PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE ENGENHARIA

CURSO DE ENGENHARIA ELÉTRICA

MONITORAMENTO DO FLUXO DE CONTROLE DE PROCESSADORES EMBARCADOS BASEADO EM PROFILING DE SOFTWARE

CLAUDIA ANTUNES ROCHA

PROF. DR. FABIAN VARGAS

Porto Alegre, Fevereiro de 2007.

Dedico toda a alegria e vitória desta etapa acadêmica à minha Mãe Marlene,
presença constante Maternal e Espiritual, em toda a minha trajetória acadêmica e de vida.
Abdicou de um futuro nas áreas de línguas e artes para se dedicar para a família e minha infância. Hoje poderia estar formando-se ao meu lado, ou melhor,
está, pois divido e dedico a Ti esta nova conquista!

Agradecimentos

À Deus e aos Amigos Protetores que em seu Nome estão sempre ao nosso lado, trazendo orientação e proteção.

Ao meu orientador Prof. Fabian Vargas, por ter me acolhido no SiSC e pela oportunidade de trabalho e aprendizado numa área que sempre me agradou, embora não tivesse nela atuado.

Aos meus Pais Daltro e Marlene, ao meu irmão Datinho, aos meus pais postiços Nelson e Eliane, e manos Kelen, Rodrigo, Hellen e a toda família de Camaquã pelo apoio, torcida, carinho recebidos, que transpuseram o reduzido convívio e ausências nestes dois anos.

Agradecimento especial ao Luciano pelo amor, apoio, companheirismo, nestes dois anos em que foi colega de pós além de esposo, dividindo toda a rotina de estudos e longos períodos de permanência na PUC.

À todos os colegas e Amigos conquistados no SiSC, que reberam uma professora de Matemática como colega, sempre dispostos a me ajudar a ingressar e conviver na sua área de competência, companheiros de "incríveis tarefas", de chimarrão e café diários, sempre num clima agradável, alegre, compartilhando conhecimentos.

Em particular ao Antônio, pelo aprendizado no uso dos kits de desenvolvimento para os testes, e por dividir seus conhecimentos e implementações para a criação do programa injetor.

Ao CNPq pelo apoio financeiro que permitiu meu ingresso no PPGEE.

RESUMO

Nos últimos anos, observa-se com grande euforia o crescimento do mercado de sistemas embarcados nas áreas econômico-sociais de grande importância, tais como a saúde, telecomunicações, automotiva e aeroespacial, entre outras. Como conseqüência, exige-se maior robustez tanto do *hardware* quanto do *software* integrante destes sistemas, além de componentes de baixo custo, principalmente memória. Dentre os tipos possíveis de falhas, as falhas que alteram o fluxo de controle de processadores que executam aplicações embarcadas, por implicarem em quase sempre em falhas catastróficas do sistema, são focadas nesta dissertação. Por falhas catastróficas, entende-se como sendo aquelas falhas que além de induzir o sistema a produzir um comportamento diferente daquele esperado para a sua função, implicam na maioria das vezes também na reinicialização do sistema como forma de recuperação da falha.

Assim, a utilização de técnicas capazes de detectar estes tipos de falhas evita que as mesmas se propaguem pelo sistema e acabem gerando saídas incorretas, pois tais falhas podem ser catastróficas para a segurança dos usuários e para a imagem e reputação das empresas. Porém, a utilização de técnicas de detecção de falhas gera um aumento na taxa de ocupação de memória do sistema, bem como provoca aumento da degradação de desempenho, o que pode ser considerado um fator crítico tratando-se de aplicações embarcadas de tempo-real.

Como alternativa para minimizar estes fatores, três hipóteses foram investigadas, sendo uma delas implementada. Assim, nesta dissertação propõe-se uma abordagem baseada em *software profiling* que analisa o grafo de fluxo de controle da aplicação, visando à otimização do número de assinaturas (*checkpoints*) a serem inseridas no código-fonte.

Para validar a abordagem proposta, foi realizada por simulação a injeção de três tipos de falhas: *jump, nop e bit-flip*, sobre diferentes programas aplicativos. Este processo de injeção de falhas foi acelerado via prototipagem do sistema em hardware, através do uso de um FPGA (*Field-Programmable Gate Array*) em uma placa comercial da Xilinx. A análise dos resultados obtidos indica que a técnica proposta reduz o número de assinaturas inseridas no código da aplicação, e portanto, minimizando o *overhead* de memória e a degradação do *desempenho* do sistema, ao passo que mantém aproximadamente inalterado nível de cobertura de falhas quando comparada a outras técnicas atualmente existentes na literatura.

ABSTRACT

In the recent years, the society observes with enthusiasm the rapid proliferation of a vast diversity of embedded systems targeted to safe-critical applications like health-care systems, telecommunication, automotive and aerospace. As a consequence, besides the need for low-cost components, mainly memory, it is also mandatory the use of more robustness hardware and software parts that integrate these systems. Among the possible types of faults, those that change the control-flow of the processors that carry out embedded applications are focused on this work. Very often, these types of faults induce in catastrophic system failure. By catastrophic failure, we mean those faults that in addition to drive the system to an unexpected behavior, it is also needed to reinitialize the system to recover from the faulty state.

Thus, the use of techniques capable of detecting these types of faults prevents them from spreading through the system and ultimately, generating incorrect outputs. Unfortunately, the use of fault detection techniques increase memory overhead and degrades system performance. These collateral effects may be critical, preventing real-time embedded systems from attaining their goals.

As possible solutions to the mentioned problems, three hypotheses were investigated, and one of them was implemented. Therefore, this work proposes an approach based on *software profiling* that analyses the control-flow graph of applications, to optimize the number of checkpoints to be inserted along with the application code.

In order to validate the proposed approach, we performed fault injection of three types of faults: *jump*, *nop* and *bit-flips*. This fault injection process was accelerated by means of hardware prototyping of the system. In this case, we used a FPGA (*Field-Programmable Gate Array*) mounted on a Xilinx commercial board. Detailed analysis of the obtained results indicates that the proposed approach reduces the number of checkpoints to be inserted along with the application code, thus, minimizing memory overhead and system performance degradation, while maintaining approximately unchanged the fault detection coverage, when compared to another existing approaches in the literature.

Sumário

Lista de Figuras

Lista de Tabelas

1. Introdução

O número de aplicações críticas embarcadas cresceu significativamente nos últimos anos. Este crescimento intensificou a pesquisa relacionada a técnicas capazes de agregarem confiabilidade e robustez aos sistemas. Exemplos reais da aplicação dos sistemas embarcados no quotidiano são os telefones celulares, os sistemas de controle de freio, tração e injeção de combustível nos automóveis de última geração, as impressoras a laser, os fornos de microondas, dentre tantos outros.

Durante o período de funcionamento destes sistemas, a probabilidade de ocorrerem falhas transientes e permanentes devido à presença de interferências dos mais variados tipos é bastante alta. Dentre as falhas mais freqüentes, salientam-se as falhas que corrompem os dados e as falhas que alteram o fluxo de controle do processador que executa a aplicação.

A utilização de técnicas capazes de detectarem estes tipos de falhas evitam que as mesmas se propaguem pelo sistema e acabe gerando saídas incorretas, o que causaria sérios prejuízos para os usuários destes sistemas e para a reputação das empresas. Estas técnicas são comumente referenciadas como CFC (*Control Flow Checking*) na literatura especializada.

1.1. Motivação

Várias metodologias baseadas em *software*, com o intuito de agregarem confiabilidade a aplicações críticas, já foram propostas na literatura e áreas de pesquisa relacionadas ao tema, serão referenciadas neste trabalho. Estas metodologias apresentam uma série de vantagens e desvantagens relacionadas ao custo, ao desempenho e ao *overhead* de área agregado aos sistemas. Por isto, quando se deseja agregar confiabilidade e robustez a um determinado sistema é necessário levar em consideração todos estes aspectos.

Na grande maioria das técnicas de CFC, o *overhead* de memória origina-se do acréscimo de instruções comumente chamadas de assinaturas (*checkpoints*) em diferentes áreas do aplicativo:

- *headers* das funções do programa, provocado pela geração das assinaturas e cálculo das assinaturas-diferença;
- em cada bloco básico, gerado pelos testes de desvio de fluxo;
- em cada bloco básico, ocasionado pelo cálculo das assinaturas de correção quando necessário.

1.2. Objetivos

Muitas técnicas existentes e referenciadas neste texto utilizam-se da construção de um grafo de forma que contenha nodos para representação de todos os blocos básicos, que conseqüentemente ficam protegidos pela técnica de detecção de falhas.

É objeto de estudo deste trabalho à análise do grafo de fluxo de controle de um aplicativo, de forma a selecionar alguns blocos básicos para receber a aplicação da técnica CFC, que será desenvolvida para este fim. Para a validação da técnica proposta, utilizaremos a técnica denominada CFCSS (*Control Flow Checking by Software Signatures*) proposta por *Nahmsuk Oh, Philip P. Shirvani* e *Edward J. McCluskey* (MCCLUSKEY, 2002), de forma a implementá-la sobre alguns e não sobre todos os blocos básicos como propõe a técnica de CFCSS original.

Determinar sobre quais nodos (blocos) estaremos aplicando a técnica de detecção de falhas simplificada em relação à técnica original (MCCLUSKEY, 2002), consiste no grande desafio deste trabalho, ou seja, determinar sobre o grafo os caminhos mais representativos, críticos, vitais para a aplicação, para sobre os mesmos aplicar a cobertura de falhas de forma a garantir a extensão desta cobertura para todos os nodos.

A presente dissertação tratará de técnicas para desenvolvimento de software robusto, baseadas na análise de fluxo de controle e de sua respectiva representação através de um grafo. A técnica proposta deverá manter a taxa de detecção de falhas tão abrangente quanto às técnicas clássicas e apresentar minimização substancial da área de memória e da degradação de desempenho.

1.3. Hipóteses de investigação

Visando alcançar os objetivos acima expostos, concentramos ações e revisão bibliográfica segundo três hipóteses de solução para o problema exposto, e que foram investigadas e analisadas segundo bibliografia e trabalhos publicados nas diferentes áreas do conhecimento envolvidas. As hipóteses investigadas foram as seguintes:

1.3.1. Mudança sobre o grafo de fluxo de controle e aplicação de técnica de CFC sobre o grafo modificado.

Aqui o foco é determinar quais blocos básicos seriam representativos do grafo como um todo, com o intuito de aplicarmos a técnica de CFC sobre um novo grafo, obtido a partir do grafo original. A forma de obtenção deste novo grafo deve estar amparada pela Teoria dos Grafos, de forma a obter fundamentação matemática para sua utilização. As idéias iniciais são a identificação daqueles blocos básicos mais executados e a identificação dos caminhos críticos dos grafos, de forma a estarmos protegendo somente estas áreas do aplicativo.

1.3.2. Mudança sobre a técnica de CFC escolhida e aplicação desta técnica modificada sobre o grafo da aplicação

Sobre este ponto de vista uma alternativa para solucionar o problema proposto seria a análise de uma técnica CFC e a partir de nova idéia, conceito, formulação matemática, técnica empírica, experimentação... e modificá-la segundo o(s) critério(s) escolhido(s).

Em oposição ao item anterior, estaríamos mantendo o grafo de controle de fluxo inalterado e fazendo alterações somente sobre uma técnica de CFC existente, e/ou proposto uma nova técnica, com o objetivo de minimizar o número de *checkpoints* (assinaturas) inseridas no código da aplicação.

1.3.3. Criação de técnicas de profiling para determinação das mudanças sobre técnicas de CFC e/ou modificações sobre o grafo de fluxo de controle.

Esta alternativa diz respeito a elaboração de técnicas de *software profiling* que a partir da análise do código da aplicação resultariam em um conjunto de regras, recomendações,

indicativos para aplicação de uma técnica de CFC, não da forma tradicional proposta na literatura, mas sim, a partir dos critérios estabelecidos pelo *software profiling*.

1.4. Organização da Dissertação

Esta dissertação está organizada em nove capítulos, dispostos como segue. No capítulo 1 apresentamos a introdução ao tema que será desenvolvido, bem como a motivação para sua execução, os objetivos desta dissertação e as hipóteses de investigação levantadas visando o alcance dos objetivos aqui propostos.

No capítulo dois iniciamos a revisão bibliográfica que extende-se até o capítulo seis, apresentando alguns tópicos que foram estudados ao longo do desenvolvimento desta dissertação, com vistas a sua adoção durante seu desenvolvimento.

O capítulo sete contém a metodologia de desenvolvimento desta dissertação, bem como a avaliação das hipóteses formuladas no capítulo um e a descrição das etapas que tratam da implementação computacional deste trabalho.

Finalmente, os resultados obtidos são apresentados no capítulo oito, segundo cada um dos aplicativos de testes utilizados. No capítulo 9 apresentamos as conclusões obtidas e trabalhos futuros relacionados ao tema desta dissertação.

2. Sistemas Embarcados

Os sistemas denominados *embarcados* podem ser caracterizados, de maneira geral, pela presença de processadores dedicados executando aplicações específicas, ou em outras palavras, representam uma combinação de *hardware* e *software* projetado para desempenhar uma função específica. O *hardware* é representado pelos processadores, memória e sensores, enquanto o *software* é usado para prover características e flexibilidade ao sistema.

Sistemas embarcados são empregados largamente em aplicações nas áreas militares, espaciais, aviação, controle automotivo, biomédica, e claro, inerente aos mesmos, existem a cobertura de um ou mais tipos de falhas, que caso ocorram, devem ser detectadas tão rapidamente quanto possíveis.

A diversidade de aplicações têm tornado o projeto de *software* embarcado uma tarefa não trivial. Em particular, existem várias restrições para s*oftwares* embarcados: temporais, consumo de energia e de memória, dentre outras. *Softwares* embarcados deverão possuir três características importantes: confiabilidade, robustez e versatilidade. Confiabilidade trata das melhores práticas para atingir qualidade de *software*. Robustez é a capacidade do sistema se em recuperar-se contra falhas. Versatilidade significa que o sistema tem capacidade de ajustar-se ou reconfigurar-se, para lidar com situações inesperadas.

Uma vantagem do *software* robusto é que ele permite a proteção contra problemas predefinidos, desde que, as possíveis entradas com erros sejam descobertas já na etapa de processo de desenvolvimento e teste. Uma desvantagem é que ele não abrange falhas induzidas por entradas erradas que não sejam especificadas.

Na medida em que a complexidade do projeto de sistemas e a *performance* dos processadores aumentam, a abordagem de projeto de sistemas embarcados cujo funcionamento e desempenho baseiam-se em tecnologias do tipo *Systens-On-Chips* (SoCs) que visam atender aplicações em tempo real, com grande exigência de desempenho, confiabilidade e robustez, tornam-se uma opção cada vez mais atrativa. Isto tem motivado esforços de pesquisa no campo

da análise de *software* embarcado, buscando, sobretudo, técnicas robustas para prover confiabilidade (WOLF, 2004).

2.1 O problema da ocorrência de falhas em sistemas embarcados

Sistemas mais robustos em relação a falhas eram, até recentemente, preocupação exclusiva de projetistas de sistemas críticos como aviões, sondas espaciais e controles industriais de tempo real. Com a espantosa popularização de redes fornecendo os mais variados serviços, aumentou a dependência tecnológica de uma grande parcela da população aos serviços oferecidos. Falhas nesses serviços podem ser catastróficas para a segurança da população ou para a imagem e reputação das empresas.

Para não ser o elo fraco de uma corrente, o mais simples dos computadores deve apresentar um mínimo de confiabilidade. Conhecer os problemas, quais soluções existem e qual o custo associado torna-se imprescindível para todos que pretendem continuar usando e expandindo seus negócios fornecendo um serviço computacional de qualidade aos seus clientes. Para desenvolvedores de *software*, projetistas de *hardware* e gerentes de rede o domínio das técnicas de tolerância à falhas torna-se essencial na seleção de tecnologias, na especificação de sistemas e na incorporação de novas funcionalidades aos seus projetos.

As falhas podem ser classificadas em permanentes, intermitentes e transientes. Falhas permanentes resultam de defeitos de fabricação ou erros residuais em componentes de *hardware* e *software*. Falhas em *hardware* intermitentes e transientes são provocadas por interferência do meio externo, tais como interferência eletromagnética (EMI) e partículas *alpha* (PRADHAN, 1996).

Falhas não-detectadas em aplicações em tempo de execução podem causar damos consideráveis, especialmente em sistemas não tolerantes a falhas. Requisitos de confiabilidade para estes sistemas são tradicionalmente implementados usando-se redundância de *hardware* (PRADHAN, 1996), o que em alguns casos, como aplicações espaciais, constituem-se numa solução viável. Nas aplicações onde métodos de redundância de *hardware* não podem ser utilizados, aplicam-se métodos de tolerância a falhas baseados em *software* (GOLOUBEVA, 2003) (ALKHALIFA, 1999) (MCCLUSKEY, 2002).

3. Teoria dos Grafos

A teoria dos grafos é o ramo da matemática que estuda as propriedades dos grafos. Neste capítulo apresentamos de forma resumida os principais conceitos relacionados a esta importante teoria.

3.1. Grafos

Um grafo é uma abstração matemática que descreve objetos interconectados e suas respectivas relações. Um grafo é um conjunto de pontos, chamados vértices (ou nodos ou nós), conectados por linhas, chamadas de arestas (ou arcos ou *links*).

Um grafo é uma estrutura de dados não-linear consistida de vértices (ou nós) que estão ligados através de arestas (ou *links*). Os nodos de um grafo podem ser tratados como simples objetos, possuindo nomes e atributos como dados e os próprios *links*.

Os grafos podem ser considerados como um modelo geral, formal, rigoroso e completo em sua teoria, o que torna possível sua utilização para especificação de vários tipos de dados, pois a maioria dos problemas no domínio da Ciência da Computação, Redes, Sistemas Distribuídos, podem ser formulados e modelados através de grafos (GROSS,1999).

Formalmente, um grafo **G=(V,A)** é um par ordenado formado por dois conjuntos finitos, de vértices V e de arco. Por sua vez, cada arco é denotado pelo par de vértices por ele ligados, isto é, **a=(v1,v2)**, onde **v1** e **v2** são dois vértices ligados pelo arco **a**.

3.1.1. Dígrafos

Um dígrafo é um grafo orientado, ou seja, os elementos do conjunto A são direcionados. Neste caso as conexões entre os vértices são chamadas de arcos e na representação destes, temos que **a=(v1,v2)** indicando que o arco **a** é direcionado a partir de **v1** para **v2.** Em um dígrafo, as relações entre os vértices não são simétricas.

É comum referir-se a um caminho em um grafo, que nada mais é do que um conjunto de arcos que possuam a mesma orientação. Podemos representar um caminho em um grafo da forma **(v1,v2,v5,v6,v4,v1)** e, neste caso temos a caracterização de um ciclo, pois **v1=vn**. Outra definição importante é a de grafos valorados, caracterizados pela existência de uma ou mais funções relacionando-se seus vértices e/ou arestas a números (valores, pesos).

Dependendo da aplicação, arcos podem ou não ter direção, pode ser permitido ou não que arcos liguem um vértice a ele próprio e vértices e/ou arcos podem ter um peso (numérico) associado. Se os arcos têm uma direção associada (indicada por uma seta na representação gráfica) temos um grafo direcionado, ou dígrafo. Estruturas que podem ser representadas por grafos estão em toda parte e muitos problemas de interesse prático podem ser formulados como questões sobre certos grafos.

Um arco como {v,w} será denotado simplesmente por **vw** ou por **wv** . Diremos que o arco **vw** incide em v e em w e que v e w são as pontas do arco. Se **vw** é um arco, diremos que os vértices v e w são vizinhos ou adjacentes. De acordo com nossa definição, um grafo não pode ter dois arcos diferentes com o mesmo par de pontas (ou seja, não pode ter arcos “paralelos”). Também não pode haver um arco com pontas coincidentes (ou seja, não pode ter “laços”). Há quem goste de enfatizar esse aspecto da definição dizendo que o grafo é “simples”.

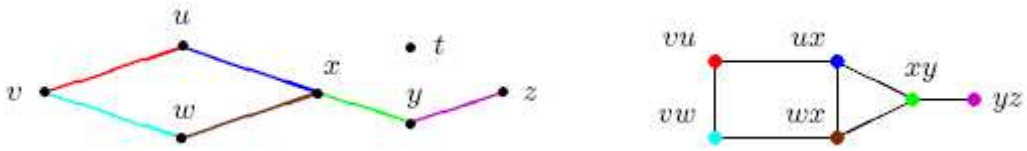


Figura 3-1 Um grafo (esquerda) e seu grafo das arestas (direita).

