PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

CARLOS ANDRÉ BECKER

DETECÇÃO DISTRIBUÍDA DE FALHAS EM SoC MULTIPROCESSADO

Prof. Dr. Fabian Luis Vargas
Orientador

Porto Alegre
2008

CARLOS ANDRÉ BECKER

DETECÇÃO DISTRIBUÍDA DE FALHAS EM SoC MULTIPROCESSADO

Dissertação apresentada como requisito para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Engenharia Elétrica da Faculdade de Engenharia da Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Dr. Fabian Luis Vargas

Porto Alegre
2008

RESUMO

A crescente evolução da área da microeletrônica nas últimas décadas acarretou um aumento expressivo da capacidade de integração de sistemas em um único *chip*, o que levou à necessidade de novas tecnologias para a análise do correto funcionamento dos sistemas. Observam-se, recentemente, novas arquiteturas de processadores, migrando de uma única CPU (Unidade Central de Processamento) para múltiplos núcleos (tipicamente, 2, 4 e 8 processadores em uma única pastilha). É neste cenário, que evolui de sistemas eletrônicos mono para multiprocessados, que este trabalho se insere, visando propor uma expansão da técnica CFCSS (*Control Flow Checking by Software Signatures*), desenvolvida por Edward J. McCluskey para sistemas monoprocessados, a uma versão aplicável a sistemas com vários processadores em um SoC (*System-on-Chip*).

Quanto à sua estrutura, este trabalho constitui-se de duas partes. A primeira apresenta a taxonomia e os conceitos básicos de sistemas tolerantes a falhas e uma revisão bibliográfica das principais técnicas de detecção de falhas em *software* em sistemas monoprocessados, além de abordar a evolução da tecnologia reprogramável. A segunda parte descreve a metodologia de desenvolvimento das plataformas de *hardware* e de *software*, bem como as etapas realizadas e as dificuldades encontradas. Além disso, apresenta a técnica CFCSS adaptada a vários processadores, o protocolo de comunicação desenvolvido para a realização dos testes e os resultados obtidos.

Assim, este trabalho demonstra caráter inovador e se justifica pela tendência de os sistemas embarcados possuírem, vários processadores e aplicações sendo executadas simultaneamente.

Palavras-chave:

Sistemas multiprocessados. *System-on-Chip* (SoC). Detecção de falhas concorrente. *On-line testing*. Detecção de falhas por *software*.

ABSTRACT

Increasing evolution in the microelectronic field within the last decades has resulted in an expansive growth of integration capacity of systems on a single chip, that has brought about the need for new technologies for analyzing the correct functioning of systems. Recently, new processor architectures have been observed moving from a single CPU (Central Processing Unit) to multiple cores (typically 2, 4 and 8 processors on a single chip). It is a scenario which evolves from mono electronic systems to multiprocessors that this paper lies, aiming at proposing an expansion of CFCSS (*Control Flow Checking by Software Signatures*) technique, which was developed by Edward J. McCluskey for monoprocessed systems, in a version applicable to systems with multiple processors in a single SoC (*System-on-Chip*).

This work is made of two parts. The first part presents taxonomy and basic concepts of fault-tolerating systems, followed by a bibliographical review of main techniques for detecting fault in software in monoprocessed systems, besides approaching reprogrammable technology evolution. The second part describes the development methodology for hardware and software platforms, as well the stages carried out and the difficulties found. In addition, it presents the CFCSS technique adapted to several processors, the communication protocol developed to carry out tests and the results obtained.

This paper presents an innovative profile and is justified by the tendency of embedded systems to have, in the near future, multiple processors and applications being executed simultaneously.

Key-words:
Multiprocessed systems. System-on-Chip (SoC). Concurrent fault detection. On-line testing. Fault detection by software.

LISTA DE FIGURAS

Figura 1: Classes primárias de falhas [1] 21

Figura 2: Criação e mecanismos de manifestação de falhas, erros e defeitos [1] 24

Figura 3: Cadeia de ameaças [1] 25

Figura 4: Blocos básicos [6] 28

Figura 5: Técnica BSSC [6] 29

Figura 6: Técnica BEEC [7] 30

Figura 7: Técnica CCA [8] 32

Figura 8: Técnica ECCA [9] 34

Figura 9: Aplicação do algoritmo de identificação de intervalos livres de saltos [10] 36

Figura 10: Geração de um bloco básico [11] 38

Figura 11: Detecção de um salto ilegal [11] 39

Figura 12: *Branch fan in node* [11] 39

Figura 13: Assinatura de tempo de execução [11] 40

Figura 14: Comparação entre blocos básicos [12] 41

Figura 15: a) Código fonte e b) Grafo do programa [13] 42

Figura 16: Arquitetura FPGA [2] 48

Figura 17-a: FPGA Crosspoint [2] 49

Figura 17-b: FPGA Plessey [2] 49

Figura 18: Implementações de $f = abd + bc\overline{d} + \overline{a}\overline{b}\overline{c}$ [2] 51

Figura 19: Quantidade e área ocupada de bloco em função do número de suas entradas [14] 51

Figura 20: Quantidade de blocos e área de roteamento ocupada em função de *K* [14] 52

Figura 21: Área total normalizada em função de *K* [2] 52

Figura 22: Topologias de *Networks on Chip*: Mesh (a), Torus (b), Ring (c), Octagonal (d) [17] 54

Figura 23: SoC com nodo de comunicação e roteamento adaptativo [24] 56

Figura 24: a) Conexão JTAG e b) *Boundary Scan* [43] 57

Figura 25: Placa de desenvolvimento XUP Virtex II Pro [30] 58

Figura 26: MPMC – Janela de configuração [31] 61

Figura 27: Diagrama simplificado do MPMC2 com dois processadores Power PC [26] .. 62

Figura 28: Diagrama simplificado do sistema implementado 63
Figura 29: Diagrama simplificado do sistema implementado 64
Figura 30: Diagrama de ligação das portas seriais (versão final) 65
Figura 31: Diagrama em blocos completo do sistema embarcado 67
Figura 32: Bloco básico para cálculo da assinatura sincronizada 70
Figura 33: Bloco básico com cálculo de assinatura por fila de espera 71
Figura 34: Diagrama em blocos simplificado do Sistema Injetor de Falhas 80
Figura 35: Relação entre tamanho da fila e tempo de execução 84

LISTA DE TABELAS

Tabela 1: Resultados dos testes com a técnica CFCSS original 81
Tabela 2: Resultados dos testes com a técnica CFCSS distribuída 82
Tabela 3: Medidas de tempo de execução variando o tamanho das filas 84
Tabela 4: Medida do tempo de execução 85
Tabela 5: Medidas do tamanho do código compilado das aplicações 85
Tabela 6: Medidas do tamanho do código fonte das aplicações 85

LISTA DE SIGLAS

ARM – Advanced Risc Machine

BEEC – Block Entry Exit Checking

BFI – Branch Free Interval

BID – Block Identifier

BRAM – Block RAM

BSSC – Block Signatured Self Checking

CCA – Control Flow Checking by Assertions

CFID – Control Flow Identifier

CLB – Configurable Logic Block

CPLD – Complex Programmable Logic Device

CPU – Central Process Unit

CCFC – Concurrent Control Flow Checking

CFCSS – Control Flow Checking by Software Signatures

CLB – Configurable Logic Block

CMOS – Complementary Metal Oxide Silicon

DCM – Digital Clock Manager

DDR – Double Data Rate

DOS – Disk Operating System

DSP – Digital Signal Processing

ECCA – Enhanced Control Flow Using Assertions

ECI – Error Capturing Instruction

EDK – Embedded Development Kit

EMI – Electro Magnetic Interference

FPOA – Field Programmable Object Array

FPGA – Field Programmable Gate Array

GAL – Generic Array Logic

ICFCSS – Improved Control Flow Checking by Software Signatures

IP – Intellectual Property

ISE – Integrated Software Environment

JTAG – Joint Test Action Group

LUT – Look-up Table

MAC – Multiplicator and Acumulator

MOS – Metal Oxide Silicon

MPGA – Mask Programmable Gate Array

MPMC – Multi Port Memory Control

NoC – Network on Chip

OPB – On-Chip Peripheral Bus

PAL – Programmable Array Logic

PC – Personal Computer

PCI – Peripheral Component Interconnect

PLA – Programmable Logic Array

PLB – Processor Local Bus

PLD – Programmable Logic Device

RAM – Random Access Memory

SEU – Single Event Upset

SoC – System on Chip

SRAM – Static Random Access Memory

TTL – Transistor Transistor Logic

UART – Universal Asynchronous Receiver/Transmitter

ULA – Unidade Lógica e Aritmética

VHDL – Very High Speed Integrated Circuit Hardware Description Language

XMD – Xilinx Microprocessor Debugger

XOR – Exclusive Or

XUP – Xilinx University Program

XUPV2P – Xilinx University Program Virtex 2 Pro

YACCA – Yet Another Control-Flow Checking using Assertions

SUMÁRIO

1	**INTRODUÇÃO**	12
2	**FUNDAMENTAÇÃO TEÓRICA**	16
2.1	CONCEITOS BÁSICOS	16
2.1.1	Sistema	16
2.1.2	Defeito	18
2.1.3	Confiabilidade	19
2.1.4	Falha	20
2.1.5	Erro	23
2.2	A PATOLOGIA DE UM DEFEITO – RELAÇÃO ENTRE FALHA, ERRO E DEFEITO	24
2.3	TÉCNICAS DE DETECÇÃO DE FALHAS POR SOFTWARE NO FLUXO DE CONTROLE DE PROCESSADORES	26
2.3.1	Técnica ECI	26
2.3.2	Técnica BSSC	27
2.3.3	Técnica BEEC	29
2.3.4	Técnica CCA	31
2.3.5	Técnica ECCA	32
2.3.6	Técnica CCFC	34
2.3.7	Técnica CFCSS	37
2.3.8	Técnica ICFCSS	41
2.3.9	Técnica YACCA	41
2.4	EVOLUÇÃO DA TECNOLOGIA REPROGRAMÁVEL	45
2.4.1	FPGA	47
2.4.1.1	Granularidade	48
2.4.1.2	Efeitos da Granularidade do Bloco Lógico na Densidade do FPGA	50
2.4.1.3	Efeitos da Granularidade do Bloco Lógico no Desempenho do FPGA	53
2.5	NOC – NETWORKS ON CHIP	53
2.6	SOC – SYSTEM ON CHIP	55

3 **METODOLOGIA E RESULTADOS** 58
3.1 METODOLOGIA 58
3.1.1 Plataforma de *Hardware* 58
3.1.2 Técnica de CFCSS Aplicada a Vários Processadores 68
3.1.3 Protocolo de Comunicação 72
3.1.4 Plataforma de *Software* e Metodologia de Teste 73
3.2 RESULTADOS 81

4 **CONCLUSÃO** 87

REFERÊNCIAS 91

1 INTRODUÇÃO

Com a crescente evolução da área da microeletrônica nas últimas décadas, observa-se um aumento crescente na capacidade de integração de sistemas em um único *chip*. Esse aumento da capacidade de integração aliado a novas metodologias de projeto possibilitou recentemente a criação de sistemas integrados complexos em uma única pastilha denominados de SoC (*System on Chip*), que podem incluir sistemas digitais, analógicos e de rádio freqüência, entre outros.

Seguindo essa tendência, a tecnologia de FPGAs (*Field Programmable Gate Array*)[2] apresenta uma grande evolução, uma vez que é capaz de integrar desde simples componentes desenvolvidos para fazer a "lógica de cola"[1] até componentes complexos, que contêm núcleos de processadores ARM e IBM PowerPC e memórias SRAM de dupla porta com grande capacidade de armazenamento.

Essa evolução segue a tendência do aumento de integração dos componentes eletrônicos, visando diminuir o consumo de energia, minimizar os custos de produção e aumentar a mobilidade dos equipamentos. Isso eleva as possibilidades de novas aplicações embarcadas e de criação de novos produtos e mercados, podendo-se, atualmente, instalar sistemas operacionais uC-Linux[47] e uCOS[48] em várias famílias de FPGAs.

Entretanto, esse aumento da densidade de integração gera desafios cada vez maiores para se garantir a testabilidade dos sistemas embarcados, visto que aumentam as possibilidades e as probabilidades de existirem falhas durante as fases tanto de projeto quanto de funcionamento do sistema. A pergunta a ser feita é a seguinte: *como garantir a funcionalidade de cada componente após sua fabricação e, em caso de falha, qual a abordagem mais eficiente para detectá-la e para corrigí-la?*

Define-se como tolerante a falhas um sistema com a capacidade de fornecer, por redundância, um serviço de acordo com as especificações, independentemente das falhas que

---

[1] Tradução do termo em inglês *glue logic*, que se refere à lógica combinacional utilizada para decodificação de endereços em barramentos de microprocessadores ou aplicada a outra função similar no circuito.

ocorreram ou que ocorrerão. Essa redundância pode ser em *hardware*, em *software* ou em uma combinação de ambos.

Sob o ponto de vista do *software,* há diversas técnicas que possibilitam o monitoramento e a detecção de falhas no fluxo de execução do processador, dentre as quais se pode citar BSSC (*Block Signatured Self Checking*) [6], ECI (*Error Capturing Instruction*) [5,7], CCA (*Control Flow Checking by Assertions*) [8], ECCA (*Enhanced Control Flow Using Assertions*) [9], YACCA (*Yet Another Control-Flow Checking using Assertions*) [13] e CFCSS (*Control Flow Checking by Software Signatures*) [11]. Essas técnicas, que visam à detecção concorrente de falhas que alteram o fluxo de controle com que o processador executa as instruções de um determinado programa em um ambiente monoprocessado, são de extrema importância, pois os ambientes reais onde sistemas embarcados devem operar são ruidosos.

Nesses ambientes, circuitos integrados estão constantemente expostos a ruídos gerados por interferência eletromagnética (EMI) irradiada e conduzida, à radiação produzida por íons de alta energia ou por partículas alfa, a disfunções da fonte de alimentação, tais como *ground bounce*, e a flutuações de V_{DD}, entre outros. Esses fenômenos geram a ocorrência de falhas transientes no sistema, as quais são comumente referenciadas como *bit-flip* ou *single-event upset – SEU*[46]. A manifestação dessas falhas ocorre mediante a alteração do conteúdo de elementos da memória do processador ou da memória RAM do sistema. Embora não sejam de caráter destrutivo, essas falhas induzem a erros no fluxo de execução de instruções à medida que o processador avança na execução do programa, o que, conseqüentemente, pode provocar a paralização ou mau funcionamento do sistema embarcado.

A pesquisa desenvolvida nesta dissertação de Mestrado tem por objetivo propor uma expansão da técnica CFCSS desenvolvida por Edward J. McCluskey para sistemas monoprocessados em uma versão aplicável para sistemas com vários processadores em um único SoC.

Em relação a outras técnicas de detecção de falhas, o CFCSS possui a vantagem de ser implementado totalmente em *software*, dispensando a utilização de componentes físicos externos, o que facilita sua utilização em sistemas cuja área de silício disponível é crítica, ao mesmo tempo em que reduz o custo final do produto pois não são necessários outros

componentes. Como desvantagem, pode-se citar o aumento do tamanho do código devido à inserção de instruções de verificação de erro, o que acarreta o aumento do tempo de execução da CPU, na área de memória necessária para armazenar tal código depois de compilado, e o aumento do consumo de potência, pois o processador dispensa mais ciclos de relógio para executar as instruções redundantes inseridas no código.

A técnica CFCSS baseia-se na divisão do programa em blocos básicos e na inserção de assinaturas digitais no início de cada bloco em tempo de compilação. Um bloco básico é uma região contínua do programa que não possui instruções de desvios. As assinaturas geradas em tempo de compilação são verificadas em tempo de execução através de operações lógicas (XOR). Cada bloco básico possui uma assinatura individual e única em todo o programa. Durante o tempo de compilação, são geradas também as diferenças de assinaturas entre os blocos básicos seguindo o fluxo normal de execução previamente projetado. Essas diferenças são armazenadas, juntamente com as assinaturas, no início de cada bloco básico. A área de dados não é modificada pela técnica de CFCSS permanecendo inalterada.

Durante o tempo de execução, um dos registradores do processador é dedicado a receber a assinatura gerada no início do primeiro bloco e, após a saída do processador deste, atualiza seu valor durante a passagem pelos blocos básicos subseqüentes. Essa atualização é feita mediante uma operação XOR entre o valor atual do registrador e a diferença de assinatura do bloco de entrada, gerada em tempo de compilação. Caso a operação XOR resulte no mesmo valor da assinatura do bloco corrente em execução pelo processador, o programa está no seu fluxo normal. Caso contrário, houve uma falha no fluxo de execução de instruções, de forma que o processador executou um desvio ilegal para o bloco corrente. A detecção desta condição gera um aviso de alerta, no qual o processador inicia um processo de atendimento de interrupção para tratamento da falha ocorrida.

Quanto à sua organização, este trabalho está organizado em duas partes. Na primeira, dedicada à *fundamentação teórica*, apresentam-se a taxonomia e os conceitos básicos de sistemas tolerantes a falhas, as definições de falha, de erro e de defeito e a relação entre eles [1], bem como se procede a uma revisão bibliográfica das principais técnicas de detecção de falhas em *software* em sistemas monoprocessados – ECI [5], BSSC [6], BEEC [7], CCA [8], ECCA [9], CCFC [10], CFCSS [11], ICFCSS [12] e YACCA [13]. Também se aborda a evolução da tecnologia reprogramável, desde a estrutura de FPGAs e sua granularidade [2,

14], até NoCs [15, 16, 17, 18, 19, 20] e, posteriormente, SoCs [21, 22, 23, 24, 25, 26, 27, 28, 29].

A segunda parte é dedicada à *metodologia e aos resultados práticos*. Descreve-se, então, a metodologia de desenvolvimento do trabalho a partir das plataformas de *hardware* e de *software*, juntamente com as dificuldades encontradas ao longo da pesquisa. Posteriormente, apresenta-se a técnica de CFCSS adaptada a vários processadores, o protocolo de comunicação interprocessador desenvolvido e os testes e resultados práticos obtidos para validar a técnica proposta.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 CONCEITOS BÁSICOS

Nesta seção, serão apresentados os conceitos básicos que serão utilizados ao longo deste trabalho, como sistema, defeito, erro, falha, bem como as relações entre eles. As definições a seguir baseiam-se nos trabalhos de Avizienis, Laprie, Randell e Landwehr [1]. Os conceitos apresentados por esses autores englobam todo o conteúdo da taxonomia de confiabilidade de sistemas na área de sistemas computacionais e de comunicações, mas somente os conceitos necessários a esta dissertação serão agora apresentados. Ao lado de cada definição, manteve-se a palavra original, em inglês, entre parênteses, a fim de se evitarem interpretações equivocadas caso sejam consultadas outras fontes de referência.

2.1.1 Sistema

Um **sistema** (*system*) é uma entidade que interage com outras entidades ou sistemas, como, por exemplo, *hardware*, *software,* pessoas ou o próprio mundo físico com seus fenômenos naturais. Definem-se estes outros sistemas como o **ambiente** (*environment*) de um dado sistema. Portanto, a **fronteira do sistema** (*system boundary*) é a fronteira comum entre o sistema e o seu ambiente.

Pode-se observar, com isso, que é muito importante a correta caracterização do ambiente em que o sistema será introduzido, pois, dessa forma, a abordagem de projeto será completamente diferente. É possível exemplificar essa afirmação citando-se dois sistemas: um sistema que irá operar dentro de um centro de processamento de dados climatizado e um outro sistema que irá operar em um satélite no espaço. O ambiente espacial é muito mais agressivo do que o ambiente terrestre devido às radiações eletromagnéticas e ionizantes [46] e a uma variação muito maior de temperatura. A fronteira entre esses dois ambientes é diferente; com isso, a metodologia de projeto, os componentes utilizados na sua construção, as interfaces e os algoritmos que integrarão o sistema devem considerar, para seu correto funcionamento, as

interferências e as suas correspondentes intensidades que o sistema receberá na fronteira com esse ambiente.

Os sistemas de computação e comunicação são caracterizados por propriedades fundamentais. Essas propriedades são funcionalidade (*funcionality*), desempenho (*performance*), confiabilidade (*dependability*), segurança (*security*) e custo (*cost*).

Com relação à funcionalidade, define-se como **função** (*function*) de um sistema o que o sistema deve realizar, e isso é descrito pelas especificações funcionais dele em termos de funcionalidade e desempenho.

Já o **comportamento (***behavior***)** do sistema é definido como o que o sistema faz para realizar sua função, e esse comportamento é descrito por uma seqüência de estados. O **estado total** (*total state*) de um dado sistema é definido como o conjunto dos seguintes estados: cálculo, comunicação, informação armazenada, interconexões e condições físicas.

A **estrutura** (*structure*) de um sistema é o que o habilita a gerar seu comportamento. Considerando o ponto de vista estrutural, um sistema é composto por uma série de componentes ligados entre si de modo a interagirem; sob esse ponto de vista, cada componente é considerado outro sistema. Essa recursão termina quando o componente é considerado **atômico (***atomic***)**, ou seja, quando já não é mais possível observar uma estrutura menor, ou ela não interessa mais e pode ser ignorada. Conseqüentemente, o estado total do sistema é o conjunto dos estados externos de seus componentes atômicos.

Para facilitar o entendimento desses conceitos, pode-se analisar a evolução tecnológica dos FPGAs [2], cujos blocos básicos inicialmente eram transistores — granularidade pequena — que podiam ser interconectados para realizar determinada função. Esses transistores podem ser considerados componentes atômicos, pois não podem ser subdivididos sem perderem sua funcionalidade. Com o aumento da capacidade de integração no silício, esses transistores passaram a ser organizados em *flip-flops,* registradores, multiplexadores, LUTs, CLBs — granularidade média — e, por fim, memórias, processadores, multiplicadores — granularidade grande. Esse aumento na granularidade tem por objetivo aumentar o tamanho dos **elementos atômicos (***atomic elements*) que compõem o FPGA, facilitando o roteamento. Observa-se que, no caso de se aumentar a granularidade, considera-se como elemento atômico

a menor estrutura correspondente ao tamanho do grão — transistor, registrador, LUT. Ao se utilizar um FPGA de grão grande, não se está interessado no estado individual de cada transistor interno que compõe esse grão, mas no estado total dos elementos atômicos.

Outro conceito importante é o de **serviço.** Um **serviço** (*service*) que é entregue por um sistema — considerado como **servidor** (*server*) — é o comportamento que é percebido pelo seu usuário. Um **usuário** (*user*) é outro sistema que recebe serviços de um outro provedor. A **interface de comunicação** (*service interface*) é a parte da fronteira do sistema servidor que entrega um determinado serviço ao usuário. A parte do estado total do servidor que é percebida na interface de comunicação — e que também pode ser observada pelo usuário — é denominada de **estado externo** (*external state*); o restante da parte do estado total do servidor não percebida na interface de serviço é denominado de **estado interno** (*internal state*). A entrega de um serviço é uma seqüência de estados externos do provedor. Pode-se notar que um sistema pode ser seqüencialmente ou simultaneamente tanto provedor como usuário em relação a outro sistema. A **interface de aplicação** (*use interface*) é a interface que recebe serviços do usuário.

É importante lembrar que um sistema pode implementar mais de uma função e entregar mais de um serviço. Serviços e funções podem ser, portanto, vistos como uma composição de itens de serviço e itens de função respectivamente.

2.1.2 Defeito

Serviço correto (*corret service*) é o serviço que é entregue quando se executa corretamente a função do sistema. Um **defeito** (*failure*) é um evento que ocorre quando o serviço entregue se desvia do serviço correto. Um serviço falha ou porque ele não completou a especificação funcional como deveria, ou porque a especificação não foi adequadamente descrita na função do sistema. Um defeito no serviço é uma transição de um serviço correto para um serviço incorreto. A transição de um serviço incorreto para um serviço correto é denominada de **restabelecimento do serviço** (*service restoration*). O intervalo de tempo em que o serviço é entregue incorretamente é chamado de **interrupção do serviço** (*service outage*).

O desvio do serviço correto pode assumir diferentes formas, que são chamadas de **modos de defeito do serviço** (*service failure modes*) e que são classificadas de acordo com a **gravidade do defeito** (*failure severities*). Considerando que um serviço é uma seqüência de estados externos do sistema, um defeito no serviço significa que pelo menos um ou mais estados externos do sistema se desviaram do estado correto do serviço. Esse desvio é chamado de **erro** (*error*). A causa que gerou o erro é chamada de **falha** (*fault*).

Quando a especificação funcional de um sistema inclui um conjunto de várias funções, o defeito em um ou mais serviços implementados nas funções pode levar o sistema a operar em um **modo degradado** (*degraded mode*), que continua a fornecer um subconjunto dos serviços necessários ao usuário. A especificação desse modo degradado pode ser identificada mediante, por exemplo, serviços lentos, limitados e de emergência. Neste caso, considera-se que o sistema sofreu um **defeito parcial** (*parcial failure*) em sua funcionalidade ou desempenho. Podem ocorrer também defeitos parciais durante a fase de desenvolvimento que interferem na confiabilidade do sistema.

2.1.3 Confiabilidade

A definição original de confiabilidade é a capacidade de entregar um serviço que pode justificavelmente ser confiável. A questão é definir o que é *confiável* sob o ponto de vista de sistemas computacionais. A definição alternativa que estabelece um critério de decisão mais objetivo com relação à confiabilidade do serviço fornecido pelo sistema é a **confiabilidade do sistema** (*dependability*), que é a capacidade de evitar defeitos no serviço mais freqüentes e mais graves que o aceitável.

Com a crescente evolução da área tecnológica das últimas décadas, a confiabilidade é um conceito integrado que engloba os seguintes atributos:

- **disponibilidade** (*availability*): prontidão para o correto serviço;
- **fidedignidade** (*reliability*): continuidade no correto serviço;
- **segurança** (*safety*): ausência de conseqüências catastróficas ao(s) usuário(s) e ao ambiente;

- **integridade** (*integrity*): ausência de alterações impróprias no sistema;
- **manutenabilidade** (*maintainability*): habilidade para suportar modificações e reparos.

