding FEH block of the function where execution currently is at. ACCE also maintains the *current function* register (represented as **F**), which stores the unique identifier of the function currently being executed. The current function register is only assigned at the extra entry function block. This process encompasses the *detection* of an illegal and erroneous due to a soft error.

Fig. 1 depicts an example of the checking and update of signatures performed in execution time that occurs in a basic block. In this example, the control-flow error occurs in the block **N2** of function **F1**, where an illegal jump incorrectly transfers the control flow to the basic block **N6** of function **F2**. When the execution reaches the footer of the block **N6** the signature register **S** is checked against the signature generated at compilation. In this case, **S** = 0111 (i.e. the previous value assigned in the header of the block **N2**). Thus, the branch test in the **N6** footer will detect that the expected signature does not match with the value of **S**, and, thus, the CFE error must signaled (step 1 in Fig. 4). In this example, the application branches to the address *f2_err*, making the application enter the **FEH_2** block (since the error was detected by a block owned by the function **F2**, the function error handler invoked is the **FEH_2**). At this point, the CFE was detected and ACCE can proceed with the correction of the detected CFE.

B. Control-Flow Error Correction

The correction process starts as soon as an illegal jump is detected by the procedure described in subsection II.A, with the control flow transferred to the FEH corresponding to the function where the CFE was found. The FEH checks if the illegal jump was originated in the function it is responsible to handle its detected errors by comparing the value of the function's identifier (**F1** or **F2**, in the example of Fig. 1) with the current function register **F**. If the error happened in the function stored in the **F** register, FEH evaluates the current value of the signature register and then transfers the control to the basic block that is the origin of the illegal jump (this origin is stored in the **S** register). On the other hand, if the illegal jump was not originated in the function where the detection has occurred, the FEH then transfers the control flow to the GEH. In this case, the GEH is responsible for identifying the function where the CFE has occurred and to transfer the control flow back to this function, so that the error is correctly treated by the function's FEH. The GEH searches the function where the error has occurred and transfers the control to its entry block, which will then sends the control flow to the proper FEH so that the error can be corrected, i.e. branching the control to the basic block where the CFE has occurred.

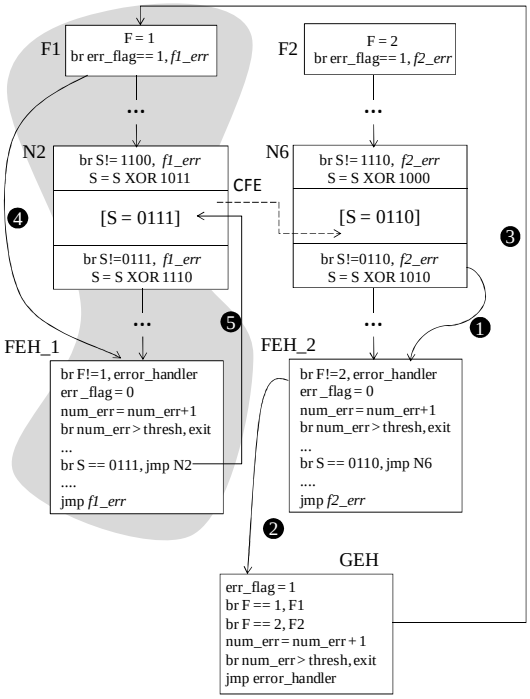

Figure 1. Depiction of how the control is transferred from a function to the basic blocks that ACCE has created when a control-flow error occurs during software execution. In this figure, there is a control flow error (dashed arrow) causing the execution to jump from the block N2 of function F1 to the block N6 of function F2.

Recalling Fig. 1, after the CFE is detected and the control is transferred to **FEH_2** (step 1) the **F** register is matched against the function identifier of the function from where the control came. However, since the CFE originated in the basic block **N2** of function **F1**, **F** = 1. Therefore, **FEH_2** is not capable of finding the basic block where the CFE originated, and then it transfers the control to the **GEH** so that the correct FEH can be found (step 2). The **GEH** searches for the function identifier stored in **F**, until it finds that it should branch to **F1** (step 3). Upon reaching the entry block **F1**, the variable *err_flag* = 1, because it was assigned to 1 in the **GEH**, meaning that there is an error that should be fixed, thus, the control branches to **FEH_1** (step 4). Now since **F** = 1, **FEH_1** knows that it is the FEH capable of handling the CFE and, as such, sets the variable *err_flag* to 0. Finally, it searches for the basic block that has the signature equals the register S. Upon finding it, the

control branches to this basic block, i.e. **N2** in Fig. 1 (step 5). This last branch restores the control flow to the point of the program right before the occurrence of the CFE. Notice that inside all the FEH and the GEH there is the variable *num_error* counting how many times the control has passed through a FEH or GEH. This acts as a threshold for the number of how many times the correction must be attempted, which is necessary to avoid an infinite loop in case the registers **F** or **S** get corrupted for any reason. This process concludes the correction of a CFE with ACCE.

III. Fault Model and Methodology

The fault model we assume is the *single bit flip*, i.e. only one bit of a word is changed when a fault is injected. ACCE is capable of handling multiple bit flips as long as the bits flipped are within the same word. Since the fault injection, as it will be discussed later, guarantees that the injected fault ultimately turned into a manifested error it does not matter how many bits are flipped, i.e. there is no silent data corruption: faults that cause a word to change its value that does not change the behavior of the program nor its output. This could happen in the case the fault flipped the bits of a dead variable.

The ACCE technique was implemented as a transformation pass in the LLVM [6] compiler, which performs all the modifications in the control-flow graph described in section II. The ACCE transformation pass was applied *after* the set of compiler optimizations, since doing in the opposite order an optimization could invalidate the ACCE generated code and semantics.

Since ACCE is a software hardening technique to detect and correct control-flow errors, the adopted fault model simulates three distinct control flow disruptions that might occur due to a control flow error. Remind that a CFE is caused by the execution of an illegal branch to a possibly wrong address. The branch errors considered in this paper are:

1. *Branch creation*: the program counter is changed, transforming an arbitrary instruction (e.g. an addition) into an unconditional branch;
2. *Branch deletion*: the program counter is set to the next program instruction to execute independently if the current instruction is a branch;
3. *Branch disruption*: the program counter is disrupted to point to a distinct and possibly wrong destination instruction address.