3.1.2. Isomorfismo

Um isomorfismo entre dois grafos G e H é uma bijeção f de V (G) em V (H) tal que dois vértices v e w são adjacentes em G se e somente se f(v) e f(w) são adjacentes em H. Dois grafos G e H são isomorfos se existe um isomorfismo entre eles. Em outras palavras, dois grafos são isomorfos se é possível alterar os nomes dos vértices de um deles de tal modo que os dois grafos fiquem iguais. Para decidir se dois grafos G e H são isomorfos, basta examinar todas as bijeções de V (G) em V (H). Se cada um dos grafos tem n vértices, esse algoritmo consome tempo

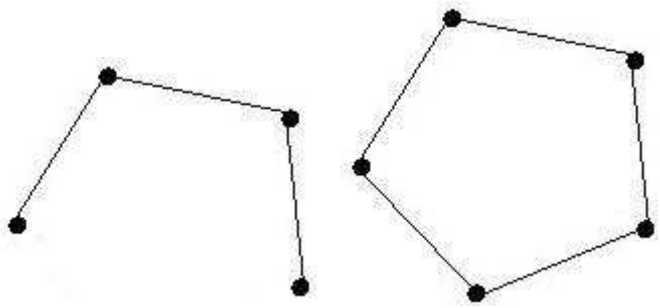
proporcional a n!. Como n! cresce explosivamente com n, esse algoritmo é decididamente insatisfatório na prática. Infelizmente, não se conhece um algoritmo substancialmente melhor.

3.1.3. Caminhos e ciclos em Grafos

Um **caminho** de G é uma seqüência de vértices de G $(v_0, v_1, ..., v_n)$ em que $v_0, v_1, ..., v_n$ são vértices de G distintos dois a dois e $\forall i \in \{1, ..., n\},\ \{v_{i-1}, v_i\} \in E(G)$.

Um caminho $(v_0, v_1, ..., v_n)$ também se diz caminho de v_0 para v_n de comprimento igual a *n*. Um caminho de comprimento zero (v_0) é, simplesmente, um vértice.

Um **ciclo** de G é uma seqüência de vértices de G $(v_0, v_1, ..., v_{n-1}, v_0)$ em que $v_0, v_1, ..., v_{n-1}$ são vértices de G distintos dois a dois e $\forall i \in \{1, ..., n-1\},\ \{v_{i-1}, v_i\}, \{v_{n-1}, v_0\} \in E(G)$. Ao número de arestas de um **ciclo** $(v_0, v_1, ..., v_{n-1}, v_0)$ chamamos comprimento do ciclo.



Caminho Ciclo

Figura 3-2 Um caminho e um ciclo.

3.1.4. Fluxos em Redes

Uma rede básica, *N*, é um dígrafo que satisfaz as seguintes condições:

1 – N tem, exatamente, uma origem e um destino;

2 – Para cada arco *e* de N está associado um número positivo *c(e)* que se designa capacidade do arco *e*, e que representa a quantidade máxima que pode passar nesse arco num determinado tempo.

Um **fluxo** em uma rede (N,A) é qualquer função de A em Z(conjunto de números inteiros). Em outras palavras, um fluxo é uma função que atribui um inteiro não-negativo a cada arco da rede. Os fluxos terão de verificar algumas condições:

1 – em cada arco, o fluxo não pode exceder a capacidade desse arco;

2 – os nós não são pontos de consumo, isto é, a soma numérica entrante em cada nó deverá ser igual à soma numérica resultante na saída deste nó, exceto os nós de origem e destino.

Um dos principais conceitos relacionados à fluxo é o seu “excesso” em cada nó. Para definir esse conceito, precisamos introduzir uma convenção de notação. Suponha que x é um fluxo e T uma parte de N. Denotamos por $\underline{\boldsymbol{T}}$ o complemento de T (ou seja, $\underline{\boldsymbol{T}} := N - T$) e por $x(\underline{\boldsymbol{T}}, T)$ a soma dos valores de x sobre os arcos que entram em T :

$$x(\underline{\boldsymbol{T}}, T) := P_{ij} \in (\underline{\boldsymbol{T}}, T)\ x_{ij}$$

É óbvio que $x(\underline{\boldsymbol{T}}, T)$ é a soma dos valores de x nos arcos que saem de T .

O **excesso**, ou **acúmulo**, de x em T é a diferença entre o que entra de T e o que sai de T :

$$x(\underline{\boldsymbol{T}}, T) - x(\underline{\boldsymbol{T}}, T)$$

3.2. Problema do caminho mínimo

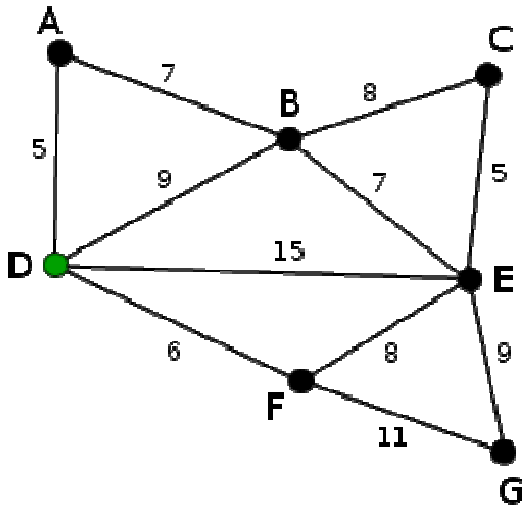


Figura 3-3 Problema do caminho mínimo.

O caminho mínimo entre D e E não é D-E, mas sim D-F-E, com uma distância de 14, conforme podemos observar na figura 3-3. Na teoria de grafos, o problema do caminho mínimo consiste na minimização do custo de travessia de um grafo entre dois nós (ou vértices), custo este dado pela soma dos pesos de cada arco percorrido.

Os algoritmos especializados em solucionar o problema do caminho mínimo são comumente chamados de algoritmos de busca de caminhos. Entre os algoritmos dessa classe, os mais conhecidos são (STANDISH ,1995):

- **Algoritmo de Dijkstra -** Resolve o problema com um vértice-fonte em grafos cujas arestas tenham peso maior ou igual a zero. Sem reduzir o desempenho, este algoritmo é capaz de determinar o caminho mínimo, partindo de um vértice de início v para todos os outros vértices do grafo;
- **Algoritmo de Bellman-Ford -** Resolve o problema para grafos com um vértice-fonte e arestas que podem ter pesos negativos;
- **Algoritmo A* -** Um algoritmo heurístico que calcula o caminho mínimo com um vértice-fonte;
- **Algoritmo de Floyd-Warshall -** Determina a distância entre todos os pares de vértices de um grafo;
- **Algoritmo de Johnson -** Determina a distância entre todos os pares de vértices de um grafo, pode ser mais veloz que o algoritmo de Floyd-Warshall em grafos esparsos.

A seguir, são comentados três destes algoritmos, que estão entre os mais utilizados na área de grafos.

3.2.1. Algoritmo de Dijkstra

O algoritmo de Dijkstra, cujo nome se origina de seu inventor, o cientista da computação Edsger Dijkstra, soluciona o problema do caminho mais curto em grafo dirigido com arestas de peso não negativo, em tempo computacional O([m+n]log n) onde m é o número de arcos e n é o número de vértices. O algoritmo que serve para resolver o mesmo problema em um grafo com pesos negativos é o algoritmo de Bellman-Ford.

Um exemplo prático de problema que pode ser resolvido pelo algoritmo de Dijkstra é:

Alguém precisa se deslocar de uma cidade para outra. Para isso, ela dispõe de várias estradas, que passam por diversas cidades. Qual delas oferece uma trajetória de menor caminho?

1º passo:

iniciam-se os valores:

para todo v □ V[G]

d[v]← ∞

π[v] ← nulo

d[s] ← 0

V[G] é o conjunto de vértices(v) que formam o Grafo G. d[v] é o vetor de distâncias de s até cada v. Admitindo-se a pior estimativa possível, o caminho infinito.

2º passo:

temos que usar dois conjuntos: S, que representa todos os vértices v onde d[v] já contem o custo do menor caminho e Q que contém todos os outros vértices.

3º passo:

realizamos uma série de relaxamentos dos arcos, de acordo com o código:

enquanto Q ≠ ø

u ← extraia-mín(Q)

S ← S □ {u}

para cada v adjacente a u

se d[v] > d[u] + w(u, v) //relaxe (u, v)

então d[v] ← d[u] + w(u, v)

π[v] ← u

w(u, v) é o peso(*weight*) da arco que vai de **u** a **v**.

u e **v** são vértices quaisquer e **s** é o vértice inicial.

extraia-mín (Q), pode ser um *heap* de mínimo ou uma lista ordenada de vértices onde obtém-se o menor elemento, ou qualquer estrutura do tipo.

No final do algoritmo teremos o menor caminho entre **s** e qualquer outro vértice de **G**.

3.2.2. Algoritmo de Bellman-Ford

O Algoritmo de Bellman-Ford é um algoritmo de busca de caminho mínimo em um dígrafo ponderado, ou seja, cujas arestas têm peso, inclusive negativo. O Algoritmo de Dijkstra resolve o mesmo problema, em um tempo menor, porém exige que todas as arestas tenham pesos positivos. Portanto, o algoritmo de Bellman-Ford é normalmente usado apenas quando existem arestas de peso negativo.

O algoritmo de Bellman-Ford executa em tempo O(v x a) onde v é o número de vértices e a o número de arestas.

```
// Define os tipos de dados para um grafo
registro vértice {
    lista arestas
    número distância
    vértice anterior
}
registro aresta {
    vértice origem
    vértice destino
    número peso
}

função BellmanFord(lista vértices, lista arestas, vértice origem)
   // Esta implementação recebe um grafo representado como uma
   // lista de vértices e arestas e modifica os vértices para
   // que seus atributos distância e anterior armazenem
   // os caminhos mais curtos.
```

Passo 1:

```
Iniciar o grafo para cada vértice v em vértices faça:
      se v é origem então:
         v.distância = 0
      senão:
         v.distância := infinito
      v.anterior := nulo
```

Passo 2:

Ajustar as arestas repetidamente para cada vértice v em vértices faça:

```
    para cada aresta uv em v. arestas faça:
        u := uv.origem
        v := uv.destino     // uv é a aresta de u para v
        se v.distância > u.distância + uv.peso então:
            v.distância := u.distância + uv.peso
            v.anterior := u
```

Passo 3:

Verificar a existência de ciclos com peso negativo para cada aresta uv em arestas faça:

```
    u := uv.origem
    v := uv.destino
    se v.distância > u.distância + uv.peso então:
        erro "O grafo contém um ciclo de peso negativo."
```

4. Uso de Grafos para detecção de falhas em fluxo de controle

Um programa pode ser dividido em conjuntos de sub-rotinas. Em linguagens como o C, uma rotina é expressa como um conjunto de declaração de variáveis, constantes, chamadas para outras rotinas, e estruturas de controle ou desvios condicionais (*if-then-else*, *while*, *continue*, etc).

A verificação de fluxo de controle é usualmente conseguida pela partição do programa aplicativo em blocos básicos (ALKHALIFA, 1999), (MCCLUSKEY, 2002), (GOLOUBEVA, 2003), (REBAUDENGO, 1999). Um bloco básico é constituído por uma ou mais instruções que não apresentam desvios condicionais. Cada bloco básico recebe uma assinatura pré-determinada, computada em tempo de execução.

A estrutura de uma rotina pode ser representada por um grafo direcionado com uma única entrada e vários nodos de saída, onde os nodos representam os blocos básicos e os arcos representam o fluxo de controle. Este grafo é chamado de grafo de fluxo de controle da rotina R.

A representação gráfica de uma aplicação através de um grafo é particularmente importante no que se refere à análise quanto ao aplicativo estar executando corretamente as seqüências de instruções previstas, pois constitui-se uma representação visual que auxilia no processo de verificação do fluxo de controle entre as instruções agrupadas em blocos.

Existem técnicas especialmente dedicadas para que o fluxo de dados ocorra corretamente, comumente chamadas de CFC (*Control Flow Checking*) na literatura especializada, abordadas no capítulo 6. Assim, como exemplo, podemos citar que uma falha inter-bloco é um *jump* a partir de um bloco básico para outro, da mesma forma uma falha intra-bloco salta ou re-executa um conjunto de instruções internas ao bloco básico corrente. Detectar falhas do tipo *jump*, entre outras, é o objetivo de técnicas CFC.

A utilização de técnicas capazes de detectarem falhas em fluxo de controle de aplicações evita que as falhas se propaguem pelo sistema e acabe gerando saídas incorretas, o que causaria sérios prejuízos para os usuários destes sistemas e para a reputação das empresas. Nesse sentido a representação do fluxo através de um grafo faz-se necessária.

5. Métodos de pesquisa sobre grafos

A seguir apresentamos um breve comentário a cerca de diferentes tipos de processamento computacional sobre grafos, concentrados em métodos de pesquisa (aplicações de algoritmos de busca), com e sem a utilização de heurísticas (RUSSELL, 2003).

5.1. Busca em Largura

Na Ciência da Computação, busca em largura (busca em largura-primeiro ou busca em amplitude) é um algoritmo de procura utilizado para realizar uma busca ou travessia numa estrutura de árvore ou grafo. Intuitivamente, você começa pelo nó raiz e explora todos os nós vizinhos. Então, para cada um desses nós mais próximos, exploramos os seus nós vizinhos inexplorados e assim por diante, até que seja encontrado o alvo da busca.

Formalmente, uma busca em largura é um método de busca não-informada (ou desinformada) que expande e examina sistematicamente todos os nós de um grafo, em busca de uma solução. Em outras palavras, podemos dizer que seu algoritmo realiza uma busca exaustiva, sem considerar o seu alvo de busca, até que ele o encontre. Ele não utiliza uma heurística.

Do ponto de vista do algoritmo, todos os nós filhos obtidos pela expansão de um nó são adicionados a uma fila (*First In First Out*). Em implementações típicas, nós que ainda não foram examinados por seus vizinhos são colocados num *container* (como por exemplo uma fila ou lista encadeada) que é chamado de “aberto”. Uma vez examinados, são colocados num *container* “fechado”.

Quando realizamos a busca do menor caminho num grafo cíclico ponderado (ou seja, que não é uma árvore), um algoritmo de busca em largura pode ser adaptado, mantendo-se um bit em cada nó para indicar qual deles já foi visitado.

A busca em largura é completa apenas se a árvore pesquisada tem um número finito de ramos – o algoritmo irá encontrar o alvo da busca caso ele exista (ou seja, ele alcança todos os nós de uma árvore). A busca em largura é ótima se o custo dos passos forem idênticos – o algoritmo encontrará a solução mais “rasa” de uma árvore de busca. No caso em que os passos possuem custos diferentes, a solução mais “rasa” não é necessariamente a melhor. A busca em largura tem complexidade linear espacial do tamanho (arestas somadas a vértices) da árvore / grafo pesquisada(o), já que ela precisa armazenar todos os nós expandidos na memória.

A busca em largura pode ser usada para identificar componentes conectados bem como para testar bipartição em grafos. O conjunto de nós alcançados pela busca em largura são os maiores componentes conectados que contém o nó inicial. Se não houverem arestas nos nós adjacentes numa mesma camada de busca, então o grafo deve conter um número ímpar de ciclos e não ser bipartido.

A busca em largura tem como objetivo verificar primeiro os nós mais próximos do nó inicial de um grafo, até chegar ao destino da busca. Para o melhor entendimento do processo de busca em largura será mostrado um exemplo meramente ilustrativo a seguir.

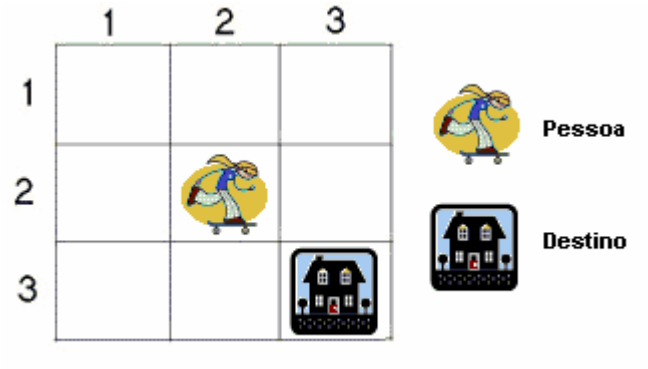


Figura 5-1 Espaço de busca em amplitude.

Na Figura 5-1 está a representação de um mapa na forma clássica. Supõe-se que a pessoa esteja restrita em não caminhar nas coordenadas das diagonais, ou seja, como ela se encontra na coordenada (2,2) do mapa, apenas poderá caminhar nas coordenadas (1,2), (2,1), (2,3) e (3,2). O objetivo da pessoa é chegar na coordenada (3,3). Através do método de busca em amplitude ajudará a pessoa a encontrar um caminho que levará ao destino, na figura 5-2 temos um grafo representando a leitura de busca para esta situação:

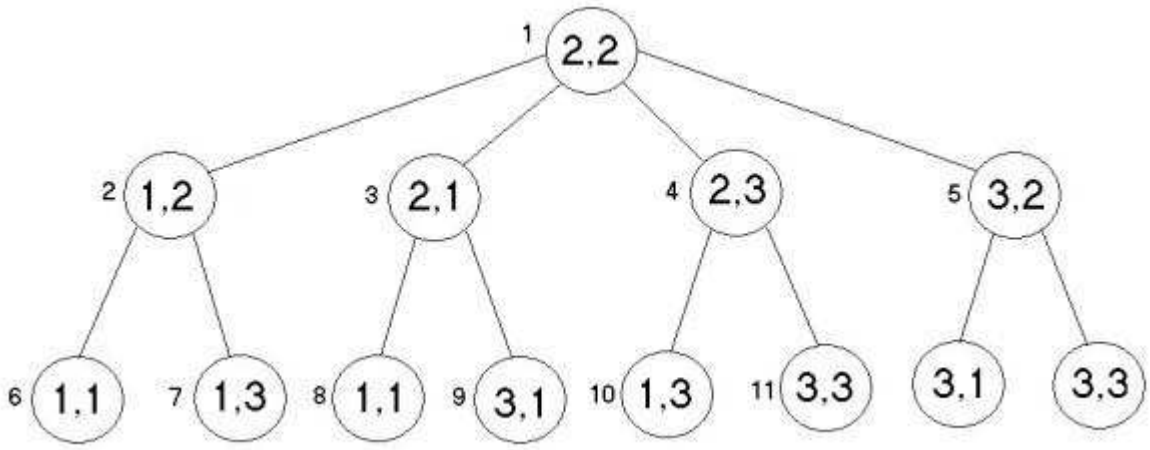


Figura 5-2 Árvore de busca de amplitude

Os números de 1 a 11 mostram a seqüência de nós visitados, e cada nó representa uma coordenada da Figura 5-2. O último nó visitado pela busca foi o (1,1), que é a coordenada destino da pessoa. Encontrado o destino, termina-se a busca aplicando o melhor caminho, neste caso é um caminho único formado pelas coordenadas (2,2), (2,3) e (3,3). Apesar de existir outro caminho, de coordenadas C1 = {(2,2),(3,2),(3,3)}, C1 não foi explorado pois o caminho C = {(2,2), (2,3),(3,3)} acima referido já fora encontrado.

Segue abaixo o algoritmo de busca em largura:

1. Crie uma variável chamada Lista-de-Nós e ajuste-a para o estado inicial.

2. Até ser encontrado um estado-meta ou Lista-de-Nós ficar vazia, faça o seguinte:

(a) Remova o primeiro elemento de Lista-de-Nós e chame-o de E. Se Lista-de-Nós estiver vazia, saia.

(b) Para a maneira como cada regra pode ser casada com o estado descrito em E, faça o seguinte:

I -Aplique a regra para gerar um novo estado;

II -Se o novo estado for um estado-meta, saia e retorne este estado;.

III -Caso o contrario, acrescente o novo estado ao final de Lista-de-Nós;

Repare que a estrutura de dados que armazena os nós é uma Fila.

5.2. Busca em Profundidade

Este método consiste em buscar prioritariamente os nós sucessores mais distantes do nó inicial de um grafo, até encontrar o destino, ou seja, este algoritmo é o oposto do busca em

amplitude, pois utiliza uma estrutura de dados denominada pilha. Para compreender melhor este método de busca, será utilizado o mesmo exemplo do tópico anterior.