Durante os últimos cinqüenta anos, foram desenvolvidas muitas técnicas para se obterem as características de confiabilidade. Essas técnicas podem ser agrupadas em quatro categorias principais:

- **Prevenção de Falhas** (*fault prevention*): modos para prevenir a ocorrência ou a introdução de falhas;
- **Tolerância a Falhas** (*fault tolerance*): modos de evitar defeitos no serviço na presença de falhas;
- **Remoção de Falhas** (*fault removal*): modos de reduzir o número e a gravidade das falhas;
- **Previsão de Falhas** (*fault forecasting*): modos de estimar o número atual, as incidências futuras e os tipos de conseqüências das falhas.

Prevenção de Falhas e Tolerância a Falhas visam assegurar a capacidade de fornecer um serviço que pode ser confiável, enquanto Remoção de Falhas e Previsão de Falhas buscam alcançar certeza nessa capacidade, garantindo que as especificações funcionais e a confiabilidade são adequadas e que o sistema consegue atingi-las.

2.1.4 Falha

Todas as falhas que podem afetar um sistema durante sua vida útil são classificadas de acordo com oito pontos de vista básicos, o que resulta nas classes elementares de falhas conforme mostra a Figura 1.

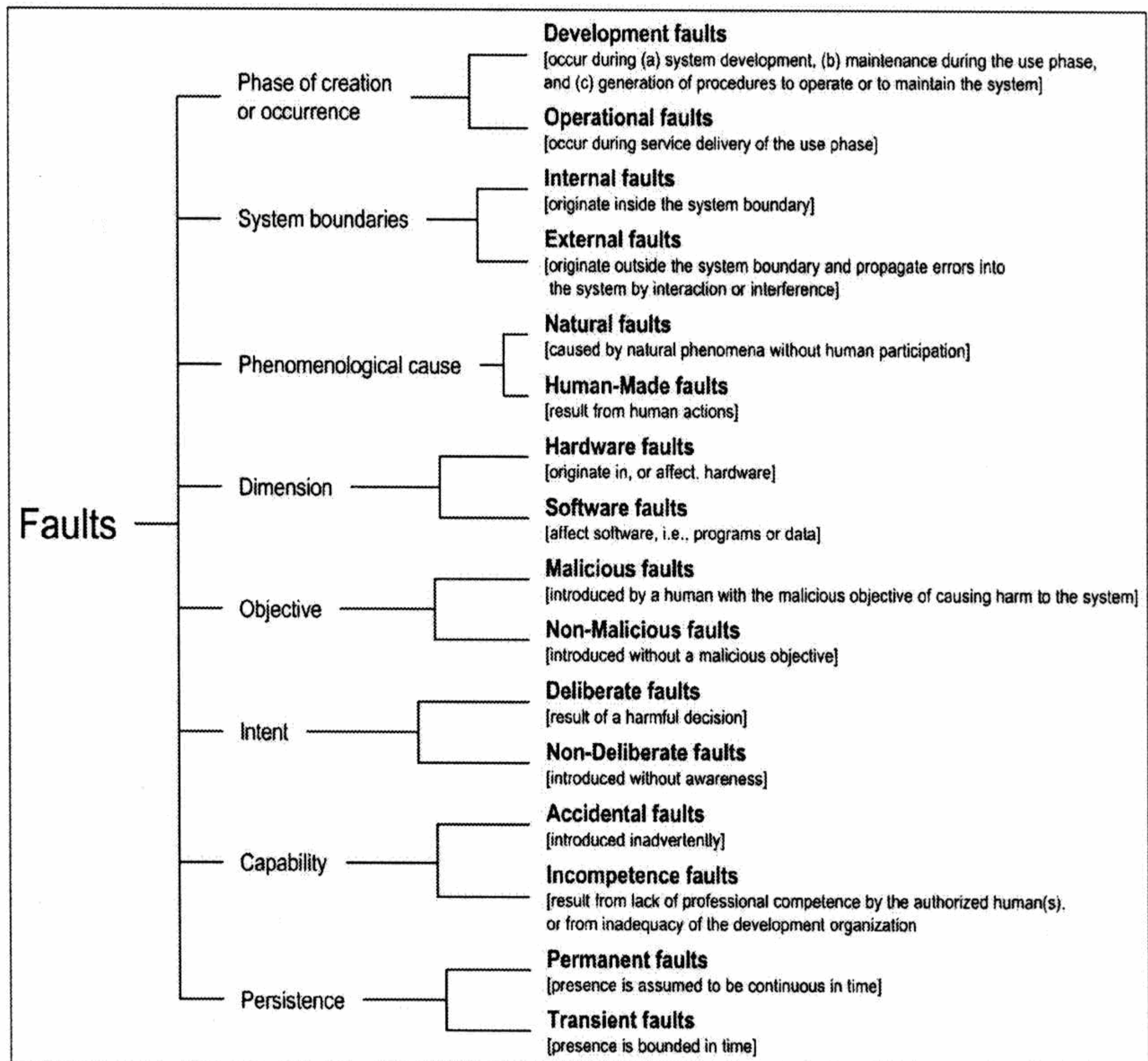


Figura 1: Classes primárias de falhas [1].

Se fosse viável realizar as oito combinações possíveis, haveria o total de 256 combinações. Os tipos de falhas mais importantes relacionados aos sistemas embarcados são

- falhas no desenvolvimento;
- falhas físicas;
- falhas de interação.

As **falhas no desenvolvimento** são todas as classes de falhas que ocorrem durante a fase de desenvolvimento, tanto em *software*, quanto em *hardware*, causadas por erros na metodologia de projeto. Incluem-se neste grupo as falhas injetadas nos sistemas devido aos *bugs* presentes nas ferramentas de desenvolvimento. Este tipo de falha ocorre, na maioria das vezes, pela pressão do mercado quanto à velocidade exigida no ciclo de desenvolvimento do projeto, conhecida como *time-to-market*. Como o foco é o tempo de desenvolvimento do

projeto, pouco tempo é investido no seu planejamento, o que ocasiona falhas conceituais, devido a pouca análise das variáveis presentes e de sua inter-relação com os demais subsistemas que integram o projeto. Na literatura especializada, citam-se casos como o do veículo lançador de satélites Ariane V [3] e o do acelerador de partículas Therac-25 utilizado em radioterapia [4]. O caso Therac-25 destaca-se por causa de uma falha no desenvolvimento do projeto que removeu as travas de segurança em *hardware* e substitui-as por *software*. Isso causou a morte de mais de dez pessoas devido a uma falha na interface do computador com o operador, que aumentava a dose de radiação sobre o paciente. O caso do foguete Ariane V apresenta um erro no desenvolvimento do projeto devido ao reuso de *software* da versão anterior do foguete. Um erro de dimensão de uma variável que controlava a velocidade horizontal do foguete provocou seu *overflow*, ocasionando o desvio de rota e a conseqüente explosão do foguete que continha dois satélites.

As **falhas físicas** são todas as classes de falhas que afetam o *hardware* do sistema, dentre as quais se incluem as devidas ao desgaste dos componentes, como, por exemplo, a eletromigração e a oxidação natural dos terminais dos componentes e das trilhas das placas de circuito impresso. Neste grupo, incluem-se as falhas geradas por fenômenos de ordem natural sem a participação humana, como as descargas atmosféricas, as interferências eletromagnéticas, as radiações ionizantes e o fenômeno de *crosstalking* entre vias do *chip*. Todas elas produzem alguma falha – permanente ou transiente – no nível físico do sistema. Importa salientar que, durante o funcionamento do sistema, as falhas naturais podem ser internas ao sistema, devido à deterioração natural dos materiais utilizados na fabricação do componente, ou externas ao sistema, originadas fora dos limites do sistema, causadas por interferências eletromagnéticas ou ionizantes que penetram pelas vias de interface do sistema (transientes na alimentação, ruídos na entrada de dados etc.).

Nos sistemas embarcados, as falhas mais importantes são as geradas por interferência eletromagnética e por falhas na metodologia e no desenvolvimento dos sistemas que compõem o SoC.

As **falhas de interação** ocorrem durante a fase de uso do sistema, portanto são todas as falhas de tipo operacional. Elas são causadas por elementos do ambiente de uso que interagem com o sistema, os quais são, portanto, externos ao sistema. A maioria dos casos de

falhas de interação é causada por ações humanas, como erros de entrada de dados e/ou de configuração.

O trabalho descrito nesta dissertação classifica-se na categoria de Tolerância a Falhas por desenvolver uma técnica de detecção distribuída de falhas no fluxo de execução de instruções de uma aplicação.

2.1.5 Erro

Um erro é definido como uma parte do estado total do sistema que pode levar a um defeito; por sua vez, um defeito ocorre quando um erro causa um desvio na entrega correta do serviço pelo sistema. A causa de um erro é chamada de falha.

Um erro é detectado se sua presença é indicada por uma mensagem ou um sinal de erro. Erros que estão presentes no sistema, mas que não foram detectados são chamados de erros latentes.

Considerando que um sistema consiste em um conjunto de componentes que interagem, o estado total do sistema é o conjunto dos estados dos componentes individuais. Essa definição implica que uma falha causa originalmente um erro no estado de um (ou mais) componente do sistema, mas o defeito no serviço não irá ocorrer enquanto o estado externo do componente não fizer parte do estado externo do sistema.

Sempre que um erro se torna parte do estado externo do componente, ocorre um defeito no serviço desse componente, mas o erro permanece interno a todo o sistema. De qualquer forma, um erro irá levar a um defeito no serviço dependendo de dois fatores:

1. da estrutura do sistema e, especialmente, da natureza de qualquer redundância que exista nele, a qual pode ser classificada de duas formas:
    a. **protetora:** redundância introduzida para fornecer tolerância a falhas, que é explicitamente intencional para prevenir um erro que leve a um defeito no serviço;

   b. **não intencional:** redundância que pode ter o mesmo – supostamente inesperado – resultado de uma redundância protetora. Na prática é difícil, se não impossível, construir um sistema que não tenha nenhuma forma de redundância e que seja tolerante a falhas;
2. do comportamento do sistema: a parte do estado do sistema que contém um erro pode nunca ser necessária para um determinado serviço, ou um erro pode ser eliminado (sobrescrito) antes de levar a um defeito.

Alguns tipos de falhas, como rajadas de radiações eletromagnéticas, podem causar erros, simultaneamente, em mais de um componente do sistema. Esses tipos de erros são chamados de múltiplos erros associados. Por sua vez, erros que afetam somente um componente do sistema são chamados de erros simples.

2.2 A PATOLOGIA DE UM DEFEITO – RELAÇÃO ENTRE FALHA, ERRO E DEFEITO

A criação e os mecanismos de manifestação de falhas, erros e defeitos são apresentados na Figura 2 e resumidos a seguir.

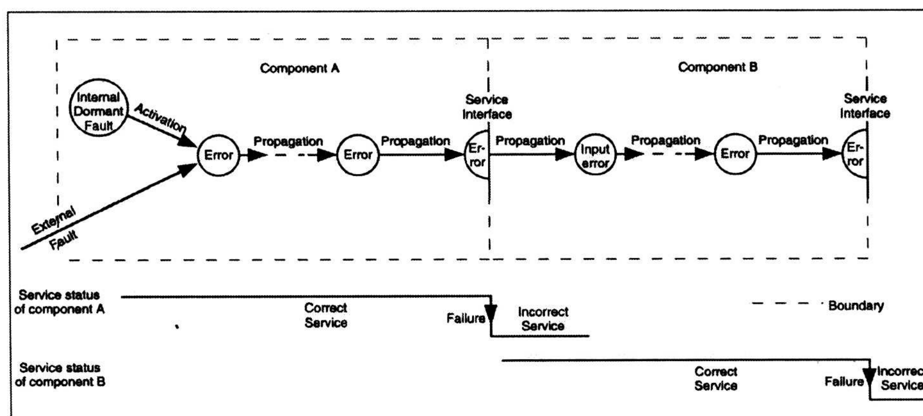


Figura 2: Criação e mecanismos de manifestação de falhas, erros e defeitos [1].

Uma falha é ativa quando produz um erro; caso contrário, diz-se que ela está **dormente**. Uma falha ativa pode ser tanto uma falha interna, quanto uma falha externa. No primeiro caso, trata-se de uma falha interna que estava previamente latente e que foi ativada ou pela execução de um processo ou por condições ambientais adversas (interferências). No segundo caso, trata-se de uma falha de origem externa. A ativação de uma falha é a aplicação

de uma entrada (conhecida como padrão de ativação) em um componente, o qual causa a ativação da falha dormente.

A propagação de um erro dentro de um dado componente (propagação interna) é causada pelo processo computacional. Um erro é sucessivamente transformado em outros erros. A propagação de um erro de um componente A para um componente B que recebe o serviço do componente A (propagação externa) ocorre quando, mediante a propagação interna, um erro alcança a interface de comunicação do componente A. Neste instante, o serviço entregue pelo componente A para o componente B torna-se incorreto, o que gera um defeito no serviço do componente A que se apresenta como uma falha externa na entrada do componente B e que propaga um erro dentro de B através de sua interface de comunicação.

Um defeito no serviço ocorre quando um erro é propagado para a interface de serviço do sistema e causa o desvio na entrega correta do serviço. Um defeito no serviço de um sistema causa uma falha permanente ou transiente para outros sistemas que recebem serviço desse sistema.

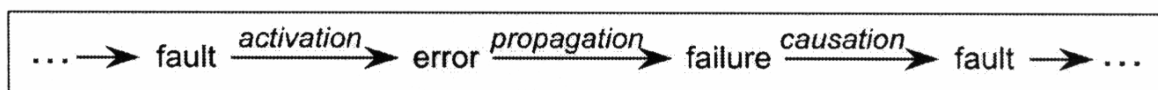


Figura 3: Cadeia de ameaças [1].

Esses mecanismos habilitam a "cadeia de ameaças", como mostra a Figura 3. As flechas nessa cadeia indicam a relação de causalidade entre falhas, erros e defeitos. Elas devem ser interpretadas genericamente: pelo mecanismo de propagação, vários erros podem ser gerados antes de um defeito ocorrer.

A habilidade de identificar o padrão de ativação de falhas que causam um ou mais erros é denominada de reprodutibilidade de ativação de falhas. As falhas podem ser categorizadas de acordo com sua reprodutibilidade de ativação. Falhas cuja ativação é passível de reprodução são chamadas de sólidas ou *hard*, enquanto falhas cuja ativação não é sistematicamente repetitível são denominadas *soft*. A grande maioria das falhas residuais no desenvolvimento de grandes sistemas de *software* é de falhas do tipo *soft*. O mecanismo de ativação dessas falhas é suficientemente complexo, e conseqüentemente sua ativação depende

de complexas combinações de estados internos e entradas externas, as quais ocorrem raramente e que são, portanto, difíceis de reproduzir.

2.3 TÉCNICAS DE DETECÇÃO DE FALHAS POR *SOFTWARE* NO FLUXO DE CONTROLE DE PROCESSADORES

Neste subcapítulo, descrevem-se diversas técnicas de detecção de falhas apresentadas na literatura especializada. As técnicas de detecção de falhas se classificam em dois grandes grupos – as técnicas de detecção de falhas em *hardware* e as em *software*.

As técnicas de detecção de falhas em *hardware* baseiam-se, de modo geral, no acréscimo de algum dispositivo eletrônico externo ou interno ao sistema, cuja função é monitorar a execução da aplicação em tempo real. Caso seja detectado algum desvio na execução correta da aplicação, haverá a sinalização da falha para que o sistema proceda ao tratamento dessa exceção.

Já as técnicas de detecção de falhas por *software* fundamentam-se, em grande parte, na introdução de trechos de código na aplicação que visam à detecção de falhas no fluxo de dados ou de instruções. As técnicas de detecção de falhas no fluxo de instruções consistem na criação e/ou na inserção de assinaturas ou de *checkpoints* localizados em pontos estratégicos da aplicação. As técnicas de detecção de falhas no fluxo de dados baseiam-se na duplicação das variáveis, na qual os dados da aplicação serão armazenados a fim de se realizar a verificação de ocorrências de falhas nesses dados.

2.3.1 Técnica ECI

A técnica ECI [5] – *Error Capturing Instruction*– é uma técnica simples de detecção de falhas no fluxo de execução, consistindo na inclusão de instruções específicas nas áreas de memória não utilizadas durante o fluxo de execução normal do programa principal. Caso haja

uma falha que leve ao desvio do fluxo de execução, a instrução de captura irá indicar a ocorrência de uma falha.

A instrução de captura é uma instrução do conjunto de instruções do processador que, quando acionada, desvia o fluxo de execução para uma rotina de captura (um *loop* infinito, por exemplo) cujo objetivo é impedir a execução errônea do programa. Para detectar o erro, um circuito de timer (*watchdog*) deve ser utilizado.

Outra maneira de detectar desvios no fluxo de execução é utilizar uma instrução de interrupção do processador, como uma instrução ECI; neste caso, o erro é detectado pelo próprio processador, que executa uma rotina de interrupção para a recuperação do sistema. As instruções ECI devem ser inseridas nas áreas de memória não utilizadas, não podendo ser colocadas na área da pilha da memória.

2.3.2 Técnica BSSC

A técnica BSSC [6] – *Block Signature Self Checking Mechanism* – baseia-se na técnica de divisão do programa em blocos básicos e na inserção de assinaturas. Por definição, um bloco básico é uma região de programa cujas instruções são executadas seqüencialmente sem a presença de instruções de salto de qualquer tipo e onde o fluxo de execução entra na parte superior e sai na parte inferior. Durante a execução do programa, as assinaturas são verificadas, possibilitando a conferência do fluxo de execução nos pontos de entrada e de saída dos blocos. Qualquer erro nessa execução pode ser detectada e tratada.

Os blocos básicos são separados por *instruções de partição*. Uma instrução de partição é qualquer instrução que pode desviar o fluxo de execução do programa ou qualquer instrução de destino de uma mudança do fluxo de execução. A Figura 4 ilustra a divisão do programa em blocos básicos e em instruções de partição.

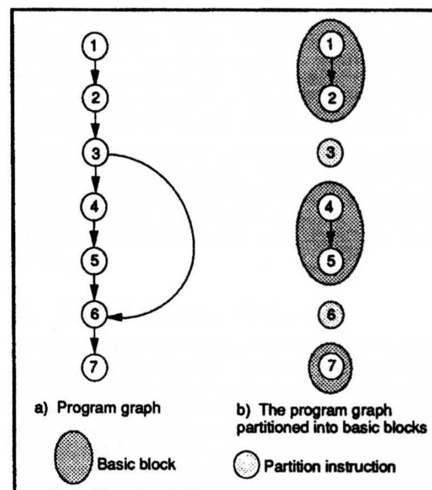


Figura 4: Blocos básicos [6].

Pode-se observar que a instrução 3 é uma instrução de salto condicional e, portanto, uma instrução de partição. Já a instrução 6 é uma instrução de partição, pois ela é uma instrução de *destino* da instrução de salto 3, mesmo não sendo uma instrução de salto condicional.

Para permitir a detecção de erros no fluxo de execução, os blocos básicos são modificados de acordo com o diagrama da Figura 5. Como se observa, uma sub-rotina é inserida no início e no fim do bloco básico. A sub-rotina no início do bloco armazena a assinatura do bloco em uma variável estática que pode ser um número previamente escolhido, o endereço de retorno da sub-rotina de entrada ou o endereço da primeira instrução do bloco básico.

A sub-rotina de saída compara a assinatura em tempo de execução com a assinatura armazenada no bloco. Se uma discrepância é detectada, a rotina de saída invoca uma rotina de exceção para recuperação do contexto (*roll back* ou outro tipo); caso contrário, a execução do programa segue normalmente.

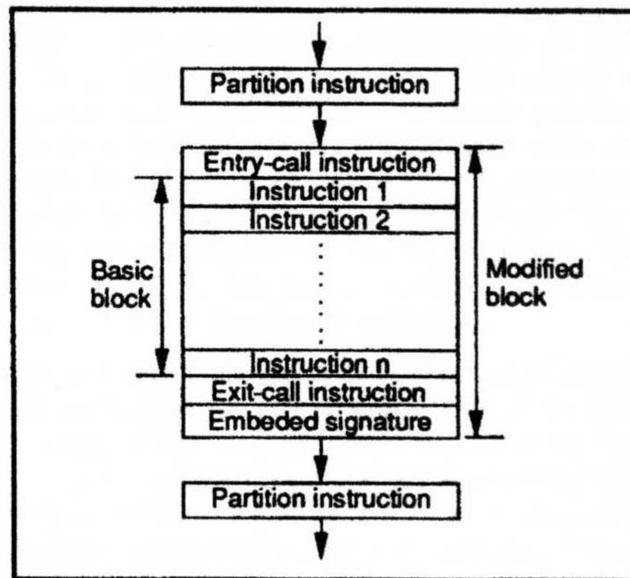


Figura 5: Técnica BSSC [6].

Um cuidado que deve ser tomado na aplicação dessa técnica se refere à alta sobrecarga (*overhead*) de memória devido à grande quantidade de blocos básicos existentes num programa. Uma solução proposta é a utilização dessa técnica em blocos básicos com um tamanho mínimo ou naqueles que serão mais executados.

2.3.3 Técnica BEEC

A técnica BEEC [7] – *Block Entry Exit Checking* – é fortemente baseada na técnica BSSC, diferindo apenas nos seguintes aspectos:

- a verificação do controle de fluxo é duplicada, pois é feita tanto na entrada quanto na saída do bloco básico, justificando, assim, o nome dessa técnica. Na técnica BSSC, o controle de fluxo é verificado apenas na saída do bloco. Caso haja um desvio indesejado no fluxo de execução do programa, o bloco **será executado** antes de essa falha ser detectada;
- o tamanho da assinatura usada é de apenas 1 byte, não de 2 bytes, como o da utilizada em BSSC.

A principal função da técnica BEEC é verificar a correta execução do fluxo da aplicação. Primeiramente, o programa é dividido em blocos básicos conforme os mesmos

critérios apresentados na técnica BSSC; assim, a única entrada legal no bloco básico é através de sua primeira instrução, e a única saída legal é através de sua última instrução. Desse modo, qualquer entrada e qualquer saída de um bloco que não sejam num desses pontos são consideradas ilegais.

Para verificar as corretas entrada e saída do bloco, a estrutura do bloco básico foi modificada, como mostra a Figura 6. O bloco modificado possui quatro partes principais: a primeira é composta pela instrução de chamada de entrada, imediatamente antes do bloco básico; a segunda, pelo bloco básico original; a terceira, também por uma instrução de chamada imediatamente após o bloco básico; finalmente, a quarta, pela assinatura do bloco embarcada no fim do bloco modificado.

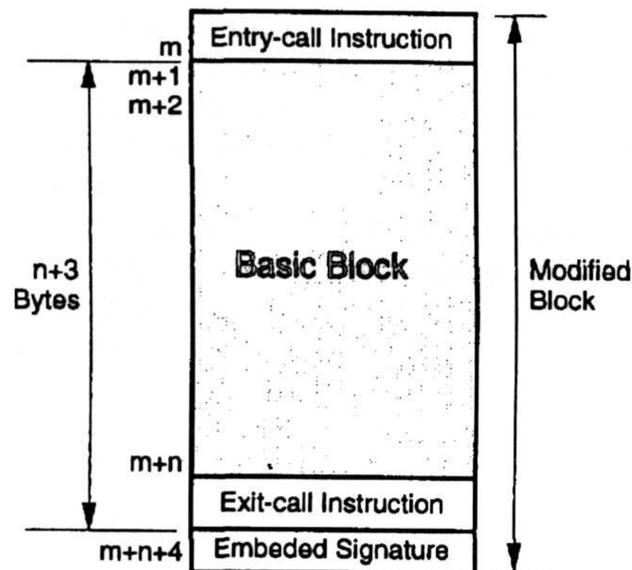


Figura 6: Técnica BEEC [7].

O valor da assinatura do bloco básico corresponde ao tamanho (em bytes) do bloco básico mais o tamanho da instrução de chamada. A rotina de entrada do bloco básico verifica se a execução do bloco anterior foi bem sucedida (entrada e saída válidas). Isso é feito examinando se o valor armazenado em uma variável estática confere com o valor da assinatura única do bloco. Em operação normal, essa assinatura é salva num *buffer* estático pela rotina de saída do bloco anterior.

A técnica BEEC possui três vantagens em relação à técnica BSSC:

- a cobertura de falhas é melhorada devido à introdução da verificação do controle de fluxo na entrada do bloco;
- o *overhead* de memória é reduzido devido ao uso de uma assinatura de um byte de comprimento e não de dois;
- o código do programa é independente da posição na memória, pois as assinaturas embarcadas não dependem dos endereços de memória.

A desvantagem corresponde a que o tempo de execução da aplicação aumenta em 4%; [7] além disso, a técnica só funciona dentro da área de programa, sendo necessário o uso de uma outra técnica auxiliar (ECI, por exemplo).

2.3.4 Técnica CCA

Na técnica CCA [8] – *Control flow Checking using Assertion* –, os intervalos livres de salto de uma linguagem de alto nível (blocos básicos) são identificados, e assinaturas na entrada e na saída desse blocos são adicionadas. O BID – *Block Identifier* – é um identificador único que representa os intervalos livres de salto. O CFID – *Control Flow Identifier* – representa as permissões no controle de fluxo e é de mesmo valor para todos os blocos que compartilham um mesmo predecessor.

A verificação do controle de fluxo é obtida verificando-se os BIDs e os CFIDs. Quando o controle de fluxo entra num bloco básico, o valor do BID é carregado. Na saída desse bloco, o BID é verificado. Se houver uma entrada ilegal pelo meio do bloco (um erro no fluxo normal), o valor atual do BID não irá conferir com o valor do BID na saída do bloco, e um erro será detectado.