We implemented a software fault injector using the GDB (Gnu Debugger) in a similar fashion as in [7], which is an accepted fault injection methodology in the embedded systems domain, in order to perform the fault injection campaigns. The steps of the fault injection process are the following:

1. The LLVM program resulting from the compilation with a set of optimization and with ACCE is translated to the assembly language of the target machine;
2. The execution trace in assembly language is extracted from the program execution with GDB;
3. A branch error (branch creation, deletion or disruption) is randomly selected. In average each branch error accounts for 1/3 of the amount of injected errors;
4. One of the instructions from the trace obtained in step 2 is chosen at random for fault injection. In this step a histogram of each instruction is computed because instructions that execute more often have a higher probability to be disrupted;
5. If the chosen instruction in step 4 executes n times, choose at random an integer number k with $1 \leq k \leq n$;
6. Using GDB, a breakpoint is inserted right before the k-th execution of the instruction selected in step 4;
7. During program execution, upon reaching the breakpoint inserted in step 6, the program counter is intentionally corrupted by flipping one of its bits to reproduce the branch error chosen in step 3;
8. The program continues its execution until it finishes.

A fault is only considered valid if it has generated a CFE, i.e. silent data corruption and segmentation faults were not considered to measure the impacts of the compiler optimizations on reliability. The experiments were performed in a 64-bit Intel Core i5 2.4 GHz desktop with 4 GB of RAM and the LLVM compiler version 2.9. For all programs versions, where each version corresponds to the program compiled with a set of optimizations plus the ACCE pass, 1,000 faults were injected using the aforementioned fault injection scheme.

In the experiments we used ten benchmark applications from the MiBench [8] embedded benchmark suite: *basicmath*, *bitcount*, 32-bit Cyclic Redundancy Check (*crc32*), *dijkstra*, Fast Fourier Transform (*fft*), *patricia*, *quicksort*, *rijndael*, *string search*, and *susan* (comprising *susan* corners, edge, and smooth). *Basicmath* performs mathematical operations, *bitcount* executes bits manipulation functions, *crc32* performs errors detection in data transmission, *dijkstra* calculates paths between pairs of nodes in a graph choosing and picks the shortest one, and *fft* performs a Fast Fourier Transform used in digital signal processing. *Patricia* is used in the network domain to represent routing tables, *quicksort* performs data sorting, *rijndael* performs cryptographic operations, s*tring search* searches for words in sentences and, finally, s*usan* is used in image processing algorithms.

IV. Impact on Reliability Caused by the Compiler

This section looks at the impacts on software reliability when an application is compiled with a set of compiler optimizations and further hardened with the ACCE method. Throughout this section the baseline for all comparisons is an application compiled with the ACCE method without any other compiler optimization. The ACCE method is a method that performs detection and correction of control-flow errors, thus all data discussed in this section considers the *correction rate* as the data to compute the efficiency metric. In this analysis we use 58 optimizations provided by the LLVM production compiler. Finally, the results were obtained using the fault model and fault injection methodology described in Section III.

The impact of the compiler optimizations when compiling for reliability is measured in this paper using the metric *Relative Improvement Percentage* (RIP) [9]. The RIP is presented in Eq. 1, where F_i is a compiler optimization, $E(F_i)$ is the error correction rate obtained for a hardened application compiled with F_i, and E_B is the error correction rate obtained for the baseline, i.e. the application compiled only with ACCE and without any optimization.

$$RIP_B(F_i) = \frac{E(F_i) - E_B}{E_B} \times 100\% \quad (1)$$

Fig. 2 shows a scatter plot of the obtained RIP for each application, with each of the 58 LLVM optimizations being a

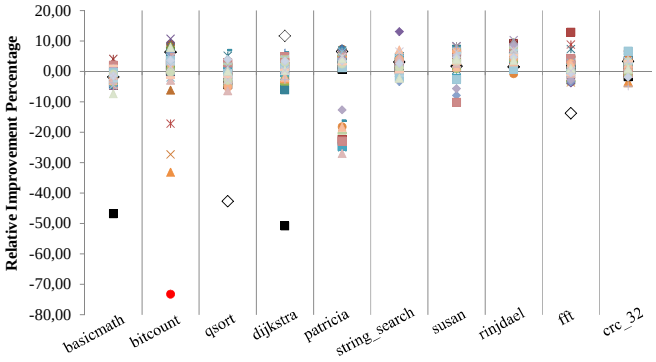

Figure 2. RIP for the error correction rate of applications hardened with ACCE under further compiler optimization. Each hardened application was compiled with a single optimization at a time, but all applications were compiled with the 58 LLVM optimizations, thus, each hardened application has 58 versions. The baseline (RIP = 0%) is the error correction rate of the hardened application compiled without any LLVM optimization.

point in the y-axis. Each point represents the hardened application compiled with a single LLVM optimization at a time. Thus, for each application have 58 different versions (points in the chart). Fig. 2 shows that several optimizations increase the RIP considerably, sometimes reaching a RIP of ~10% above the baseline. This is a great result, which shows that reliability can be increased for free just picking appropriate optimizations that facilitates for ACCE the process of error detection and correction. However, we also see that some optimizations totally jeopardize reliability, reaching a RIP of −73.27% (bottom filled circle for *bitcount*).

It is also possible to gather evidence that the structure of the application also influences how an optimization impacts on the RIP of reliability. Let us consider the *block-placement* optimization, which is represented by the white diamond in Fig. 2. In the case of the *qsort* application, *block-placement* has a RIP of −42.75% and a RIP of +11.68%. It can be noticed that other optimizations also have this behavior (increasing RIP for some applications and decreasing it for others). It also happens that some hardened applications are less sensitive to compiler optimizations, as it is the case of the *crc_32* one, where the RIP is within the ± 5% interval around the baseline.