Baseado no mesmo problema exemplificado na busca em largura pela figura 5-2, a Busca em Profundidade resultará na seguinte árvore de busca representado na figura 5-3:

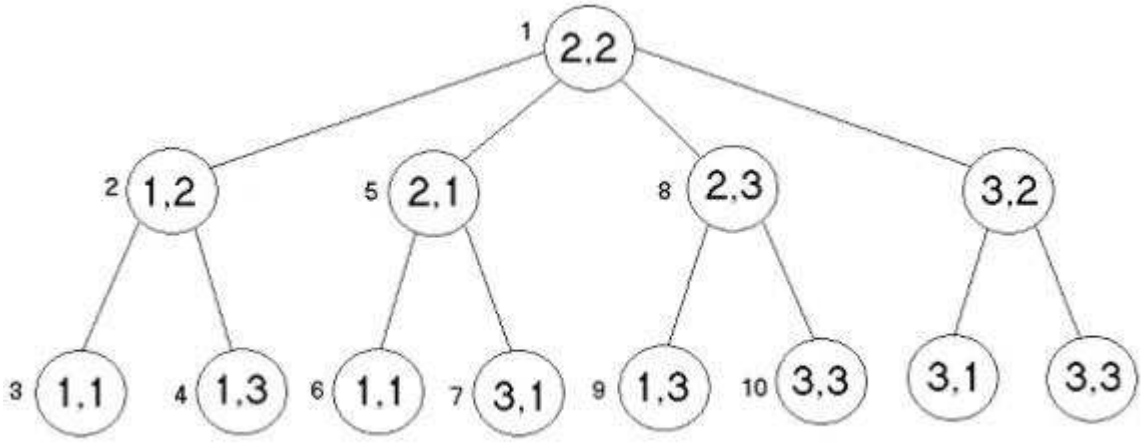


Figura 5-3 Árvore de busca em profundidade.

Os números de 1 à 10 revelam a seqüência de busca, enquanto cada nó da árvore representa a coordenada da figura 5-3. Percebe-se que este método é necessário um limite na sua busca, ou seja, determinar a profundidade da sua busca, pois este método tem a possibilidade de checar infinitos sucessores em apenas um galho da árvore de busca, ou pode deixar menos eficiente a busca, como ocorre na figura 5-4:

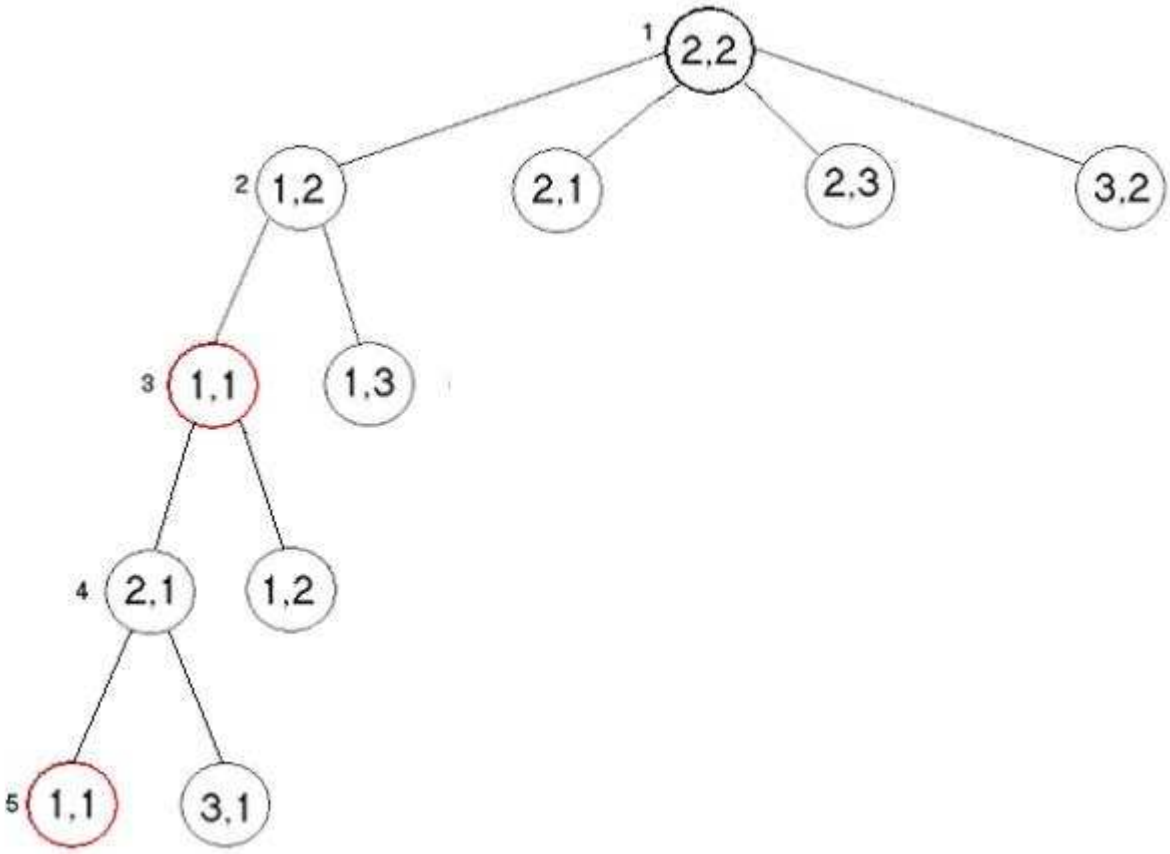


Figura 5-4 Árvore de busca em profundidade sem solução.

Repare que na figura 5-4 as coordenadas (1,1) repetem durante o processo de busca e assim repetindo os mesmos nós sucessores, caindo em um *looping* infinito. Segue abaixo o algoritmo de busca em profundidade:

1. Se o estado inicial é um estado de meta, saia e retorne sucesso.

2. Caso contrário, faça o seguinte até a sinalização de sucesso ou fracasso:

(a) Gere um sucessor, E, do estado inicial. Se não houver mais sucessores, sinalize fracasso.

(b) Chame Busca em Profundidade com E, como estado inicial.

(c) Se for retornado sucesso, sinalize sucesso. Caso contrário, continue nesse laço (*loop*).

5.3. Best First Search

Os métodos de busca em largura e em profundidade tornam-se menos eficientes devido à inexistência de uma função que direcione a seleção de nós para um caminho mais otimizado. Justamente com o intuito de otimizar estes métodos de busca são utilizadas as chamadas heurísticas através de algoritmos de busca denominados de *Best First Search*.

Trabalhar com heurística, em linhas gerais, envolve métodos ou algoritmos exploratórios para definição de problemas em que as soluções são descobertas pela avaliação do progresso obtido na busca de um resultado final. Trata-se de métodos em que, embora a exploração seja feita de forma algorítmica, o progresso é obtido pela avaliação puramente empírica do resultado.

Visando problemas diferentes, foram elaborados vários métodos matemáticos para tais contextos, chamados métodos heurísticos que são utilizados nos algoritmos de busca de melhor caminho. As técnicas heurísticas são usadas em problemas em que a complexidade da solução do algoritmo disponível é a função exponencial de algum parâmetro; quando o valor deste cresce, o problema torna-se rapidamente mais complexo. Uma alternativa heurística será praticável se a complexidade do cálculo depender, por exemplo, polinomialmente do mesmo parâmetro.

A heurística pode ser considerada uma probabilidade, um chute, uma suposição a respeito da resolução de um problema. O ser humano utiliza heurística instintivamente toda vez que resolve um problema.

Por exemplo, quando uma pessoa está procurando uma vaga para estacionar, ela analisa as condições de trânsito, e como andam as vagas próximas. Com base nestas informações ela pode deduzir se vai existir uma vaga próxima ao local onde ela deseja ir ou não. No entanto, a pessoa nunca vai saber com certeza se há uma vaga próxima ao seu destino ou não. Esta probabilidade é uma forma de heurística.

Agora pense que existe uma função matemática h(x) a qual quanto menor o seu valor maior é a probabilidade de algo ser correto. Com este conceito em mente pode-se pensar em um algoritmo que quando resolva um problema utilize tal função em prol de minimizar sua área de busca.

Relembrando o exemplo anterior utilizado para explicar a busca em largura e em profundidade, para tal pode-se criar uma heurística que funcione da seguinte maneira:

H = 10 * (valor absoluto (origemX - destinoX) + valor absoluto (origemY – destinoY))

Gostaríamos de salientar que em se tratando de heurísticas, estamos aqui exemplificando através da heurística de Manhattan, mas para a resolução de problemas de busca poderemos adotar uma nova heurística, criada a partir do contexto da situação a que se destina resolver, caso as heurísticas já consagradas na literatura especializada não apresentem resultados satisfatórios ou como o esperado. Tal método é denominado de método Manhattam e será comentado na próxima seção.

Aplicando tal função ao exemplo tem-se na figura 5-5 os seguintes valores para os nós:

	1	2	3
1	40	30	20
2	30	20	10
3	20	10	0

Figura 5-5 – Valores da função h onde o nó (3,3) é o destino.

Abrindo a busca tem-se a seguinte árvore de resultados ilustrada na figura 5-6, onde a cada interação o algoritmo escolhe o menor valor de h:

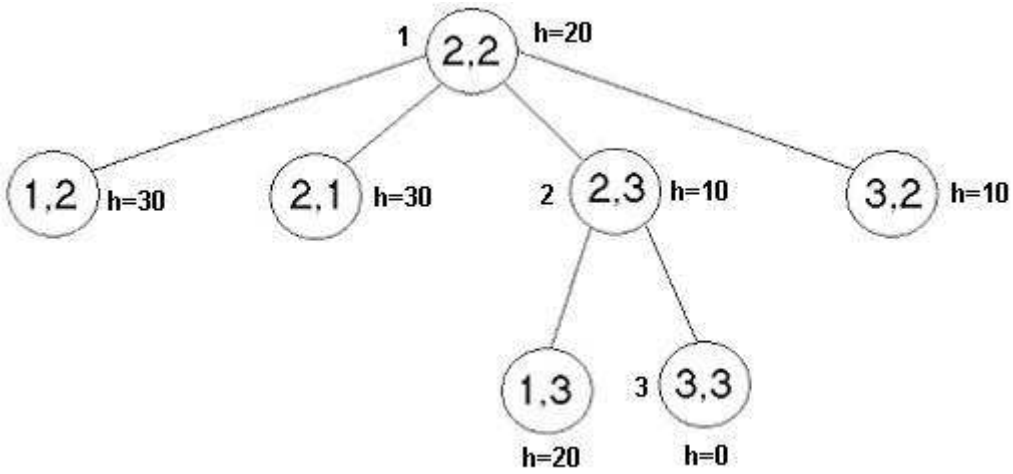


Figura 5-6 – Árvore de busca do BFS com valores de H.

Adicionando-se uma outra função denominada g(n) que representa o custo do nó inicial até o nó “n” obtêm-se a seguinte fórmula:

f(n)=g(n) + h(n)

Nesta fórmula o algoritmo utilizará o elemento com menor valor de f(n), ou seja, o elemento que tem o melhor equilíbrio entre o custo e probabilidade.

No exemplo em questão tem-se a representação da figura 5-7:

	1	2	3
1	20 + 40	10 + 30	20 + 20
2	10 + 30	20 + 0	10 + 10
3	20 + 20	10 + 10	0 + 20

Figura 5-7 – Valores da função F.

Percebe-se que a função g não provoca mudança neste exemplo, no entanto ela tem um propósito, evitar que o algoritmo fique abrindo filhos em uma direção que apesar de manter uma boa heurística nunca atinge o destino. A função g faz com que o algoritmo procura por outras ramificações onde o custo não é tão grande.

Seu funcionamento será mais bem entendido durante a explicação do algoritmo A*.

Abrindo a busca têm-se a seguinte árvore de resultados ilustrada na figura 5-8, onde a cada interação o algoritmo escolhe o menor valor de h:

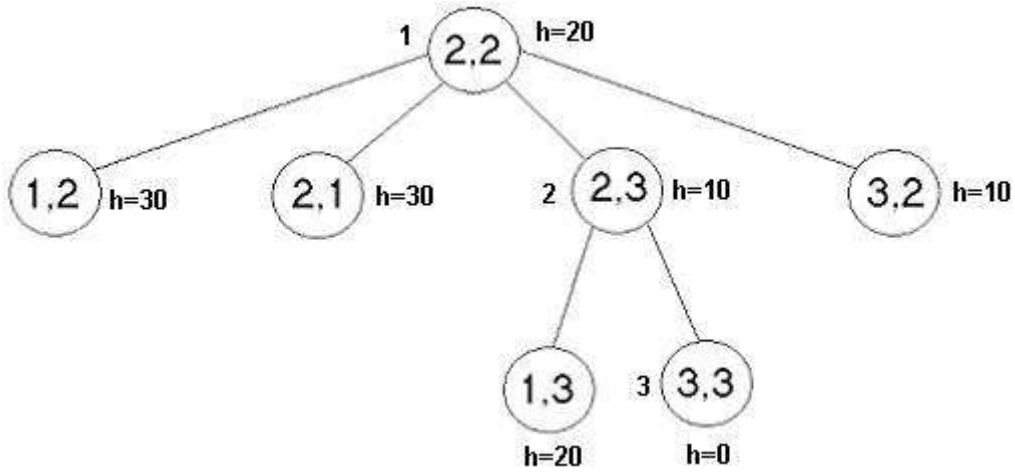


Figura 5-8 Árvore de busca do BFS com valores de H.

Adicionando uma outra função denominada g(n) que representa o custo do nó inicial até o nó “n” caímos na seguinte formula:

f(n)=g(n) + h(n)

Nela o algoritmo agora pegara o elemento com menor valor de f(n), ou seja, o elemento que tem o melhor equilíbrio entre o custo e probabilidade.

Percebe-se que a função g não provoca mudança neste exemplo, no entanto ela tem um propósito, evitar que o algoritmo fique abrindo filhos em uma direção que apesar de manter uma boa heurística nunca atinge o destino. A função g faz com que o algoritmo procura por outras ramificações onde o custo não é tão grande.

Seu funcionamento será mais bem entendido durante a explicação do algoritmo A*.

5.4. Método Manhattan

O método Manhattam é o mais utilizado em jogos, por requerer menor quantidade de cálculos e ser um dos melhores em expansão de nós e custos de caminho. Sua desvantagem é que não encontrará sempre o melhor caminho, embora encontre um dos melhores, além de ser um método não balanceado, por dar um peso maior aos valores de G comparado a H, que prejudica a aplicação de penalidades para curvas (utilizado em situações em que se deseja que não sejam feitas curvas bruscas no caminho a ser utilizado) e a aplicação de áreas de influência (utilizado para determinar a área de domínio de dois times diferentes em um mesmo mapa) por utilizarem G, exagerando ambos, além de não lidar com diagonais.

H = 10 * (valorasbsoluto (origemX - destinoX) + valorabsoluto (origemY – destinoY))

Para a fórmula a cima, considere os seguintes valores apresentados na tabela 5-1.

Tabela 5-1 Valores X,Y para Origem e Destino.

Origem X	Origem Y	Destino X	Destino Y	H
10	10	30	30	400

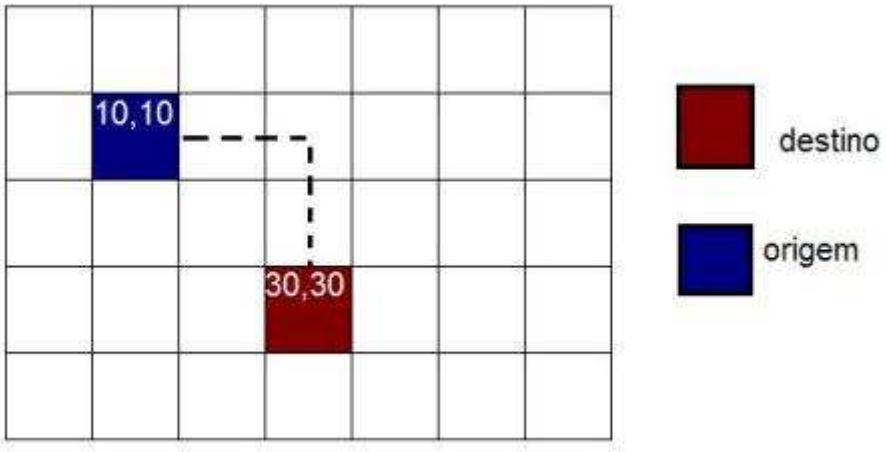


Figura 5-9 Caminho percorrido pelo método Manhattan.

5.5. Método Diagonal

Consiste da utilização de diagonais com um valor igual ao de horizontais e verticais, assim, colocando em equilíbrio os valores de H e G. Tem como desvantagem o fato de utilizar mais cálculos que outros métodos. A sua vantagem é que é ótimo para a aplicação de penalidades para curvas e a aplicação de áreas de influência.

Consiste da fórmula:

xDistância = valorabsoluto (origemX – destinoX)

yDistância = valorabsoluto (origemY – destinoY)

Se xDistância > yDistância

H = 14 * yDistância + 10 * (xDistância – yDistância)

Ou

H = 14 * xDistância + 10* (yDistância – xDistância)

Fim

Para a fórmula a cima, considere os seguintes valores apresentados na tabela 5-2.

Tabela 5-2 Valores X,Y para Origem e Destino.

Origem X	Origem Y	Destino X	Destino Y	H
10	10	30	30	280

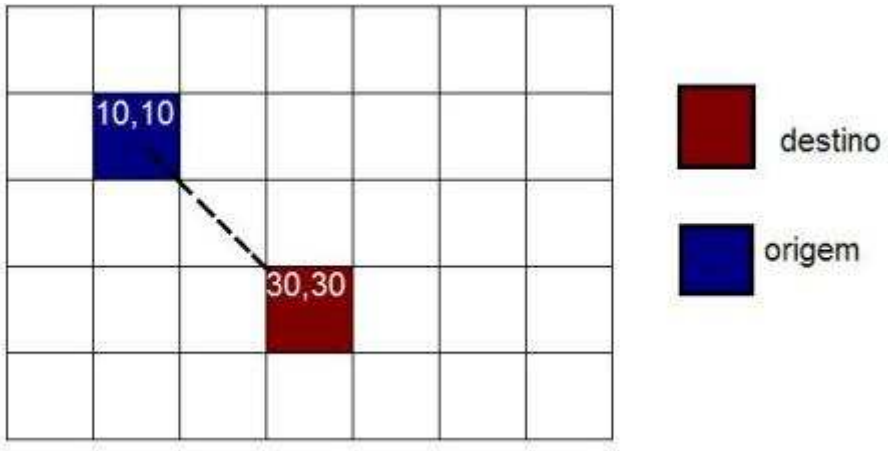


Figura 5-10 Caminho percorrido pelo método diagonal.

5.6. Algoritmo A*

Algoritmo A* (lê-se A estrela) é um algoritmo para Busca de Caminho. Ele busca o caminho em um grafo de um vértice inicial até um vértice final. Ele é a combinação de aproximações heurísticas como do algoritmo *Best-First Search* e da formalidade do Algoritmo de Dijkstra. Sua aplicação vai desde aplicativos para encontrar rotas de deslocamento entre localidades a resolução de problemas, como a resolução de um quebra-cabeças. Ele é muito usado em jogos.

Durante a procura nós precisamos manter os caminhos em duas listas de estados (chamados nodos, pois são nodos em uma árvore). A primeira lista é a lista aberta. Nela se armazena nodos que nós geramos usando as regras de um nodo existente, mas ainda não sabemos até onde ele conduz. A segunda lista é de nodos que nós geramos e que também exploramos aonde eles vão. Cada estado é armazenado junto com alguns dados extras necessários em nossa procura. A primeira coisa que é requerida é um ponteiro pai. Precisamos saber como chegamos a este nodo. Quando o resultado final for achado, será gerado um caminho de atrás para o início. Sabendo qual é o pai do nodo é que nos permite fazer isto. O nodo também tem três valores usados durante a procura que é o custo de se chegar ao nodo, a estimativa heurística e o total dos outros dois. O custo de um nodo é fácil de calcular. A cada movimento pode ser atribuído um custo. No exemplo mostrado anteriormente o custo de um nodo é sempre um, mas como nós queremos saber o custo do caminho todo, somamos os custos de todos os

nodos desde o começo para o nodo que estamos, ou seja, quando geramos um novo nodo, seu custo será o seu mais o custo de seu pai.

A estimativa heurística pode ser muito mais complexa do que o que vimos. Nós podemos adivinhar o quanto é bom o nodo e o quanto ele está próximo do destino. No exemplo, usamos as diferenças de x e y entre o nodo atual e o nodo destino como uma heurística. Finalmente o valor total de um nodo, que representa o quanto está bom em termos de quanto custa para chegar lá e quão perto da meta nós estamos. É usado ao longo da procura para caminho mais promissor.