O CFID, utilizado para garantir a execução dos blocos na ordem correta, é armazenado numa fila de dois elementos, que é inicializada para conter o valor do CFID do primeiro bloco. Após a entrada no bloco, o CFID do próximo bloco é colocado na fila. Na saída do bloco, é retirado da fila e comparado com seu valor atual para verificar se são iguais; caso isso ocorra, o fluxo está sendo executado corretamente.

As operações de colocação e de retirada da fila devem verificar o estado da fila (cheia, vazia, posição) e, então, realizar a operação desejada. A latência na detecção de falhas é menor à medida que a verificação do *status* da fila é realizada com maior antecedência.

Os procedimentos para atribuir BIDs e CFIDs aos intervalos livres de saltos são ilustrados para uma estrutura *if-then-else* na Figura 7. Na figura, NEXT representa o CFID do bloco seguinte à construção. Na coluna ao lado, é mostrada a verificação dos IDs para o caminho correspondente à instrução THEN.

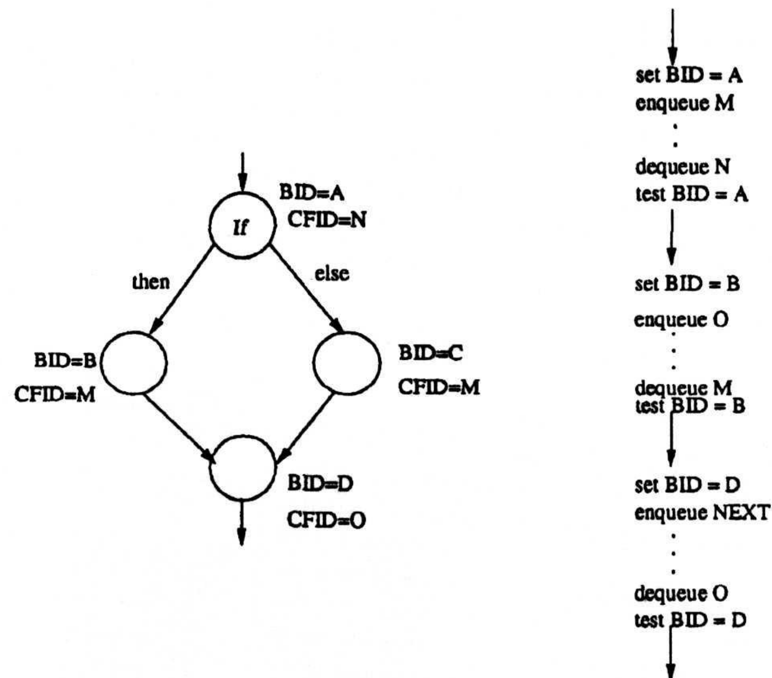


Figura 7: Técnica CCA [8].

2.3.5 Técnica ECCA

A técnica ECCA [9] – *Enhanced Control-flow Checking using Assertion* – é uma versão sofisticada da técnica CCA para aplicações em sistemas de tempo real distribuídos na detecção de falhas no fluxo de execução. O método ECCA divide o programa num conjunto de blocos, que são coleções consecutivas de instruções livres de saltos (BFIs), nas quais existe um único ponto de entrada e de saída. Para máxima cobertura, um bloco pode ser do tamanho de apenas um BFI; entretanto, isso irá causar um *overhead* máximo, pois cada BFI irá conter duas assinaturas.

Um número primo único maior que dois, chamado de BID (*Block IDentifier*), é atribuído a cada bloco. Após essa etapa, duas linhas de código são inseridas em cada bloco. A primeira linha é uma simples atribuição executada na entrada do bloco da seguinte forma:

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

onde *id* é uma variável inteira global, atualizada durante o tempo de execução na entrada e na saída de cada bloco. *BID* é o *Block IDentifier*, um número primo maior que dois gerado em tempo de compilação e único para cada bloco. Pode-se observar que $\overline{id \bmod BID} = 0$ ou $id \bmod 2 = 0$ irá produzir um erro de divisão por zero.

A segunda linha, inserida no fim do bloco, é uma atribuição na seguinte forma:

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

onde NEXT é um número inteiro que contém o produtório dos BIDs de todos os blocos, cujo acesso é permitido a partir do bloco atual. Observa-se que $\overline{\overline{(id - BID)}}$ pode ser 0 ou 1.

A primeira atribuição é chamada de SET, e a segunda, de TEST, pois a primeira prepara a variável *id* na entrada do bloco, e a segunda verifica se o fluxo segue a ordem correta.

A Figura 8 apresenta um exemplo de aplicação da técnica ECCA numa estrutura *if-then-else*. Pode-se observar que o valor de NEXT do bloco 2 corresponde a 77, que é o produto dos BIDs do bloco três (BID=7) e do bloco quatro (BID=11). Isso se deve ao fato de o bloco três possuir uma estrutura do tipo *while* e, portanto, de existir a possibilidade de o bloco três ser ou não ser executado dependendo do resultado obtido no comando *while*. No bloco três, o valor de NEXT é também 77, porque o bloco três pode retornar para si mesmo ou para o bloco quatro.

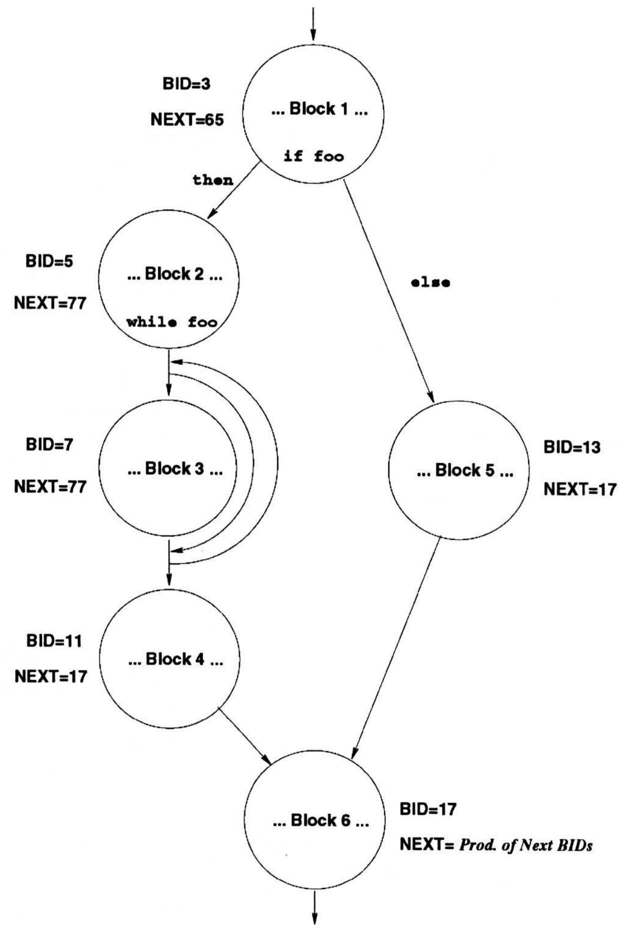


Figura 8: Técnica ECCA [9].

Para se melhorar o desempenho da técnica ECCA, pode-se juntar blocos que não sejam pontos críticos de verificação. Um exemplo é unir os blocos dois, três e quatro num único bloco (caso a verificação nesses blocos possa ser dispensada), economizando, desse modo, tempo de execução. Uma técnica eficiente é determinar quais trechos do código da aplicação são mais executados e, nessas partes, inserir os códigos de verificação. A razão é estatisticamente demonstrada, pois áreas de código que são mais executadas possuem maior probabilidade de receber interferências, necessitando de maior proteção.

2.3.6 Técnica CCFC

A técnica CCFC [10] – *Concurrent Control Flow Checking* – primeiramente divide o programa em blocos básicos e cria uma base de dados contendo os possíveis caminhos que podem ser seguidos durante a fase de execução do programa. Na segunda etapa, são inseridos

trechos de código para realizar a verificação do fluxo de execução. No momento em que o programa é rodado, os caminhos que estão sendo executados são comparados com aqueles descritos anteriormente na base de dados, e qualquer discrepância nas informações indica uma falha no fluxo de execução. Essa abordagem é genérica e pode detectar todos os saltos ilegais no programa como também qualquer salto ilegal dentro do bloco ou após a saída deste.

O bloco é definido como o máximo conjunto de instruções ordenadas que podem ser executadas seqüencialmente sem instruções de salto condicionais. A instrução final do bloco é uma instrução de salto condicional ou uma instrução que recebe o fluxo de controle de dois ou mais pontos do programa.

Um programa pode ser representado por um grafo G{V,B}, chamado de *grafo do programa*, onde $V=\{v_1, v_2, v_3, \ldots ,v_n\}$ é um conjunto de vértices, no qual cada vértice representa um bloco básico, e $B=\{b_{ij} \mid b_{ij}$ é um salto direto e legal de v_i para $v_j\}$. O vértice v_j está no conjunto de sucessores de v_i , denominados de suc(v_i), se e somente se $b_{ij} \in B$. Se $b_{ij} \notin B$, o salto é considerado ilegal, ou seja, houve um desvio não previsto durante a execução do programa.

Um intervalo livre de saltos $I_k=\{V_k, B_k\}$, k=1, 2, ... , n de um grafo de programa G={V,B} é um subgrafo máximo formado pelo subconjunto V_k de V e pelo subconjunto B_k de B, que possuem uma única entrada e que não podem possuir laços ou saltos para trás. O vértice de entrada desse subconjunto é chamado de *header* do intervalo livre de saltos, cujo conjunto é representado por H. O conjunto R representa os intervalos livres de salto restantes identificados no programa, mas ainda não incluídos nos conjuntos V_k identificados.

Através da aplicação do algoritmo de divisão do programa em blocos básicos durante a primeira etapa, obtém-se um grafo da aplicação, como exemplificado na Figura 9:

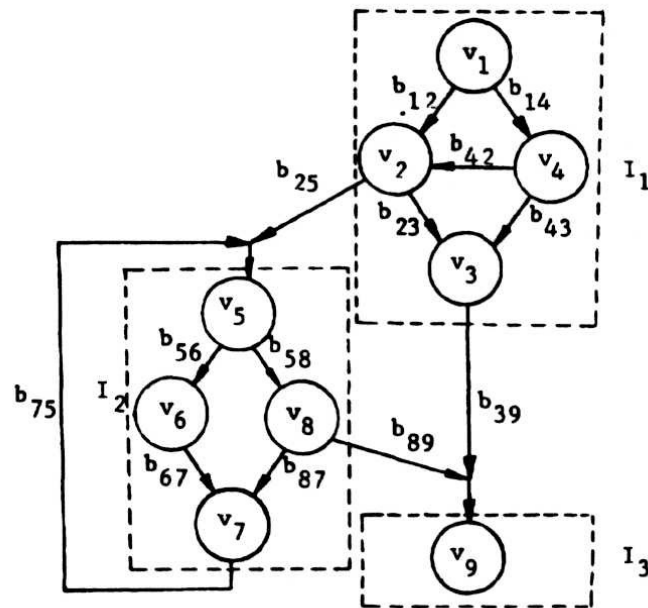


Figura 9: Aplicação do algoritmo de identificação de intervalos livres de saltos [10].

Pode-se observar que, no programa exemplo, foram identificados três intervalos livres de salto – I_1, I_2 e I_3 –, identificados pelas linhas tracejadas. Os saltos b_{25}, b_{75}, b_{39}, e b_{89} são considerados como transferência de controle entre os intervalos livres de salto e não são incluídos nesses intervalos. O conjunto H (*headers*) é definido como H={v_1,v_2,v_9} e é o conjunto dos intervalos livres de salto.

A verificação do controle de fluxo concorrente pode ser conseguida mediante o monitoramento da seqüência de instruções executadas. Nessa abordagem, o fluxo de execução é verificado a partir da base de dados dos blocos livres de salto gerada em tempo de compilação; entretanto, esse método causa uma sobrecarga que não é tolerável em sistemas de tempo real.

A técnica de CCFC está baseada em duas etapas. A primeira fase é a construção de uma base de dados, como mostrada na Figura 9, que permita a verificação dinâmica do fluxo de execução do programa em tempo de execução. A base de dados contém todas as informações dos caminhos a serem percorridos.

A base de dados consiste dos registros dos intervalos livres de salto (CIID), dos caminhos previstos, da identificação dos caminhos possíveis e do próximo intervalo livre de saltos (NIID).

Cada intervalo livre de salto é rotulado com um número distinto (ID), e cada vértice que compõe um intervalo livre de salto é identificado com um número primo distinto. As identificações dos caminhos são obtidas multiplicando-se as identificações dos vértices daquele caminho individualmente. Caso os números gerados sejam muito grandes devido às multiplicações, é possível reduzir sua magnitude, aplicando-se as propriedades dos logaritmos na etapa de geração das identificações dos caminhos.

A segunda fase consiste na inserção de código em trechos do programa em consideração, para realizar a verificação cruzada com a base de dados gerada, a fim de se detectar qualquer desvio não previsto (autorizado) no fluxo de execução. Qualquer discrepância irá indicar um erro no fluxo de execução.

Para diminuir a sobrecarga de trabalho, as verificações na base de dados são inseridas entre os intervalos livres de saltos. As vantagens de usar os intervalos livres de salto na construção da base de dados é que a estrutura gerada é fixa e independe do número de vezes que os intervalos livres de saltos são executados; portanto, além de o monitoramento do fluxo de controle ser muito mais simples, os caminhos totais a serem armazenados se tornam bem menores.

2.3.7 Técnica CFCSS

A técnica de CFCSS [11] – *Control Flow Checking by Software Signatures* – baseia-se na divisão do programa em blocos básicos e na inserção de assinaturas digitais no início de cada bloco em tempo de compilação. Um bloco básico é uma região contínua do programa que não possui instruções de desvios, como mostra a Figura 10.

As assinaturas geradas em tempo de compilação são verificadas em tempo de execução mediante operações lógicas XOR. Cada bloco básico possui uma assinatura

individual e única em todo o programa. Durante o tempo de compilação, são geradas também as diferenças de assinaturas entre os blocos básicos seguindo o fluxo normal de execução previamente projetado. Essas diferenças são armazenadas em cada bloco básico.

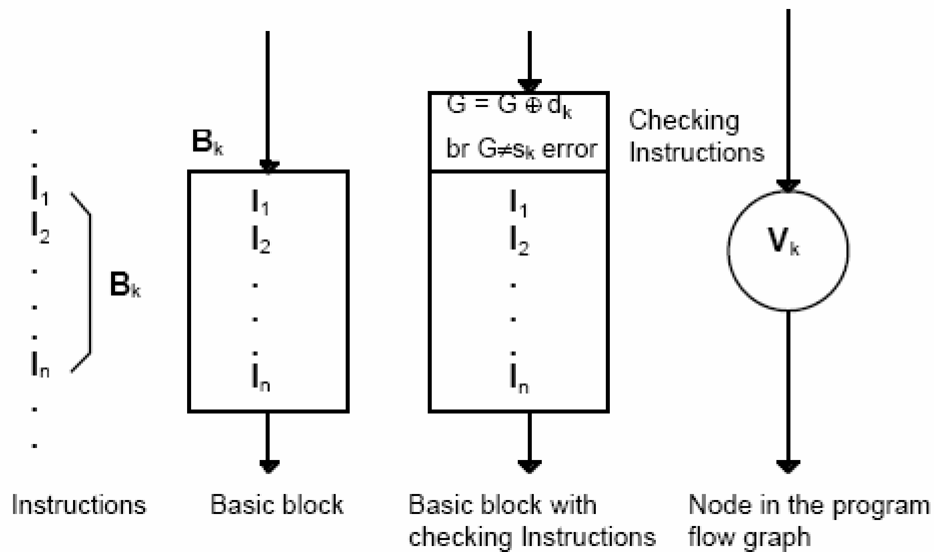


Figura 10: Geração de um bloco básico [11].

Durante o tempo de execução, como mostra a Figura 11, um dos registradores internos do processador é dedicado a receber a assinatura gerada no início do primeiro bloco e, após a saída deste, atualiza seu valor durante a passagem pelos blocos básicos seguintes. Essa atualização é feita mediante uma operação XOR entre o valor atual do registrador e a diferença de assinatura do bloco de entrada gerada em tempo de compilação. Caso a operação XOR dê como resultado o valor da assinatura do bloco de entrada atual, no qual o fluxo de execução se encontra, o programa está no seu fluxo normal de execução. Caso contrário, houve uma falha, e o fluxo de execução veio de um bloco não previsto (houve um salto incorreto); desse modo, um aviso de alerta é gerado (interrupção, *reset* ou mensagem).

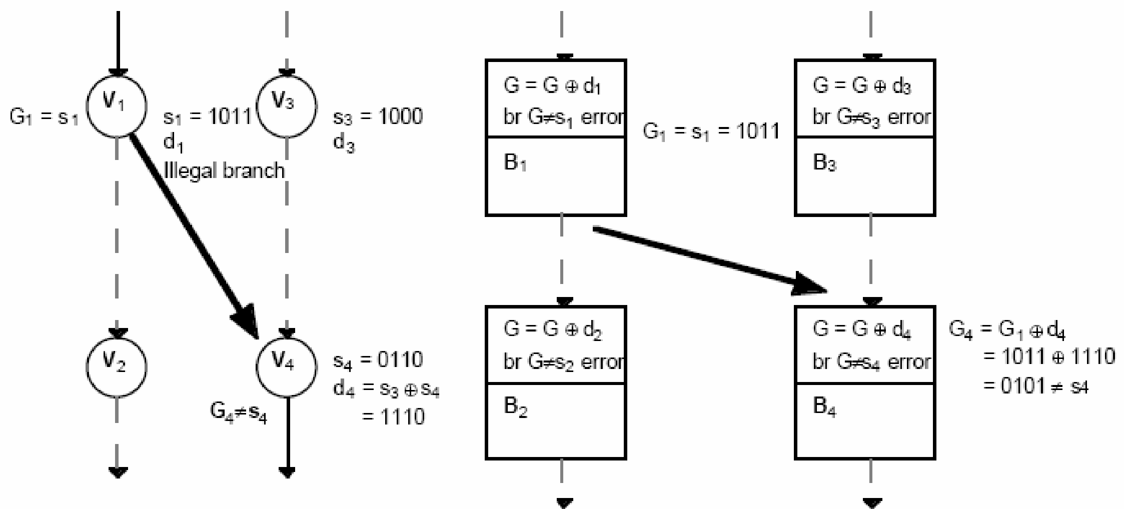


Figura 11: Detecção de um salto ilegal [11].

Um caso que deve, também, ser analisado é aquele em que um determinado bloco recebe o fluxo de dois ou de mais caminhos diferentes (nó V_5). Esse tipo de nodo convergente, mostrado na Figura 12, recebe o nome de *branch fan in node*.

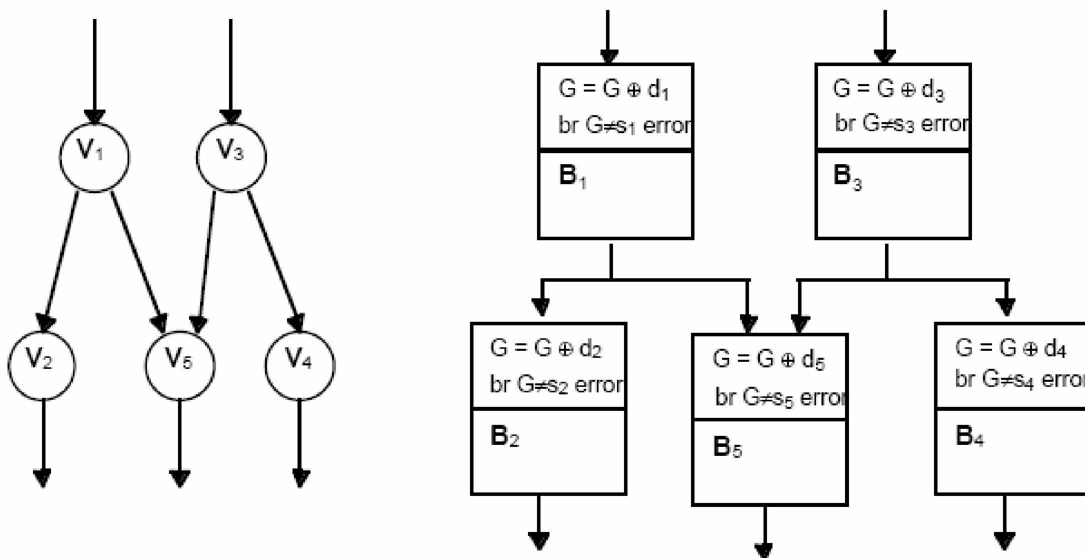


Figura 12: *Branch fan in node* [11].

Esse tipo de nodo recebe um tratamento diferente, pois é preciso que a assinatura recebida permita fluxos provenientes de ambos os nós (no exemplo, V_1 e V_3). Aparentemente, a solução seria colocar a mesma assinatura nos nós V_1 e V_3; entretanto, essa atitude permitiria que saltos dos nós $V_{1\text{-}4}$ e dos nós $V_{3\text{-}2}$ não fossem detectados, o que invalidaria a técnica de detecção. Para solucionar esse problema, é necessária a criação de uma assinatura extra (D),

mostrada na Figura 13, chamada de assinatura de ajuste de tempo de execução (*run time adjusting signature*). Essa assinatura extra visa permitir o ajuste da assinatura local para que ambos os nós sejam considerados fluxos válidos – e não apenas um –, evitando, com isso, uma falsa indicação de erro (denominada de *aliasing*) no fluxo de execução e mantendo a robustez do sistema.

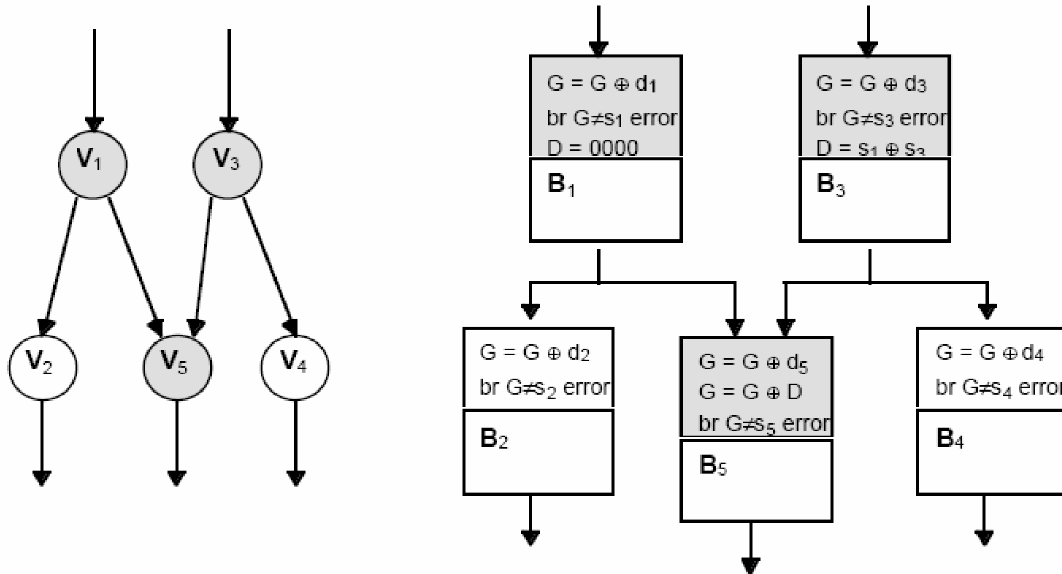


Figura 13: Assinatura de tempo de execução [11].

Pode-se observar que, no exemplo, a assinatura de ajuste (D) é obtida mediante uma operação XOR com as assinaturas dos nós predecessores (V_1 e V_3). Além disso, no nó V_1, D=0, uma vez que a diferença de ajuste das assinaturas em V_5 foi feita utilizando como referência o caminho dos nós $V_{1\text{-}5}$; portanto, não é necessário ajuste por esse caminho. Entretanto, no nó V_3, D=s_1 XOR s_3, já que se deve ajustar a diferença de assinaturas do caminho dos nós $V_{3\text{-}5}$. No nó V_5, após o cálculo normal da assinatura local, deve-se compensar o valor obtido realizando mais uma operação XOR com o valor da assinatura armazenado em D e, então, verificar se as assinaturas do nó V_5 e a gerada pelo fluxo de execução são as mesmas.

2.3.8 Técnica ICFCSS

Uma variação da técnica CFCSS é a técnica ICFCSS [12] – *Improved Control Flow Checking by Software Signature*. Essa técnica modifica o bloco básico para evitar o problema de *aliasing* causado pelos nós com múltiplos *fan-in*. A técnica modifica o parâmetro D distribuindo-o dentro do bloco básico antes de cada instrução de salto, como mostra a Figura 14-b.

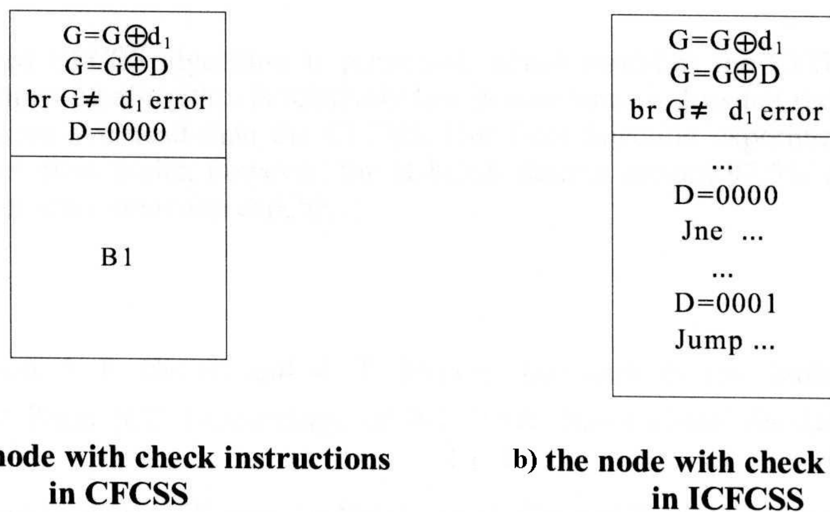


Figura 14: Comparação entre blocos básicos [12].