Usually compiler optimizations are applied in bulk, using several of them during compilation. Therefore, it is important to also examine if successive optimization passes compromise or increase software reliability of a hardened application. Fig. 4 presents the error correction rate RIP where the hardened application was compiled with a subset of the 58 LLVM optimizations. In this experiment we used six sizes of subsets of optimizations: 10, 20, 30, 40, 50, and 58. The RIP shown in Fig. 3 is the average RIP of five random subsets, i.e. it is an average of distinct subsets of the same size. Taking the average and picking the optimizations at random reproduces the effects of indiscriminately picking the compiler optimizations or, at least, choosing optimizations with the object of optimizing performance without previous knowledge of how the chosen optimizations influence together the software reliability.

It is possible to see that the cumulative effect of compiler optimizations in the error correction RIP is in most of the cases deleterious, but for a few exceptions. Fig. 3 confirms that some applications are less sensitive to the effects of compiler optimizations, e.g. the *crc32* has its RIP within the [−1.11%, 0.73%] around the baseline. On the other hand, *basicmath*, *bitcount*, and *patricia* are jeopardized. Interesting to notice that

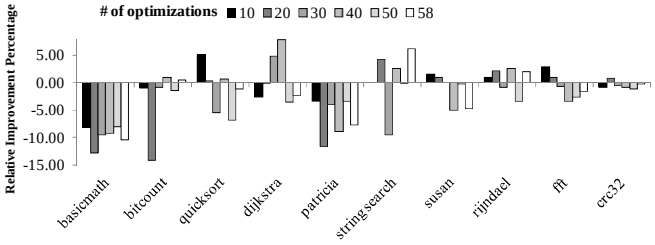

Figure 3. Relative Improvement Percentage of random subsets of the 58 LLVM optimizations with a varying number of optimizations for each different subset: 10, 20, 30, 40, 50, 58 optimizations. The RIP for each subset was measured taking the average of 6 random subsets for each subset size. Hence, distinct possible optimizations subsets were considered. The baseline (RIP = 0%) is the error correction rate of the hardened application compiled without any LLVM optimization.

the RIP in case of picking a subset of optimizations is not subject to the much severe reduction that was measured when only a single optimization was used (Fig. 2), evidencing that the composition of distinct optimization might be beneficial for reliability because the composition cancel the bad effects of individual optimizations.

V. Conclusions and Future Work

In this paper we have shown how the reliability of radiation hardened software is affected by compiler optimizations. This study is important for aerospace systems because the compiler has great importance in software production, being, as supported by the experiments presented herein, a non-negligible source of noise. Thus, compiler optimizations cannot be applied without a careful planning if the software is going to be hardened against radiation. As far as we are aware, this is the first study that measures this impact. Results show that reliability can be enhanced for free, just by picking the appropriate optimizations for a given application. However, it is not clear *why* some applications benefit more when they are optimized for performance and, subsequently hardened against radiation. Therefore, as future work we are studying what characteristics benefit radiation hardening, and how those structures can be automatically identified in the compiler.

Rereferences

[1] ITRS. ITRS 2009 Roadmap. Technical report, Int. Technology Roadmap for Semiconductors, 2009.

[2] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. 2005. Micro, 25(6):10–16, Nov.

[3] E. Normand. Single event upset at ground level. 1996. IEEE Trans. on Nuclear Science, 43(6): 2742–2750, Dec.

[4] P. Rech et al. Neutron-induced soft-errors in graphic processing units. 2012. In IEEE Radiation Effects Data Work. (REDW ’12) IEEE, 6 pp.

[5] R. Vemu, S. Gurumurthy, and J. Abraham. ACCE: Automatic correction of control-flow errors. In ITC ’07. IEEE Int. Test Conf., p. 1–10, 2007.

[6] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In CGO ’04: Proc. of the int. symp. on Code generation and optimization, pages 75–, USA, 2004. IEEE.

[7] N. Krishnamurthy, V. Jhaveri, and J. A. Abraham. A design methodology for software fault injection in embedded systems. In DCIA ’98: Proc. Workshop on Dependable Computing and its appl., IFIP.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In WWC-4. ‘01: Proc. of the IEEE Int. Workshop of Workload Characterization, 3–14, 2001. IEEE.

[9] Zhelong Pan and Rudolf Eigenmann. 2006. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In Proceedings of the International Symposium on Code Generation and Optimization (CGO '06). IEEE, 319-332.

Compiler Optimizations Do Impact the Reliability of Control-Flow Radiation Hardened Embedded Software

Rafael B. Parizi, Ronaldo R. Ferreira, Luigi Carro, and Álvaro F. Moreira

Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
{rbparizi, rrferreira, carro, afmoreira}@inf.ufrgs.br

Abstract. This paper characterizes how compiler optimizations impact software control-flow reliability when the optimized application is compiled with a technique to enable the software itself to detect and correct radiation induced soft-errors occurring in branches. Supported by a comprehensive fault injection campaign using an established benchmark suite in the embedded systems domain, we show that the careful selection of the available compiler optimizations is necessary to avoid a significant decrease of software reliability while sustaining the performance boost those optimizations provide.

Keywords: compiler optimization, compiler orchestration, embedded systems, fault tolerance, LLVM, radiation, reliability, soft errors, tuning.

1 Introduction

Compiler optimizations are taken for granted in modern software development, enabling applications to execute more efficiently in the target hardware architecture. Modern architectures have complex inner structures designed to boost performance, and if the software developer were to be aware of all those inner details, performance optimization would jeopardize the development processes. Compiler optimizations are transparent to the developer, who picks the appropriate ones to the results s/he wants to achieve, or, as it is more common, letting this task to the compiler itself by flagging if it should be less or more aggressive in terms of performance.

Industry already offers microprocessors built with 22 nm transistors, with a prediction that transistor's size will reach 7.4 nm by 2014 [1]. This aggressive technology scaling creates a big challenge concerning the reliability of microprocessors using newest technologies. Smaller transistors are more likely to be disrupted by transient sources of errors caused by radiation, known as *soft-errors* [2]. Radiation particles originated from cosmic rays when striking a circuit induce bit flips during software execution, and since transistors are becoming smaller there is a higher probability that transistors will be disrupted by a single radiation particle with smaller transistors requiring a smaller amount of charge to disrupt their stored logical value. The newest technologies are so sensitive to radiation that their usage will be compromised even at the sea level, as predicted in the literature [3]. In [4] it is shown that modern 22nm GPU cards are susceptible to such an error rate that makes their usage unfeasible in

critical embedded systems. However, industry is already investing in GPU architectures as the platform of choice for high performance and low power embedded computing, such as the ARM Mali® embedded GPU [5].