6. Monitoramento do Fluxo de Controle de Processadores Baseado em Assinaturas de Software

A estratégia para detecção de falhas é fortemente dependente da aplicação, o que dificulta o desenvolvimento de técnicas sistemáticas para sua detecção. Entretanto, existem vários estudos focados sobre uma classe especial de falhas chamada falhas de fluxo de controle, que ocorrem quando o processador salta para uma instrução que não é a instrução correta.

Um erro no fluxo de controle é dito ter ocorrido se o processador executa uma seqüência incorreta de instruções devido a uma falha. Estes erros podem ser causados por falhas transientes ou permanentes detectadas pelo programa, endereços de circuitos, ou sistema de memória. Falhas são inevitáveis, mas as conseqüências das falhas, ou seja, o colapso do sistema, a interrupção no fornecimento do serviço e a perda de dados, podem ser evitados pelo uso adequado de técnicas viáveis e de fácil compreensão.

Assim, para evitar que estas falhas sejam propagadas e gerem saídas incorretas, aconselha-se a utilização de técnicas específicas capazes de monitorarem os dados manipulados e o fluxo de execução do programa. O conhecimento dessas técnicas habilita o usuário a implementar as mais simples, ou exigir dos fornecedores soluções que as incorporem. Entretanto, as técnicas que toleram falhas têm certo custo associado. O domínio da área auxilia usuários e desenvolvedores de sistemas na avaliação da relação custo benefício para o seu caso específico, e determinar qual a melhor técnica para seu orçamento.

6.1. Técnicas de Detecção de Erros em Dados

As falhas em dados provocam alterações indesejadas no conteúdo das variáveis. Assim para evitar que essas falhas se propaguem no sistema, surgem várias metodologias de tolerância

a falhas capazes de detectá-las. A seguir serão apresentadas duas técnicas propostas na literatura para a detecção de erros em dados.

6.2. Técnica ED^{4}I

Error Detection by Diverse Data and Duplicated Instructions - ED^{4}I, proposta por Oh. Nahmsuk (NAHMSUK, 2002) é uma técnica SIHFT que provê a detecção de falhas permanentes e temporárias executando dois programas "diferentes", que apresentam a mesma funcionalidade e diferentes conjuntos de dados, e comparam as suas saídas. Basicamente, ED^{4}I determina que cada número *x*, do programa original seja mapeado em novo número *x´* de tal forma que os resultados das duas versões coincidam. O mapeamento é feito através da equação (6.1).

$$x´ = k.x \quad (6.1)$$

Onde: *k* determina a probabilidade de detecção de falha e a integridade dos dados do sistema.

6.3. Regras de transformação do algoritmo

Antes de apresentar as regras de transformação definidas em ED^{4}I, algumas definições e terminologias pertinentes serão apresentadas.

Segundo Oh. Nahmsuk (NAHMSUK, 2002) um bloco básico é uma seqüência de instruções consecutivas em que o fluxo de controle entra no início e sai no fim sem encontrar nenhum desvio até o final.

Abaixo segue a notação utilizada em ED^{4}I:

V = {v1, v2, ... , vn}: conjunto dos vértices que representam os blocos básicos;

E = {(i,j)| (i,j) é um desvio de vi para vj}: conjunto dos limites que representam os possíveis desvios entre os blocos básicos;

Assim um programa pode ser representado por um grafo P = {V, E}.

A figura 6.1 apresenta um determinado programa seguido de seu grafo de fluxo. Neste caso, este programa possui quatro blocos básicos, *v1*, *v2*, *v3* e *v4* e o conjunto dos desvios possíveis é definido por E = {(1,2), (2.3), (3,2),(2,4)}.

As regras definidas em ED^{4}I transformam *P*, programa original em *P´*. *P´* resulta da multiplicação de todas as variáveis e constantes do programa original por um fator k e portanto se *x* é *k* vezes maior que *y*, *x* é *k* múltiplo de *y*. Assim esta técnica determina que sejam executadas as transformações abaixo descritas:

Transformação da expressão: este passo transforma as expressões dentro *P* para novas expressões em P´, de tal forma que cada variável ou constante de *P´* é o valor de *P* multiplicado por *k*. Como os valores de P´ são diferentes dos valores originais, quando comparamos os dois valores em um instrução condicional, a relação de desigualdade pode necessitar de uma alteração. Por exemplo, dada a instrução condicional *if (i < 5)* em *P*, ela necessita ser alterada para *if (i > - 10)* em *P´* quando *k = -2*. De outra forma, o fluxo de controle determinado pelas instruções condicionais em *P´* devem ser diferentes do fluxo de controle em *P*, e portanto o resultado da computação do programa *P´* não será *k* múltiplo do código original.

Transformação das condições de desvio: este passo ajusta a relação desigual na instrução condicional em *P´* tal que o fluxo de controle em *P* é *P´* são idênticos.

Assim, um programa multiplicado por um fator *k* é um programa com um novo grafo denotado por *Pg´ {V´, E´}* que por sua vez é isomórfico a *Pg*. Dado que *S* e *S´* representam o conjunto das variáveis de *P* e *P´* respectivamente e *n* o número de vértices (blocos básicos) executados, então *S(n)* e *S´(n)* são definidos como :

S(n): o conjunto de variáveis em *S* após *n* blocos básicos serem executados;
S´(n): o conjunto de variáveis em *S´* após *n* blocos básicos serem executados.

A partir da análise da figura 6.1 é possível definir o conjunto de variáveis, ou seja, *S = {i, x, y, z}*. Assim, após o programa ser iniciado e um bloco básico ser executado, *n = 1* e *S(1) = {i = 0, x = 1, y = 5, z = 0}* porque as quatro instruções do primeiro bloco são executadas. Após *v2* e *v3* serem executadas, *n = 3* e *S(3) = {i = 1, x = 1, y = 5, z = 1}*.

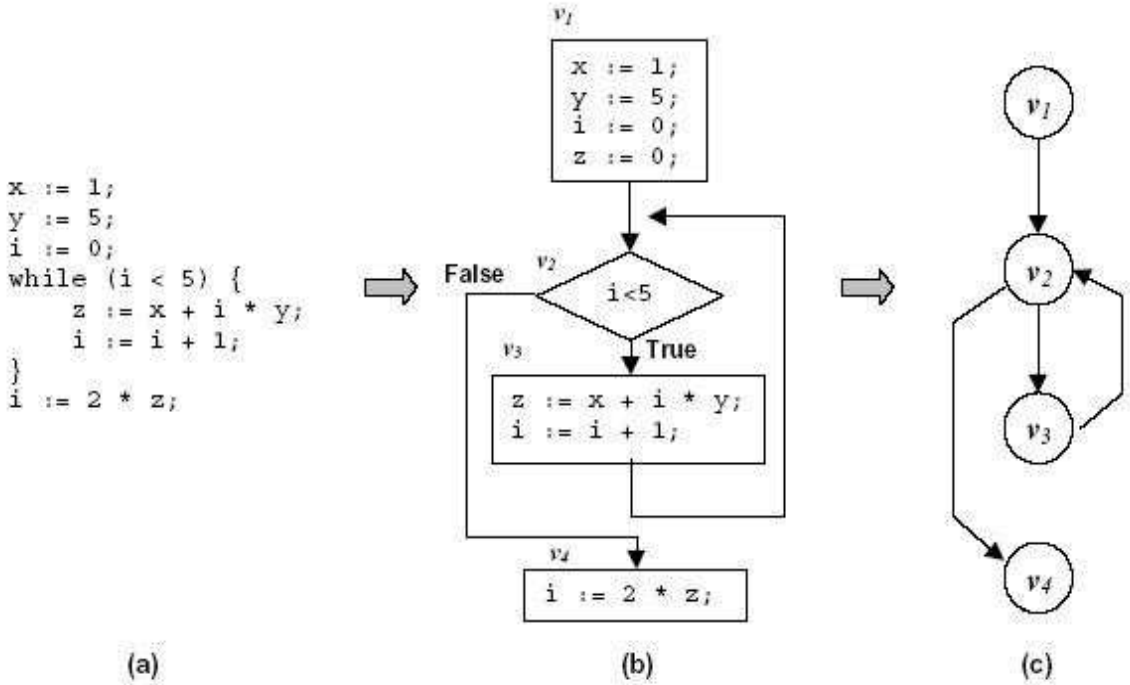


Figura 6-1 (a) um programa; (b) o grafo de fluxo e (c) o grafo do programa Pg. (NAHMSUK, 2002)

Em resumo, a transformação do programa deve satisfazer as condições abaixo: *Pg* e *Pg´* são isomórficos;

$k.S(n) = S'(n)$, para $\forall\ n > 0$

Onde: *k.S(n)* é obtido multiplicando todos os elementos em *S(n)* por *k*.

A condição (1) define que o fluxo de controle nos dois programas deve ser idêntico e a condição (2) define que todas as variáveis de *P´* são sempre múltiplas de k do programa original *P*.

A figura 6-2 e 6-3 mostra o programa da figura 6-1 transformado a partir da utilização de um fator $k = -2$.

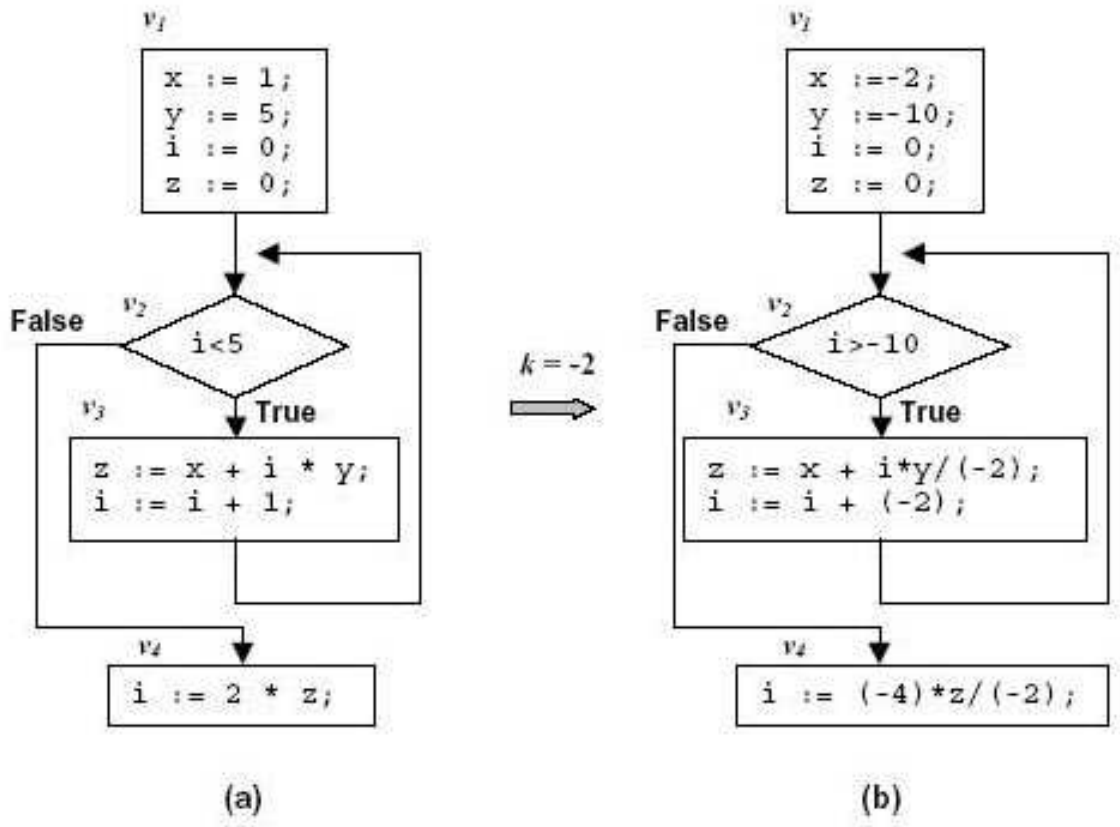


Figura 6-2 (a) grafo do programa original e (b) grafo do programa transformado com *k = -2*. (NAHMSUK, 2002)

```
x := 1;
y := 5;
i := 0;
while (i < 5) {
    z := x + i * y;
    i := i + 1;
}
i := 2 * z;
```

(a)

k = -2

```
x := -2;
y := -10;
i := 0;
while (i > -10) {
    z := x + i * y /(-2);
    i := i + (-2);
}
i := (-4) * z / (-2);
```

(b)

Figura 6-3 (a) o programa original e (b) o programa transformado com *k = -2*. (NAHMSUK, 2002)

Salienta-se que o fator *k* é determinado levando-se em consideração e conseqüentemente evitando um eventual *overflow* durante a execução do programa transformado.

Quanto à cobertura de falhas, esta técnica é capaz de detectar falhas temporárias através da comparação dos resultados obtidos a partir da execução das duas versões do código e falhas permanentes. Além disso, esta técnica pode detectar falhas permanentes que afetam o caminho dos dados em unidades funcionais através da execução de dois programas com dados diferentes utilizando diferentes partes das unidades funcionais e comparando os resultados.

6.4. Técnica proposta por M. Rebaudengo

M. Rebaudengo (REBAUDENGO, 2004) propõe uma solução via *software* para a detecção e correção de falhas transientes afetando dados manipulados pelo programa. Para isto, esta técnica apresenta várias regras de transformação que são aplicadas ao código fonte do programa alvo. Para aplicar esta técnica em um determinado programa, deve-se seguir as seguintes regras:

Cada variável *x* deve ser duplicada: *x0* e *x1* representarão as duas cópias. Cada operação de escrita executada em x deve ser executada também em *x0* e *x1*. Assim, dois conjuntos de variáveis são gerados: o conjunto 0 e o conjunto 1.

Após cada operação de leitura em x, as duas cópias *x0* e *x1* devem ser verificadas, e se alguma inconsistência é detectada, uma rotina de correção de erro deve ser ativada.

A figura 6.4 mostra à esquerda o código original de um determinado programa escrito em C ANSI e à direita o mesmo código acrescido das modificações acima descritas.

Código original	**Código modificado**

a = b;	a0 = b0;
	a1 = b1;
	if (b0 != b1)
	  error();

a = b + c;	a0 = b0 + c0;
	a1 = b1 + c1;
	if ((b0!=b1) || (c0!=c1))
	  error();

Figura 6-4 Código original e código tolerante a falhas. (REBAUDENGO, 1999)

Esta técnica detecta falhas que possam vir a ocorrer durante o processo de manipulação das variáveis pelo processador ou durante o período em que estas estiverem armazenadas na memória.

6.5. Técnicas de Detecção de Erros de Fluxo de Controle

Assim como as falhas em dados, as falhas de fluxo de controle podem ser geradas a partir de ruídos externos e consistem na execução das instruções em uma seqüência incorreta, ou seja, a execução do programa não segue o fluxo correto de execução.

Salienta-se que a grande maioria das soluções via *software*, definidas para detecção de erros de fluxo de controle baseiam-se fundamentalmente na divisão do código do programa em blocos básicos e na utilização da notação, abaixo descrita, para representá-lo.

Oh Nahmsuk (NAHMSUK, 2002) sugere a seguinte notação:

P = {V,E}: representa o grafo do programa;

V = {*vi*, *i* = 1, 2, .., n}: representa o conjunto de blocos básicos;

E = {*ei*, *ej*, ..., *ek*}: representa as linhas entre os blocos básicos (conjunto de desvios);

vi: representa o bloco básico *i*;

ei: representa a linha de união entre os blocos básicos (desvios);

bri,j: desvio entre *vi* e *vj*;

suc(*vi*): conjunto de nós sucessores do nó vi;

pred(*vi*): conjunto de nós predecessores do nó vi.

Assim, um programa P = {V,E}, onde V = {*v1*, *v2*, *v3*, ..., *vn*} e E = {*e1*, *e2*, e3, ..., *em*}. Cada nó *vi* representa um bloco básico e está associado a um conjunto de sucessores e predecessores, representados por *suc*(*vi*) e *pred*(*vi*) e cada linha *ei* representa um desvio *bri,j*, ou seja um desvio entre *vi* e *vj*.

A seguir serão descritas diferentes metodologias capazes de detectarem falhas de fluxo de controle propostas na literatura.

6.6. Técnica CCA (Control Flow Checking by Assertions)

A técnica CCA proposta em G. A. Kanawati (KANAWATI, 1996) e Z. Alkhalif (ALKHALIF, 1999) é uma solução em *software* que provê a detecção de erros de fluxo de controle em algoritmos. Ela consiste na divisão do código do programa em intervalos livres de desvio (*branch free intervals* - BFIs), na inserção de dois identificadores para cada um dos BFIs e na verificação dos identificadores durante o período de execução do código.

O primeiro identificador é denominado identificador do intervalo livre de desvio (*branch-free interval IDdentifier* - BID) e armazena um valor único que identifica o BFI. Já o segundo, denominado identificador de fluxo de controle (*control flow IDentifier* - CFID), representa os desvios permitidos, ou seja, indica se os desvios de fluxo de controle estão ocorrendo na seqüência correta. O CFID é armazenado em duas filas de elementos que são inicializadas com o CFID do primeiro BFI. Na entrada do BFI, o CFID do próximo BFI é colocado na fila e na saída do mesmo o CFID é retirado da fila. A fila que monitora o valor CFID provê a redução da média de latência de detecção da falha. A figura 6.5 mostra a estrutura IF-THEN-ELSE acrescida das instruções de verificação da técnica CCA.

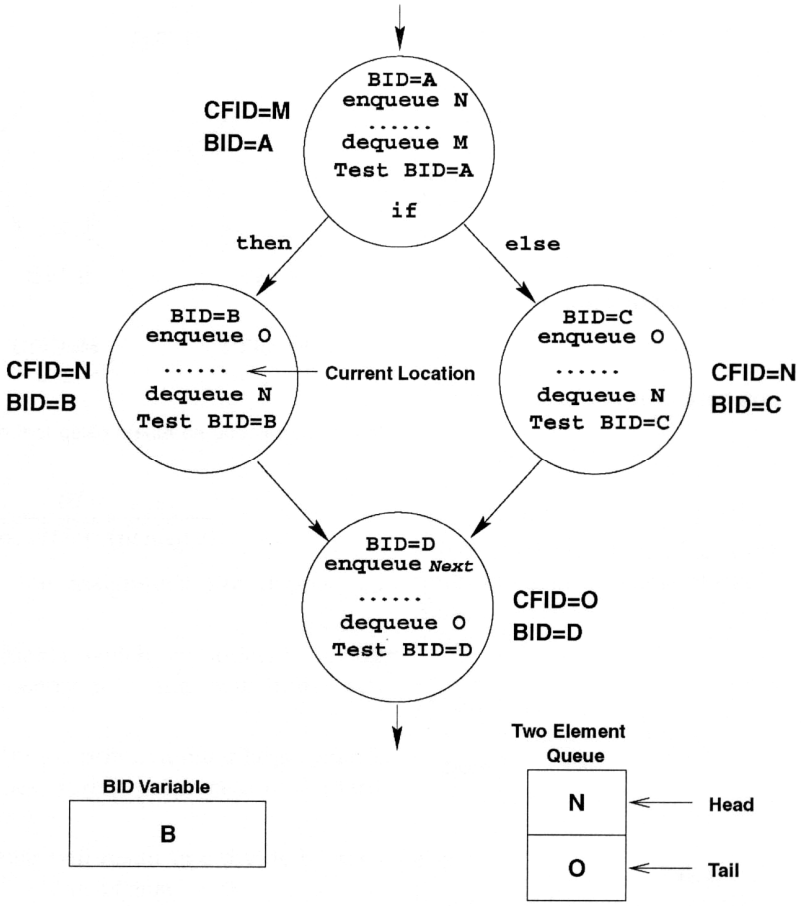


Figura 6-5 Instruções e verificação dos IDs para estrutura IF-THEN-ELSE com CCA. (ALKHALIFA, 1999)

Quanto à cobertura de falhas, a técnica acima descrita é capaz de detectar todos os erros de fluxo de controle simples e a maioria dos erros múltiplos. Basicamente, as falhas podem gerar:

- um desvio de dentro de um BFI;.
- um desvio para dentro de um BFI;.
- um desvio para um BFI ilegal, ou seja, um desvio para um BFI que não pertence ao grupo de sucessores do BFI atual.

6.7. Técnica ECCA (Enhanced Control Flow using Assertions)

Segundo Z. Alkhalifa (ALKHALIFA, 1997), a técnica ECCA é uma solução em *software* capaz de detectar erros no fluxo de controle de algoritmos que se baseia fundamentalmente em inserir instruções de teste e atualização no código da aplicação a fim de torná-lo tolerante a falhas e conseqüentemente mais confiável e robusto.