Essa mudança no modo de tratar os nós com múltiplos *fan-in*, inserindo instruções de ajuste, faz com que as taxas de detecção da técnica CFCSS subam de 92% para 97,5%, usando a técnica ICFCSS proposta.

2.3.9 Técnica YACCA

A técnica YACCA [13] – *Yet Another Control-Flow Checking using Assertions* – verifica o controle de fluxo usando variáveis do tipo inteira globais e dedicadas chamadas de *code*, que contêm uma assinatura associada em tempo de execução ao nodo atual no grafo do fluxo do programa. De modo similar a outras técnicas de controle de fluxo, YACCA divide o

programa em blocos básicos, que são definidos como conjuntos de instruções executadas seqüencialmente, onde o fluxo de execução entra na primeira e sai na última instrução.

Um programa *P* pode ser representado por um grafo composto pelo conjunto de nodos *V* e por um conjunto de bordas *E*, onde P={V, E}, $V=\{v_1, v_2, v_3, ... ,v_n\}$ e $E=\{e_1, e_2, e_3, ... ,e_m\}$. Cada nodo v_i representa um bloco básico, e cada borda e_i, um salto $br_{i,j}$ de v_i, v_j. As bordas $br_{i,j}$ não são necessariamente instruções de salto explícito, mas podem ser também chamadas de sub-rotinas e de instruções de retorno.

Considerando o grafo do programa P={V, E}, é possível, para cada nó v_i, definir $suc(v_i)$ como o conjunto de nodos sucessores de v_i e $pred(v_i)$ como o conjunto de nodos predecessores de v_i. Um nodo v_j pertence ao $suc(v_i)$ se e somente se $br_{i,j}$ está contido em E. Do mesmo modo, v_j pertence ao $pred(v_i)$ se e somente se $br_{j,i}$ está contido em E.

Considerando a representação do programa pelo seu grafo P={V,E}, então, durante a execução de P, $br_{i,j}$ é ilegal se $br_{i,j}$ não está incluído em E. Este salto ilegal indica um erro no fluxo de controle que pode ser causado por uma falha transiente ou permanente.

A Figura 15 apresenta um trecho de código que exemplifica a maneira de se criar o grafo de um programa.

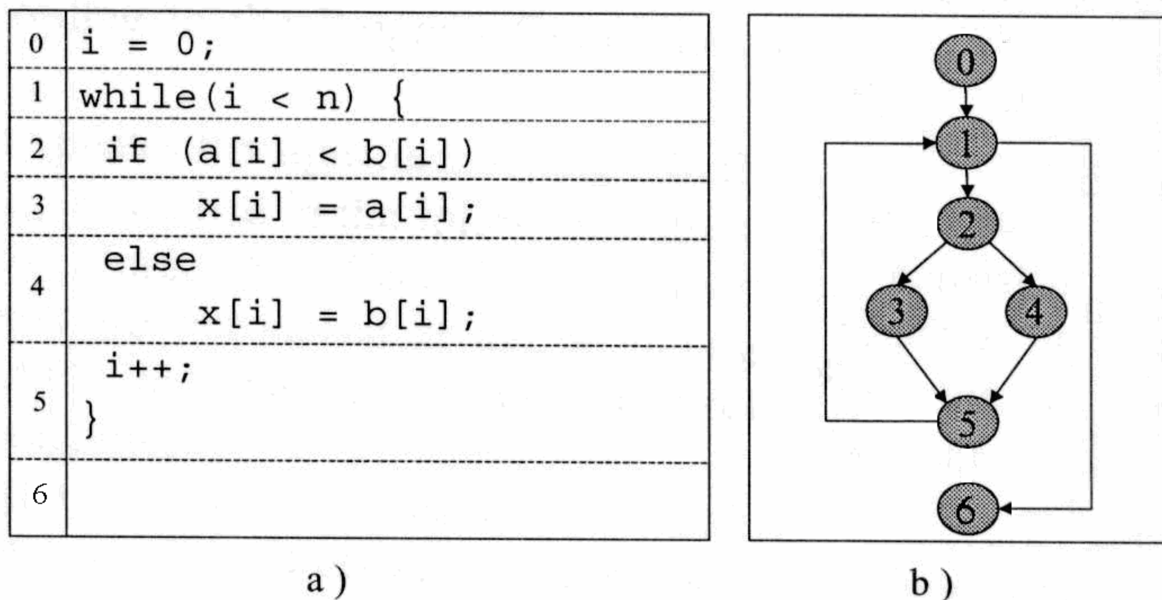


Figura 15: a) Código fonte e b) Grafo do programa[2] [13].

[2] O grafo original publicado na referência está errado, sendo apresentado aqui a versão corrigida.

Cada bloco básico é identificado por uma assinatura única definida em tempo de compilação, e são introduzidas as seguintes instruções em cada bloco básico v_i:

- uma instrução de teste controla a assinatura do bloco básico anterior e verifica se o acesso é permitido de acordo com o grafo do programa, ou seja, se $v_j \in pred(v_i)$;
- uma instrução de atribuição atualiza a assinatura, fazendo seu valor corresponder ao valor de B_i.

As novidades apresentadas pela técnica YACCA estão na definição das instruções de teste e de atribuição e em sua colocação no início e no fim de cada bloco básico, permitindo a cobertura das falhas de controle de fluxo, incluindo aquelas que não passam pelas fronteiras dos blocos básicos.

Além disso, a técnica incorpora uma regra adicional que detecta possíveis falhas que afetem operandos de decisão dentro de saltos condicionais. A regra consiste na inserção de uma linha adicional de teste repetida no início de cada bloco básico para ambas as cláusulas (verdadeira e falsa, caso existam). Se as duas linhas do teste (a original e a introduzida) produzirem resultados diferentes, um erro foi detectado.

Quando o fluxo do programa entra num bloco básico, v_i vindo de um bloco básico v_j, uma instrução verifica se a transição foi legal verificando se o salto $br_{j,i}$ pertence ao conjunto E de fronteiras do grafo do programa. Dado um nó v_i, é possível encontrar um conjunto $E_i \subseteq E$, composto pela lista de saltos entrando no nodo v_i. Uma constante chamada $PREVIOUS_i$ é calculada a partir do produto das assinaturas correspondentes à lista de nodos entrando no nó v_i de acordo com a seguinte regra:

$$PREVIOUS_i = \prod B_j, \forall\, v_j \, e \, br_{j,i} \in E_i$$

A fim de verificar se o salto é legal, a instrução de teste verifica se $PREVIOUS_i$ é um múltiplo da assinatura atual examinando o resto da divisão usando a seguinte expressão:

if (PREVIOUS$_i$ % code) error();

Considerando o ponto de vista da implementação, essa instrução pode ser substituída pelo seguinte código, que elimina a operação de divisão, considerada crítica nessa situação devido à grande probabilidade de ocorrerem falhas nesse processo:

if ((code != B_a) && (code != B_b) && (...) && (code != B_n)) error();
com o conjunto $E_i = \{br_{a,i}, br_{b,i}, ... ,br_{n,i}\}$.

Com a finalidade de se evitar a inserção de novas instruções de salto no código original, as quais podem ser também afetadas pela introdução de falhas, uma outra otimização também pode ser introduzida. Uma variável global chamada *ERR_CODE* é definida e inicializada com 0. Durante a execução da instrução de teste em cada bloco básico, essa variável é atualizada com a seguinte equação booleana:

ERR_CODE |=*((code != B_a) && (code != B_b) && (...) && (code != B_n));*
com o conjunto $E_i = \{br_{a,i}, br_{b,i}, ... ,br_{n,i}\}$.

A cada bloco básico v_i é atribuído um valor de assinatura correspondente ao valor atual do bloco. Uma solução simples pode ser adotada usando a seguinte instrução:

code=B_i;

O problema dessa atribuição é que as falhas causadas por saltos errados são indetectáveis. Por essa razão, a solução adotada faz a atribuição ao nodo v_i usando uma função que depende da assinatura atual. Para um dado nó v_i, uma fórmula genérica pode ser expressa como

$$code = (code \,\&\, M1) \oplus M2 \qquad (1),$$

onde M1 representa uma constante que depende das assinaturas dos nós que pertencem ao conjunto *pred(v_i)*, enquanto M2 representa uma constante que depende da assinatura do nodo atual e de todas as assinaturas pertencentes ao conjunto *pred(v_i).*

Os valores apropriados de M1 e de M2 precisam ser definidos de modo a obterem a mudança correta no valor da assinatura B_i para o nodo v_i. Como exemplo, dado um nó v_i, se o

conjunto de seus predecessores *pred(v_i)* é composto de um único nó v_j, $M1 = -1$ e $M2 = B_j \oplus B_i$, e, nesse caso, a fórmula genérica (1) fica

$$code = code \oplus (B_j \oplus B_i)$$

Supondo ausência de erros, antes da execução desta instrução, *code*=B_j, e, após a execução, o novo valor é *code*=B_i. Todas essas operações são realizadas em tempo de compilação pois todos os operandos são valores constantes.

Como segundo exemplo, considerando o conjunto de predecessores *pred(v_i)* composto por dois nodos v_j e v_k, a fórmula genérica (1) fica

$$code = (code \,\&\, (B_j \overline{\oplus} B_k)) \oplus (B_j \,\&\, (B_j \overline{\oplus} B_k) \oplus B_i).$$

Se não houver erros, antes da execução da instrução, o valor de *code* pode ser tanto B_i como B_j, mas em qualquer caso, após a execução, o valor irá corresponder a B_i. Para assegurar a ausência de *aliasing*, M1 nunca deve assumir o valor zero.

A principal diferença entre a técnica YACCA e a técnica CFCSS é que, na técnica YACCA, a assinatura é calculada sem nenhum ajuste dependente dos possíveis *fanouts* múltiplos.

2.4 EVOLUÇÃO DA TECNOLOGIA REPROGRAMÁVEL

Com o advento da tecnologia planar sobre o silício desenvolvida por Jack Kilby, tornou-se possível a construção de circuitos integrados numa única pastilha. O setor da eletrônica digital, especificamente a informática, beneficiou-se grandemente com o surgimento da possibilidade de se colocarem vários transistores em um único substrato de silício. Com a necessidade constante do aumento de desempenho dos computadores da época — cujo uso era permitido somente a grandes empresas e corporações, já que eram construídos a partir de transistores, relés e válvulas —, da diminuição de suas dimensões, da redução do

seu consumo de energia elétrica e do seu custo de fabricação, o surgimento dos circuitos integrados preencheu essas necessidades superando até as melhores estimativas.

A primeira grande mudança de paradigma nessa época foi o surgimento das primeiras famílias lógicas construídas a partir de diodos, resistores e transistores. Podem-se citar duas grandes famílias que se destacaram nessa época: a família TTL (*Transistor Transistor Logic*), construída a partir de transistores de junção bipolar em lâminas de silício (*waffers*) com orientação cristalina 1-1-1, e a família CMOS (*Complementary Metal Oxide Silicon*), construída a partir de transistores de efeito de campo tipo MOS em lâminas de silício com orientação cristalina 1-0-0 e mais eficiente, em termos de consumo de energia, do que a família TTL.

Durante essa época, construíam-se circuitos integrados que realizavam funções lógicas fundamentais como NAND, NOR, AND, OR, NOT e XOR com a quantidade de entradas variando de duas a até oito. Além dessas portas lógicas fundamentais, começaram a surgir rapidamente outros circuitos básicos, como *flip-flops*, registradores, *buffers* e também unidades lógicas aritméticas (ULA). O objetivo era sempre a construção de circuitos digitais cada vez menores, mais eficientes e mais rápidos.

Com o desenvolvimento da física de semicondutores e da tecnologia microeletrônica e de suas técnicas de produção, tornou-se possível o aumento da miniaturização dos circuitos, o que permitiu integrar uma quantidade cada vez maior de transistores em uma única pastilha. Isso possibilitou o surgimento do microprocessador, que antes era construído com circuitos integrados discretos.

Por sua vez, com o surgimento do microprocessador, foi possível a construção do microcomputador, e, como conseqüência, houve uma popularização do uso dessa tecnologia pelo público em geral, devido, principalmente, à redução de custos e ao aumento da capacidade de integração.

Visando à redução de custos e ao aumento da versatilidade dos circuitos digitais produzidos, a indústria começou a introduzir no mercado os primeiros dispositivos lógicos programáveis. Esses circuitos objetivavam permitir ao projetista criar funções lógicas personalizadas e também substituir vários componentes utilizados na “lógica de cola”, isto é,

nos circuitos combinacionais empregados na etapa de decodificação de endereços para acesso a dispositivos de I/O em sistemas microprocessados. As primeiras versões desses componentes eram as PAL (*Programmable Array Logic*), as PLA (*Programmable Logic Array*) e as GAL (*Generic Array Logic*). As evoluções desses componentes levaram aos conhecidos CPLD (*Complex Programmable Logic Devices*).

Uma desvantagem das primeiras PAL/PLA era o fato de esses componentes poderem ser programados uma única vez; desse modo, para cada nova função, era necessário programar um novo dispositivo.

2.4.1 FPGA

Com o desenvolvimento da tecnologia e da necessidade de maior integração e interligação entre as funções lógicas, surgiu uma nova classe de componentes, o FPGA (*Field Programmable Gate Array*). De acordo com [2, 14], a arquitetura de um FPGA é semelhante à arquitetura de um MPGA (*Mask Programmable Gate Array*), diferindo apenas no modo de programação. O MPGA utiliza máscaras no processo de fabricação microeletrônico para fazer as interconexões das camadas de metal, enquanto o FPGA é programado eletricamente. O FPGA, como mostra a Figura 16, consiste de uma matriz de blocos lógicos, interligados por chaves de passagem programáveis, que podem ser configuradas para realizar uma determinada função.

A grande vantagem do FPGA sobre outros dispositivos lógicos programáveis (PLD) está na sua capacidade de roteamento. No FPGA, cada entrada e cada saída de um bloco lógico passam por diversos blocos de roteamento (*Switch Box*), enquanto, num PLD, a função lógica é programada mediante a utilização de dois níveis de lógica AND/OR. Num FPGA, a função lógica é criada a partir de múltiplos níveis de portas lógicas com baixo *fan-in*, o que é geralmente mais compacto do que topologias de dois níveis, como no caso do PLD.

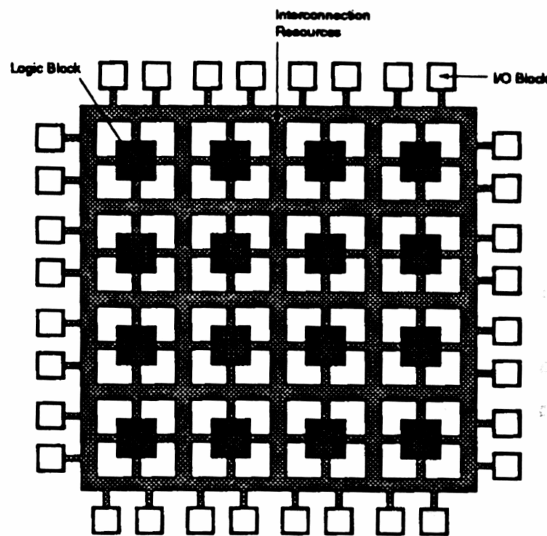


Figura 16: Arquitetura FPGA [2].

2.4.1.1 Granularidade

Um bloco lógico de um FPGA pode ser tão simples quanto um transistor ou tão complexo quanto um microprocessador. O bloco lógico é capaz de realizar várias funções lógicas seqüenciais e combinacionais diferentes. Essa variação de complexidade do bloco lógico é denominada de granularidade. Na literatura, são apresentados três tipos de grãos: o pequeno, o médio e o grande.

O grão pequeno (*fine grain*) são blocos lógicos formados por transistores ou por portas lógicas fundamentais NAND ou NOR com o acréscimo de *flip-flops* e multiplexadores. A Figura 17a/b ilustra esse tipo de bloco lógico programado para realizar a função *f(a,b,c)=a.b+#c*.

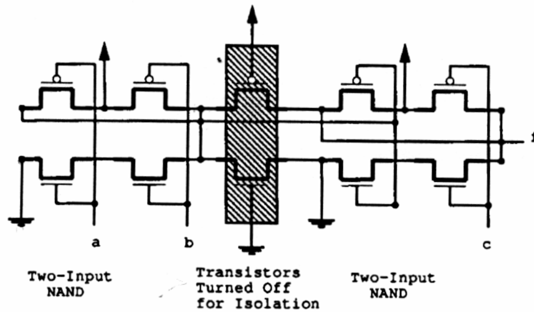


Figura 17-a: FPGA Crosspoint [2].

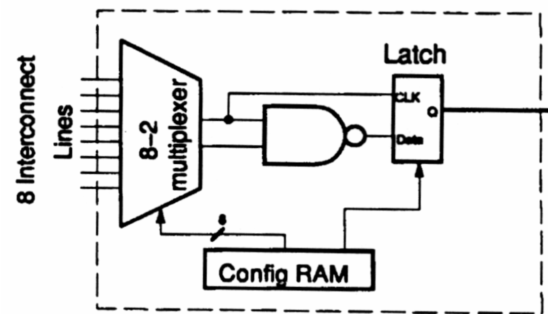


Figura 17-b: FPGA Plessey [2].

A principal desvantagem dos FPGAs com blocos de grão pequeno é que eles necessitam de um número relativamente grande de conexões, implicando um custo elevado de recursos de roteamento no FPGA. Esse custo se manifesta no aumento do *delay* dos sinais elétricos entre os módulos e na área ocupada pelo circuito. Como resultado, FPGAs que empregam blocos de grão pequeno são, em geral, lentos e atingem baixas densidades.

Para se contornar o problema do custo de roteamento e aumentar a capacidade de integração, foram desenvolvidos os FPGAs de grão médio. Um FPGA de grão médio é composto por multiplexadores, *flip-flops* e LUTs (*Look-up Table*). Os blocos lógicos baseados em multiplexador possuem a vantagem de fornecer um alto grau de funcionalidade para um número relativamente pequeno de transistores.

As LUTs, por sua vez, são formadas por células de SRAM que armazenam a tabela verdade da função a ser implementada. Uma tabela verdade de K-entradas é armazenada numa SRAM de 2^K x 1 bit. As linhas de endereço da SRAM funcionam como entrada da função, e a saída de dados da SRAM fornece o valor de saída da função lógica. A vantagem do uso de LUTs é que elas apresentam alto grau de funcionalidade [35]. Uma LUT de K entradas pode implementar **qualquer** função de K entradas, e existem 2^{2^K} funções possíveis. A principal desvantagem é que as LUTs se tornam inaceitavelmente grandes acima de cinco entradas, considerando que o número de células de memória para uma LUT de K-entradas é de 2^K. Outro aspecto importante é que, embora o número de funções que podem ser implementadas cresça rapidamente, essas funções adicionais possíveis de serem implementadas não são utilizadas com freqüência e são difíceis de serem exploradas pelas ferramentas de síntese lógica; portanto, a LUT se torna subutilizada.

Chegou-se à conclusão, finalmente, de que o número ideal de entradas que uma LUT deve possuir para atender, com máxima eficiência, a todos os quesitos importantes – densidade [36], área ocupada [37, 38, 39], *delay* [41, 42] e área de roteamento [40] – é a LUT de 4 entradas, como se pode observar na análise a seguir.

2.4.1.2 Efeitos da Granularidade do Bloco Lógico na Densidade do FPGA

Nos últimos anos, vários esforços foram feitos no sentido de se determinar a escolha do tamanho do bloco lógico ideal para um FPGA que otimizasse densidade e desempenho. Várias publicações foram feitas nesse sentido, como [14], entre outras. Considerando o efeito da granularidade do bloco lógico na densidade do FPGA, quanto maior a granularidade, menor é a quantidade de blocos lógicos necessários para implementar uma função lógica. Por outro lado, um bloco lógico com granularidade maior irá necessitar de uma quantidade extra de transistores, ocupando, assim, uma área maior. Essa relação de proporção inversa sugere um ponto de granularidade "ótimo" de bloco lógico cuja área ocupada para a implementação do circuito seja minimizada.

Como exemplo do efeito da funcionalidade do bloco lógico na área ocupada, consideremos a Figura 18. Supondo que a função $f = abd + bc\overline{d} + \overline{a}\overline{b}\overline{c}$ seja implementada com blocos lógicos com granularidades diferentes, podemos observar o efeito da quantidade dos blocos lógicos utilizados. A Figura 18 mostra a implementação dessa função em três versões utilizando LUTs de duas *(a)*, de três *(b)* e de quatro *(c)* entradas. Observa-se que a implementação com LUTs de duas entradas necessita de sete blocos, a de três entradas, de três blocos, e a de quatro entradas, de um bloco. Pode-se observar, também, que a área ocupada – quantidade total de bits utilizados nas LUTs – diminui à medida que o tamanho do bloco aumenta. Outra observação importante é que o atraso de propagação do sinal (*delay*) também diminui, pois não apenas a quantidade de níveis lógicos diminui, como também há menos interconexões a serem feitas, portanto menos atrasos em redes RC devido às resistências e capacitâncias parasita.

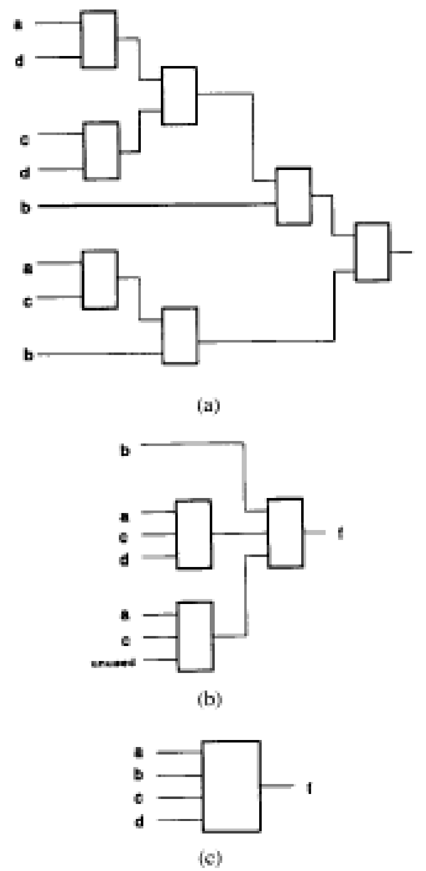


Figura 18: Implementações de $f = abd + bc\overline{d} + \overline{a}\overline{b}\overline{c}$ [2].

A Figura 19 mostra um resultado experimental para a relação entre granularidade e área ocupada para uma implementação de teste.

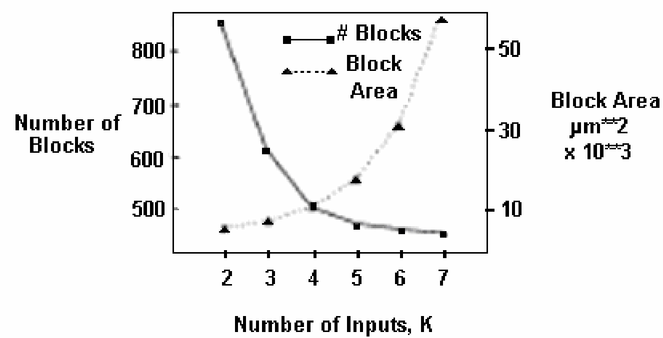


Figura 19: Quantidade e área ocupada de bloco em função do número de suas entradas [14].

Pode-se observar que, para o valor de *K* igual a quatro entradas, obtém-se o melhor aproveitamento do bloco em termos de área ocupada e quantidade utilizada.

A área de lógica ativa utilizada no bloco lógico é apenas uma parte da área total utilizada, que é composta pela área de lógica ativa mais a área de roteamento. A área dedicada ao roteamento dos blocos lógicos é geralmente maior e ocupa entre 70% a 90% da área total utilizada. A Figura 20 apresenta a área de roteamento utilizada e o número de blocos lógicos utilizados *versus K*.

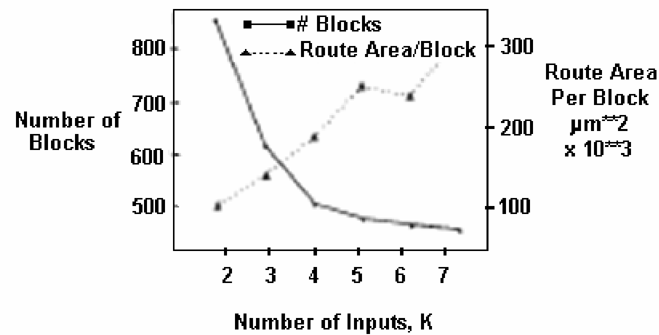


Figura 20: Quantidade de blocos e área de roteamento ocupada em função de *K* [14].

A área total de *chip* necessária para um FPGA é a soma da área de bloco lógico e da área de roteamento. O gráfico da área total pode ser observado na Figura 21.

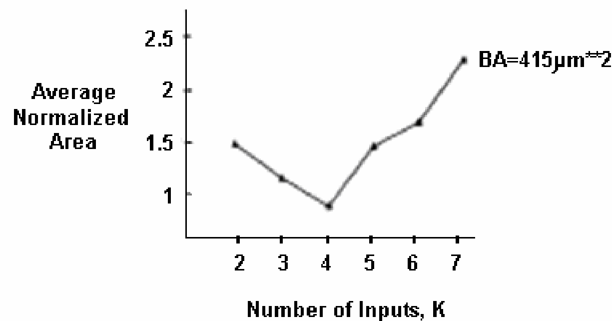


Figura 21: Área total normalizada em função de *K* [2].