The classical solution to harden systems against radiation is the use of *spatial redundancy*, i.e. the replication of hardware modules. However, spatial redundancy is prohibitive for embedded systems which usually cannot afford extra costs of hardware area and power. The increase on power is a severe problem, because it is expected that 21% of the entire chip area must be turned off during its operation to meet the available power budget, and an impressive chip area of 50% at 8 nm [6]. This creates the *dark silicon* problem [6]: a huge area of the circuit cannot be used during its lifecycle. This problem gets worse when the microprocessor has redundant units, because system's reliability could be compromised if redundant units were turned off. The current solution to this problem is to use radiation hardened microprocessors, which are designed to endure radiation. The problem with this approach is the low availability and high pricing of those radiation hardened components. For instance, a 25 MHz microprocessor has a unitary price of U$ 200,000.00 [7]. This high pricing makes the use of radiation hardened microprocessors unfeasible for embedded systems used in aircrafts, not to say about cars and low-end medical devices such as pacemakers. For these critical embedded systems where cost is the major constraint a cheaper but yet effective approach for reliability against radiation is necessary.

Software-Implemented Hardware Fault-Tolerance (SIHFT) [8] is an approach for radiation reliability that adds redundancy in terms of extra instructions or data to the application, keeping the hardware unchanged. SIHFT techniques work by modifying the original program by adding *checking mechanisms* to it. SIHFT are classified either as *control-flow* or as *data-flow*. The former is designed to detect when an illegal jump has occurred during application execution to possibly proceed with the resolution of the correct jump address or at least signaling that such an error has occurred. The latter checks if a data variable being read is correct or not. While the effects of data-flow SIHFT methods are clear (usually the duplication of program variables or the addition of variable checksums solve the problem), the impacts of the control-flow ones is yet not well understood. Because control-flow methods modify the program's control-flow graph (CFG), which happens to be the same artifact used by compiler optimizations, the efficiency of control-flow reliability techniques might be influenced by the optimizations in an unpredictable way.

In this paper we evaluate how the cumulative usage of compiler optimizations influence reliability of applications hardened with the state-of-the-art *Automatic Correction of Control-flow Errors* (ACCE) [9] control-flow SIHFT technique, which was chosen because it is the current most efficient method in terms of reliability, attaining an error correction rate of ~70%. The application set we use in this paper is drawn from the MiBench [10] suite. For the sake of clarity, the ACCE technique is briefly reviewed in Section 2. Section 3 presents the fault model we assume and the methodology used in this paper. Finally, Section 4 presents the impact of individual and cumulative optimization passes using the LLVM [11] as the production compiler.

2 Automatic Correction of Control-flow Errors

ACCE [9] is a software technique for reliability that detects and corrects control-flow errors (CFE) due to random and arbitrary bit flips that might occur during software execution. The hardening of an application with ACCE is done at compilation, since it is implemented as a transformation pass in the compiler. ACCE modifies the applications' basic blocks with the insertion of extra instructions that perform the error detection and correction during software execution. In this section we briefly explain how ACCE works in two separate subsections, one dedicated to error detection and the other to error correction in the subsections 2.1 and 2.2, respectively. The reader should refer to the ACCE article for a detailed presentation and experimental evaluation [9]. The fault model that ACCE assumes is further described in Section 3.

2.1 Control-Flow Error Detection

ACCE performs online detection of CFEs by checking the signatures in the beginning and in the end of each basic block of the control-flow graph, thus, ACCE is classified as a *signature checking* SIHFT technique as termed in the literature. The basic block signatures are computed and generated during compilation; the signature generation is critical because it needs to compute non-aliased signatures between the basic block, i.e. each block must be unambiguously identified. In addition, for each basic block found in the CFG two additional code regions are added, the *header* and the *footer*. The signature checking during execution takes place inside these code regions. Fig. 1 shows two basic blocks (labeled as **N2** and **N6**) with the additional code regions. The top region corresponds to the header and the bottom to the footer. Still at compilation ACCE creates for each function in the application two additional blocks, the function entry block and the *Function Error Handler* (FEH). For instance, Fig. 1 depicts a portion of two functions, *f1* and *f2*, both owning *entry blocks* labeled as **F1** and **F2**, and function error handlers, labeled as **FEH_1** and **FEH_2**, respectively. Finally, ACCE creates a last extra block, the *Global Error Handler* (GEH), which can only be reached from a FEH block. The role of these blocks will be presented soon.

At runtime ACCE maintains a global *signature register* (represented as **S**), which is constantly updated to contain the signature of the basic block that the execution has reached. Therefore, during the execution of the *header* and *footer* code regions of each basic block, the value of the signature register is compared with the signatures generated during compilation for those code regions and, if those values do not match, a control flow error has just been detected and the control should be transferred to the corresponding FEH block of the function where execution currently is at. ACCE also maintains the *current function* register (represented as **F**), which stores the unique identifier of the function currently being executed. The current function register is only assigned at the extra entry function block. This process encompasses the *detection* of an illegal and erroneous due to a soft error.