A aplicação da técnica ECCA é feita a partir da realização dos seguintes passos:

1. dividir o programa em um conjunto de intervalos livres de desvio (*branch free intervals* - BFI´s)

2. atribuir um número primo único, denominado identificador de intervalos livres de desvio (*branch free intervals IDentifier* – BID), para cada BFI.

3. inserir no início do BFI a instrução (6.2) e no fim a instrução (6.3).

$$id \leftarrow \frac{BID}{\overline{\overline{(id \bmod BID)}}.(id \bmod 2)}$$

(6.2)

$$id \leftarrow NEXT + \overline{\overline{(id - BID)}}$$

(6.3)

Onde: *id* representa a variável global atualizada durante o tempo de execução do algoritmo na entrada e na saída do BFI, ou seja, monitora o fluxo de execução do algoritmo; BID representa a assinatura de cada BFI, ou seja, BID armazena um número primo único para cada BFI gerado em tempo de pré-processamento; NEXT representa o somatório de todos os BID dos BFI que podem ser sucessores do BFI atual gerado em tempo de pré-processamento.

Assim, um programa escrito em linguagem C possui a seguinte estrutura após ser pré-processado pela técnica ECCA.

/* Início do BFI */

id = <BID> / ((!(id%<BID>)) * (id%2));

… corpo do **BFI** …

id = <NEXT> + !!(id-<BID>);

/* Fim do BFI */

A figura 6-6 E de um programa em C dividido em BFIs.

A figura 6-6 representa o mesmo programa acrescido das instruções de controle definidas pela técnica, ou seja, após o pré-processamento. Finalmente a figura 6-7 mostra o diagrama de bloco do mesmo programa.

```
/* Inicio do código original*/
...        BFI1 ...
if foo
{
... BFI2 ...
}
else
{
... BFI3 ...
}
... BFI4 ...
/* Fim do código original */
```

Figura 6-7 Representação do código original. (ALKHALIFA, 1997)

```
/* Início do código pré-processado*/
...
/* <BID> é 3 */
... BFI1...
id = 35+!!(id-3);
/* Observe que NEXT vale 35 e resulta da multiplicação do BID = 5 do BFI 2 com o
BID = 7 do BFI 3. */
if foo
{
/* <BID> é 5 */
                id = 5 / ((!(id%5)*(id%2));
```

```
                    ... BFI2 ...
                    id = 11+!!(id-5);
}
else
/* <BID> é 7 */
                     id = 7/((!(id%7))*(id%2));
                    … BFI3 …
                     id = 11+!!(id-7);
}
/* <BID> é 11 */
id = 11/((!(id%11))*(id%2));
... BFI4 ...
/* Fim do código pré-processado */
```

Figura 6-8 Representação do código tolerante a falhas de acordo com a técnica ECCA. (ALKHALIFA, 1997)

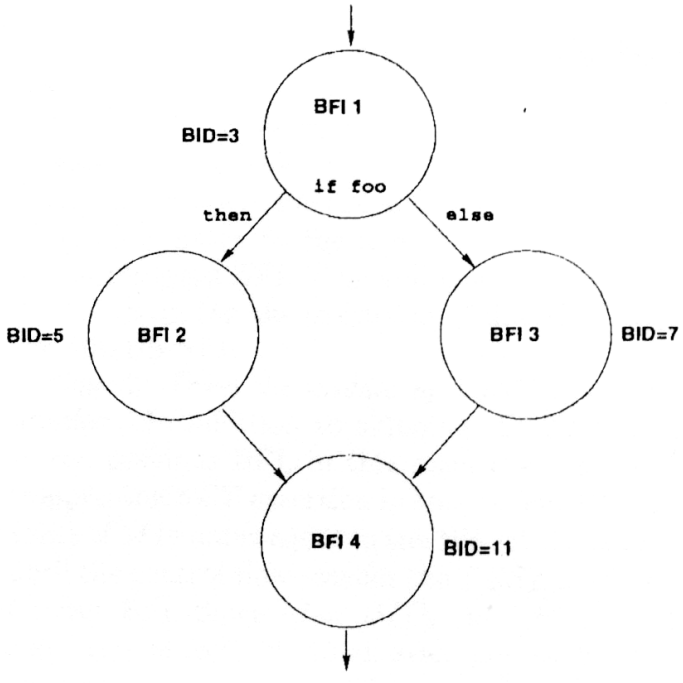


Figura 6-9 Diagrama de bloco do código tolerante a falhas de acordo com a técnica ECCA. (ALKHALIFA, 1997)

Quanto à cobertura de falhas, esta técnica apresenta a mesma capacidade de detecção da técnica CCA, ou seja, é capaz de detectar todos os erros de controle de fluxo simples e a maioria dos múltiplos. Entretanto, ECCA apresenta algumas vantagens em relação a CCA, são elas:

ECCA apresenta um menor *overhead* em termos de espaço e tempo, pois são inseridas apenas duas linhas de código por BFI;

ECCA utiliza apenas uma variável ao invés de duas filas e uma variável;

A figura 6.10 mostra a redução de erros múltiplos de fluxo de controle onde, dado três BFI´s A, B e C; um erro de fluxo de controle múltiplo pode ocorrer somente se ocorrer um desvio ilegal de dentro do BFI A para dentro do BFI B, e então do BFI B ocorrer um desvio para dentro do BFI C. Diante desta situação, o erro será classificado como um erro de A para C, ou seja, um erro de fluxo de controle simples.

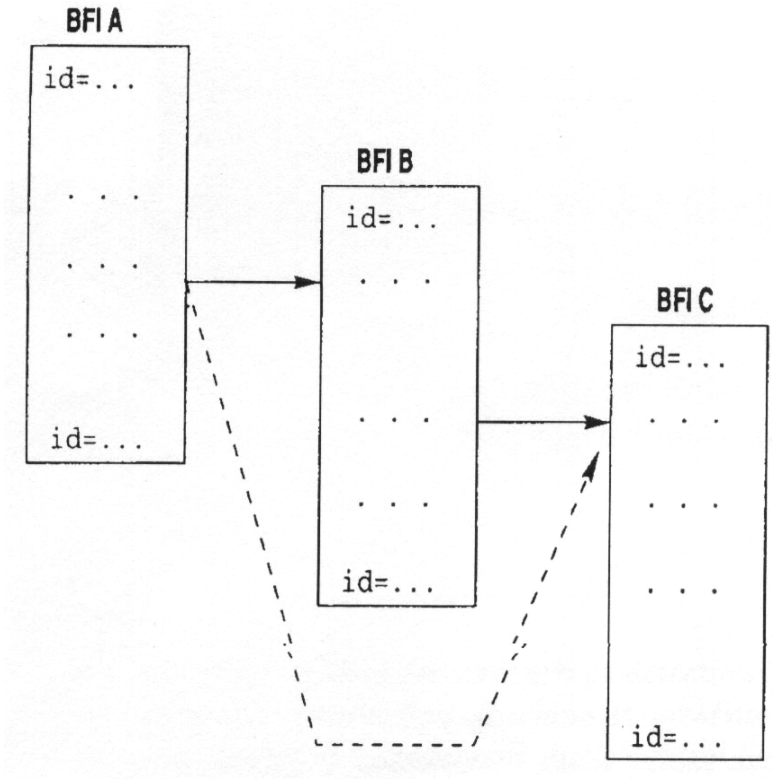


Figura 6.10 Redução de erros de fluxo de controle múltiplos.

Assim, comparando as técnicas acima descritas com as soluções baseadas em *hardware* anteriormente propostos, as técnicas CCA e ECCA apresentam as seguintes vantagens:

- São portáveis, ou seja, podem ser implementadas para múltiplas plataformas, pois são aplicadas em alto nível;
- Representam soluções implementadas puramente em *software*;

– Diminuem o *overhead* relacionado ao custo e ao desempenho.

6.8. Técnica CFCSS (*Control Flow Checking by Software Signatures*)

A técnica de CFCSS, proposta por E. McCluskey (MCCLUSKEY, 2002) é uma solução via *software* que provê a detecção de erros de fluxo de controle. Assim como as técnicas anteriormente descritas, a CFCSS baseia-se fundamentalmente na divisão do programa em blocos básicos, na atribuição de assinaturas para cada um deles e no monitoramento das assinaturas em tempo de execução.

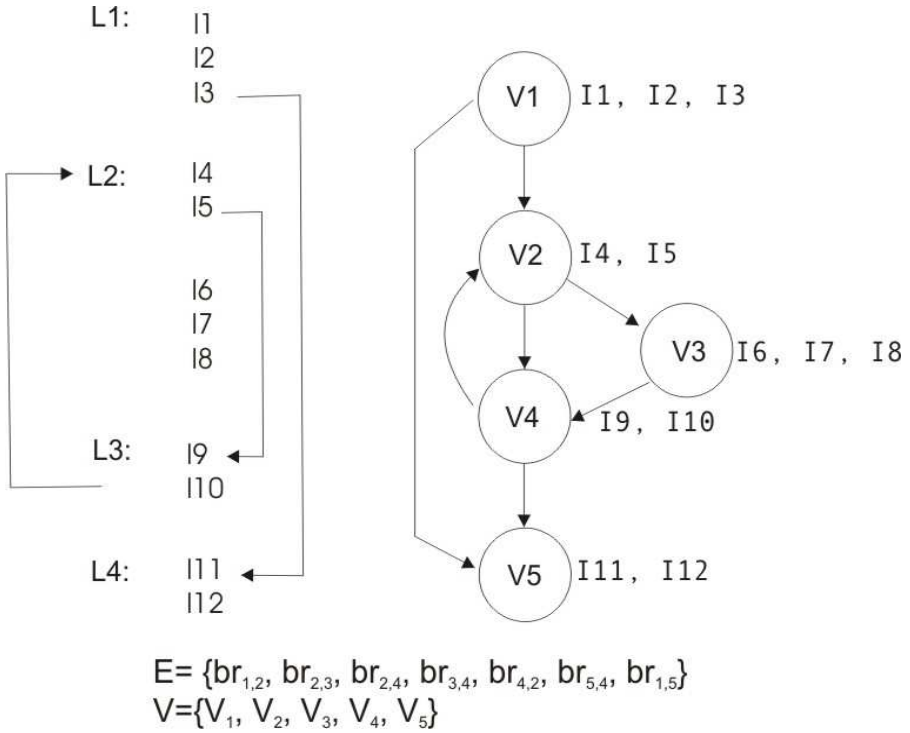


Figura 6-11 Seqüência de instruções e seu respectivo grafo. (MCCLUSKEY, 2002)

Em relação à notação adotada, a técnica de CFCSS utiliza basicamente a terminologia mencionada anteriormente. Assim, a técnica determina que o programa alvo seja dividido em blocos básicos e que a partir desta divisão seja construído um grafo que representa o fluxo de execução do mesmo. A figura 6-11 mostra um programa e seu respectivo grafo de execução.

A técnica de CFCSS verifica em tempo real o fluxo de controle do programa através do monitoramento e verificação das assinaturas, sendo definida pelos seguintes passos:

Dividir o programa em blocos básicos (*vi*);

Construir o grafo que representa o fluxo de execução do programa;

Atribuir aleatoriamente um valor único de assinatura (*si*) para cada bloco básico em tempo de compilação para representar o bloco básico *vi*;

Calcular em tempo de compilação a assinatura diferença (*di*) definida com base nas assinaturas do(s) predecessor(es) de *vi* (*pred(vi)*) e na assinatura de *vi*. Este cálculo é realizado através da seguinte fórmula: $di = ss \oplus sd$, onde *ss* representa a assinatura do nó fonte (predecessor) e *sd* representam a assinatura do nó destino (no caso o próprio nó *vi*). Assim, dado o bloco básico *vi* e seu conjunto de predecessores igual a *pred(vi)={vj}*, a assinatura *di* é calculada a partir da formula: $di = sj \oplus si$.

Quando necessário, calcular em tempo de compilação a assinatura de ajuste (D). Esta assinatura é utilizada quando um determinado nó *vi* possuir mais de um predecessor.

Além dos passos acima descritos, uma assinatura G deve ser calculada em tempo de execução e armazenada em uma variável dedicada denominada registrador de assinatura global (*Global Signature Register* - GSR). Assim toda vez que o controle é transferido de um bloco básico para outro a assinatura G é atualizada através da função de assinatura *f* dada pela equação (6.4).

$$f = f(G, di) = G \oplus dd \text{, onde } dd = ss \oplus sd$$

(6.4)

A figura 6-12 ilustra um exemplo de um desvio legal que ocorre do nó *v1* para *v2*. Observe que antes do desvio $G = G1 = s1$ e após o desvio G é atualizado a partir do seguinte cálculo: $f = f(G1, d2) = G1 \oplus d2$, onde $d2 = s1 \oplus s2$ e assim $G2 = s1 \oplus (s1 \oplus s2) = s2$, ou seja, para o nó *v2*, G2 = *s2*.

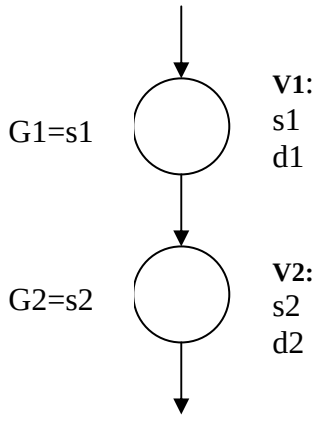


Figura 6-12 Exemplo de um desvio legal de *v1* para *v2*. (MCCLUSKEY, 2002)

A figura 6.12 ilustra um exemplo de desvio ilegal de *v1* para *v4*. Observe que antes do desvio, $G = G1 = s1$ e após o desvio, G é atualizado a partir do seguinte cálculo $f = f(G1, d4) = G1 \oplus d4$, onde $d4 = s3 \oplus s4$ e assim $G4 = s1 \oplus (s3 \oplus s4) \neq s4$, ou seja, o erro de fluxo de controle será detectado pois, G4 é diferente de *s4*.

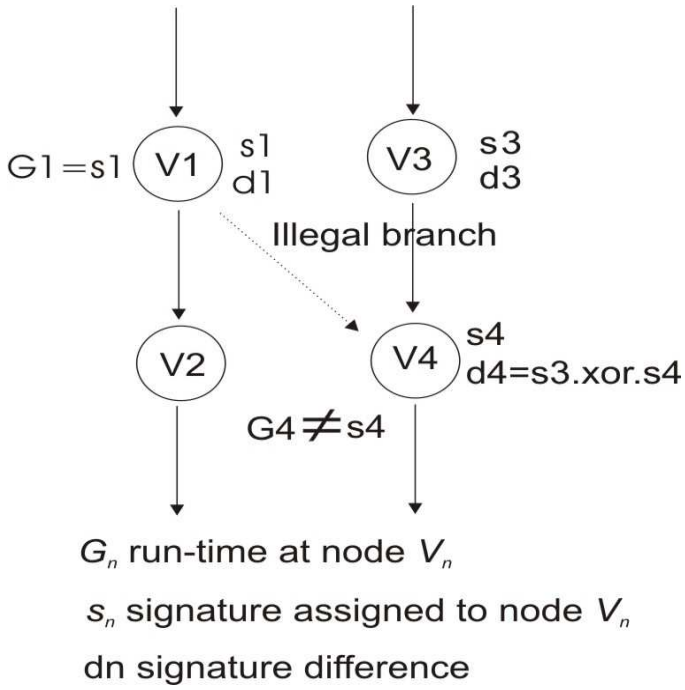


Figura 6-13 Exemplo de um desvio legal de *v1* para *v4*. (MCCLUSKEY, 2002)

Assim, no topo de cada bloco básico devem ser inseridas as instruções de verificação abaixo definidas:

1. Função assinatura: G = (G ⊕ *dk*);

2. Instrução de verificação de desvio: “Br (G ≠ *sk*) error”.

A figura 6-14 ilustra respectivamente a representação gráfica das instruções, do bloco básico, do bloco básico acrescido das instruções de verificação e a representação utilizada para um nó em um grafo de fluxo de execução.

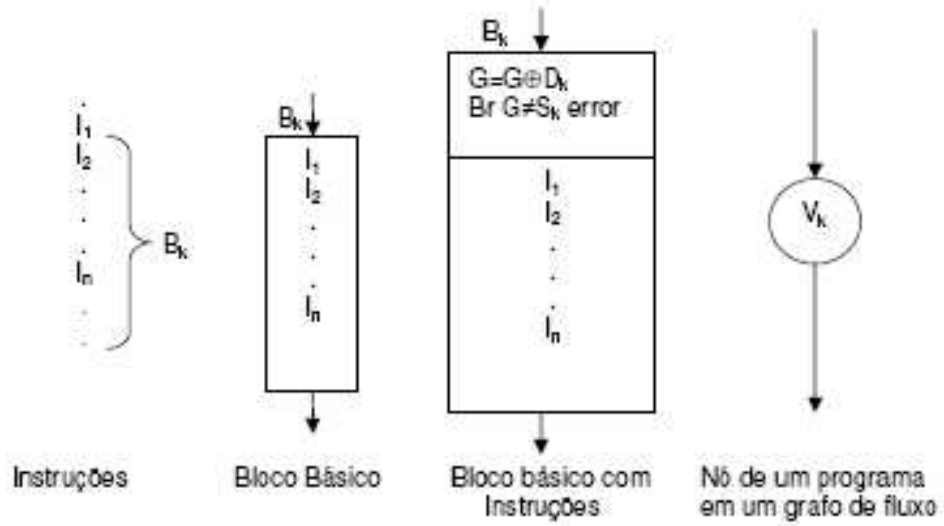


Figura 6-14 Representação gráfica. (MCCLUSKEY, 2002)

Entretanto, quando um determinado bloco básico possui mais de um predecessor (grafos com nós convergentes), é necessário inserir no código uma assinatura extra denotada como *D*. A Figura 6-15 ilustra claramente o problema que surge quando um determinado bloco básico possui mais de um predecessor. Observe que os blocos *v1* e *v3* desviam para o nó *v5*. Logo, *d5* seria um resultado de $\sigma 1 \oplus \sigma 5$. Caso o desvio seja *br1,5*, o *G5* = $\Gamma 1 \oplus \delta 5 = \sigma 1 \oplus \sigma 1 \oplus \sigma 5 = \sigma 5$, ou seja, o G5 será igual a s5 e conseqüentemente nenhum erro será observado. Contudo, ocorrerá um erro se o desvio for *br3,5*, pois o G5 = G3 $\oplus\ \delta 5 \Rightarrow \sigma 3 \oplus \sigma 1 \oplus \sigma 5 \neq \sigma 5$ devido a $\sigma 3 \neq \sigma 1$.

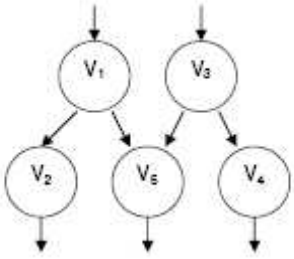


Figura 6-15 – Exemplo de um bloco básico com mais de um predecessor.

Neste contexto, a fim de solucionar o problema acima ilustrado, surge a assinatura *D*, que deve ser inserida após a instrução que realiza o cálculo de atualização de *G*. A figura 1.3.6 ilustra um exemplo de utilização da assinatura *D*. Observe que no bloco básico B5 é adicionada a instrução $G = G \oplus D$ após a instrução que calcula a atualização da assinatura G, dada por $G = G$

⊕ *d5*. D é determinado a partir dos nós fontes *v1* e *v3*, ou seja, *D = s1* ⊕ s3. Portanto, inicialmente, *d5 = s1* ⊕ s5 e D no bloco B1 apresentam o valor 0000. Assim, após o desvio *br1,5*, *G5 = G* ⊕ *d5* e *G5 = G* ⊕ *D = s5 = 0000 = s5* e após o desvio br3,5, *G5 = G3* ⊕ *d5 = s3* ⊕ *(s1* ⊕ *s5)* e *G = G5* ⊕ *D = s3* ⊕ *(s1* ⊕ *s5)* ⊕ *(s1* ⊕ *s3) = s5*.