Observa-se que a área total é mínima para *K*=4, o que demonstra, claramente, que o melhor rendimento em área se obtém com blocos lógicos de quatro entradas.

2.4.1.3 Efeitos da Granularidade do Bloco Lógico no Desempenho do FPGA

A granularidade do bloco lógico afeta também o desempenho do FPGA, visto que a quantidade de blocos lógicos utilizados depende do tamanho deles, e, quanto maior o número de blocos lógicos usados, maior será o número de níveis lógicos do circuito. Isso implica um

caminho crítico – *critical path* – maior; logo, o atraso de propagação do sinal de entrada será maior, o que diminuirá a freqüência de operação do circuito.

A partir de todas as considerações feitas, é possível estabelecer conclusões importantes:

- FPGAs de grão pequeno possuem a desvantagem de necessitar de grande área de roteamento, o que limita seu uso em aplicações que necessitem de grande quantidade de blocos lógicos;
- FPGAs de grão médio possuem LUT com tamanho otimizado de 4 entradas, mas apresentam os mesmos problemas de limitação de roteamento, devido, principalmente, aos atrasos de propagação em aplicações de médio porte e, especialmente, nas de DSP.

2.5 NOC – NETWORKS ON CHIP

Com o aumento da capacidade de integração e devido às limitações na estrutura interna dos FPGA, principalmente no que diz respeito ao roteamento, uma solução apresentada foi a utilização de técnicas de redes de computadores, já aplicadas e validadas, que utilizam processamento paralelo e distribuído [15, 16, 17].

A idéia consiste na reprodução das topologias de redes já consagradas dentro do FPGA, de modo que um grupo de LUTs ou um bloco funcional dedicado possam ser utilizados na criação de *elementos* ou de *unidades de processamento* dedicadas denominadas PE (*Processing Elements*), como mostra a Figura 22. Essas unidades de processamento podem ser desde ULAs programáveis até núcleos de processadores completos (*firmcores*) licenciados, dependendo do espaço disponível e da aplicação alvo do FPGA.

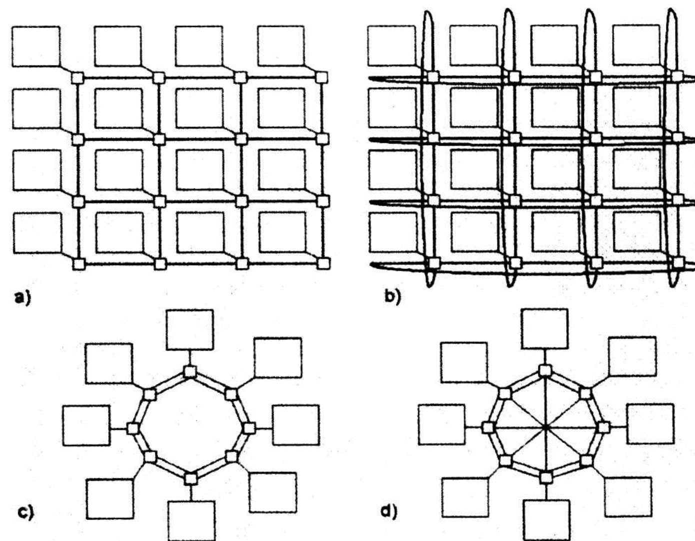


Figura 22: Topologias de *Networks on Chip*: Mesh (a), Torus (b), Ring (c), Octagonal (d) [17].

Uma das contribuições das NoCs foi a possibilidade de integração dos conceitos desenvolvidos em computação paralela e distribuída em uma única pastilha, o que possibilitou, além disso, a aplicação dos FPGAs em áreas como DSP. Já existem dispositivos no mercado que fazem uso dessa abordagem, como o FPOA (*Field Programmable Object Array*), desenvolvido pela empresa MathStar [18, 19], cujos blocos básicos são unidades MAC (*Multiplicator and Acumulator*) com ULAs de ponto flutuante cuja freqüência de operação garantida é de 1GHz.

Além disso, a pesquisa em NoCs desenvolveu também as arquiteturas de barramento [20], pois, devido à necessidade de grande fluxo de dados entre os nodos da rede *intrachip*, novas arquiteturas de barramentos que atendessem a essa demanda tornaram-se fundamentais para o funcionamento dessas redes. As características necessárias mais importantes nesses barramentos são as altas taxas de transferência de dados, a conectividade entre diversos tipos de nodos e a flexibilidade de uso em diversas topologias de rede.

2.6 SOC – SYSTEM ON CHIP

Com a necessidade cada vez maior de sistemas portáteis e de menor custo e consumo e devido ao aumento de sua complexidade e da capacidade de integração no silício, a tendência natural foi a evolução para sistemas em uma única pastilha (SoCs) [21].

O paradigma do SoC veio ao encontro das necessidades do mercado em relação ao tempo de desenvolvimento de um produto, conceito conhecido como *time-to-market*. Além disso, devido à grande complexidade dos dispositivos projetados, o reaproveitamento de módulos já validados tornou-se fundamental.

Neste cenário, considerando-se também o desenvolvimento dos barramentos internos [22, 23] e das arquiteturas de rede pesquisados em NoC, criou-se um ambiente propício ao fortalecimento do paradigma de SoC [24]. Pode-se traçar um paralelo e observar que o paradigma de SoC está para a metodologia de desenvolvimento do *hardware* assim como o paradigma de orientação a objetos está para o desenvolvimento de software considerando **o ponto de vista da reusabilidade**.

Sob este ponto de vista, a granularidade do FPGA deixa de ser homogênea e passa a ter grãos de diferentes tamanhos. Num FPGA orientado à arquitetura SoC, podemos encontrar, além das LUTs tradicionais, microprocessadores, memórias, multiplicadores, ULAs dedicadas e outras estruturas específicas que visam a facilitar a realização do projeto, diminuir o tempo de construção do sistema e otimizar o roteamento, como mostra a Figura 23. Os fabricantes de FPGA atentos a essas mudanças já incorporam núcleos licenciados de microprocessadores em seus produtos, como no caso da Altera [25], com o processador ARM, e da Xilinx [26], com o processador Power PC.

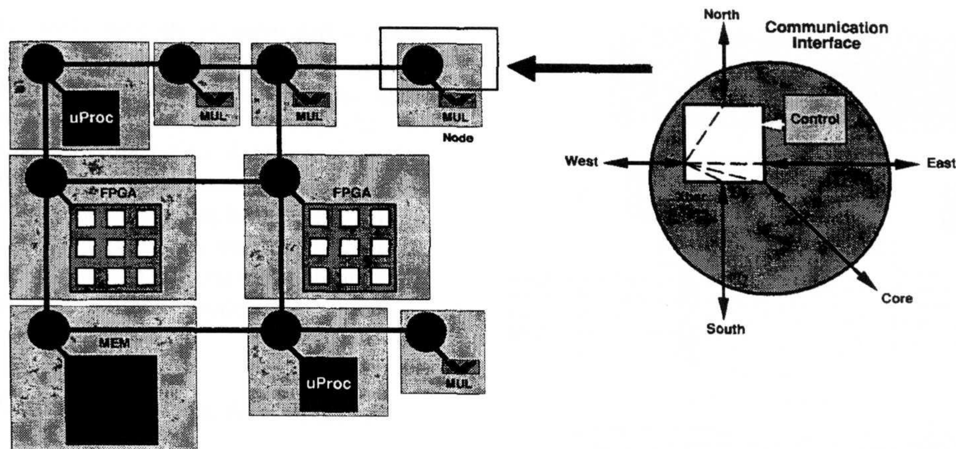


Figura 23: SoC com nodo de comunicação e roteamento adaptativo [24].

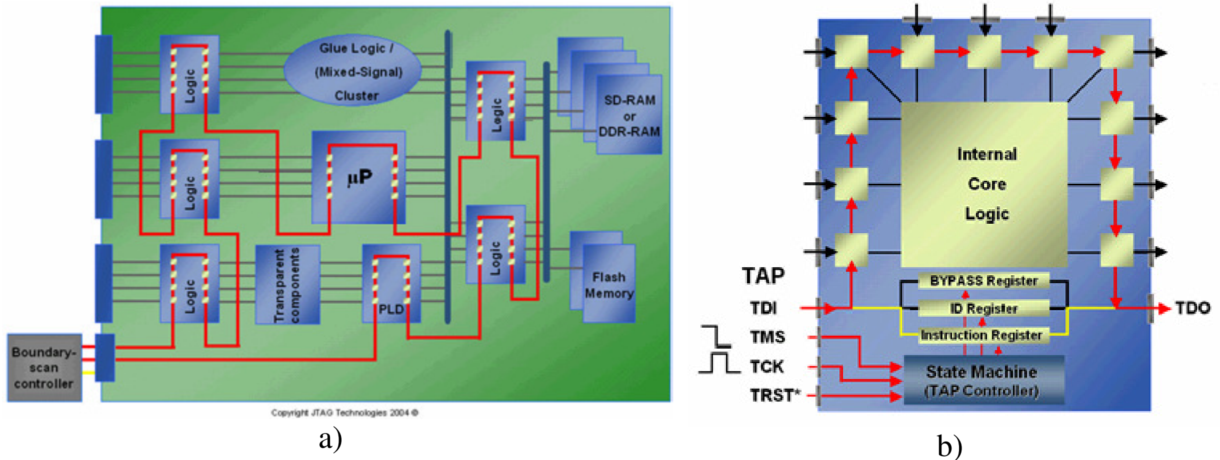
Além disso, com o desenvolvimento das linguagens de descrição de *hardware* (VHDL, Verilog etc.), o projetista pode criar seus próprios núcleos de *hardware* (*softcores)* para aplicações específicas ou licenciá-los em uma biblioteca (*IP cores*).

Com o rápido desenvolvimento dessas tecnologias, houve também uma necessidade de melhorar as metodologias de projeto e o método de teste, pois, com o aumento do grau de integração, a necessidade de garantir a testabilidade dos sistemas se tornou um fator crucial. Caso houvesse uma falha no sistema, como poderia ser possível detectar o local da falha, se não há meios de se injetarem os padrões de teste na entrada do módulo?

A resposta a essa questão foi o desenvolvimento do *Boundary Scan* e do JTAG, que são abordagens técnicas que permitem o acesso a todos os módulos do circuito, possibilitando o seu teste individual. Com o aumento da densidade de integração, a capacidade de se testar isoladamente um determinado componente dentro do sistema se torna um problema complexo, pois fica difícil o acesso aos terminais dos componentes para a realização do teste.

Para solucionar esse problema, foi desenvolvido o padrão JTAG (IEEE 1149.1) [27], que permite o teste dos componentes internos de um SoC. Já o *Boundary Scan Test* [28, 29] serve para testar os componentes externos que se ligam ao SoC de modo a ser possível injetar padrões de teste nos dispositivos em volta do SoC, melhorando a controlabilidade e a observabilidade do sistema. A figura 24 mostra um diagrama do padrão JTAG e do *Boundary*

Scan Test, que resolveram o problema de acesso aos componentes internos do sistema com baixo custo e grande versatilidade.



a)

b)

Figura 24: a) Conexão JTAG e b) *Boundary Scan* [43].

Dessa forma, neste capítulo, fundamentou-se a taxonomia básica de sistemas tolerantes a falhas e as definições de sistema, falha, erro e defeito, bem como sua relação. Além disso, revisaram-se as principais técnicas para detecção de falhas por *software* em sistemas monoprocessados.

Como a plataforma de desenvolvimento deste trabalho é um FPGA, abordaram-se a evolução da tecnologia reprogramável e os principais parâmetros que influenciam no desempenho desses dispositivos, bem como procedeu-se a uma apresentação dos paradigmas SoC e NoC e das técnicas de Boundary Scan e JTAG para permitir a testabilidade do sistema.

3 METODOLOGIA E RESULTADOS

3.1 METODOLOGIA

3.1.1 Plataforma de *Hardware*

Inicialmente, realizou-se um estudo das ferramentas de projeto disponíveis EDK 8.1i e ISE 8.1i, realizando-se vários tutoriais com o objetivo de analisar o fluxo de projeto Xilinx e seu compilador C. Para tanto, utilizou-se como plataforma de *hardware* a placa XUP Virtex II PRO Development System, fabricada pela Digilent Inc, como mostra a Figura 25.

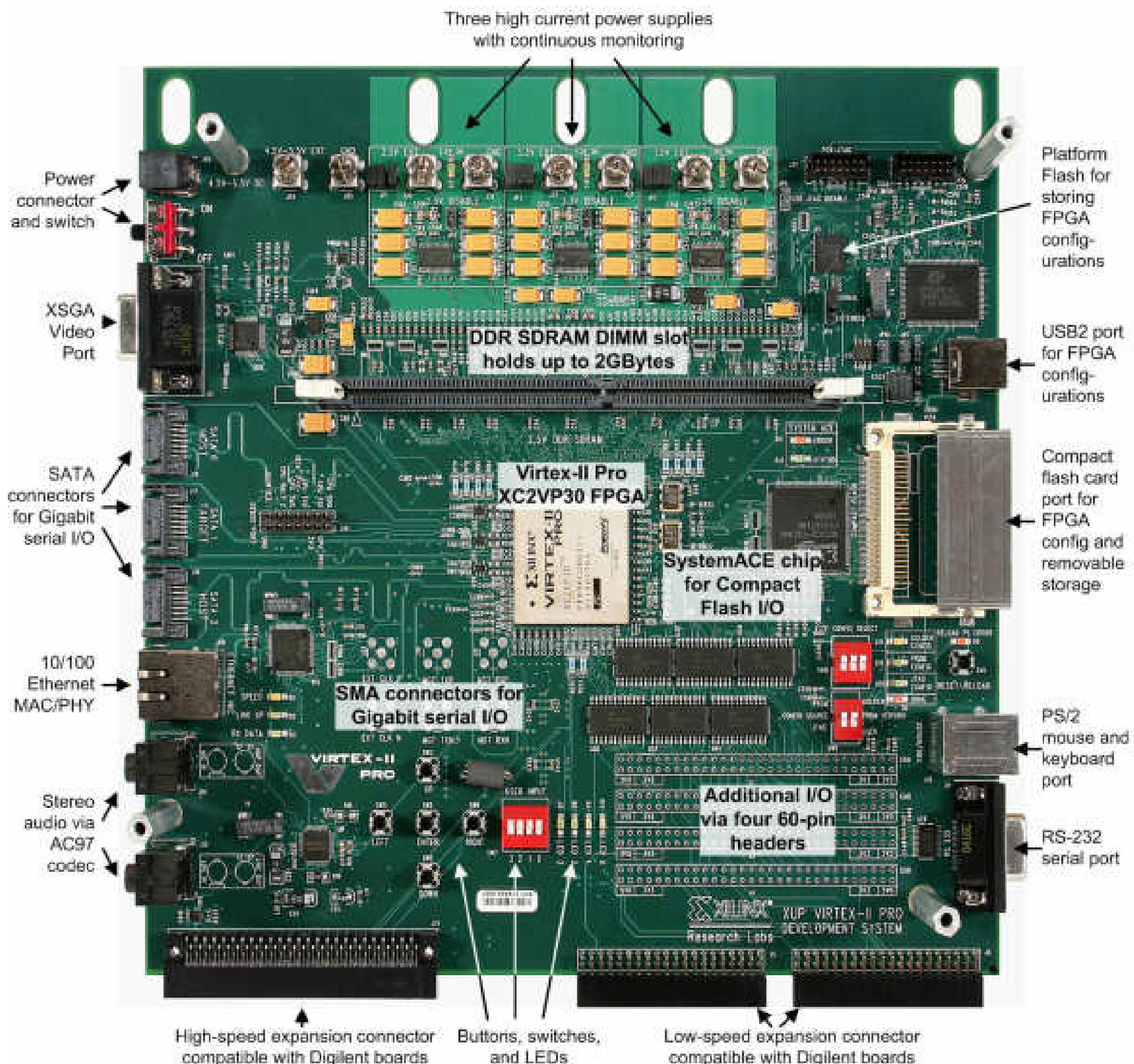


Figura 25: Placa de desenvolvimento XUP Virtex II Pro [30].

Após a conclusão dessa etapa, partiu-se para o desenvolvimento da arquitetura de *hardware,* base para o projeto de pesquisa. A meta inicial era a utilização de dois processadores Power PC 405 (PPC-405), trabalhando-se com o sistema operacional Linux v. 2.6 embarcado.

Testou-se, inicialmente, a metodologia de projeto para embarcar o sistema operacional Linux na placa de desenvolvimento, utilizando apenas um processador PPC-405 e a memória DDR externa de 256 MB. Empregou-se o fluxo de projeto da ferramenta EDK para criar uma plataforma de *hardware* adequada a esse teste. Após a síntese desse *hardware*, compilou-se o *kernel* do Linux com a ferramenta *Busy Box* [44].

Para se compilar o *kernel* do Linux, foi necessário realizar o *download* da distribuição uCLinux, que pode ser encontrada em [45]. Fez-se o *download* dos seguintes arquivos:

- uClinux-dist-20051110.tar.gz;
- linuxppc-2.4-20051021.tgz;
- powerpc-elf-tools-20060320.tgz;
- vendors-20051021.tgz.

Após o *download* dos arquivos, realizaram-se os seguintes procedimentos para gerar a imagem do sistema operacional:

1) descompactar a distribuição uClinux:

   tar zxfv uClinux-dist-20051110.tar.gz

2) remover os diretórios linux-2.* de dentro do diretório uClinux:

   *rm –rf linux-2.**

3) descompactar o *kernel* do Linux dentro do mesmo diretório descrito no procedimento 1:

   tar zxfv linuxppc-2.4-20051021.tgz

4) renomear linuxppc-2.4 para linux-2.4.x :

   mv linuxppc-2.4 linux-2.4.x

5) descompactar o arquivo *vendors* dentro do diretório descrito no procedimento 1:

   tar zxfv vendors-20051021.tgz

6) descompactar o *cross-compiler*:

   tar zxfv powerpc-elf-tools-20060320.tgz

7) exportar a variável de ambiente PATH do diretório onde o *cross-compiler* está instalado:

export PATH=$PATH:/<path_para_o_cross_compiler>/powerpc-elf-tools/bin

8) substituir o arquivo *auto-config.in* em linux-2.4/arch/ppc/platforms/xilinx-auto/auto-config.in pelo mesmo arquivo gerado na ferramenta EDK da Xilinx. Nesse arquivo, estão os endereços do *hardware* criado para uso do sistema operacional.

9) executar o *menu* de opções de módulos a serem integrados no sistema:

make menuconfig

10) após a seleção dos módulos, executar o *link* com o *kernel*:

make dep && make

Após a realização desses procedimentos, copiou-se a imagem do sistema operacional do diretório */images* para o PC onde estava instalada a ferramenta EDK. Em seguida, procedeu-se ao carregamento do *hardware* (*bitstream*) e do *software* (*kernel*) na plataforma, e o resultado foi o correto funcionamento do sistema operacional, incluindo acesso à internet através de endereço IP.

Após essa etapa, iniciou-se o estudo da viabilidade de se utilizarem os dois Power PC rodando paralelamente com o sistema Linux. Verificou-se que era preciso dividir a memória RAM de 256 MB em duas partes de 128MB**,** para que cada sistema operacional rodasse isoladamente. Entretanto, havia à disposição, na placa, somente um *slot* de memória. Procuraram-se, no *site* do fabricante da FPGA, soluções para esse problema, e encontrou-se um gerenciador de memória que permite compartilhar a memória DDR com várias aplicações — o MPMC (*Multi Port Memory Control*) [31].

O MPMC é um núcleo de *hardware* desenvolvido pela Xilinx que permite compartilhar a memória DDR ou DDR2 com vários dispositivos**,** fornecidos na ferramenta de desenvolvimento EDK. Sua utilização viabiliza a criação de sistemas multiprocessados, como mostra a Figura 26. Neste sentido, pode-se ligar dispositivos OPB (*On-Chip Peripheral Bus*) e PLB (*Processor Local Bus*), entre outros, aos barramentos dos processadores Power PC e Microblaze.

Para se utilizar o MPMC, é necessário configurar alguns parâmetros, como mostra a Figura 26. A primeira etapa consiste em definir a freqüência de operação da memória, a

largura do barramento de dados e a quantidade de sinais do tipo *chip select* (CS). Na segunda etapa, configura-se o estilo do banco de memória (SIMM ou DIMM), o tipo, a densidade, o fabricante, o código e a velocidade. Na terceira parte, configuram-se as 8 portas de acordo com o tipo de barramento a que o módulo será conectado (PLB, OPB, ISPLB, DSPLB) e as respectivas faixas de endereços.

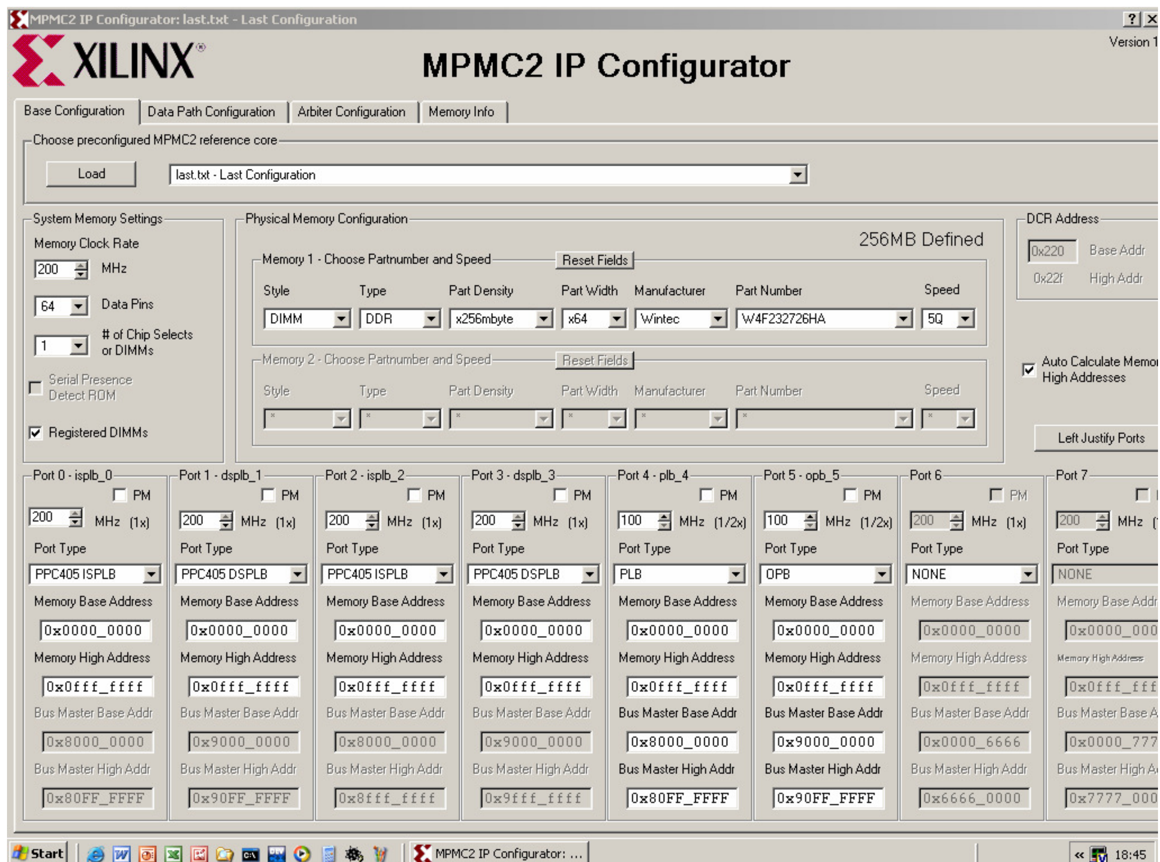


Figura 26: MPMC – Janela de configuração [31].

De posse do núcleo do MPMC, realizou-se um estudo para sua instalação e utilização na ferramenta de desenvolvimento. Em seguida, realizou-se uma pesquisa à procura de uma arquitetura que possibilitasse a conexão de dois processadores Power PC no barramento do MPMC, conforme o modelo proposto pelo fabricante, o qual se adequava às necessidades de *hardware* do projeto, como mostra a Figura 27.

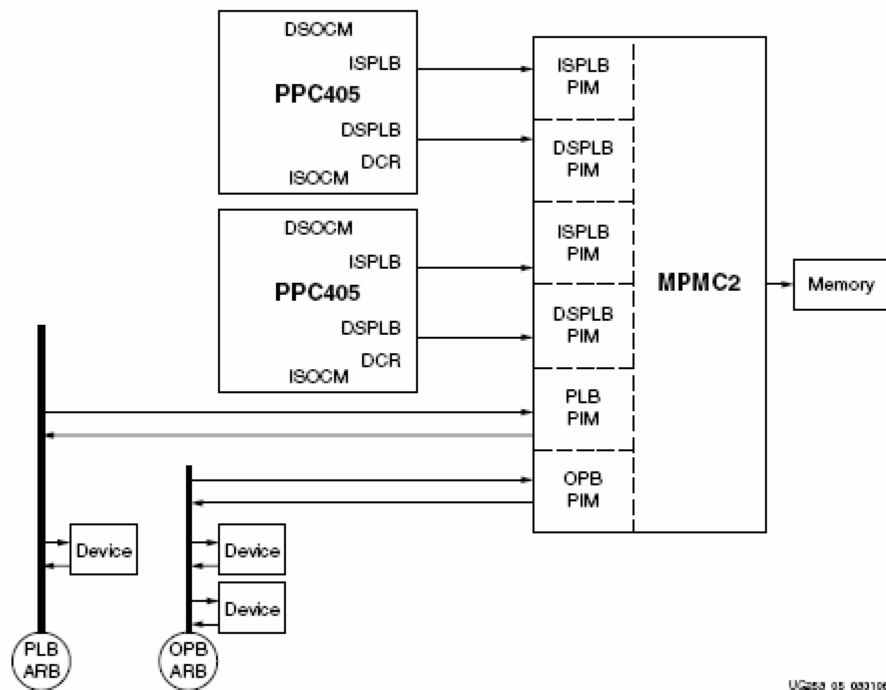


Figura 27: Diagrama simplificado do MPMC2 com dois processadores Power PC [26].