Fig. 1 depicts an example of the checking and update of signatures performed in execution time that occurs in a basic block. In this example, the control-flow error occurs in the block **N2** of function **F1**, where an illegal jump incorrectly transfers the

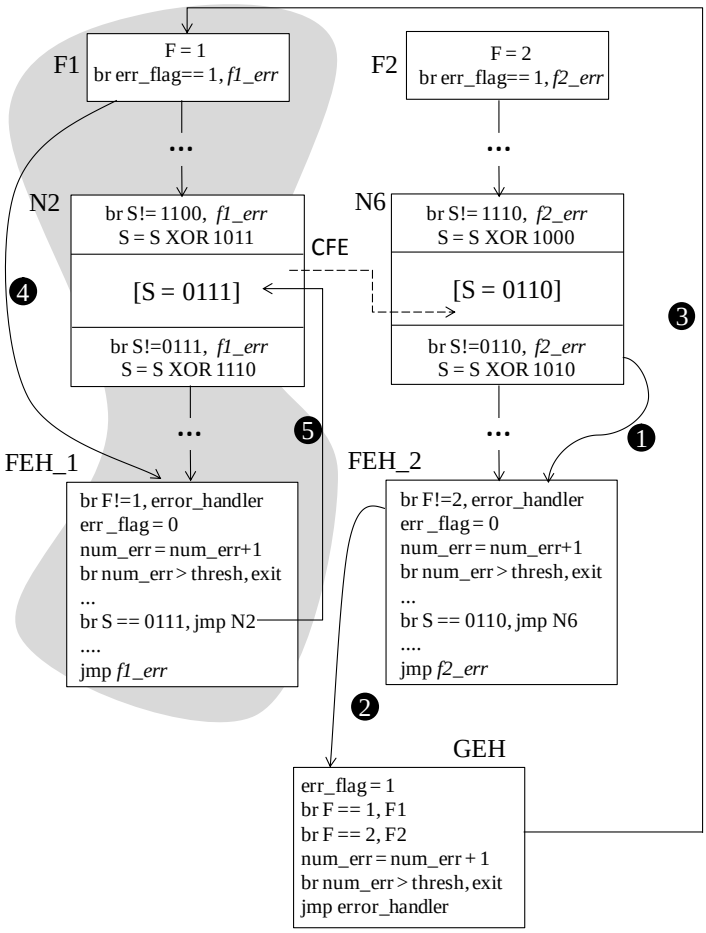

Fig. 1. Depiction of how the control is transferred from a function to the basic blocks that ACCE has created when a control-flow error occurs during software execution. In this figure, there is a control flow error (dashed arrow) causing the execution to jump from the block N2 of function F1 to the block N6 of function F2.

control flow to the basic block **N6** of function **F2**. When the execution reaches the footer of the block **N6** the signature register **S** is checked against the signature generated at compilation. In this case, **S** = 0111 (i.e. the previous value assigned in the header of the block **N2**). Thus, the branch test in the **N6** footer will detect that the expected signature does not match with the value of **S**, and, thus, the CFE error must signaled (step 1 in Fig. 1). In this example, the application branches to the address *f2_err*, making the application enter the **FEH_2** block (since the error was detected by a block owned by the function **F2**, the function error handler invoked is the **FEH_2**). At this point, the CFE was detected and ACCE can proceed with the correction of the detected CFE.

2.2 Control-Flow Error Correction

The correction process starts as soon as an illegal jump is detected by the procedure described in subsection 2.1, with the control flow transferred to the FEH corresponding to the function where the CFE was found. The FEH checks if the illegal jump was originated in the function it is responsible to handle its detected errors by comparing the value of the function's identifier (**F1** or **F2**, in the example of Fig. 1) with the current function register **F**. If the error happened in the function stored in the **F** register, FEH evaluates the current value of the signature register and then transfers the control to the basic block that is the origin of the illegal jump (this origin is stored in the **S** register). On the other hand, if the illegal jump was not originated in the function where the detection has occurred, the FEH then transfers the control flow to the GEH. In this case, the GEH is responsible for identifying the function where the CFE has occurred and to transfer the control flow back to this function, so that the error is correctly treated by the function's FEH. The GEH searches the function where the error has occurred and transfers the control to its entry block, which will then sends the control flow to the proper FEH so that the error can be corrected, i.e. branching the control to the basic block where the CFE has occurred.

Recalling the example depicted in Fig. 1, after the CFE is detected and the control is transferred to **FEH_2** (step 1) the **F** register is matched against the function identifier of the function from where the control came. However, since the CFE originated in the basic block **N2** of function **F1**, **F** = 1. Therefore, **FEH_2** is not capable of finding the basic block where the CFE originated, and then it transfers the control to the **GEH** so that the correct FEH can be found (step 2). The **GEH** searches for the function identifier stored in **F**, until it finds that it should branch to **F1** (step 3). Upon reaching the entry block **F1**, the variable *err_flag* = 1, because it was assigned to 1 in the **GEH**, meaning that there is an error that should be fixed, thus, the control branches to **FEH_1** (step 4). Now since **F** = 1, **FEH_1** knows that it is the FEH capable of handling the CFE and, as such, sets the variable *err_flag* to 0. Finally, it searches for the basic block that has the signature equals the register S. Upon finding it, the control branches to this basic block, i.e. **N2** in Fig. 1 (step 5). This last branch restores the control flow to the point of the program right before the occurrence of the CFE. Notice that inside all the FEH and the GEH there is the variable *num_error* counting how many times the control has passed through a FEH or GEH. This acts as a threshold for the number of how many times the correction must be attempted, which is necessary to avoid an infinite loop in case the registers **F** or **S** get corrupted for any reason. This process concludes the correction of a CFE with ACCE.

3 Fault Model and Experimental Methodology

The fault model we assume in the experiments is the *single bit flip*, i.e. only one bit of a word is changed when a fault is injected. ACCE is capable of handling multiple bit flip as long as the bits flipped is within a same word. Since the fault injection, as it will be discussed later, guarantees that the injected fault ultimately turned into a manifested error it does not matter how many bits are flipped, i.e. there is no silent data

corruption: faults that cause a word to change its value that does not change the behavior of the program nor its output. This could happen in the case the fault flipped the bits of a dead variable.

The ACCE technique was implemented as a transformation pass in the LLVM [11] production compiler, which performs all the modifications in the control-flow graph described in section 2 using the LLVM Intermediate Representation (LLVM-IR). The ACCE transformation pass was applied *after* the set of compiler optimizations, since doing in the opposite order a compiler optimization could invalidate the ACCE generated code and semantics.

Since ACCE is a SIHFT technique to detect and correct control-flow errors, the adopted fault model simulates three distinct control flow disruptions that might occur due to a control flow error. Remind that a CFE is caused by the execution of an illegal branch to a possibly wrong address. The branch errors considered in this paper are:

1. *Branch creation*: the program counter is changed, transforming an arbitrary instruction (e.g. an addition) into an unconditional branch;
2. *Branch deletion*: the program counter is set to the next program instruction to execute independently if the current instruction is a branch;
3. *Branch disruption*: the program counter is disrupted to point to a distinct and possibly wrong destination instruction address.