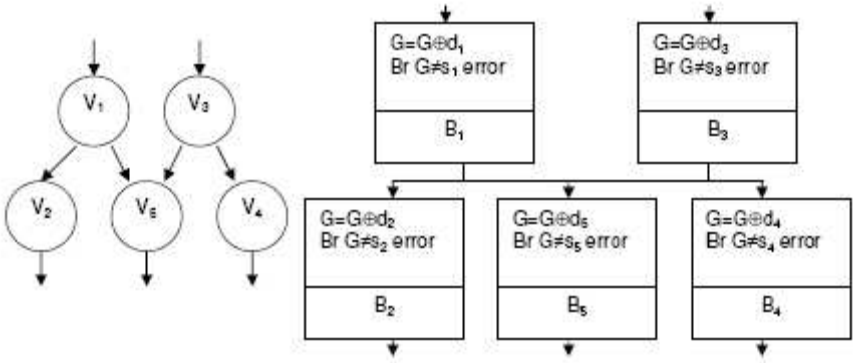


Figura 6-16 – Exemplo de utilização da assinatura D utilizada para solucionar o problema de nós convergentes.

Neste contexto, surge o algoritmo abaixo, denominado algoritmo A, para nortear a implementação da técnica de CFCSS em um determinado programa.

Algoritmo A:

Passo 1: Identificar todos os blocos básicos, o grafo de fluxo do programa e o número de nós do grafo.

Passo 2: Atribuir um *si* para cada vi, em que si ≠ *sj* se *i* ≠ *j*, i,*j*=1,2,...,N.

Passo 3: Para cada *vj*, *j* = 1,2,...,N.

3.1. Para *vj* cujo *pred(vj)* é somente um nó *vi*, então *dj = si* ⊕ *sj*.

3.2. Para *vj* cujo *pred(vj)* é um conjunto de nós *vi*,1, *vi*,2,...,*viM*, *dj* é determinado por um dos nós como *dj = si*,*1* ⊕ *sj*. Para *vi*,*m*, *m*=1,2,...,M, inserir uma instrução Di, *m = si*,*1*⊕*si*,m em *vi*,*m*.

3.3. Inserir uma instrução *G =G* ⊕ *dj* no começo de *vj*.

3.4. Se *vj* é um nó com mais de um predecessor, então inserir a instrução G = G ⊕ D após *G = G* ⊕ *dj* no nó *vj*.

3.5. Inserir uma instrução "*br(G* ≠ *sj) error*" após as instruções dos dois últimos passos.

Passo 4: Fim Algoritmo.

Quanto à cobertura de falhas, a técnica de CFCSS é capaz de detectar: (1) um desvio ilegal feito para a função de assinatura – primeira linha do nó; (2) um desvio ilegal feito para a instrução "*br(G ≠ s)error*" – segunda linha do nó; (3) um desvio ilegal para o corpo do nó; (4) um desvio inserido dentro do próprio nó, isto é, dentro do próprio bloco básico; (5) a exclusão de uma instrução de desvio incondicional do nó.

E. McCluskey (MCCLUSKEY, 2002) sugere uma pequena modificação no algoritmo A, a fim de diminuir a degradação no desempenho em função da agregação desta técnica. A alteração deve ser feita no item 3.5, ou seja, ao invés de inserirmos a instrução "br(G ≠ *sj*) error" em todos os blocos básicos deve-se eleger alguns blocos para receberem a instrução de verificação. Porém, esta solução aumenta a latência para detecção de falhas de fluxo de controle e conseqüentemente pode aumentar o número de falhas não detectadas pela técnica. Em resumo, o sistema pode gerar saídas erradas até que uma instrução "*br(G ≠ sj) error*" seja executada pelo processador, ou mesmo entrar em *crash* e pendurar.

6.9. Técnica YACCA (Yet Another Control-Flown Checking using Assertions)

A técnica YACCA, proposta por O. Goloubeva (GOLOUBEVA, 2003) consiste em uma solução via *software* capaz de detectar erros de fluxo de controle através do monitoramento de assinaturas inseridas nos programas. A técnica define um conjunto de assinaturas que são geradas em tempo de compilação e uma assinatura atualizada em tempo de execução. O conjunto de assinaturas geradas em tempo de compilação são os identificadores dos blocos básicos e a assinatura gerada em tempo de execução é armazenada em uma variável dedicada e está associada ao valor do bloco básico atual.

A aplicação da técnica baseia-se nos seguintes passos:

- Dividir o programa em blocos básicos e definir seu grafo de fluxo de execução;
- Associar, em tempo de compilação, a cada bloco básico uma assinatura única;
- Inserir em cada bloco básico (*vi*) as seguintes instruções:
  - uma instrução de teste que controla a assinatura do bloco básico anterior (predecessor) e verifica se o desvio ocorrido é válido de acordo com o grafo do programa, ou seja, verifica se *vj* pertence ao grupo de *pred*(*vi*);

– uma instrução de atualização que calcula a variável *code* com o novo valor referente ao bloco atual.

Assim, as duas instruções inseridas nos blocos básicos do programa: instrução de teste e instrução do cálculo da assinatura (*code*), são baseadas em regras e pertencem respectivamente ao conjunto de teste e ao conjunto das assinaturas.

Regras para definir o conjunto de teste:

Durante a execução do programa, quando ocorre uma transição do bloco básico *vj* para *vi* é necessário verificar se esta transição é legal, ou seja, se *brj, i* pertence ao conjunto *Ei*.

Para que este controle seja realizado, a técnica sugere a criação de uma variável denominada PREVIOUSi que contém o produto de todas as assinaturas dos nós predecessores de *vi*, equação (6.5), e a inserção da instrução de teste no início de cada bloco básico, equação (6.6).

PREVIOUSi = Π *Bj*, $\forall$*vj* com desvio *brj,i* $\in$ *Ei*

(6.5)

Onde: *Bj* corresponde a assinatura(s) do(s) nó(s) predecessor(es) de *vi*; *Ei* é o conjunto que contém os desvios legais para vi; *brj,i* representa o desvio de *vj* para *vi*.

if (PREVIOUSi % CODE) error()

(6.6)

Devido à complexidade da instrução acima, equação (6.4), seu processamento torna-se bastante crítico, por isto, são sugeridas duas soluções alternativas representadas pelas equações (6.7) e (6.8).

if ((CODE!= Ba) && (CODE!= Bb) && (...) && (CODE!= Bn)) error()

(6.7)

Onde: Ei = {bra,i; brb,i; ...; brn,i}

ERR_CODE |= ((CODE!= Ba) && (CODE!= Bb) && (...) && (CODE!= Bn))

(6.8)

Onde: Ei = {bra,i; brb,i; ...; brn,i}

Regras para definir o conjunto de atualização das assinaturas:

Dado um determinado bloco básico *vi* a variável *code* será igual a *Bi*, onde *Bi* corresponde à assinatura de *vi*.

A fórmula genérica para o cálculo de *code* é dada pela equação (6.7).

$code = (code \ \& \ M1) \oplus M2$

(6.7)

Onde, M1 representa uma constante calculada a partir das assinaturas dos nós que formam o conjunto dos *pred(vi).* Já M2 representa uma constante gerada a partir da assinatura do nó atual e dos nós que formam o conjunto dos *pred(vi)*.

Os exemplos abaixo demonstram claramente o cálculo da variável *code*.

Exemplo 1: Dado *vi*, seu conjunto de predecessores é:

$pred(vi) = \{vj\}$

$M1 = -1$ e $M2 = Bj \oplus Bi$

Assim $code = code \oplus (Bj \oplus Bi)$

Exemplo 2: Dado *vi*, seu conjunto de predecessores é:

$pred(vi) = \{vj, vk\}$

$M1 = (Bj \ \overline{\oplus} \ Bk)$ e $M2 = (Bj \ \& \ (Bj \ \overline{\oplus} \ Bk) \oplus Bi)$

E assim, $code = (code \ \& \ (Bj \ \overline{\oplus} \ Bk)) \oplus (Bj \ \& \ (Bj \ \overline{\oplus} \ Bk) \oplus Bi))$

A figura 6.16 mostra uma aplicação modificada a partir das regras de transformação de código definidas pela técnica YACCA.

```
ERR_CODE |= (code != 0);
code = code ^ 1; /* code = 1 */
x0 = 1;
y = 5;
i = 0;
while( i < 5 ) {
        ERR_CODE |= (code != 1) && (code != 3);
```

```
                code = (code & 5) ^ 3; /* code = 2 */
                z = x+i*y;
                i = i+1;
                ERR_CODE |= (code != 2);
                code = code ^ 1; /* code = 3 */
        }
        ERR_CODE |= (code != 1) && (code != 3);
        i = 2*z;
```

Figura 6-16 Código modificado a partir da técnica YACCA.

Quanto à cobertura de falhas, esta técnica é capaz de detectar os seguintes tipos de erros de controle:

- um desvio de *vi* para o bloco básico *vj* que, por sua vez, não pertence ao conjunto de *suc*(*vi*);
- um desvio de *vi* para o início do bloco básico *vj* que, por sua vez, pertence ao conjunto de *suc*(*vi*);
- um desvio de *vi* para algum lugar do bloco *vj* que pertence ao conjunto de *suc*(*vi*).

7. Metodologia

7.1. Estudo de caso

A partir da análise teórica e que foi referenciada nos capítulos anteriores, decidiu-se a estratégia de abordagem para a obtenção da solução do problema levantado. Esta decisão considerou o fator tempo, e também a aplicabilidade e confiabilidade dos estudos, pois desejava-se apresentar resultados obtidos, mesmo que não representativos da totalidade de testes necessários já nesta dissertação.

Desta forma, concluiu-se com relação a cada uma das hipóteses anteriormente levantadas:

a) Modificação sobre o grafo de uma aplicação - a possibilidade de trabalharmos com o grafo de uma aplicação qualquer, e a partir deste, realizarmos modificações em sua estrutura, requer profundo estudo e principalmente aplicação da Teoria dos Grafos e de conceitos Topológicos, pois, certamente, deveríamos trabalhar com lemas, teoremas que precisariam ser demonstrados, isto não seria passível de ser desenvolvidos em nove meses de dissertação.
b) Modificação conceitual/estrutural de uma técnica de CFC e/ou proposta de uma nova técnica – para testes e validação de uma nova técnica e/ou teoria, necessitaríamos de um intervalo de tempo maior do que nove meses, principalmente no que se refere a execução de testes e análise de resultados obtidos, para verificar a qualidade da nova técnica proposta.
c) Criação de técnicas de *profiling* para determinação das mudanças sobre técnicas de CFC e modificações sobre o grafo de fluxo de controle – foram examinadas as freqüências de execuções de cada bloco básico, com o objetivo de identificarmos, em cada aplicação analisada, somente os blocos básicos mais executados, a partir de vetores de entrada gerados randomicamente.

Dentre as técnicas de detecção de erros em fluxo de controle adotamos como estudo de caso a técnica de CFCSS (MCCLUSKEY, 2002), reconhecida e validada no meio acadêmico.

Neste trabalho, estamos propondo a utilização da técnica de CFCSS de forma parcial, ou seja, fazendo a proteção do aplicativo para alguns e não para todos os blocos básicos que a constituem, como propõe a técnica original. E estaremos nos referindo a esta proposição de técnica como McCluskey simplificado, que conterá a técnica original cobrindo determinados blocos básicos do aplicativo original, que serão selecionados segundo a técnica de *profiling* descrita no item a seguir.

7.2. Técnica de *profiling* escolhida

Aplicativos do tipo *profilers* (STANDISH,1995) são ferramentas de instrumentação e análise que recolhem informação sobre o desempenho de um programa enquanto este é executado. Fornecem uma visão de alto nível sobre a execução do mesmo, permitindo a identificação das partes mais críticas em termos de desempenho, podendo-se concentrar nestas últimas os esforços de otimização pretendidos.

Inicialmente as técnicas de *profiling* foram propostas pela Engenharia de Software a poucas décadas atrás, e vinham sendo usadas desde então basicamente para observar, agrupar, e analisar dados sobre as características do comportamento de programas em tempo de execução **(**DIEP, 2005).

Para a Engenharia de Software, o emprego de *software profiling* é valioso, pois viabiliza o conhecimento de uma aplicação visando sua melhoria após a fase de desenvolvimento, ou seja, quando efetivamente as aplicações estão em campo. O emprego de *software profiling* demonstra como o *software* está sendo utilizado atualmente **(**ORSO, 2003), quais configurações estão sendo empregadas **(**MEMON, 2004), qual desenvolvimento supostamente pode não funcionar **(**ELBAUM, 2005), onde atividades de validação estão faltando **(**ELBAUM, 2004), ou quais cenários são mais suscetíveis á ocorrência de falhas **(**LIBLIT, 2003).

Adotamos como critério de seleção dos blocos básicos a contagem do número de execução de cada bloco. Para tal seleção, foram realizadas 10 baterias de execuções do aplicativo em avaliação, sendo que em cada uma delas o aplicativo recebeu vetores de 1000 entradas geradas randomicamente. Isto perfaz um total 10.000 diferentes execuções, onde em cada uma

das baterias identificamos os blocos básicos mais executados e o final obtivemos o valor médio para todas as baterias de execuções.

A partir desta avaliação, foram identificados os blocos básicos que receberam a técnica de McCluskey simplificada, segundo dois critérios inicialmente:

a) para aplicativos com número de blocos reduzidos serão selecionados os blocos básicos que tiverem número de execuções expressivo e destacado dos demais;

b) para aplicativos com número de blocos mais expressivos, além dos blocos referenciados acima, serão também considerados para efeito da geração do McCluskey simplificado, os blocos intermediários entre dois blocos mais executados. Com isto, pretendemos proteger áreas consideradas vitais e igualmente importantes do aplicativo.

Existe uma ferramenta para ambiente de desenvolvimento baseada no compilador GCC para realizar *profiling* de código, designada por GCOV(GCOV, 2007). Esta ferramenta permite analisar o resultado de cobertura de código compilado com o GCC. A compilação deve incluir as opções "–fprofile-arc-ftest-coverage".

Ao executarmos o programa desta forma compilado, é gerado o arquivo nomeaplicação.c.gcov que contem o próprio código fonte compilado à esquerda de cada linha algumas informações, entre elas o número de execuções de cada linha do código. Linhas que apresentam o mesmo número de execuções identificam um mesmo bloco básico. A seguir, ilustramos um trecho do arquivo gerado pelo GCOV.

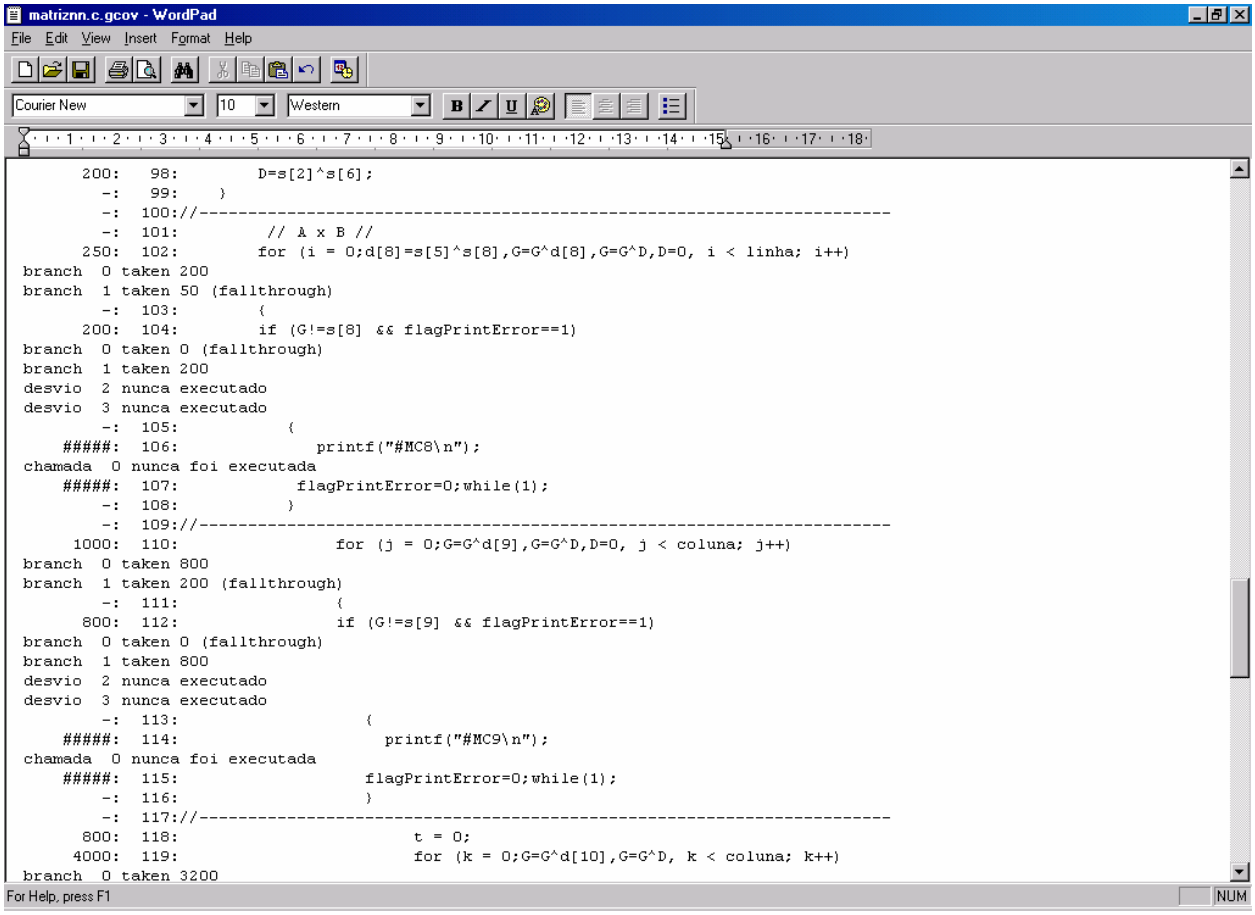


Figura 7-1 Saída produzida pela compilação do programa matriznn.c usando-se o GCOV.

Como vemos na figura 7-1 o arquivo produzido apresenta-se bastante poluído no que se refere à identificação do número de execuções de cada um dos blocos básicos, uma vez que estas informações encontram-se inseridas no arquivo juntamente com o próprio código-fonte do aplicativo. Por isto fez-se necessário a geração de um programa que execute a extração do número do bloco e do número de execuções do arquivo .gcov . Para atender a esta necessidade criamos o programa "extrator" em linguagem de programação C, que faz a extração destas informações automaticamente para um arquivo texto.

Para facilitar e automatizar a análise dos blocos básicos mais executados, foram criados programas em linguagem de programação BASH (NEVES, 2001), para realizar a compilação do aplicativo com diretivas GCOV, execução do mesmo a partir de vetores de entrada “n” vezes e finalmente chamada do aplicativo "extrator" para análise dos blocos básicos mais executados.

7.3. Testes de Injeção de Falhas em *Software*

Com o objetivo de analisar as taxas de cobertura de falhas da técnica de McCluskey simplificado, para validação da técnica proposta neste trabalho, foram realizados os seguintes testes de injeção de falhas via s*oftware*, em tempo de execução na área de memória física da placa de desenvolvimento *Spartan-3 Starter Board* (DIGILENT, 2007) (HSUEK,1997):

a) inserção de instruções de desvio (*jump*);
b) inversão de um bit (*bit-flip);*
c) inserção de instruções de *nop (No OPeration).*

Resumidamente, a injeção das falhas ocorria em tempo de execução, onde a partir da geração de um ponto de parada randômico, o aplicativo era congelado, uma posição de memória também gerada randomicamente era alterada recebendo a inserção de uma falha (*jump*, *nop* ou *bit-flip*). A partir desta inserção o aplicativo continuava a sua execução, e as saídas obtidas direcionadas para arquivo de *log* para posterior análise. As saídas do aplicativo foram posteriormente analisadas para obtenção da contagem do número falhas detectadas e não-detectadas, que foram ilustradas nos gráficos apresentados no capítulo seguinte.

Podemos visualizar o programa injetor de falhas em ação e as saída do aplicativo em testes na figura a seguir.