Devido à falta de documentação mais detalhada e de maior experiência prática em como realizar a conexão do núcleo do MPMC na placa de desenvolvimento utilizada (XUPV2P), não se conseguiu, ainda, uma estrutura de *hardware* estável para utilização prática. O problema reside principalmente no ajuste correto do sincronismo entre os sinais de acesso à memória DDR e a geração de sinais de *clocks* adequados com suas fases correspondentes.

Como a utilização do *core* MPMC não foi viável, a solução encontrada foi a criação de uma outra arquitetura que permitisse não só a utilização dos dois processadores embarcados como também a comunicação entre ambos. A solução encontrada foi a utilização das *Block RAM* internas do FPGA de modo que cada processador tivesse sua própria memória local. Isso limita o tamanho do código da aplicação, mas permite a validação do conceito proposto neste trabalho.

Cabe salientar que foi lançado, em setembro de 2007, o MPMC 3, cujo *core* já vem disponível no *software* EDK 9.1i. Entretanto, até o momento, não se obteve a licença para se utilizar essa versão do *software*; assim, ainda não foi possível testar a nova versão desse gerenciador de memória, o que pode constituir uma proposta de trabalho futuro.

Para a comunicação entre os processadores, também foi utilizada uma *Block RAM*, que faz parte da estrutura interna da Virtex II Pro, mas que foi configurada como RAM de dupla porta. Desse modo, cada porta da RAM foi conectada no barramento de cada um dos processadores, permitindo a comunicação entre ambos, conforme mostra a Figura 28.

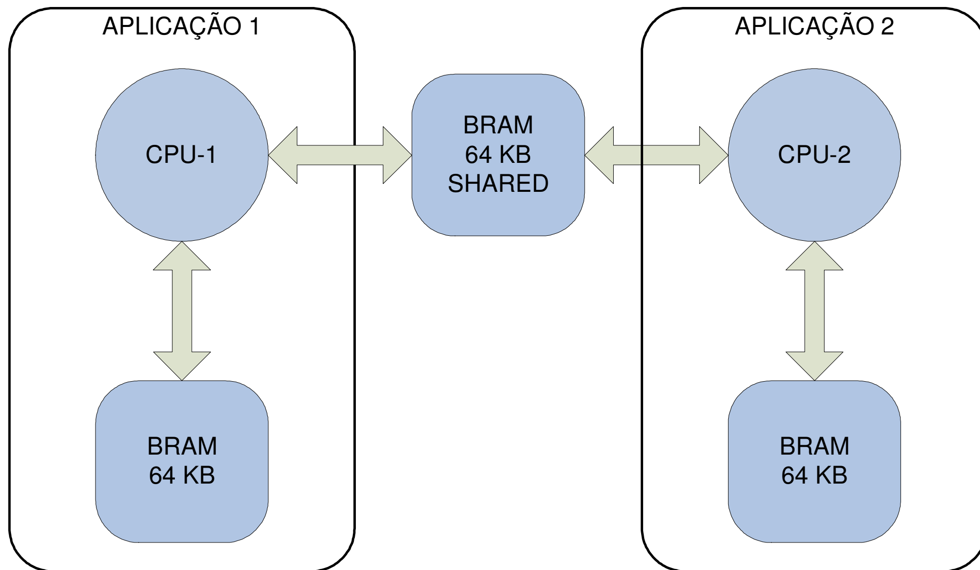


Figura 28: Diagrama simplificado do sistema implementado.

Para coletar os dados da saída de cada processador, utilizou-se a porta serial cujo *hardware* de conversão TTL para RS232 (MAX 3232) acompanha a placa de desenvolvimento, conforme ilustra a Figura 25.

A concepção inicial era utilizar a memória compartilhada para enviar os dados até o processador que possuía conexão com a porta serial e, então, enviar os dados até o PC, conforme ilustra o diagrama da Figura 29 (setas em verde indicam comunicação da CPU-1 com o PC). Desta forma, foi desenvolvido o *software* para a comunicação e adaptaram-se as rotinas de teste para essa topologia de *hardware*. Cabe salientar que, nesta arquitetura de comunicação dos processadores da placa de desenvolvimento XUP Virtex II Pro (Figura 25) com o PC externo, somente um dos dois processadores Power-PC (CPU-1) possuía acesso remoto, via porta serial, com o PC externo. Nesta arquitetura, a CPU-1 operava como *Master*, e a CPU-2, como *Slave*.

Os testes iniciais demonstraram o bom funcionamento da comunicação entre os processadores e o PC e serviram como base para validar o protocolo de comunicação

desenvolvido especialmente para o injetor de falhas, cuja estrutura é abordada na seção referente ao desenvolvimento do *software*.

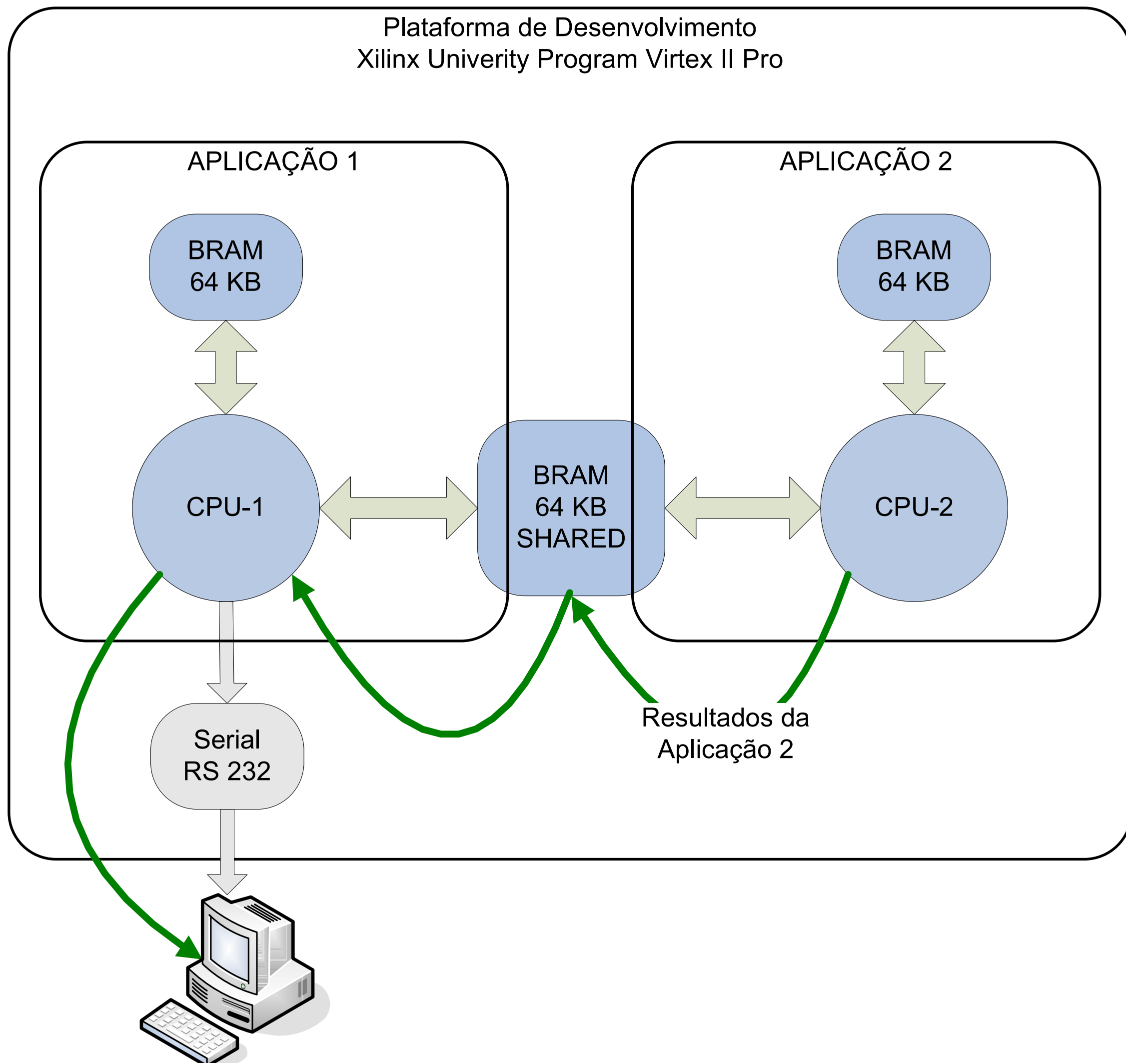


Figura 29: Diagrama simplificado do sistema implementado.

Entretanto, observou-se que, durante a fase de injeção de falhas na CPU-1, o envio de dados da CPU-2 ficava comprometido, uma vez que a falha injetada interferia na rotina de comunicação, que retirava os dados da memória compartilhada da CPU-2 e os enviava para a porta serial da placa. Esse problema, que não havia sido previsto durante a fase de desenvolvimento, foi causado pelo fato de que, ao se injetar a falha na CPU-1 (*Master*), ela fazia com que a aplicação que estava rodando nesta CPU saísse de seu estado normal de funcionamento e, como conseqüência, deixasse de executar a rotina de comunicação como programada. Esse comportamento irregular causava uma falha na comunicação da CPU-2

(*slave)* com o PC, o que impedia o recebimento dos dados, causando, desse modo, erros de leitura.

A solução encontrada foi a criação de um outro canal de comunicação serial dedicado entre a CPU-2 e o computador externo, como mostra a Figura 30. Isso implicou uma reformulação da plataforma de *hardware* e, principalmente, do *software* tanto das aplicações, como do injetor de falhas que rodava dentro do PC.

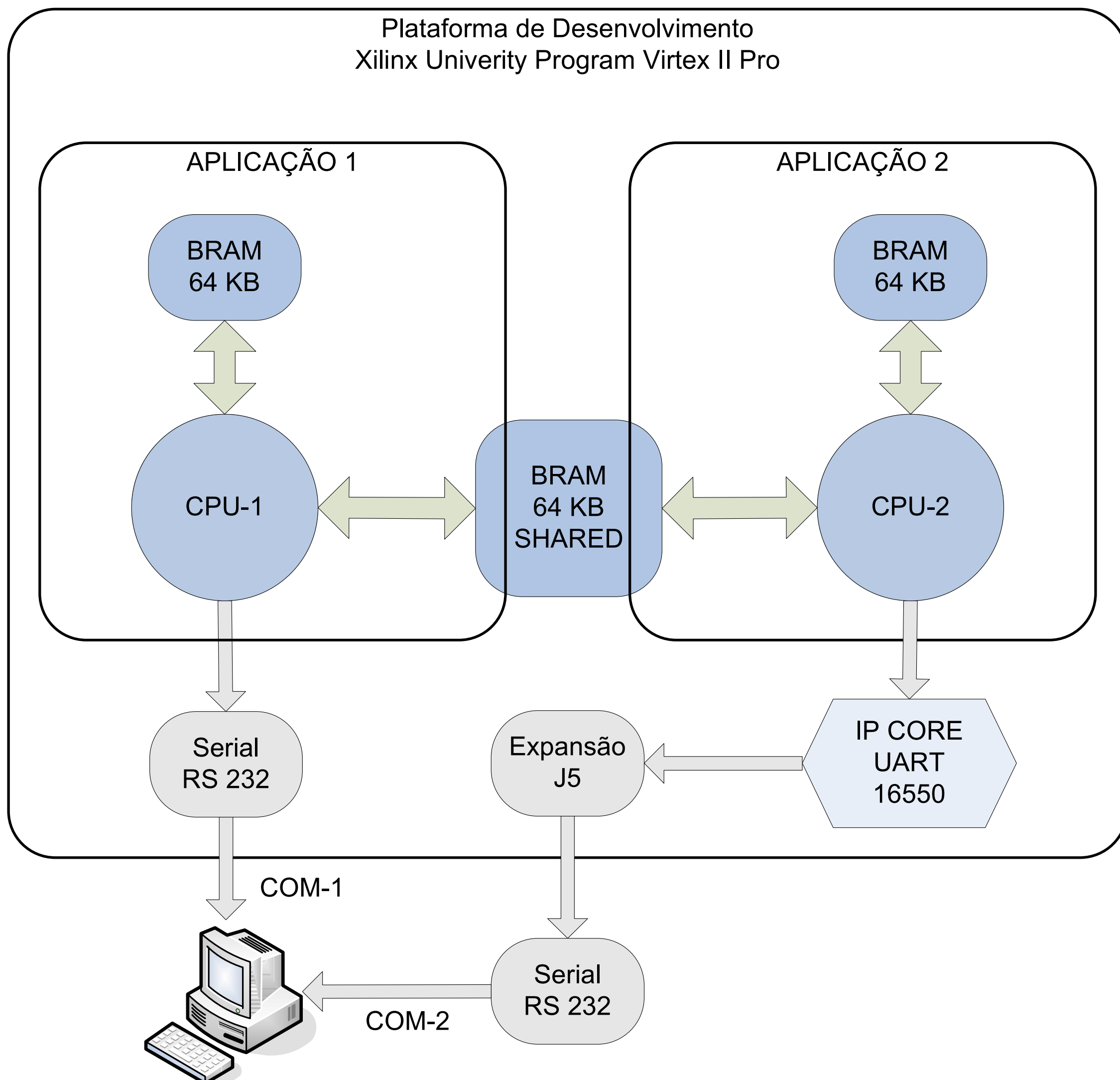


Figura 30: Diagrama de ligação das portas seriais (versão final).

A primeira mudança foi a inclusão do *core* UART 16550, que faz todo o protocolo de comunicação serial. As conexões de comunicação (Tx e Rx) foram mapeadas no conector J5 para permitir a comunicação externa com o PC. Após isso, a fim de se converterem os sinais,

foi construído o circuito de conversão de nível TTL para RS232, usando o circuito integrado MAX 232. O esquemático utilizado encontra-se disponível no *datasheet* do componente [32].

Foi adquirida e instalada, internamente no barramento PCI do PC, uma placa de comunicação serial extra (RS232 9 pinos) para aumentar a quantidade de portas seriais disponíveis. Após os testes iniciais, verificou-se o correto funcionamento do sistema, utilizando-se como janela de visualização o programa *hiper terminal*.

Realizaram-se vários testes a fim de se conferir o correto funcionamento das portas serial ao receberem dados simultaneamente vindos dos dois processadores embarcados com o objetivo de avaliar a resposta do PC a essa situação. Todos os testes realizados obtiveram bons resultados.

O diagrama detalhado do *hardware* encontra-se na Figura 31. Pode-se observar que o sistema é composto por dois bancos de BRAM com seus respectivos controladores ligados em cada processador através do barramento PLB. Para a comunicação entre as CPUs, há um módulo controlador de BRAM (*MEMORY_UNIT*) em cada unidade de memória que utiliza as duas portas de uma das BRAMs (A e B), permitindo a comunicação entre os barramentos PLB dos dois processadores.

O módulo DCM (*Digital Clock Manager*), localizado no bloco *dcm_module*, é o núcleo responsável por gerar os sinais de *clock* para os processadores e os sinais de sincronismo para os barramentos (OPB e PLB). O núcleo de *reset*, localizado no bloco *proc_sys_reset*, gera os sinais para reinicialização dos diversos níveis do Power PC (*Chip*, CPU e *Core*) e dos barramentos. Os módulos de comunicação UART-16550 fazem a comunicação individual entre os processadores e o PC através do canal de comunicação serial RS232, e os módulos de interrupção geram os sinais para o correto funcionamento do sistema.

É importante salientar, caso se esteja interessado em embarcar Linux nesta plataforma, que se deve acrescentar o módulo IP de *timer*, que é essencial para o funcionamento dessa arquitetura, uma vez que ele fornece o *hardware* para as rotinas de temporização do sistema operacional [44, 45].

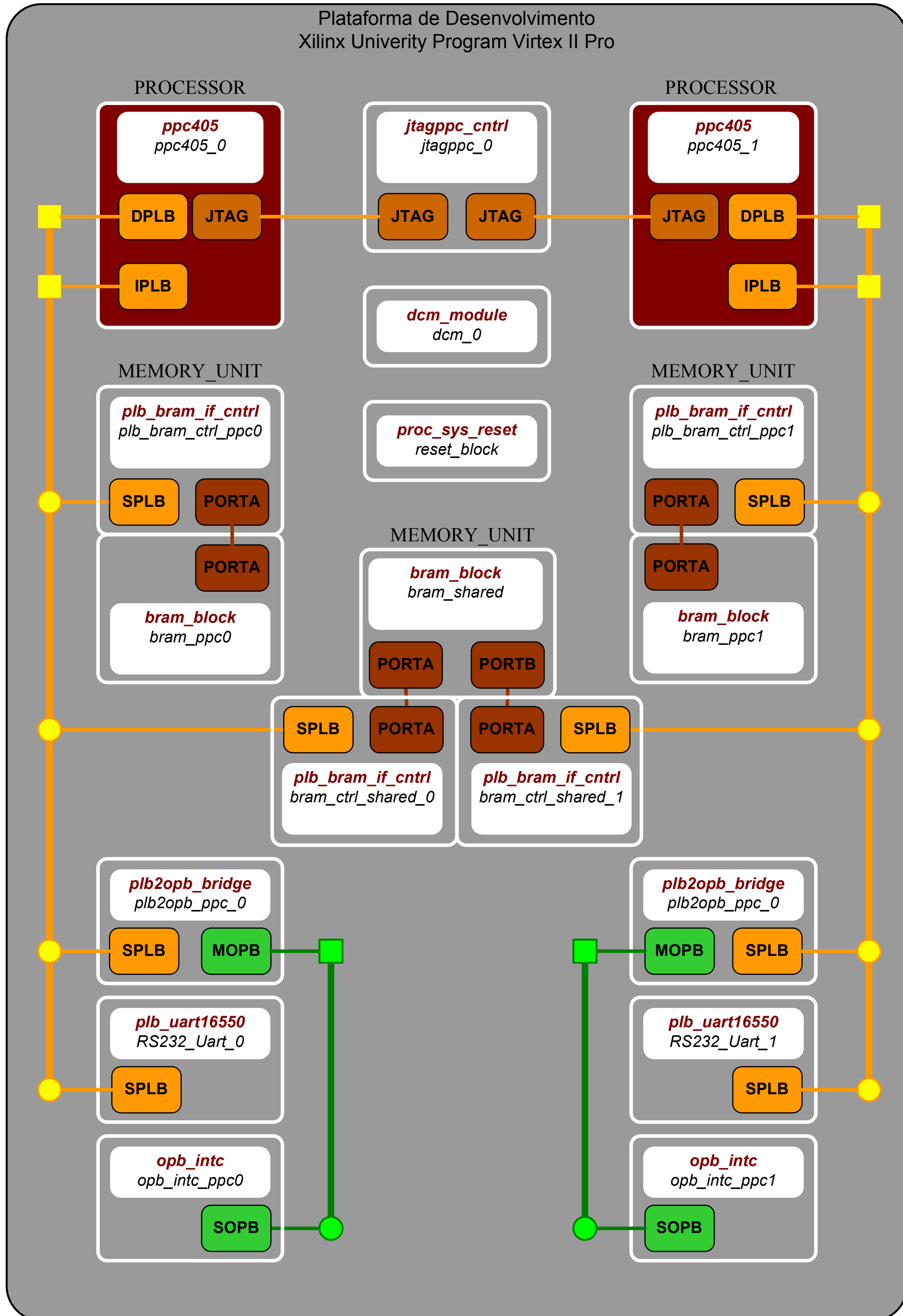


Figura 31: Diagrama em blocos completo do sistema embarcado.

3.1.2 Técnica de CFCSS Aplicada a Vários Processadores

A técnica de CFCSS, tal como originalmente proposta, visa à detecção de falhas que alteram o fluxo de controle do processador durante a execução das instruções de um dado programa de *software*. Portanto, essa técnica é proposta para sistemas monoprocessados. Como o objetivo deste trabalho é a adaptação dessa técnica para operar em sistemas multiprocessados, cada processador deve distribuir a assinatura gerada em cada um de seus blocos básicos ao(s) outro(s) processador(es) que compõe(m) o sistema embarcado. Desse modo, caso haja uma falha no fluxo de execução da aplicação em um processador e este não a detecte, o outro processador do sistema poderá indicar essa falha.

Uma das principais vantagens dessa técnica corresponde à detecção de falhas por uma entidade diferente daquela em que se pretende garantir maior robustez, ou seja, por outro dispositivo de *hardware* (isto é, outro processador remotamente conectado na rede). Essa arquitetura é intrinsecamente mais robusta à ocorrência de falhas e deve apresentar maior confiabilidade. Como exemplo, pode-se citar um processador (diz-se processador P1) que é corrompido por uma falha e que, em conseqüência, entra em um laço infinito, de forma que, para tirá-lo desta condição, um comando externo de *reset* torna-se obrigatório. Neste caso, a técnica CFCSS executada no próprio processador (P1) é incapaz de detectar a condição de falha, mas a adaptação proposta neste trabalho para sistemas multiprocessados, na qual a técnica de CFCSS é verificada em entidades de *hardware* distintas (isto é, diferentes processadores), permite que esse tipo de falha, em que o processador entra em laço infinito ou entra em um estado em que ele perde o controle sobre si mesmo, seja automaticamente detectado pelos outros processadores do sistema.

A técnica CFCSS adaptada para sistemas multiprocessados consiste no envio das assinaturas da aplicação geradas em tempo de execução de cada bloco básico a uma fila de espera para que possam ser conferidas por outro processador que não seja o executor do fluxo. As características fundamentais dessa técnica proposta consistem

a) no método de envio dos dados às filas;
b) no tamanho da fila;
c) na localização da fila na memória do sistema
d) e na maneira como ocorrerá a comunicação entre os processadores.

Conforme se observou, é fundamental assegurar que o gerenciador da fila esteja protegido de qualquer tipo de falha, uma vez que a ocorrência de uma falha nessa rotina compromete o funcionamento do sistema.

Até o momento, foram idealizadas três abordagens para a implementação da proposta do método de CFCSS para sistemas com vários processadores embarcados. No entanto, somente a segunda abordagem (cálculo da assinatura com verificação redundante) foi implementada, por ser, a *priori*, mais eficiente em termos de tempo de execução, pois não necessita esperar que a assinatura no outro processador seja calculada para continuar com a execução normal do programa como descrita na primeira abordagem (cálculo da assinatura em modo sincronizado) permitindo que processadores de diferentes freqüências possam trabalhar no mesmo sistema embarcado sem que um processador tenha que esperar pela execução do outro processador. A terceira abordagem foi também descartada por não apresentar verificação de assinatura em cada bloco básico, deixando assim espaços não protegidos na área de código. A seguir, apresentam-se e analisam-se as três abordagens.

Abordagem 1: Cálculo da assinatura em modo sincronizado.

Nesta abordagem, mostrada na Figura 32, são inseridos semáforos em cada bloco, os quais instruem o processador(*master*) a aguardar pelo cálculo da assinatura realizada no outro processador(*slave*). Cada processador deve esperar o término da execução de cada bloco do outro processador, para que este realize a conferência da assinatura enviada. Neste caso, dependendo da diferença de tempo no processamento de cada bloco, haverá muito atraso na execução do programa no processador que está executando o menor bloco, pois este deverá aguardar pelo término da execução do bloco maior do outro processador para seguir executando a aplicação.

Como mostra a Figura 32, cada vez que o fluxo de execução da CPU entra em um bloco básico de sua aplicação, um semáforo (lock-1, lock-2) é setado indicando a entrada no bloco básico. A seguir, os dados para o cálculo da assinatura do bloco (s, d e D) são enviados ao outro processador para conferência, juntamente com a assinatura de tempo de execução G, calculada localmente. Pode-se observar que o fluxo de execução na CPU remota fica interrompido enquanto a transferência dos dados da assinatura não é completada. Quando todos os dados foram enviados às respectivas filas, é realizado o cálculo da assinatura enviada

remotamente e, também, é feita a comparação entre a assinatura calculada e a recebida. Caso sejam iguais, o programa segue seu fluxo normal; caso contrário, é chamada a rotina de tratamento de erros.