We implemented a software fault injector using the GDB (GNU Debugger) in a similar fashion as [12], which is an accepted fault injection methodology in the embedded systems domain, in order to perform the fault injection campaigns. The steps of the fault injection process are the following:

1. The LLVM-IR program resulting from the compilation with a set of optimization and with ACCE is translated to the assembly language of the target machine;
2. The execution trace in assembly language is extracted from the program execution with GDB;
3. A branch error (branch creation, deletion or disruption) is randomly selected. In average each branch error accounts for 1/3 of the amount of injected errors;
4. One of the instructions from the trace obtained in step 2 is chosen at random for fault injection. In this step a histogram of each instruction is computed because instructions that execute more often have a higher probability to be disrupted;
5. If the chosen instruction in step 4 executes n times, choose at random an integer number k with $1 \leq k \leq n$;
6. Using GDB, a breakpoint is inserted right before the k-th execution of the instruction selected in step 4;
7. During program execution, upon reaching the breakpoint inserted in step 6, the program counter is intentionally corrupted by flipping one of its bits to reproduce the branch error chosen in step 3;
8. The program continues its execution until it finishes.

A fault is only considered valid if it has generated a CFE, i.e. silent data corruption and segmentation faults were not considered to measure the impacts of the compiler optimizations on reliability. All the experiments in this paper were performed in a 64-bit Intel Core i5 2.4 GHz desktop with 4 GB of RAM and the LLVM compiler version 2.9. For all programs versions, where each version corresponds to the program compiled with a set of optimizations plus the ACCE pass, 1,000 faults were injected using the aforementioned fault injection scheme. In the experiments we considered ten benchmark applications from the MiBench [10] embedded benchmark suite: *basicmath*, *bitcount*, *crc32*, *dijkstra*, *fft*, *patricia*, *quicksort*, *rijndael*, *string search*, and *susan* (comprising *susan* corners, edge, and smooth).

4 Impact of Compiler Optimizations on Control-Flow Reliability of Embedded Software

This section looks at the impacts on software reliability when an application is compiled with a set of compiler optimizations and further hardened with the ACCE method. Throughout this section the baseline for all comparisons is an application compiled with the ACCE method without any other compiler optimization. ACCE performs detection and correction of control-flow errors, thus all data discussed in this section considers the *correction rate* as the data to compute the efficiency metric. In this analysis we use 58 optimizations provided by the LLVM production compiler. Finally, the results were obtained using the fault model and fault injection methodology described in section 3.

The impact of the compiler optimizations when compiling for reliability is measured in this paper using the metric *Relative Improvement Percentage* (RIP) [13]. The RIP is presented in Eq. 1, where F_i is a compiler optimization, $E(F_i)$ is the error correction rate obtained for a hardened application compiled with F_i, and E_B is the error correction rate obtained for the baseline, i.e. the application compiled only with ACCE and without any optimization.

$$RIP_B(F_i) = \frac{E(F_i) - E_B}{E_B} \times 100\% \tag{1}$$

Fig. 2 shows a scatter plot of the obtained RIP for each application, with each of the 58 LLVM optimizations being a point in the y-axis. Each point represents the hardened application compiled with a single LLVM optimization at a time. Thus, for each application have 58 different versions (points in the chart). Fig. 2 shows that several optimizations increase the RIP considerably, sometimes reaching a RIP of ~10%. This is a great result, which shows that reliability can be increased for free just picking appropriate optimizations that facilitates for ACCE the process of error detection and correction. However, we also see that some optimizations totally jeopardize reliability, reaching a RIP of −73.27% (bottom filled red circle for *bitcount*).

It is also possible to gather evidence that the structure of the application also influences how an optimization impacts on the RIP of reliability. Let us consider the *block-placement* optimization, which is represented by the white diamond in Fig. 2. In

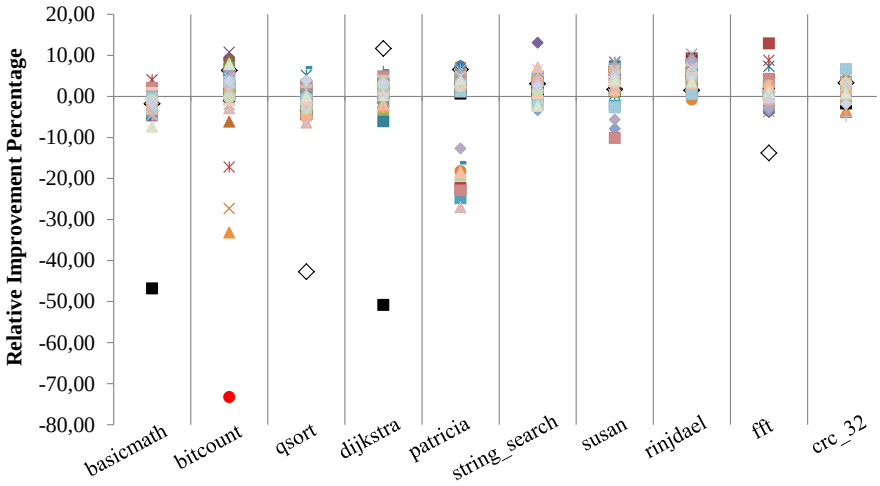

Fig. 2. Relative Improvement Percentage for the error correction rate of applications hardened with ACCE under further compiler optimization. Each hardened application was compiled with a single optimization at a time, but all applications were compiled with the 58 LLVM optimizations, thus, each hardened application has 58 versions. The baseline (RIP = 0%) is the error correction rate of the hardened application compiled without any LLVM optimization. Each point in the chart represents the application with one optimization protected with ACCE.

the case of the *qsort* application, *block-placement* has a RIP of −42.75% and a RIP of +11.68%. The reader can notice that other optimizations also have this behavior (increasing RIP for sovme applications and decreasing it for others). It also happens that some hardened applications are less sensitive to compiler optimizations, as it is the case of the *crc_32* one, where the RIP is within the ± 5% interval around the baseline.