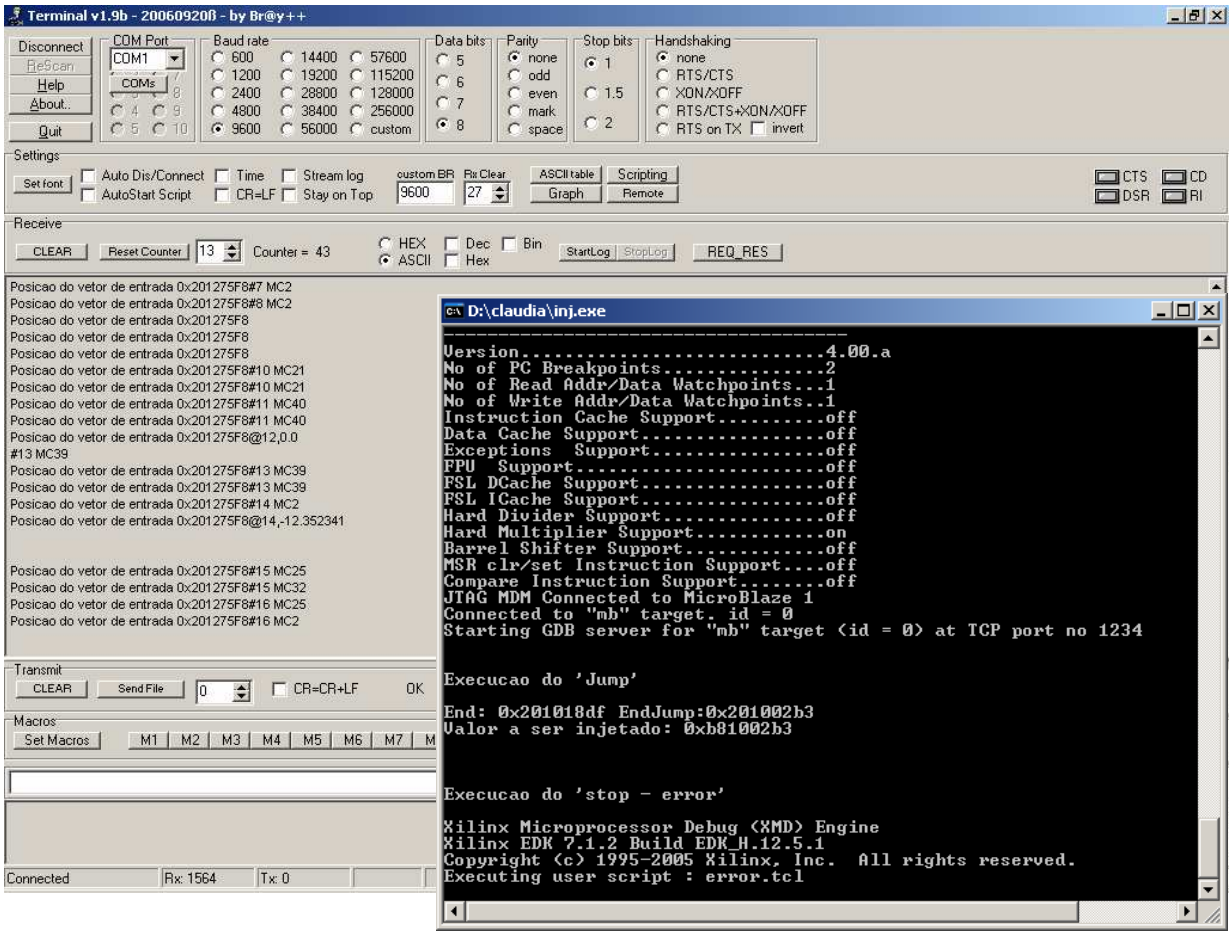


Figura 7-2 Execução do programa Injetor e observação das saídas do programa e mensagens de detecção da falha pela técnica CFC.

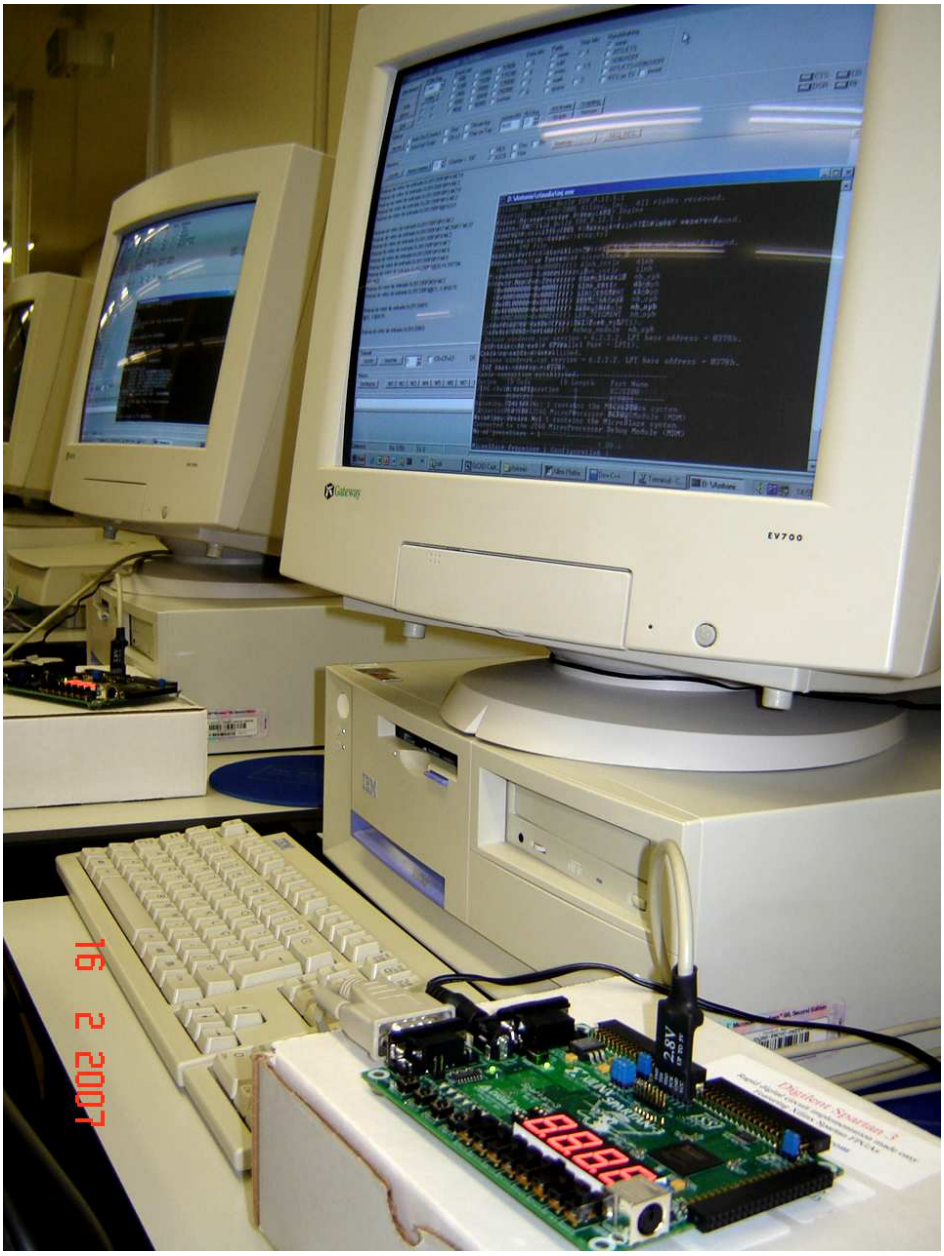


Figura 7-3 Realização das simulações de injeção de falhas no Laboratório SiSC (PUC RS).

Já em fase de final de testes temos uma nova configuração do programa Injetor, que dinamiza o processo de configuração inicial de todos os parâmetros necessários para injeção das falhas, conforme figura 7-4 a seguir.

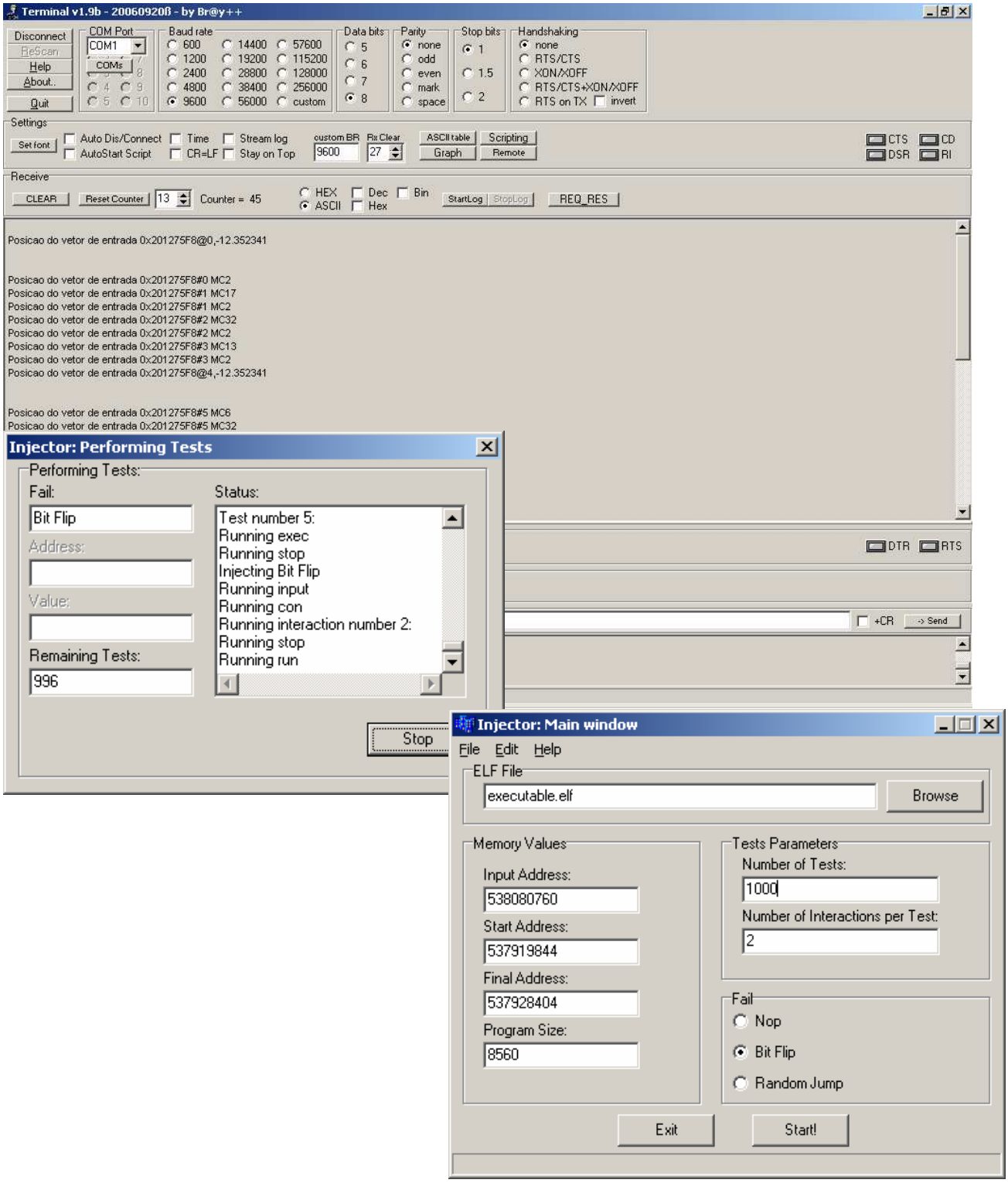


Figura 7-4 – Nova interface do programa Injetor.

8. Resultados

Os experimentos práticos foram realizados para verificar a validação da abordagem proposta em termos de balanceamento de *overhead* de memória e degradação de desempenho em relação às taxas de cobertura de falhas. Três aplicações da técnica CFCSS (MCCLUSKEY,2004) foram implementadas visando este propósito, na forma de duas versões cada uma, CFCSS completo e CFCSS simplificado. As aplicações utilizadas nos testes foram:

a) *Curve Fitting* (CF) – programa que realiza o ajuste de 10 pontos decimais a 5 tipos de curvas, pelo método dos mínimos quadrados;
b) Multiplicação de Matrizes (MM) – multiplicação de matrizes de ordem 4;
c) *InsertSort* (IS) – programa que ordena vetores de 10 números inteiros.

Inicialmente, foram geradas as 3 aplicações acima, cada uma contendo a técnica de CFC completa (CFCSS completo). Na seqüência, foi aplicada a técnica de *profiling* proposta para cada um dos aplicativos, de forma a determinar com que freqüência eram executados cada um dos blocos básicos. Para este propósito, 10.000 vetores de entradas para testes foram gerados randomicamente, divididos em 10 baterias de 1000 testes cada uma.

Para determinar o conjunto de blocos básicos formadores do grafo de fluxo de controle simplificado, três limitações de 50%, 20% e 25% foram arbitrariamente definidas para CF, MM e IS respectivamente. Isto implica que qualquer bloco básico que apresentou uma freqüência de execução igual ou superior que 5000 vezes das 10.000 executadas, foi selecionado para constituir o grafo simplificado do código CF, 800 vezes no caso do código MM, e 500 vezes no caso do código do programa IS. Os histogramas a seguir ilustram os resultados obtidos após a aplicação das técnicas de *profiling.*

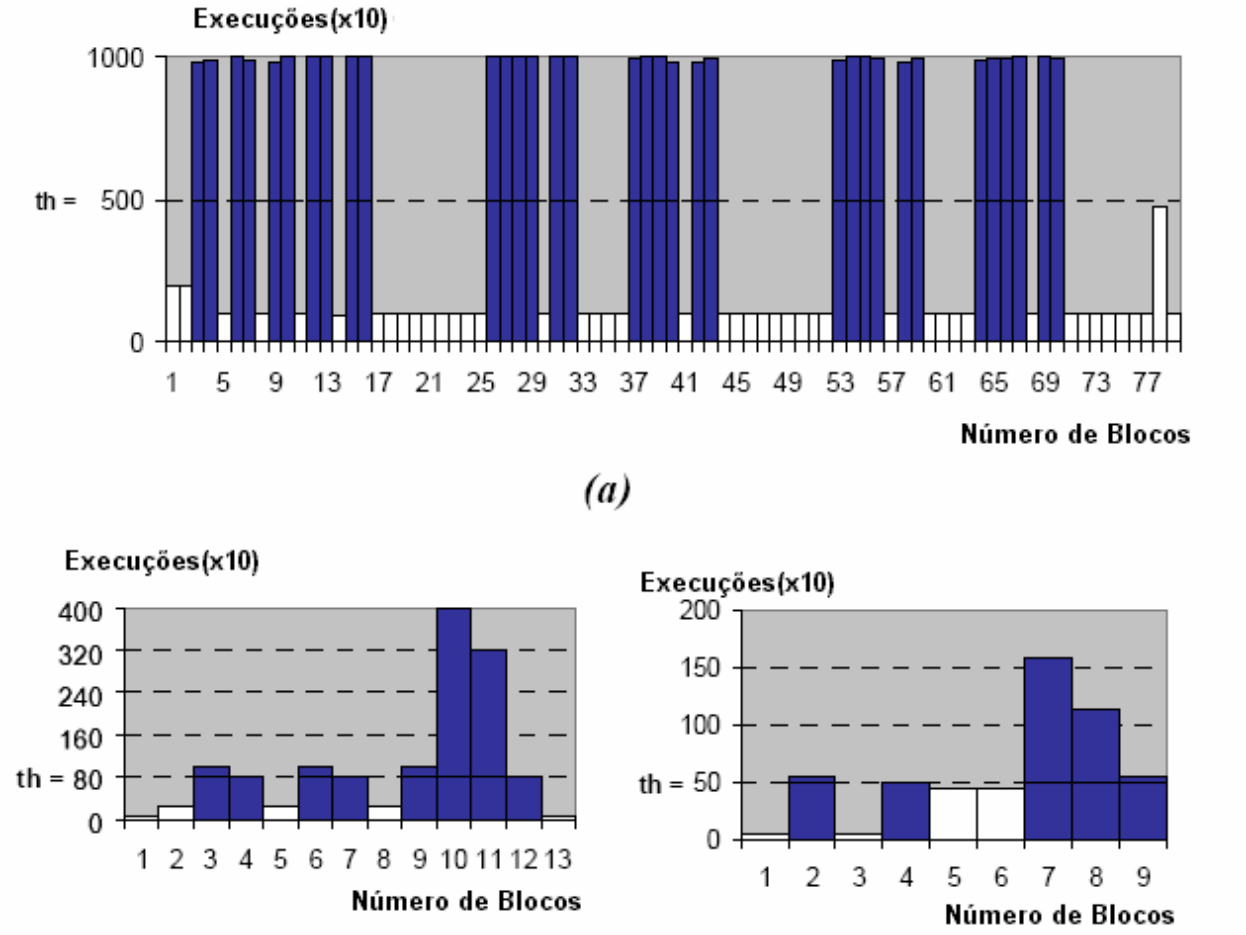


Figura 8-1 – Histograma de execução dos nodos dos grafos *(a)* CF, *(b)* MM e *(c)* IS.

A Figura 8-1 apresenta o grafo de freqüência de execução dos nodos vistos na Figura 8-2. Os nodos mais executados (com freqüência de execução superior ao limitador) são mostrados em cor cinza. Cada nodo representa um bloco básico e cada salto para outro bloco representa uma estrutura condicional durante a execução do código.

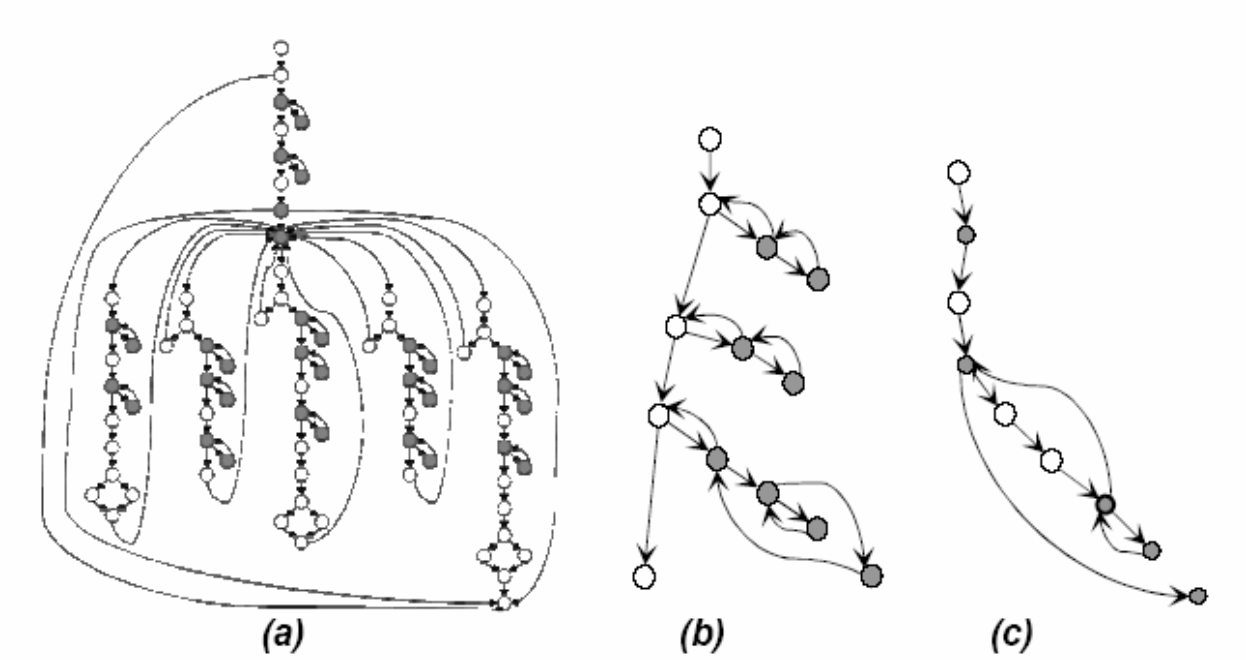


Figura 8-2 – Grafos dos aplicativos *(a)* CF, *(b)* MM e *(c)* IS.

A partir da análise dos histogramas e dos grafos acima obtidos pela aplicação das técnicas de *profiling*, foram geradas as três versões de CFCSS simplificado para cada uma das aplicações, CF, MM e IS. As diferenças entre as versões CFCSS completo e simplificado estão sumarizadas na tabela abaixo.

Tabela 8-1 Resumo dos histogramas obtidos a partir da execução das técnicas de *profiling.*

Aplicação	Número de nodos com assinaturas		**Nós Removidos (%)**
	McCluskey Completo	McCluskey Simplificado	
CF	**79**	**34**	**56,96**
MM	**13**	**8**	**38,46**
IS	**9**	**5**	**44,44**

Uma vez gerada as versões de CFCSS simplificado e somando-se às versões originais usadas na etapa de *profiling*, passamos a contar com um conjunto de 6 versões de aplicativos para a realização dos testes, conforme tabela abaixo.

Tabela 8-2 Demonstrativo das diferentes versões de aplicativos para os testes.

Versões McCluskey	CF	MM	IS
Completa			
Simplificada			

A Figura 8-3 apresenta resultados comparativos de *overhead* de memória (a) e degradação de desempenho (b) entre as versões da técnica CFCSS referenciadas como completo e simplificado, para os aplicativos CF, MM e IS.