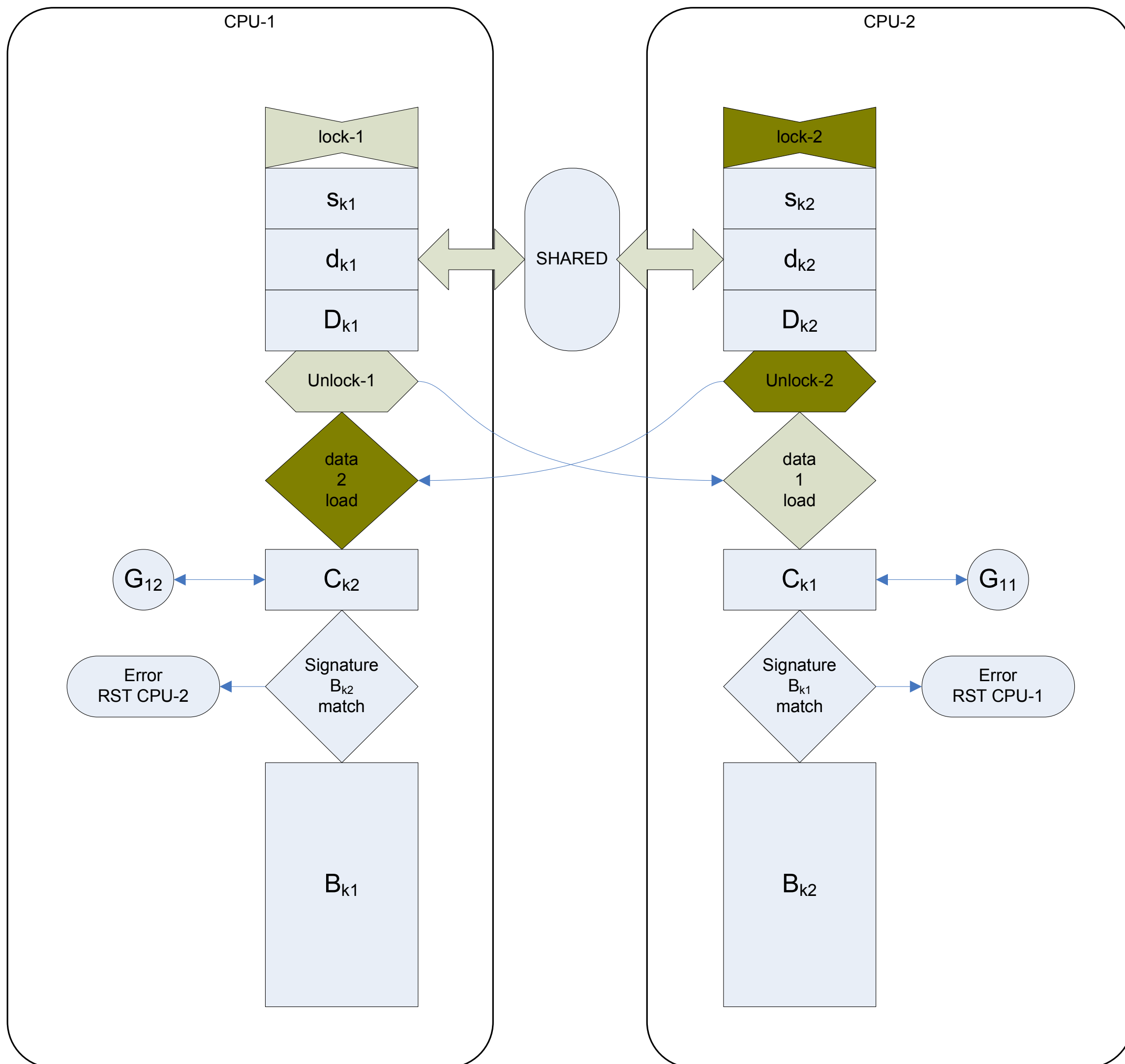


Figura 32: Bloco básico para cálculo da assinatura sincronizada.

Abordagem 2: Cálculo da assinatura com verificação redundante.

Nesta abordagem, mostrada na Figura 33, cada processador realiza a verificação de assinatura do seu próprio processo, como no método original de CFCSS, enviando, também, uma cópia da assinatura do bloco B_k e do valor atual da assinatura em tempo de execução (G_k), através de uma fila, ao outro processador, que apenas confere o valor, indicando falha quando essas assinaturas não conferem. Esta abordagem permite que laços infinitos sejam detectados e interrompidos, visto que laços dentro de um bloco básico irão interromper o

envio de assinaturas ao outro processador. Isso esvaziará a fila de assinaturas do outro processador, indicando a detecção de uma possível falha. A implementação de filas para armazenar as assinaturas diminui o impacto no tempo de execução da tarefa nos processadores, pois estes ficam livres para executar suas aplicações o mais rápido possível sem precisar esperar pelo resultado do outro processador. Importa salientar o cuidado no correto dimensionamento da fila de armazenamento das assinaturas para se evitar que o processador precise aguardar por liberação de espaço.

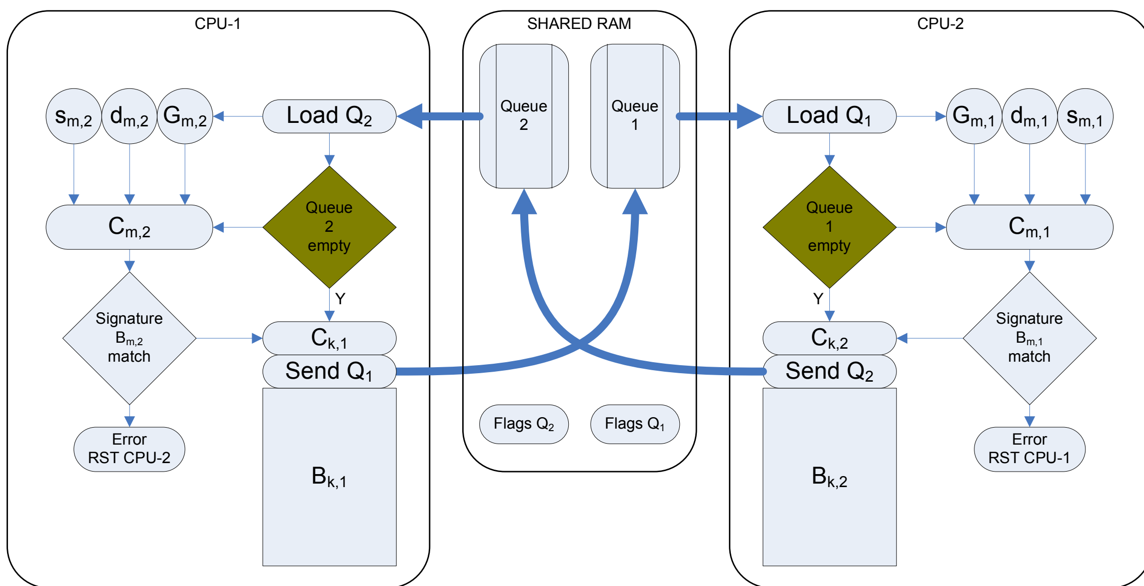


Figura 33: Bloco básico com cálculo de assinatura por fila de espera.

Uma possível implementação em alto nível desta técnica é a utilização de um sistema operacional embarcado como plataforma, em que haverá uma tarefa para a aplicação 1, outra para a aplicação 2 e uma terceira, que gerenciará as filas de assinaturas na memória compartilhada, conferindo as assinaturas e indicando falhas quando estas ocorrerem. Esse conceito foi testado, mas não se obteve sucesso em sua implementação por causa da incompatibilidade entre o núcleo de IP MPMC2 e a ferramenta EDK 8.1i, o que corresponde a uma possibilidade de trabalho futuro, já que, de acordo com o fabricante, esse problema de incompatibilidade já foi solucionado.

Uma variação possível desta técnica é a atribuição de um endereço da fila para cada assinatura de bloco básico. A fila receberá a assinatura do bloco e a assinatura em tempo de execução (G), sabendo-se assim, *a priori,* o tamanho da fila de armazenamento das assinaturas – que será o dobro da quantidade de blocos básicos do sistema. Cada vez que o

fluxo de execução passa por um bloco, sua assinatura é enviada a um endereço específico da fila. Torna-se mais difícil implementar esta abordagem à medida que se aumenta o tamanho do programa aplicativo, pois há o limite de memória interna (*BRAM*) disponível no FPGA.

Abordagem 3: Cálculo da assinatura com verificação ciclo a ciclo.

Nesta abordagem, as assinaturas a serem processadas são colocadas em uma fila de espera como no modelo anterior, com a diferença de que, apenas em cada ciclo completado pelo programa, as assinaturas são processadas e conferidas. Neste caso, o tempo de resposta fica comprometido, visto que uma falha em um bloco só será detectada no final do ciclo pelo outro processador; além disso, é necessário um "tempo extra" para calcular as assinaturas na fila de espera, perdendo-se, dessa forma, a capacidade de detecção de falha em tempo real. Outra desvantagem é que, dependendo do tamanho do programa, a fila de armazenamento das assinaturas na memória compartilhada deve ser precisamente dimensionada para evitar sobrecarga.

3.1.3 Protocolo de Comunicação

Para realizar a correta comunicação entre as aplicações e o PC, desenvolveu-se um protocolo de comunicação serial que tem a função de enviar os dados gerados na aplicação para sua futura conferência, assim como o seu estado de execução, além de sinais que visam permitir a medida de tempo e do correto funcionamento da aplicação.

A fim de facilitar o uso da plataforma de *software*, o protocolo deve ser simples e eficiente, de modo a simplificar a programação tanto de envio como de recebimento das informações. Foi definida a seguinte estrutura para o protocolo de comunicação.

Todos os comandos devem vir precedidos do caractere *underscore* "_", seguidos de uma letra maiúscula, que define o tipo de informação, e de um número, que indica o número da aplicação a que pertence o comando. As funções de cada letra são apresentadas a seguir:

• **_T<n>** indica o fim de um ciclo completo da aplicação, servindo como *checkpoint* para medida de tempo de execução e como indicador de *timeout* caso a aplicação pare de responder:

Exemplos: **_T1** indica fim do ciclo da aplicação no processador 1;
_T2 indica fim do ciclo da aplicação no processador 2.

• **_F<n>** indica que uma falha foi detectada pela técnica de falha utilizada no próprio fluxo de controle da aplicação.

• **_M<n>** indica que a assinatura enviada não confere com os valores informados.

Exemplo: **_M1** indica que as assinaturas na fila provenientes da aplicação rodando no processador 2 não conferem, ou seja, o processador 1 detectou remotamente falha no fluxo da aplicação rodando no processador 2;

_M2 indica que as assinaturas na fila provenientes da aplicação rodando no processador 1 não conferem, ou seja, o processador 2 detectou remotamente falha no fluxo da aplicação rodando no processador 1.

• **_O<n>:<dado>** transmite os dados da aplicação para o PC. Por esse comando, a aplicação pode enviar os dados gerados para o programa de controle no PC.

Exemplos: **_O1:890** indica que a aplicação do processador 1 envia o dado 890;
_O2:1.23 indica que a aplicação do processador 2 envia o dado 1,23.

• **_E<n>** indica o fim da aplicação.

Com esse protocolo, é possível utilizar não somente dois processadores no sistema, mas vários, uma vez que *n* pode assumir valores de 0 a 9 (formato DOS). Esse valor fica limitado ao formato do nome do arquivo onde serão armazenados os resultados do teste.

3.1.4 Plataforma de *Software* e Metodologia de Teste

Depois do desenvolvimento da plataforma de *hardware* e da execução dos testes preliminares para a validação desta, iniciou-se o desenvolvimento de uma nova plataforma de

software para o injetor de falhas (INJETOR). Como base para essa nova versão do programa INJETOR, utilizaram-se as rotinas de leitura das portas seriais desenvolvidas, as funções de *download* do *bitstream* criadas, o protocolo de comunicação gerado e as funções de transmissão dos comandos do programa EDK desenvolvidos, na versão anterior do *software* INJETOR, para a comunicação através de uma única porta serial.

Para se realizar o *download* manual do sistema e executá-lo corretamente na placa de desenvolvimento, é necessário iniciar o programa XMD, que acompanha o pacote EDK, e digitar na janela de console os seguintes comandos:

- realizar o *download* do *bitstream* com o comando

impact -batch download.cmd

- carregar os parâmetros do *hardware* do projeto

xload xmp <nome_do_arquivo>.xmp

- conectar o programa a uma das CPUs embarcadas *ppc0* ou *ppc1*, utilizando o comando

ppc0:

connect ppc hw -cable type xilinx_platformusb port USB2 frequency 750000 -debugdevice cpunr 1

ppc1:

connect ppc hw -cable type xilinx_platformusb port USB2 frequency 750000 -debugdevice cpunr 2

- realizar o *download* do programa aplicativo

dow executable.elf

- executar o programa digitando o comando

run

Como as aplicações utilizam a porta serial, os dados podem ser observados utilizando-se o programa *hiper terminal*. A fim de parar ou de recomeçar a execução da aplicação no ponto em que ela se encontra, utilizam-se também os comandos *stop* e *con*.

Para realizar a injeção de falhas, é necessário automatizar o procedimento acima descrito, incluindo os comandos para injetar a falha (escrita de um comando em um determinado endereço) e, também, rotinas de análise das respostas produzidas pela aplicação

para a tomada de decisão correta durante a fase de injeção. Após o final da bateria de testes, executam-se rotinas para o levantamento estatístico das respostas produzidas.

Devido à natureza concorrente do processo de leitura, pela CPU do PC, dos dados enviados pela aplicação através das duas portas seriais – uma vez que as leituras dos dados das portas COM1 e COM2 deveriam ocorrer ao mesmo tempo –, ficou evidente, desde o início da concepção do *software*, que seria necessário utilizar *threads*.

Iniciou-se o projeto do *software* com um pequeno experimento para realizar a leitura das portas seriais, utilizando *threads*, a fim de se estudarem a sintaxe e a estrutura dos comandos referentes ao disparo de *threads* e à leitura das portas seriais com o Sistema Operacional Windows. O objetivo inicial era a leitura das duas portas seriais e sua apresentação em uma janela dedicada para este fim. Os comandos utilizados na criação das *threads* foram as funções *pthread_create( )* e *pthread_join( ).*

Para se utilizar *pthread_create( )*, deve-se criar uma função de tipo ponteiro – *nome_da_função( ). Desse modo, quando for executado o comando *pthread_create( ),* será criada uma nova tarefa concorrente em que se executará a função; a tarefa que fez a chamada continuará rodando normalmente. A função *pthread_join( )* faz o programa que realizou a chamada da tarefa aguardar sua finalização, para então seguir o fluxo de execução.

Ao serem realizados os testes preliminares, observou-se que a segunda porta serial, que estava associada à segunda tarefa, ficava bloqueada. Por isso, testou-se, em um único processo, a leitura simultânea das duas portas seriais para se isolar a causa do problema, a qual correspondia a uma incompatibilidade no uso da função *pthread_create( )* no ambiente Windows, uma vez que ela fora desenvolvida, originalmente, para Linux [33].

Tendo em vista não só esse problema como também a necessidade de se abrir uma janela de console para cada processo de leitura serial a fim de visualizar as respostas obtidas, pesquisaram-se outras alternativas para a solução do problema. A solução encontrada foi o uso da função *_spawnl( )*, desenvolvida para trabalhar no ambiente Windows, que pode ser associada com as funções *FreeConsole( )*, *AllocConsole( )* e *TerminateProcess( )* [34].

A etapa seguinte foi a criação de duas janelas de console independentes, associadas a cada uma das portas seriais, e a apresentação de sua leitura a fim de se testar e de se entender o princípio de funcionamento das funções pesquisadas. Todos os testes foram satisfatórios.

A função *FreeConsole( )* desconecta a saída de dados da execução do programa da janela de console atual, e a função *AllocConsole( )* cria uma janela de console dedicada e independente para a apresentação dos dados do programa. Isso permite que um programa com a função de supervisor abra e feche programas a qualquer momento.

Para realizar a comunicação entre os processos, utilizou-se a função *CreateFileMapping( )*, que transforma uma área alocada da RAM do sistema em uma memória compartilhada. Desse modo, utilizando a função *CopyMemory( )*, é possível escrever na área da memória compartilhada, a partir de qualquer processo. Para fazer a leitura da memória, basta utilizar o ponteiro correspondente do endereço a ser lido. É importante salientar que o tipo de dado da memória compartilhada é *char*, sendo necessária a devida conversão no caso de utilização de variáveis numéricas.

Com o objetivo de se proceder à mesma tentativa de leitura das portas seriais simultaneamente à apresentação dos dados lidos em janelas de console exclusivas para cada porta, realizou-se um ensaio para a verificação do funcionamento dessas funções, cujos resultados foram satisfatórios.

A meta seguinte, após a validação das leituras concorrentes das portas seriais, foi efetuar o controle de cada processador embarcado na placa de desenvolvimento. Esse procedimento visava desviar a entrada padrão de dados pelo teclado e entregá-la ao programa supervisor. Para a realização dessa etapa, procedeu-se a um estudo do funcionamento de *pipe*s.

A função em linguagem C que permite a criação de um *pipe* é a função *popen( )*, que funciona da mesma maneira que a função *open( )*. Para realizar a escrita no *pipe,* simulando a digitação no teclado, deve-se usar duas funções: a função *fputs( )* juntamente com a função *fflush( )*. O seguinte trecho de código exemplifica a maneira de se utilizarem as funções:

Para abrir um programa conectado a um *pipe*, utilizou-se o seguinte trecho de código em linguagem C:

prg=popen("xmd", "w");

A *string "xmd"* representa o nome do programa que se deseja abrir: no caso, o programa XMD. O campo *"w"* representa o tipo de acesso: no caso, somente escrita.

Para se escrever no *pipe*, usaram-se as seguintes instruções:

fputs("run\n",prg);

fflush(prg);

Pode-se observar que, na função *fputs( )*, a *string "run\n"* representa a digitação do comando *run* no EDK, onde *prg* é um ponteiro do tipo *FILE* que foi carregado pela função *popen( )*. Verifica-se que *\n* representa o pressionamento da tecla ENTER, indispensável para o correto funcionamento do envio do comando. A função seguinte, *fflush( )*, executa a "descarga" do *pipe*, isto é, força a execução imediata do comando, enviado ao *buffer* de entrada do *pipe* pela função *fputs( )*.

Utilizando os dois comandos (*_spawnl( )* e *popen( )*), criou-se um programa para realizar o *download* do *bitstream* e dos aplicativos na placa de prototipagem através do programa EDK e para proceder à leitura das portas seriais. Esse programa de teste tinha como objetivo verificar o funcionamento integrado dos conceitos pesquisados, visando à construção de um programa automatizado de injeção de falhas. Os resultados obtidos foram muito satisfatórios e serviram como base para o desenvolvimento do sistema de injeção de falhas automatizado.

O sistema de injeção de falhas automatizado baseia-se em um programa que tem como objetivo a injeção de falhas no *software* da aplicação alvo dentro do FPGA. É importante salientar que esse sistema foi primeiramente projetado para a injeção de falhas utilizando as ferramentas da Xilinx (Power PC e Microblaze). Entretanto, deve-se ressaltar que, devido à estrutura modular projetada, o sistema pode ser facilmente adaptado para outros fabricantes de FPGA. Para se proceder a essa adaptação, basta editar o código fonte, substituindo os *strings* de comandos de console enviados à ferramenta EDK da Xilinx através das funções *fputs*() pelos comandos utilizados na ferramenta do outro fabricante (Altera, Lattice, Aldec etc.).

Além disso, é preciso substituir o comando de chamada do programa EDK pelo programa de carregamento do *bitstream* do fabricante do FPGA e modificar a seqüência de chamada dos comandos, caso isso seja necessário.

O diagrama em blocos do sistema de injeção de falhas automatizado é apresentado na Figura 34. Pode-se observar que o módulo INJETOR é o módulo principal do sistema, realizando a chamada de cada módulo de acordo com a necessidade.

Primeiramente, o módulo INJETOR faz a análise dos arquivos executáveis de cada aplicação (*executable0.elf* e *executable1.elf*), localizando o endereço inicial de execução e o tamanho da aplicação. Esses dados são utilizados na etapa de injeção de falhas, pois o módulo INJETOR precisa identificar o endereço inicial e o tamanho da aplicação para poder sortear um endereço dentro da região válida do programa.

Na etapa seguinte, o programa injetor chama as rotinas de medida de tempo de execução das aplicações em cada processador (TIMER1 e TIMER2). Essas rotinas determinam o tempo necessário que cada aplicação leva para realizar um ciclo de execução completo nos respectivos processadores. O tempo medido será utilizado para se estimar um tempo limite de *timeout*, caso a aplicação não envie mais sinais de funcionamento, fazendo com que o módulo injetor reinicie a aplicação no processador que falhou. Caso o tempo de *timeout* seja atingido sem que o processador sob teste termine a execução do programa, conclui-se que, em função da falha injetada, ele entrou em um laço infinito de programa ou em alguma seqüência ilegal de execução do código que o impediram de terminar corretamente a execução da aplicação.

Após a medida de tempo de execução, realiza-se a coleta de dados produzida pelas aplicações através das rotinas GDATA1 e GDATA2. Os dados coletados são salvos nos arquivos *ref-1.log* e *ref-2.log*, que contêm os dados não corrompidos pela injeção de falhas. Os dados salvos nesses arquivos são utilizados na etapa de análise dos resultados.

Após o levantamento dos tempos de execução, dos dados de referência e da faixa de endereços válidos para a injeção de falhas, o módulo de injeção de falhas (SABOT) paralisa a execução das aplicações nos dois processadores, sorteia um endereço válido e, após, realiza a escrita na memória mediante a substituição da instrução corrente pela instrução de falha (NOP ou JUMP, dependendo do tipo de falha desejada). No caso de a falha do tipo JUMP ser

injetada, realiza-se mais um sorteio dentro da faixa de endereços válidos para se obter o endereço de destino do salto.

Após a execução da bateria de testes, realiza-se a análise dos resultados, usando o módulo de totalização dos resultados obtidos (TOTAL). Cada arquivo de teste gerado na etapa anterior é verificado em busca de erros nos dados gerados, comparando-se os dados lidos com os dados do arquivo de referência. Além disso, procuram-se as palavras-chave TIMEOUT, DETECTED e CFCD. A palavra-chave TIMEOUT é salva no arquivo do teste toda vez que a aplicação exceder o limite de tempo estabelecido de espera para a realização de um ciclo completo de execução. Esse tempo limite foi estabelecido como três vezes o tempo médio de execução, podendo ser alterado caso necessário.

A declaração TIMEOUT indica que a aplicação travou a execução, deixando de enviar dados para o módulo de injeção de falhas. Caso isso ocorra, o endereço que gerou o mau funcionamento é salvo no arquivo *excluded.txt*, o que evita que esse endereço seja novamente utilizado, pois, a cada sorteio, o endereço sorteado para a injeção da falha é verificado com os endereços do arquivo de exclusão. Caso o endereço sorteado esteja presente no arquivo de exclusão, novo sorteio é realizado até ser escolhido um endereço que não esteja na lista de exclusão.

Para não se perder a contagem dos testes realizados no caso de a aplicação travar, o número do teste é atualizado no arquivo *test.log* a cada novo ciclo de teste. Caso a aplicação trave, o programa injetor reinicializa o módulo de injeção de falha, e este realiza a leitura do arquivo *teste.log* do número do último teste realizado e reinicia a contagem dos testes a partir desse número.

Quando uma falha é detectada através da técnica de controle de fluxo utilizada, o programa injetor recebe um sinal da aplicação e grava, no arquivo gerado dos dados da amostra, a palavra DETECTED, reiniciando, após, novo ciclo de injeção para a nova amostra.

Em caso de detecção de falha pelo processador remoto (*slave*), a declaração CFCD é salva no arquivo gerado da amostra toda vez que o conjunto de assinaturas enviado ao outro processador foi corrompido por alguma razão e este realizou a detecção da falha. Isso

possibilita a verificação do fluxo de controle por ambos os processadores, permitindo a diferenciação do modo como a falha foi detectada.

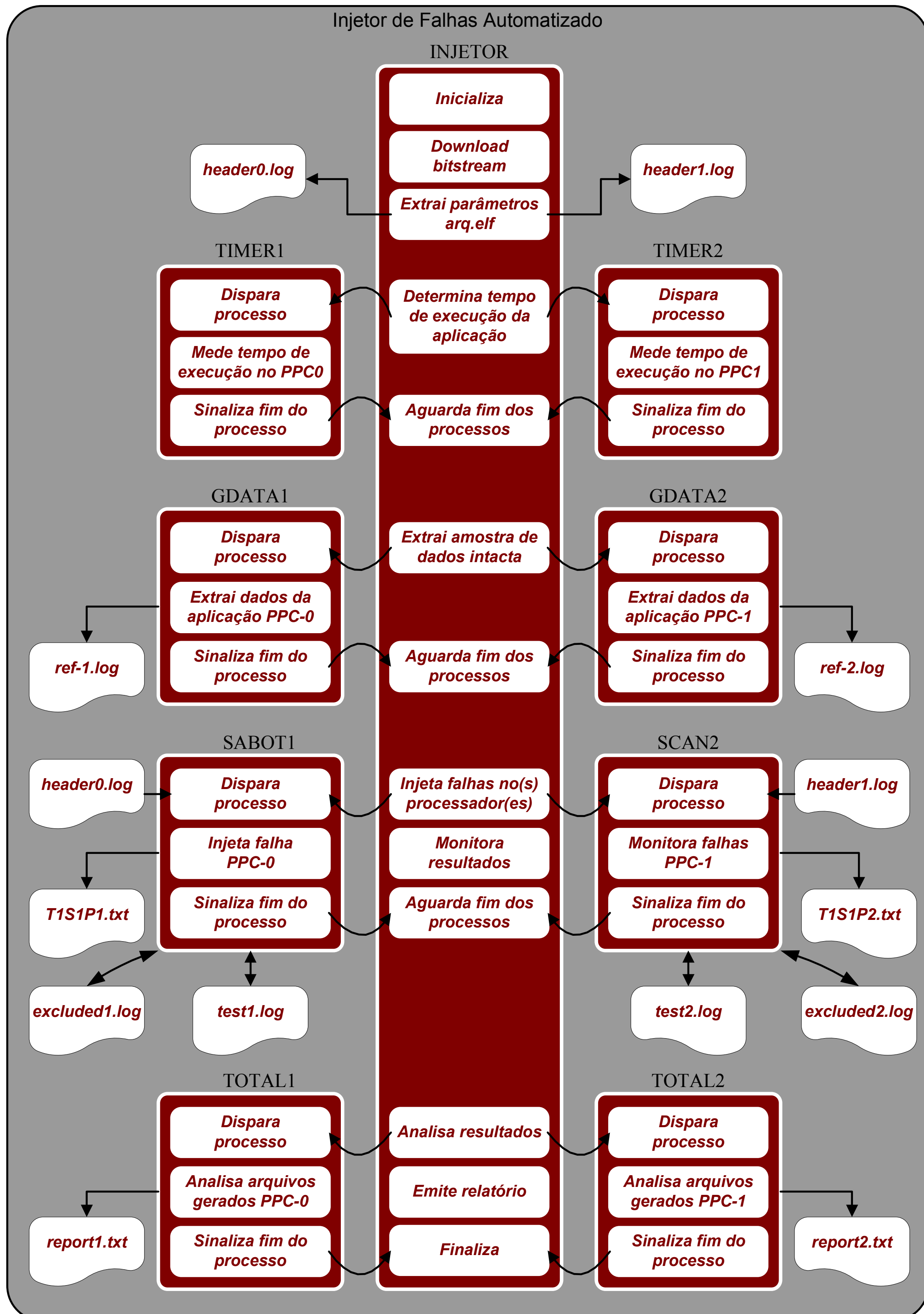


Figura 34: Diagrama em blocos simplificado do Sistema Injetor de Falhas.