Fig. 3 depicts the RIP of a selected subset of the 58 LLVM optimizations, making it clear that even within a small subset the variation in RIP for reliability is far from

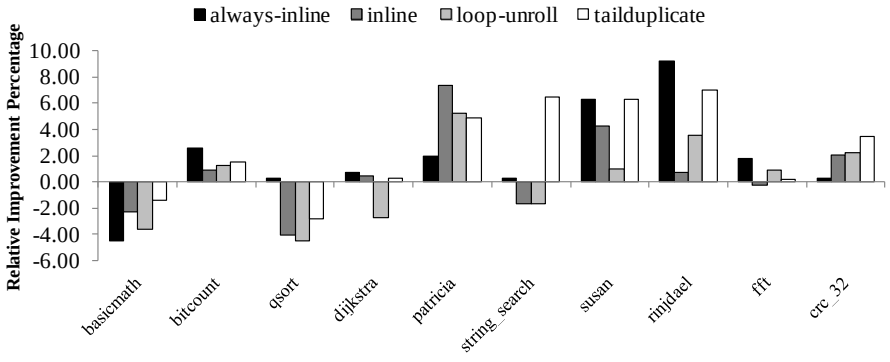

Fig. 3. Relative Improvement Percentage of a selected subset of the 58 LLVM optimizations. The baseline (RIP = 0%) is the error correction rate of the hardened application compiled without any LLVM optimization.

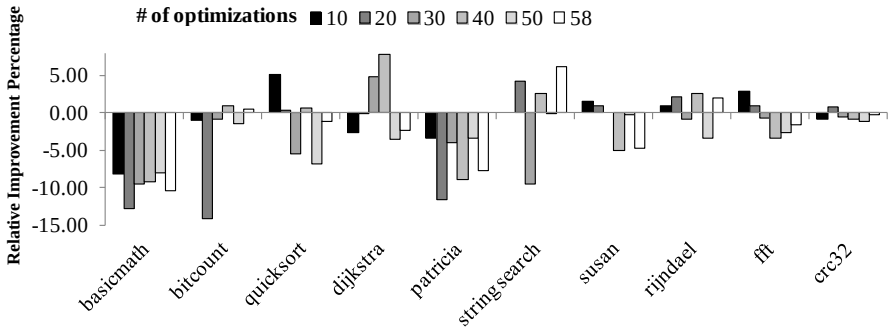

Fig. 4. Relative Improvement Percentage of random subsets of the 58 LLVM optimizations with a varying number of optimizations for each different subset: 10, 20, 30, 40, 50, 58 optimizations. The RIP for each subset was measured taking the average of 6 random subsets for each subset size. Hence, distinct possible optimizations subsets were considered. The baseline (RIP = 0%) is the error correction rate of the hardened application compiled without any LLVM optimization.

negligible. For instance, the *always-inline* LLVM optimization has an error correction RIP interval of [−4.55%, +9.24%].

Usually compiler optimizations are applied in bulk, using several of them during compilation. Therefore, it is important to also examine if successive optimization passes could compromise or increase software reliability of a hardened application. Fig. 4 presents the error correction rate RIP where the hardened application was compiled with a subset of the 58 LLVM optimizations. In this experiment we used six sizes of subsets: 10, 20, 30, 40, 50, and 58. The RIP shown in Fig. 4 is the average of five random subsets, i.e. it is an average of distinct subsets of the same size. Taking the average and picking the optimizations at random reproduces the effects of indiscriminately picking the compiler optimizations or, at least, choosing optimizations with the object of optimizing performance without previous knowledge of how the chosen optimizations influence together the software reliability.

It is possible to see that the cumulative effect of compiler optimizations in the error correction RIP is in most of the cases deleterious, but for a few exceptions. Fig. 4 confirms that some applications are less sensitive to the effects of compiler optimizations, e.g. the *crc32* has its RIP within the [−1.11%, 0.73%]. On the other hand, *basicmath*, *bitcount*, and *patricia* are jeopardized. Interesting to notice that the RIP in case of picking a subset of optimizations is not subject to the much severe reduction that was measured when only a single optimization was used (Fig. 2), evidencing that the composition of distinct optimization may be beneficial for reliability.

Based on the data and experiments discussed in this section it is clear that choosing of compiler optimizations requires the software designer to take into consideration that some optimizations may not be adequate in terms of reliability for a given application. Moreover, data shows that a given optimization is not only by itself a source of reliability reduction; reliability is also dependent of the application being hardened and how a given optimization facilitates or not the work of the ACCE technique.

5 Related Work

Much attention has been devoted to the impact of compiler optimizations on program performance in the literature. However, the understanding of how those optimizations work together and how they influence each other is a rather recent research topic. The *Combined Elimination* (CE) [13] is an analysis approach to identify the best sequence of optimizations of for a given application set using the GCC compiler. The authors discuss that simple *orchestration* schemes between the optimizations can achieve near-optimal results as if it was performed an exhaustive search in all the design space created by the optimizations. CE is a greedy approach that firstly compiles the programs with a single optimization, using this version as the baseline. From those baseline versions the set of *Relative Improvement Percentage* (RIP) is calculated, which is the percentage that the program's performance is reduced/increased (section 4 discussed RIP in details). With the RIP at hand for all baselines, the CE starts removing the optimizations with negative RIP, until the total RIP of all optimizations applied into a program do not reduce. CE was evaluated in different architectures, achieving an average RIP of 3% for the SPEC2000, and up to 10% in case of the Pentium IV for the floating point applications.

The *Compiler Optimization Level Exploration* (COLE) [14] is another approach to achieve performance increase by selecting a proper optimization sequence. COLE uses a population-based multi-objective optimization algorithm to construct a Paretto optimal set of optimizations for a given application using the GCC compiler. The data found with COLE give some insightful results about how the optimization. For instance, 25% of the GCC optimizations appear in at most one Paretto set, and some of them appear in all sets. Therefore, 75% of all optimizations do not contribute to improve the performance, meaning that they can be safely ignored! COLE also shows that the quality of an optimization is highly tied with the application set.