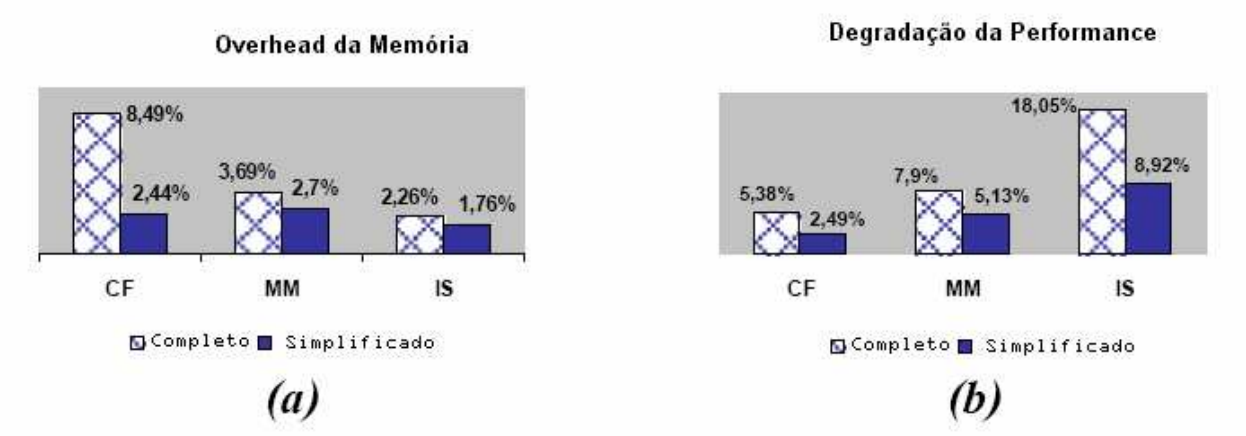


Figura 8-3 (a)*Overhead* de memória e (b) Degradação de *performance*.

Como observado, ao aplicarmos a técnica CFCSS de forma simplificada segundo a técnica de *profiling* executada, resulta um *overhead* de memória de 2,44%, 2,7% e 1,76% para CF, MM e IS, respectivamente. Em contrapartida, se a técnica CFCSS é aplicada sobre todos os blocos básicos de cada aplicativo, os valores de *overhead* de memória são incrementados para 8,49%,3,69% e 2,26%.

Similarmente, a figura 8-3 (b) sumariza os resultados da degradação de *performance*. Foram obtidos os valores de 2,49%, 5,13% e 8,92% para CF, MM e IS ao aplicarmos a técnica CFCSS de forma simplificada, enquanto que se a técnica CFCSS é aplicada sobre todos os nodos do grafo de fluxo de controle, os valores de degradação de *performance* são incrementados para 5,38% , 7,9% e 18,05%, respectivamente.

A partir destes dados, notamos que a técnica simplificada apresentou redução substancial tanto em *overhead* de código quanto em degradação de *performance*. Agora analisaremos as taxas de detecção obtidas ao final da etapa de injeção de falhas nos aplicativos CF, MM e IS.

Na etapa seguinte, simulações de injeção de falhas em *software*, foram realizadas 5 baterias de testes de injeção de falhas para o aplicativo CF e 4 baterias de testes par os aplicativos MM e IS. A distribuição de aplicativos durante os testes de injeção pode ser vista na tabela 8-3. Cada bateria foi composta por 1000 execuções de cada aplicativo, a partir de vetores de entrada gerados randomicamente.

A tabela 8-3 demonstra o volume de 78.000 execuções realizadas durante os testes. Como pode ser observado, foram executados 39.000 testes para cada uma das versões de McCluskey completo e simplificado. Do ponto de vista do número de diferentes execuções/injeções de falhas por aplicativo, temos 30.000 testes para CF e 24.000 para MM e IS.

Tabela 8-3 Número de execuções de baterias de 1000 testes por aplicativos/versão/falha.

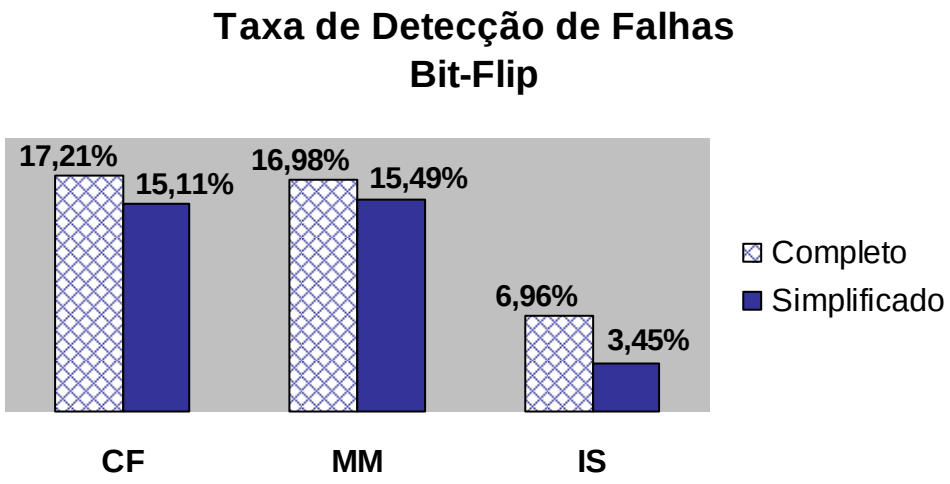
Versão	**MC Completo**			**MC Simplificado**		
Tipo deFalha	**Jump**	**Nop**	**Bit-Flip**	**Jump**	**Nop**	**Bit-Flip**
CF	5x	5x	5x	5x	5x	5x
MM	4x	4x	4x	4x	4x	4x
IS	4x	4x	4x	4x	4x	4x

Três tipos de falhas foram injetadas durante a execução da aplicação no processador *MicroBlaze* mapeado dentro de uma *Spartan 500E FPGA IC* (DIGILENT,2007): *Bit-Flip*, substituição de instruções por uma instrução *Jump* em endereços gerados randomicamente no código, e uma instrução substituída por um *Nop*. Estas falhas foram injetadas individualmente por um procedimento (função) de alta prioridade. Este procedimento interrompe o processador em um momento gerado randomicamente, e produzia uma falha que também é selecionada randomicamente nos registros do processador ou instruções em endereços de memória.

Para o aplicativo CF, os percentuais representam a média obtida a partir de 5 baterias de 1000 testes, em um total de 5000 testes para cada tipos de falha (*Bit-Flip*,*Jump*,*Nop*), ou seja, 15.000 injeções de falhas para a versão CFCSS completa e 15.000 injeções de falhas para a simplificada, totalizando 30.000.

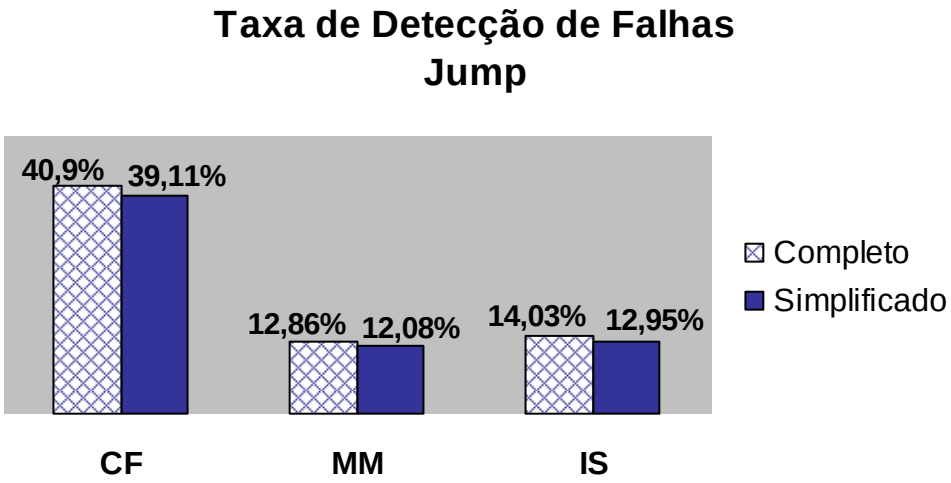
Já para os aplicativos MM e IS, os percentuais representam a média obtida a partir de 4 baterias de 1000 testes, em um total de 4000 testes por falha. Desta forma, tivemos 12.000 injeções de falhas para a versão CFCSS completa e 12.000 injeções de falhas para a simplificada, totalizando 24.000 testes para o aplicativo MM e 24000 testes sobre o aplicativo IS.

A seguir, apresentamos os resultados obtidos segundo cada tipo de falha usada nos testes, sumarizados nas figuras 8-4, 8-5 e 8-6.



Figuras 8-4 Resumo da capacidade de detecção de falhas do tipo *Bit-Flip*.

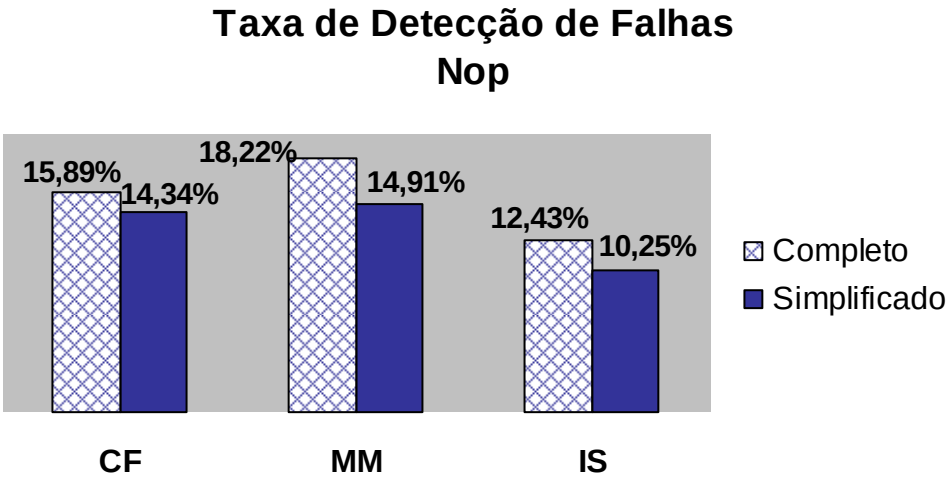
Como observado na figura 8-4, ao aplicarmos a técnica CFCSS sobre todos os blocos básicos de cada aplicativo (CFCSS completo), os valores percentuais de detecção de falhas obtidas são 17,21%,16,98% e 6,96% para CF, MM e IS, respectivamente. Mas se aplicarmos a técnica simplificada proposta, segundo a técnica de *profiling* executada, obtivemos os percentuais de 15,11%, 15,49% e 3,45%, para CF, MM e IS.



Figuras 8-5 Resumo da capacidade de detecção de falhas do tipo *Jump*.

Similarmente, a figura 8-5 sumariza os resultados para falhas do tipo *Jump*. Foram obtidos os valores percentuais de 39,11% , 12,8% e 12,95% para CF, MM e IS ao aplicarmos a técnica CFCSS de forma simplificada, enquanto que se a técnica CFCSS é aplicada sobre todos

os nodos do grafo de fluxo de controle, os percentuais de detecção de falhas são incrementados para 40,9% , 12,86% e 14,03%, respectivamente.



Figuras 8-6 – Resumo da capacidade de detecção de falhas do tipo *Nop.*

Como observado na figura 8-6, ao aplicarmos a técnica CFCSS de forma simplificada segundo a técnica de *profiling* executada, os percentuais obtidos foram 14,34%, 14,91% e 10,25% para CF, MM e IS. Em contrapartida, se a técnica CFCSS é aplicada sobre todos os blocos básicos de cada aplicativo, os valores percentuais são incrementados para 15,89%,18,22% e 12,43%.

Ao final dos testes usando-se falhas dos tipos *Bit-Flip*, *Jump* e *Nop*, nota-se que as taxas de cobertura de falhas apresentadas pelas duas técnicas, completa e simplificada, não apresentam grande discrepância entre seus valores.

Mas cabe destacar que as taxas de cobertura de falhas apresentadas entre as duas versões CFC para falhas do tipo *Jump* possuem valores cuja diferença numérica é reduzida, indicando que a aplicação da técnica simplificada segundo a técnica de *profiling* estabelecida é tão exitosa, no que se refere à cobertura de falhas, quanto a técnica CFCSS original (completa).

9. Conclusões

Embora as técnicas existentes para a monitoração do fluxo de controle de processadores sejam eficientes em termos de cobertura e latência de falhas, elas provocam efeitos colaterais indesejados no sistema embarcado, tais como maior consumo de memória e maior queda de desempenho.

Estas técnicas agregam aos sistemas embarcados um aumento no *overhead* de código binário pela inserção de assinaturas (*checkpoints*) e uma degradação de desempenho, gerada pelo aumento no tempo de execução provocado pelos sucessivos testes que as técnicas incorporam ao código, em sua grande maioria.

Desta forma, foi apresentada nesta dissertação uma proposta para aplicação de uma técnica CFC modificada segundo técnicas de *profiling* que reúne redução em *overhead* de código binário e redução de degradação de desempenho, apresentando também níveis de confiabilidade e proteção de código satisfatórios se comparada a técnica original. Isto é fundamental para aplicações embarcadas, que apresentam fortes restrições de taxas de ocupação de memória e também necessitam de desempenho compatível com a velocidade de resposta que devem apresentar para atender as funcionalidades a que se destinam.

Apresentamos então, neste trabalho, uma solução para este cenário, baseada na redução do número de assinaturas, balanceando de forma criteriosa o aumento de redundância no código através do uso de técnicas de *software profiling*.

A proposta aqui apresentada consiste da inserção de assinaturas somente nos blocos básicos que apresentam maior freqüência de execução. A redução no número de blocos básicos protegidos pela técnica CFC, apresentou significativa redução sobre o consumo de memória e sobre a queda de desempenho, normalmente inerentes à aplicação de técnicas CFC, conforme os teste de injeção de falhas em *software* realizados.

Os resultados obtidos indicam que a versão modificada minimiza o número de assinaturas inseridas no código da aplicação, logo, minimiza o *overhead* de memória e a degradação de *performance*, ao mesmo tempo em que apresenta valores médios de taxa de

detecção de falhas similares e pouco discrepantes dos níveis de cobertura de falhas da técnica original. Isto viabiliza e torna interessante a aplicação da técnica aqui proposta para detecção de falhas de fluxo de controle de sistemas embarcados.

9.1. Trabalhos Futuros

Como sugestão de trabalhos futuros, apresentamos atividades que inicialmente constavam do planejamento inicial, mas que devido ao tempo consumido nas etapas de implementação computacional e execução dos testes, não foram incluídas neste trabalho. Estas atividades estão diretamente relacionadas à revisão bibliográfica apresentada e são elas:

a) implementação da técnica *profiling* proposta sobre outras técnicas CFC, tais como YACCA, ECCA, etc.;
b) uso de heurísticas para tornar o grafo de fluxo de controle valorado, visando a análise posterior do mesmo com relação à aplicação de algoritmos de busca de caminhos em grafos;
c) criação de heurística baseada em fatores considerados críticos para aplicações embarcadas, segundo o contexto de detecção de falhas de fluxo de controle visando à obtenção de maiores taxas de detecção.

Podemos ainda apresentar como sugestão um estudo estatístico a cerca da geração de vetores de entrada para realização dos testes de *software*. Existem modelos estatísticos que apresentam várias distribuições de dados, com diferentes concentrações de pontos, e parece-nos importante investigar este aspecto, neste momento, com o objetivo de otimizar a execução dos testes de injeção de falhas.

10. Referências Bibliográficas

(ALKHALIFA, 1997) Z. Alkhalifa, V.S.S. Nair, *Design of a Portable Control-Flow Checking Technique*, IEEE High-Assurance Systems Engineering Workshop, 1997.

(ALKHALIFA, 1999) Z Alkhalifa, VSS Nair, N Krishnamurthy, JA Abraham. *Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection*, IEEE Trans. On Parallel and Distributed Systems, 1999.

(DIEP, 2005) Diep M.; Elbaum, S.; Cohen, M. *Profiling Deployed Software: Strategic Probe Placement*. . Technical Report CSE-05-08-01/CSE-2005-005, Dept. of Computer Science and Engineering, Univ. of Nebraska-Lincoln, Lincoln, NE, USA, Aug. 2005.

(DIGILENT, 2007) Digilent - *Plataforma de Desenvolvimento*. Disponível em http://www.digilentinc.com/Products/Detail.cfm?Nav1=Products&Nav2=Programmable&Prod=S3BOARD. Último acesso em Fev/2007.

(ELBAUM, 2004) Elbaum S.; Hardojo, M. *An Empirical Study of Profiling Strategies for Released Software and Their Impact on Testing Activities*. International Symposium on Software Testing and Analysis, ACM, June 2004. pp. 65-75.

(ELBAUM, 2005) Elbaum S.; Diep, M. *Profiling Deployed Software: Assessing Strategies and Testing Opportunities*. IEEE Trans. on Software Engineering, 31(4), 2005. pp. 312-327.

(GCOV, 2007) GCOV:*Test coverage program GCC.* Disponível em http://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html#Gcov-Intro . Último acesso em Fev/2007.

(GOLOUBEVA, 2003) O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante. *Soft-error Detection Using Control Flow Assertions*, IEEE Int.l Symp. on Defect and Fault Tolerance in VLSI Systems, 2003, pp. 581-588

(GOLOUBEVA, 2005) Goloubeva O, Rebaudengo M, Reorda M S, Violante M. *Improved Software-Based Processor Control-Flow Errors Detection Technique.* Ann. Reliab. and Maint. Symp. Alexandria, USA. 2005;583-589.

(GROSS, 1999) Gross, Jonathan. *Graph Theory and its applications*. Boca Raton : CRC, 1999.

(HSUEH, 1997) M. Hsueh, T. K. Tsai, R. K. Lyer. *Fault Injection Techniques and tools*. Computer, vol. 30, n. 4 pp. 75-82, Abril 1997.

(KANAWATI, 1996) G. A. Kanawati, V.S.S. Nair, N. Krishanmurthy, J. A. Agraham. *Evaluation of Integrated System-Level Checks for On-Line Error Detection*, IEEE Computer Performance and Dependability Symposium, 1996, pp. 292-301.

(MCCLUSKEY, 2002) N. Oh, P. Shirvani, Edward .J. McCluskey. *Control-Flow Checking by Software Signatures*, IEEE Transactions on Reliability, Vol. 51, No. 2, March 2002, pp. 111-122

(MEMON, 2004) Memon A.; Porter, A.; Yilmaz, C.; Nagarajan, A.; Schmidt,D.; Natarajan, B.; *Skoll: Distributed Continuous Quality Assurance*. International Conference on Software Engineering, May 2004. pp. 449-458.

(NAHMUSK, 2002) Nahmsuk Oh, Subhasish Mitra and Edward J. McCluskey. *ED4I: Error Detection by Diverse Data and Duplicated Instructions*, IEEE Trans. on Computers, Vol 51, nº 2, Feb. 2002, pp. 180-199.

(NEVES, 2001) J. C. NEVES, Julio Cezar. *Programação Shell.* Rio de Janeiro:Brasport,2ed,2001

(ORSO, 2003) Orso A.; Apiwattanapong, T.; Harrold M. J. Leveraging *Field Data for Impact Analysis and Regression Testing.Foundations of Software Engineering*, ACM, Sept. 2003.pp. 128-137.

(PRADHAN, 1996) D. K. Pradhan. *Fault-Tolerant Computer System Design*, Prentice-Hall, 1996.

(REBAUDENGO, 1999) M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante. *Soft-error Detection through Software Fault-Tolerance Techniques*, IEEE Int.l Symp. on Defect and Fault Tolerance in VLSI Systems, 1999.

(REBAUDENGO, 2004) M. Rebaudengo, M. Sonza Reorda, M. Violante. *A New Approach to Software-Implemented Fault Tolerance*, Journal of Electronic Testing: Theory and Applications, 2004.

(RUSSELL, 2003) Russell, Stuart J.. *Artificial intelligence : a modern approach*. 2. ed. Upper Saddle River, NJ : Prentice Hall, c2003.

(SHIRVANI, 2002) N. Oh, P. Shirvani, and E. McCluskey. *Control-flow checking by software signatures*. IEEE Transactions on Reliability, 51(1):111–122, 2002.

(STANDISH ,1995) STANDISH, T.A. Data Structures, *Algorithms & Software Principles in C*. Addison-Wesley: Irvine, 1995.

(WOLF, 2004) Wolf, W. *Embedded is the new paradigm(s)*. Computer, 37,99-101,2004
http://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html#Gcov-Intro