A próxima seção apresenta os resultados obtidos com as injeções de falhas do tipo JUMP e NOP no sistema, bem como uma análise dos problemas observados.

3.2 RESULTADOS

Os testes realizados foram agrupados em baterias de 100 amostras para falhas do tipo JUMP e NOP, cujos resultados foram organizados como mostra a Tabela 1. Foram realizados 1200 testes, ou seja, 12 baterias de 100 amostras cada, utilizando-se a técnica proposta. Além disso, foram realizados 1200 testes com a mesma aplicação sem a técnica de detecção distribuída, a fim de se proceder a uma comparação entre os resultados. A aplicação utilizada foi um algoritmo Multiplicador de Matrizes.

	Falhas do tipo NOP			***Falhas do tipo JUMP***		
Bateria #	Erro nos dados	*Crashes* no sistema	Falhas detectadas (%)	Erro nos dados	*Crashes* no sistema	Falhas detectadas (%)
1	8	4	47	11	2	70
2	1	1	44	4	1	62
3	7	1	47	8	1	48
4	9	1	39	16	2	27
5	10	2	14	10	2	67
6	15	0	47	9	1	71
7	11	2	24	11	0	66
8	8	4	8	8	0	63
9	11	1	40	7	1	56
10	12	1	29	14	2	69
11	8	4	16	13	1	71
12	7	4	22	4	1	61
Média	**8,91**	**2,08**	**31,41**	**9,58**	**1,17**	**60,91**

Tabela 1: Resultados dos testes com a técnica CFCSS original.

	Falhas do tipo NOP			Falhas do tipo JUMP		
Bateria #	Erro nos dados	*Crashes* no sistema	Falhas detectadas (%)	Erro nos dados	*Crashes* no sistema	Falhas detectadas (%)
1	0	40	0	2	21	25
2	0	32	0	5	18	31
3	0	22	0	3	10	31
4	0	28	0	5	11	40
5	0	26	0	4	23	8
6	0	26	0	5	21	5
7	0	24	0	9	18	10
8	0	33	0	1	15	8
9	1	27	0	1	22	7
10	0	25	0	1	26	6
11	0	26	0	6	13	43
12	1	22	0	3	17	16
Média	**0,17**	**27,58**	**0**	**3,75**	**17,91**	**19,17**

Tabela 2: Resultados dos testes com a técnica CFCSS distribuída.

Conforme se verifica na Tabela 1, para a técnica de CFCSS original, o percentual de detecção para falhas do tipo NOP foi de 31,41% e, para falhas do tipo JUMP, de 60,91%. De acordo com a Tabela 2, para a técnica de CFCSS modificada, o percentual de detecção para falhas do tipo NOP foi de 0% e, para falhas do tipo JUMP, de 19,17%.

Os resultados apresentados nas Tabelas 1 e 2 foram classificados da seguinte forma: todas as ocorrências de erro, em relação aos valores corretos, nos resultados da multiplicação da matriz, as quais apareceram nos arquivos dos testes realizados, foram somadas na coluna "Erro nos dados". Esses erros são devidos a falhas de execução no fluxo de controle que não foram detectados pela técnica de detecção implementada.

Já a coluna "*Crashes* no sistema" apresenta o total de ocorrências em que a injeção de uma falha provocou a interrupção do funcionamento da aplicação. Por sua vez, a coluna

“Falhas detectadas” apresenta o total de falhas injetadas que foram detectadas pela técnica de CFCSS implementada na aplicação.

Observou-se, também, que a taxa de falhas do tipo NOP detectadas remotamente pelo segundo processador com a aplicação rodando no primeiro processador foi de 0%. Isso se justifica, pois o algoritmo modificado rodando no primeiro processador (*master*) envia, primeiramente, a cópia da assinatura de CFCSS à fila de comunicação – a fim de ser conferida pelo segundo processador (*slave*) – para então, imediatamente, calcular a assinatura do seu próprio fluxo de execução. Importa salientar, neste momento, que, devido ao tamanho da fila, ocorre um atraso na transmissão da assinatura para o segundo processador; portanto, a assinatura gerada pelo fluxo de execução é calculada, em primeiro lugar, no processador (*master*), e sua cópia, enviada ao segundo processador (*slave*), é calculada posteriormente, uma vez que a fila implementada é do tipo FIFO (*First In First Out*). Desse modo, justifica-se, devido ao atraso de comunicação da fila, que nenhum erro no fluxo de controle do primeiro processador (*master*) tenha sido detectado no segundo processador (*slave*). Isso quer dizer que a detecção de falha local ocorre sempre antes que a detecção no outro processador da rede.

Além disso, verifica-se que, para as falhas do tipo JUMP, a taxa de detecção para a técnica modificada foi inferior (19,17%) à da técnica original (60,91%). Isso se deve ao fato de que a parte do código da aplicação que implementa a rotina de tratamento das filas não foi protegida pela técnica de CFCSS, mas apenas o módulo principal, isto é, a aplicação do Multiplicador de Matrizes propriamente dito. Isso fez com que, caso houvesse um salto a essa região durante a etapa de injeção de falha, ela não fosse detectada, causando um *crash* no sistema. Para comprovar essa teoria, pode-se observar que a taxa de *crash* no sistema foi de 17,91% para técnica de CFCSS modificada contra 1,17% para a técnica original. A razão para que as rotinas de tratamento das filas não fossem protegidas pela técnica de CFCSS foi o pouco espaço em memória nas BRAM (2448Kb) disponíveis no FPGA.

Realizou-se, ainda, uma medida do impacto do tamanho da fila no tempo de execução da aplicação. Observa-se que a redução do tamanho da fila aumenta o tempo de execução da aplicação, pois ela necessita aguardar a liberação da fila por mais tempo. A Tabela 3 apresenta as medidas de tempo de execução das aplicações realizadas, modificando o tamanho da fila de comunicação entre os processadores.

Na Figura 35, pode-se observar o gráfico que mostra o efeito do tamanho da fila no tempo de execução da aplicação. Verifica-se que o tempo de execução se mantém aproximadamente constante para filas com tamanho de até 128 *bytes* e aumenta para valores inferiores a este.

Tamanho da fila (bytes)	***Tempo de execução (ms)***	
	CPU-1	CPU-2
1024	7384	7381
512	7381	7378
256	7387	7387
128	7390	7387
64	7597	7559
32	8254	8336
16	9605	9585

Tabela 3: Medidas de tempo de execução variando o tamanho das filas.

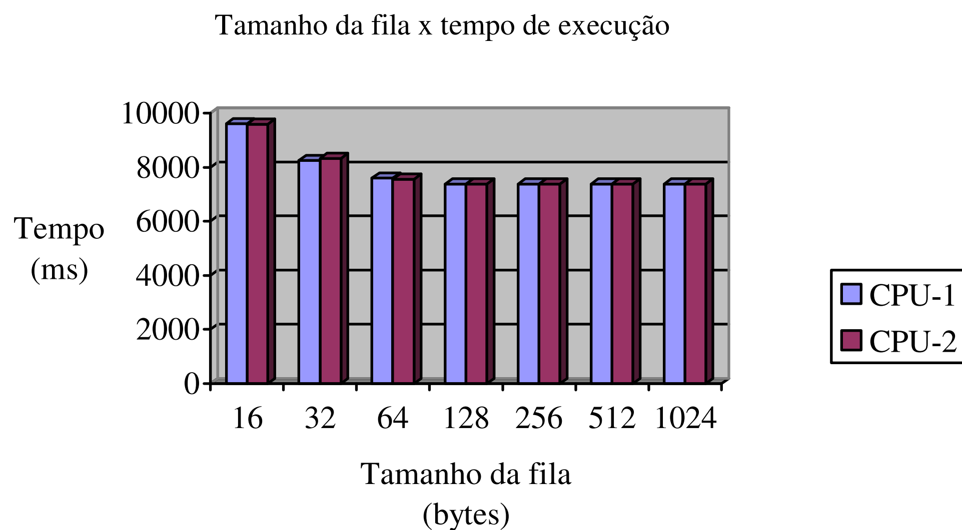


Figura 35: Relação entre tamanho da fila e tempo de execução.

Para efeitos de comparação de desempenho, mediu-se o tempo de execução da aplicação sem a implementação da técnica de detecção distribuída de falhas, apenas utilizando a técnica de CFCSS não modificada. Os resultados obtidos são apresentados na Tabela 4.

Técnica utilizada	*Tempo de execução da aplicação (ms)*
CFCSS sem assinatura distribuída	3428
CFCSS com assinatura distribuída	7381
Variação	+115,32%

Tabela 4: Medida do tempo de execução.

Pode-se observar que o tempo de execução com a técnica de verificação distribuída foi mais que o dobro (115,31%) do tempo de execução sem a técnica de verificação distribuída, devido principalmente ao tempo de espera na fila de comunicação, cujo tamanho, no experimento, era de 1024 *bytes*.

Além das medidas do tempo de execução, realizou-se uma comparação do tamanho da aplicação com e sem a técnica de detecção distribuída para observar o impacto no tamanho da aplicação. Os resultados obtidos são apresentados nas Tabelas 5 e 6.

Técnica utilizada	*Tamanho do arq. executável da aplicação (Bytes)*
CFCSS sem assinatura distribuída	40702
CFCSS com assinatura distribuída	48149
Variação	+18.29%

Tabela 5: Medidas do tamanho do código compilado das aplicações.

Técnica utilizada	*Tamanho do código fonte da aplicação (Bytes)*
CFCSS sem assinatura distribuída	7026
CFCSS com assinatura distribuída	14966
Variação	+113%

Tabela 6: Medidas do tamanho do código fonte das aplicações.

Verifica-se que o tamanho da aplicação já compilada (arquivo executável) aumentou 18,29% em relação ao código compilado de CFCSS sem a técnica de distribuição de assinaturas implementada, pois, na técnica de CFCSS distribuída, são acrescentadas as rotinas de verificação e de controle das filas, aumentando, conseqüentemente, o tamanho da aplicação. Já quanto ao código fonte, como mostra a Tabela 6, o aumento do tamanho do código foi de 113% devido aos mesmos motivos já apresentados em relação ao código fonte compilado.

O próximo capítulo apresenta as conclusões a respeito das observações feitas até o momento e aponta possibilidades para trabalhos futuros.

4 CONCLUSÃO

Com o objetivo de contribuir para o aprimoramento das técnicas de detecção de falhas existentes, as quais são dedicadas para a detecção de falhas em sistemas monoprocessados, este trabalho propôs sua expansão para sistemas multiprocessados. Para atingir esse objetivo, utilizou-se como estudo-de-caso a técnica CFCSS (*Control Flow Checking by Software Signatures*), desenvolvida por McCluskey para sistemas monoprocessados, em uma versão para sistemas com vários processadores em um único *chip*. Justifica-se esta pesquisa pela tendência de os sistemas embarcados possuírem, em futuro próximo, múltiplos processadores contendo várias aplicações sendo executadas simultaneamente.

A idéia por trás dessa nova técnica foi fazer com que cada processador verificasse as assinaturas geradas pela aplicação executada no outro processador, trocando informações através de uma estrutura de memória compartilhada. Isso permite, caso haja uma falha no *software* ou no *hardware* de um dos processadores, que o outro processador possa detectá-la e a partir daí, tomar alguma atitude em relação ao problema, reportando a falha a níveis superiores de implementação do sistema (por exemplo, o sistema operacional) ou forçando a reinicialização do processador em estado de falha.

Essa técnica pode ser ampliada futuramente em uma versão para computação distribuída em *clusters* de núcleos de processadores implementados em um mesmo *chip* ou em *chips* diferentes montados em uma mesma placa de circuito impresso, cada um executando sua respectiva aplicação. Além disso, ressalta-se que, após se colocar em funcionamento o núcleo gerenciador de memória MPMC utilizando a ferramenta EDK 9.1i, essa técnica poderá ser validada através de prototipação de SoCs operando sob o controle de operacionais de uso comercial como uC-Linux e uC-OS.

Em relação à sua estrutura, este trabalho apresentou duas partes. A primeira – *Fundamentação Teórica* – abordou a taxonomia e os conceitos básicos de sistemas tolerantes a falhas (as definições de falha, de erro e de defeito e a relação entre eles) e realizou uma revisão bibliográfica das principais técnicas de detecção de falhas em *software* em sistemas monoprocessados – ECI, BSSC, BEEC, CCA, ECCA, CCFC, CFCSS, ICFCSS e YACCA.

Além disso, descreveu-se a evolução da tecnologia reprogramável: da estrutura de FPGAs e sua granularidade até NoCs e SoCs.

A segunda parte – *Metodologia e Resultados* – descreveu a metodologia de desenvolvimento do trabalho, baseada em plataformas de *hardware* e de *software*, bem como as dificuldades encontradas ao longo da pesquisa. Em primeiro lugar, a plataforma de *hardware* foi desenvolvida utilizando-se a placa de desenvolvimento XUPV2Pro da Digilent, que possui um FPGA Virtex II Pro da Xilinx com dois processadores IBM PowerPC 405 embarcados. A cada processador PowerPC foi acoplado um bloco de memória (*BRAM*) dedicado, enquanto a comunicação entre os processadores foi estabelecida através de blocos de memória (*BRAM*) de dupla porta que estão embarcados no FPGA. Já a comunicação de cada PowerPC da placa de desenvolvimento com o ambiente externo foi implementada através de um núcleo de IP UART-16550 ligado em cada processador, o que permitiu que dois canais seriais de comunicação RS-232 fossem conectados a um PC externo.

Em segundo lugar, a plataforma de *software* que roda no PC foi composta por cinco módulos. O módulo INJETOR é responsável pelo *download* do *bitstream*, pela inicialização e seqüência de execução do teste e pela verificação dos dados de teste obtidos. Por sua vez, o módulo TIMER mede o tempo de um ciclo de execução da aplicação, o qual serve como base para o cálculo do tempo de *timeout*. Já o módulo GDATA adquire e salva, em um arquivo, uma cópia dos dados gerados pela aplicação sem a injeção de falhas para posterior comparação com os dados coletados durante a fase de injeção de falhas, enquanto o módulo SABOT é responsável pela injeção de falhas do tipo NOP e JUMP na aplicação. Finalmente, o módulo TOTAL faz o levantamento estatístico dos dados gerados durante a fase de injeção de falhas.

Finalmente, adaptou-se a técnica de CFCSS para vários processadores, propôs-se um protocolo de comunicação interprocessador e realizou-se uma série de testes práticos visando validar a técnica proposta. Ao final, os resultados obtidos foram discutidos, e trabalhos futuros foram apresentados.

A partir da descrição e da análise dos resultados, é possível concluir que a técnica de detecção distribuída de falhas não apresenta bons resultados quando utilizada em sistemas embarcados com vários processadores, sem a presença de um sistema operacional, devido,

principalmente, ao seu elevado custo de tempo de execução, que se deve, principalmente, ao atraso de espera nas filas de comunicação entre os processadores, as quais ficam cheias rapidamente. As filas cheias atrasam a execução das aplicações, pois as aplicações precisam esperar até que haja espaço disponível nas filas para o envio dos dados da assinatura em tempo de execução, a fim de serem conferidos pelo outro processador embarcado no sistema.

A presença de um sistema operacional possibilitaria que o tamanho das filas fosse maior mediante a utilização da memória DDR externa, além de permitir a criação de uma rotina dedicada (*daemon*) à conferência das assinaturas, que poderia ser executada toda vez que um dos processadores ficasse ocioso. Esse gerenciamento de recursos ficaria a cargo do sistema operacional.

Outra alternativa seria a criação, em *hardware*, de uma fila e de um sistema de conferência das assinaturas, o que diminuiria a carga de *software* extra da aplicação. Essa abordagem, entretanto, extrapola o objetivo deste trabalho, que é o desenvolvimento de uma técnica puramente em *software*, podendo, contudo, ser investigada como trabalho futuro.

No entanto, apesar dos resultados alcançados, houve avanços em relação a outros aspectos importantes, ressaltando-se a própria plataforma de teste, que, por ser modular, permite a injeção de falhas em diferentes núcleos de processadores com mínimas modificações no código fonte (PowerPC e Microblaze, por exemplo). Essa plataforma pode servir de base a outras pesquisas, com outros fabricantes, como, por exemplo, ALTERA. Além disso, o código fonte inicial para testes de injeção de falhas e o protocolo de comunicação interprocessador foram implementados e validados.

Finalmente, durante a pesquisa, desenvolveu-se a metodologia para embarcar Linux na plataforma de *hardware* com um único processador no sistema. Essa metodologia foi validada e comprovada experimentalmente, estando disponível para utilização em trabalhos futuros.

Como trabalho futuro, sugere-se a aplicação da mesma técnica, modificando-se apenas o sistema de gerência das filas de comunicação, colocando-o sob controle de um sistema operacional. Recomenda-se a utilização da ferramenta EDK 9.1i com a nova versão do sistema gerenciador de memória MPMC descrito neste trabalho, a fim de se poder utilizar a memória DDR externa da placa de prototipagem XUPV2P. Isso permite a utilização de uma

rotina específica para a gerência das filas e para a conferência das assinaturas sem degradar a execução das diversas tarefas do processador.

Sugere-se, também, a utilização da mesma técnica em uma versão híbrida *software / hardware*, desenvolvendo-se a fila, seu controle e a conferência das assinaturas em *hardware* através de um núcleo de IP dedicado sem sistema operacional, a fim de se avaliar o impacto no desempenho e no tamanho do *hardware* sintetizado e realizar uma comparação com a técnica desenvolvida puramente em *software* apresentada neste trabalho.

REFERÊNCIAS

[1] Avizienis, A.; Laprie, J-C; Randell, B; Landwehr, C. *Basic Concepts and Taxonomy of Dependable and Secure Computing*. IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, January-March 2004. pp. 11-33.

[2] Rose, J.; Gamal, A. E.; Sangiovanni-Vincentelli, A. *Architeture of Field-Programmable Gate Arrays*. Proceedings of the IEEE, Vol. 81, No. 7, July 1993. pp. 1013-1029.

[3] Le Lann, G. *An Analysis of the Ariane 5 Flight 501 Failure – A System Engineering Perspective*. IEEE, 1997. pp. 339-346.

[4] Leveson, N. G.; Turner, C. S. *An Investigation of the Therac-25 Accidents*. IEEE Computer, July 1993. pp. 18-41.

[5] Halse, R. G.; Preece, C. *Erroneous execution and recovery in microprocessor systems*. Software & Microsystems, Vol. 4, No. 3, June 1985. pp. 63-70.

[6] Miremadi, G.; Karlsson, J.; Gunneflo, U.; Torin, J. *Two Software Techniques for On-Line Error Detection*. IEEE, 1992. pp. 328-335.

[7] Miremadi, G.; Torin, J. *Evaluating Processor-Behavior and Three Error-Detection Mechanisms Using Physical Fault-Injection*. IEEE, 1995. pp. 441-454.

[8] Kanawati, G.; Nair, V. S. S.; Krishnamurthy, N.; Abraham, J. A. *Evaluation of Integrated System-Level Checks for On-Line Error Detection*. IEEE, 1996, pp. 292-301.

[9] Alkhalifa, Z.; Nair, V. S. S.; Krishnamurthy, N.; Abraham, J. A. *Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection*. IEEE, 1999. pp. 627-641.

[10] Yau, S. S.; Chen, F-C. *An Approach to Concurrent Control Flow Checking*. IEEE, 1980. pp. 126-137.

[11] Oh, N.; Shirvani, P. P.; McCluskey, E. J. *Control-Flow Checking by Software Signatures*. IEEE, 2002. pp. 111-122.

[12] Wu, Y-X.; Gu, G.-C.; Wang, K.-H. *An improved CFCSS Control Flow Checking Algorithm*. IEEE, 2007. pp. 284-287.

[13] Goloubeva, O.; Rebaudengo, M.; Reorda, M. S.; Violante, M. *Soft-error Detection Using Control Flow Assertions*. IEEE, 2003.

[14] Rose, J.; Francis, R. J.; Lewis, D.; Chow, P. *Architeture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency*. IEEE, 1990, pp. 1217-1225.

[15] Hartenstein, R. *A Decade of Reconfigurable Computing: a Visionary Retrospective*. Proceedings of the Conference on Design, Automation and Test in Europe. IEEE, 2001, pp. 1-9.

[16] Hartenstein, R. *Coarse Grain Reconfigurable Architetures*. Proceedings of the 2001 Conference on Asia South Pacific Design Automation. ACM, 2001, pp. 1-6.

[17] Nurmi, J. *Network-on-Chip: A New Paradigm for System-on-Chip Design*. IEEE, 2005, pp. 2-6.

[18] Helgemo, D. R. *Digital Signal Processing at 1GHz in a Field-Programable Object Array*. IEEE, 2003, pp. 57-60.

[19] Quénot, G. M.; Kraljié, I. C.; Sérot, J.; Zavidovique, B. *A Reconfigurable Compute Engine for Real-Time Vision Automata Prototyping*. IEEE, 1994, pp. 91-100.

[20] Wolf, W.; Xu, J.; Henkel, J.; Chakradhar, S. *A Methodology for Design, Modelling, and Analysis of the Networks-on-Chip*. IEEE, 2005, pp. 1778-1781.

[21] Sgroi, M.; Sheets, M.; Mihal, A.; Keutzer, K.; Malik, S.; Rabaey, J., Sangiovanni-Vincentelli, A. *Adressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design*. Proceedings of the 38th Conference on Design Automation. ACM, 2001, pp. 667-672.

[22] Flynn, D. *AMBA: Enabling Reusable On-Chip Designs*. IEEE Micro, 1997, pp. 20-27.

[23] Hofmann, R.; Drerup, B. *Next Generation Coreconnect Processor Local Bus Architeture*. IEEE, 2002, pp. 221-225.

[24] Liang, J.; Swaminathan, S.; Tessier, R. *aSoc: A Scalable, Single-Chip Communication Architeture.* IEEE, 2000, pp. 37-46.

[25] www.altera.com. último acesso 30/7/2008

[26] www.xilinx.com. último acesso 30/07/2008

[27] Test Techonology Standards Committee. *IEEE Standard Test Access Port and Boundary-Scan Architeture*. IEEE, 2001, pp. 1-208.

[28] Burgess Jr., R.; Nagaraj, P.; Waseq, M. N. *The Boundary Scan: A Standard Worth Pursuing in the VLSI Design Environment*. IEEE, 1995, pp. 11-12.

[29] Zhu, G.; Yan, X.; Zhou, Y.; Guo, Y. *The Algorithms of Inserting Boundary-Scan Circuit Automatically*. IEEE, 1998, pp. 527-531.

[30] www.digilentinc.com. último acesso 30/7/2008

[31] www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf. último acesso 30/7/2008

[32] *MAX 232, MAX 2321. Dual EIA-232 Drivers/Receivers*. Texas Instruments, 2002.

[33] www.cplusplus.com. último acesso 30/07/2008

[34] www.msdn.com. último acesso 30/07/2008

[35] Hill, D.; Woo, N.-S. *The benefits of flexibility in lookup table FPGAs*. In Moore, W.; Luk, W., Eds. *FPGAs*, edited from the *Oxford 1991 Int. Workshop on Field Programmable Logic and Applications*, Abingdon, 1991, pp. 127-136.

[36] Kouloheris, J.; Gamal, A. E. *FPGA performance vs. cell granularity*. Custom Integrated Circuits Conf. 91. CICC, 1991.

[37] Kouloheris, J.; Gamal, A. E. *FPGA area versus cell granularity – PLA cells*. Custom Integrated Circuits Conf. 92. CICC, 1992, pp. 6.2.1-6.2.4.

[38] Kouloheris, J.; Gamal, A. E. *FPGA area versus cell granularity – lookup tables and PLA cells*. FPGA 92, ACM First Int. Workshop on Field Programmable Gate Arrays. 1992, pp. 9-14.

[39] Rose, J.; Francis, R. J.; Chow, P.; Lewis, D. *The Effect of Logic Block Complexity on Area of Programmable Gate Arrays*. Proc. 1989 Custom Integrated Circuits Conf. 1989, pp. 5.3.1-5.3.5.

[40] Rose, J. S.; Brown, S. *The Effect of Switch Box Flexibility on Routability of Field Programmable Gate Arrays*. Proc. Custom Integrated Circuits Conf. 1990, pp. 27.5.1-27.5.4.

[41] Singh, S.; Rose, J. S.; Lewis, D.; Chung, K.; Chow, P. *Optimization of Field Programmable Gate Array Logic Block Architecture for Speed.* Custom Integrated Circuits Conference 91. CICC, 1991, pp. 6.1.1-6.1.6.

[42] Singh, S.; Rose, J. S.; Lewis, D.; Chow, P. *The Effect of Logic Block Architecture on FPGA Performance*. IEEE JSSC, 1992, pp. 281-287.

[43] www.jtag.com. último acesso 30/07/2008

[44] http://busybox.net. último acesso 30/07/2008

[45] www.itee.uq.edu.au/~jwilliams/mblaze-uclinux/powerpc_linux/. último acesso 30/07/2008

[46] Karlsson, J.; Gunneflo, U.; Lidén, P.; Torin, J. *Two Fault Injection Techniques for Test of Fault Handling Mechanisms*. International Test Conference 1991. IEEE, 1991, pp. 140-149.

[47]http://www.latticesemi.com/corporate/newscenter/productnews/2008/r080219announcesuclinuxsu.cfm último acesso 25/08/2008.

[48]http://www.latticesemi.com/corporate/newscenter/productnews/2006/r061023rtossupportacceler.cfm último acesso 25/08/2008.