The *Architectural Vulnerability Factor* (AVF) [15] is a metric to estimate the probability that the bits in a given hardware structure will be corrupted by a soft-error when executing a certain application. The AVF is calculated as the total time the vulnerable bits remains in the hardware architecture. For example, the register file has a 100% AVF, because all of its bits are vulnerable in case of a soft-error. This metric is influence by the application due to liveness: for instance, a dead variable has a 0% AVF because it is not used in a computation. The authors in [16] evaluate the impact of the GCC optimizations in the AVF metric by trying to reduce the *AVF-delay-square-product* (ADS) introduced by the authors. The ADS relates considers a linear relation of the AVF between the square of the performance in cycles, clearly prioritizing performance over reliability. It is reported that the –O3 optimization level is detrimental both to the AVF and performance, because for the benchmarks considered (MiBench) have increased the number of loads executed. Again, the *patricia* application was the one with the highest reduction in the AVF at 13%.

In [17] the authors analyze the impact of compiler optimizations on data reliability in terms of variable liveness. *Liveness* of a variable is the time period between the variable is written and it is last read before a new write operation. The authors conclude that the liveness is not related only with the compiler optimization, but it also

depends on the application being compiled, which is in accordance with the discussion we made in section 4. The paper shows that some optimizations tend to extend the time a variable is stored in a register instead of memory. The goal behind this is obvious: it is much faster to fetch the value of a variable when it is in the register than in memory. However, the memory is usually more protected than registers because of cheap and efficient Error Correction Code (ECC) schemes, and, thus, thinking about reliability it is not a good idea to expose a variable in a register for a longer time. The solution to that could be the application of ECC such as Huffman to the program variables itself. Decimal Hamming (DH) [18] is a software technique that does that for a class of programs where the program's output is a linear function of the input. The generalization of efficient data-flow SIHFT techniques such as DH (i.e. ECC of program variables) is still an open research problem.

6 Conclusions and Future Work

In this paper we characterized the problem of compiling embedded software for reliability, given that compiler optimizations do impact the coverage rate. The study presented in this paper makes clear that choosing optimizations indiscriminately can decrease software reliability to unacceptable levels, probably avoiding the software to be deployed as originally planned. Embedded software and systems deployed in space applications must always be certified evidencing that they support harsh radiation environments, and given the increasing technology scaling, other safety critical embedded systems might have to tolerate radiation induced errors in a near future. Therefore, the embedded software engineer must be very careful when compiling safety critical embedded software.

Design space exploration (DSE) for embedded systems usually considers "classical" non-functional requirements, such as energy consumption and performance. However, this paper has shown the need for automatic DSE methods to consider reliability when pruning the design space of feasible solutions. This could be realized with the support of compiler orchestration during the DSE step. As future work we are studying how to efficiently extend automatic DSE algorithms to implement compiler orchestration for reliability against radiation induced errors.

Acknowledgments

This work is supported by CAPES foundation of the Ministry of Education, CNPq research council of the Ministry of Science and Technology, and FAPERGS research agency of the State of Rio Grande do Sul, Brazil. R. Ferreira was supported with a doctoral research grant from the Deutscher Akademischer Austauschdienst (DAAD) and from the Fraunhofer-Gesellschaft, Germany.

References

1. ITRS. ITRS 2009 Roadmap. *International Technology Roadmap for Semiconductors, Tech. Rep*, 2009.
2. S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. 2005. *Micro*, 25(6):10–16, Nov.
3. E. Normand. Single event upset at ground level. 1996. *IEEE Trans. on Nuclear Science*, 43(6): 2742–2750, Dec.
4. P. Rech et al. Neutron-induced soft-errors in graphic processing units. 2012. In *IEEE Radiation Effects Data Workshop.* (REDW '12) IEEE, 6 pp.
5. ARM Mali Graphics Hardware, http://www.arm.com/products/multimedia/mali-graphics-hardware/index.php.
6. H. Esmaeizadeh, et al, "Dark silicon and the end of multicore scaling," in ISCA '11: Proc. of the 38th Int. Symp. on Comp. Arch., 2011, 365–376.
7. P. C. Mehlitz and J. Penix. Expecting the unexpected – radiation hardened software. 2005. In Infocom @ American Inst. of Aeronautics and Astronautics.
8. O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, Software-Implemented Hardware Fault Tolerance. New York, NY, USA: Springer, 2006.
9. R. Vemu, S. Gurumurthy, and J. Abraham. ACCE: Automatic correction of control-flow errors. In *ITC '07. IEEE Int. Test Conf*., pages 1–10, 2007.
10. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC-4. '01: Proc. of the IEEE Int. Workshop of Workload Characterization*, 3–14, 2001. IEEE.
11. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proc. of the int. symp. on Code generation and optimization*, pages 75–, Washington, DC, USA, 2004. IEEE.
12. N. Krishnamurthy, V. Jhaveri, and J. A. Abraham. A design methodology for software fault injection in embedded systems. In *DCIA '98: Proc. of the Workshop on Dependable Computing and its applications*, IFIP.
13. Zhelong Pan and Rudolf Eigenmann. 2006. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In Proceedings of the International Symposium on Code Generation and Optimization (CGO '06). IEEE, 319-332.
14. Kenneth Hoste and Lieven Eeckhout. 2008. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (CGO '08). ACM, 165-174.
15. S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. of the 36th annual IEEE/ACM Int. Symp. on Microarchitecture*, (MICRO 36). IEEE, 29–41.
16. T. M. Jones, M.F. P O'Boyle, O. Ergin, 2008. Evaluating the Effects of Compiler Optimisations on AVF. In *Workshop on Interaction Between Compilers and Computer Architecture* (INTERACT-12).
17. S. Bergaoui and R. Leveugle. 2011. Impact of Software Optimization on Variable Lifetimes in a Microprocessor-Based System. In Proceedings of the 2011 Sixth IEEE International Symposium on Electronic Design, Test and Application (DELTA '11). 56-61.
18. C. Argyrides, R. Ferreira, C. Lisboa, and L. Carro. Decimal hamming: a novel software-implemented technique to cope with soft errors. In *Proc. of the 26th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotech. Sys*., DFT '11. IEEE, 2011, 11-17.