Detecção e Correção de Falhas Transitórias Através da Descrição de Programas Usando Matrizes

Ronaldo R. Ferreira, Álvaro F. Moreira, Luigi Carro

[1]Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre – RS – Brasil

{rrferreira, afmoreira, carro}@inf.ufrgs.br

Abstract. *Advances in transistor manufacturing have enabled technology scaling along the years, sustaining Moore's law. In this scenario, embedded systems will face restricted resources available to deploy fault-tolerance due the increase of power consumption that these techniques require. In this paper, we claim the detection and correction (D&C) of failures at system level by using matrices to encode whole programs and algorithms. With such encoding, it is possible to employ estabilished D&C techniques of errors occurring in matrices, running with unexpressive overhead of power and energy. We evaluated this proposal using two case studies significant for the embedded system domain. We observed in some cases an overhead of only 5% in performance and 8% in program size.*

Resumo. *Os avanços na fabricação de transistores têm permitido reduzir o tamanho da tecnologia, sustentando a Lei de Moore. Neste cenário, os sistemas embarcados serão projetados com margem reduzida para a implantação de técnicas de tolerância a falhas devido ao aumento no consumo de potência que essas técnicas requerem. Neste artigo, defendemos a detecção e correção (D&C) de falhas em nível de sistema através da codificação de quaisquer programas e algoritmos com matrizes. Essa codificação possibilita empregarmos técnicas estabelecidas de D&C de erros em matrizes, incorrendo em acréscimo inexpressivo de potência e energia. Avaliamos a nossa proposta através de dois estudos de caso relevantes para o domínio de sistemas embarcados, para os quais observamos em alguns casos decréscimo de somente 5% em desempenho e de aumento 8% em tamanho de programa.*

1. Introdução

Os avanços na fabricação de transistores têm permitido reduzir o tamanho da tecnologia, sustentando a Lei de Moore. Com a tecnologia atual de 45 nm amplamente disponível e com as futuras possuindo nodos com tecnologia de 32 nm e 22 nm, as falhas transitórias causadas por radiação gerarão problemas em qualquer produto eletrônico que as utilize. Dado que a distância entre os transistores diminui rapidamente, uma partícula que venha a atingir o núcleo do circuito integrado interferirá em diversas unidades lógicas, acarretando em falhas que perduram durante diversos ciclos de relógio [Lisboa et al. 2007].

Neste cenário, os sistemas embarcados serão requisitados em seus limite de operação; eles deverão oferecer diversos serviços demandando baixíssimo gasto energético e

fornecendo alto desempenho, mesmo na presença de múltiplas falhas. Técnicas clássicas de tolerância a falhas tais como Redundância Modular Tripla (TMR) e Duplicação com Comparação (DwC) impõem acréscimos em área no fator de 3 ou 2 vezes devido a hardware adicional para a implementação dessas técnicas [Huang and Abraham 1984]. Além disso, o alto consumo de potência dessas técnicas as tornam impraticáveis para serem implantadas no domínio de sistemas embarcados [Argyrides et al. 2009].

Em um sistema no qual a disponibilidade de potência e energia são escassas, a solução mais viável para tratar falhas transitórias é detectá-las e corrigi-las em software, em nível de sistema [Argyrides et al. 2009, Huang and Abraham 1984, Pattabiraman et al. 2007, Vemu et al. 2007, Oh et al. 2002]. Deixar essa tarefa para o desenvolvedor é impraticável, pois acarretaria em acréscimos elevados nos tempos de desenvolvimento e teste do software, além na complexidade do mesmo, o que comprometeria o *time-to-market*. Portanto, o mecanismo de tolerância a falhas deve estar incorporado na linguagem de programação, implementado pelo seu compilador ou interpretador. Assim, exime-se o desenvolvedor de software de tratar essas falhas, beneficiando o processo de desenvolvimento e reduzindo o *time-to-market*.

Neste trabalho apresentamos os fundamentos essenciais para se realizar detecção e correção de falhas transitórias utilizando construções algébricas formais, adotando matrizes como maneira de se descrever quaisquer algoritmos e programas, e propomos também a incorporação desse formalismo em uma linguagem de programação. Linguagens baseadas em matrizes, por exemplo, Matlab, são amplamente adotadas pela indústria de sistemas embarcados. Apresentamos aqui esses fundamentos através de dois algoritmos importantes para o domínio de software embarcado: geração do código de Huffman [Huffman 1952] e as transformadas MDCT (Modified Discrete Cosine Transform) e IMDCT (Inverse MDCT) [Princen et al. 1987]. Por fim, apresentamos como podemos eficientemente detectar e corrigir falhas transitórias usando somente operações sobre matrizes, fornecendo suporte para a implementação da técnica proposta.

A contribuição principal deste trabalho é um mecanismo para detectar e corrigir falhas em nível de sistema, o qual incorre em acréscimos mínimos em área, desempenho e potência. Alcançamos esse objetivo através da definição de programas e algoritmos completamente com matrizes; sobre essas é possível utilizar técnicas de verificação de baixo gasto energético. Os resultados experimentais dos dois estudos de caso radicalmente diferentes ilustram os princípios do método e sua generalidade para outras aplicações.

Este texto está organizado da seguinte maneira: a seção 2 discute os trabalhos relacionados; a seção 3 revisa a técnica utilizada para detectar e corrigir erros em matrizes; a seção 4 apresenta os estudos de caso e introduz a matemática necessária para se expressar programação dinâmica com operações sobre matrizes; a seção 5 apresenta os resultados experimentais usados para demonstrar a viabilidade da técnica proposta; e a seção 6 discute as conclusões e os trabalho futuros.

2. Trabalhos Relacionados

Os autores em [Blum et al. 1989] provaram que para qualquer anel parcialmente ordenado, existe uma máquina universal correspondente sobre esse anel. Como apresentaremos na seção 4.1, os autores em [Atallah et al. 1989] descrevem qualquer algoritmo baseado em árvores através de um semi-anel de matrizes. Portanto, caso exista um anel

parcialmente ordenado de matrizes, sua máquina correspondente, de acordo com o modelo de Blum [Blum et al. 1989], é universal, bem como a nossa abordagem. A construção de um anel para qualquer tipo de algoritmo, não somente os baseados em árvores e os naturalmente representáveis por matrizes, é um dos nossos trabalho futuros. Este artigo estuda a viabilidade prática de uma abordagem completamente baseada em matrizes.

Tolerância a falhas baseada em algoritmo (ABFT) [Huang and Abraham 1984] refere-se ao caso no qual a técnica de tolerância a falhas é incorporada à computação dos dados. Os autores em [Huang and Abraham 1984] utilizaram ABFT para corrigirem operações sobre matrizes baseados em *checksums*. Os autores enfatizaram que para se utilizar ABFT a operação sendo protegida deve obrigatoriamente preservar a propriedade do *checksum*, o que não é sempre o caso. Eles detectam e corrigem vários elementos com erro na matriz resultante, mas dependem para tal de uma rede inter-conectada com diversos processadores. Neste trabalho, a detecção e a correção são independentes das operações e da organização do hardware.

Verificação de programas [Blum and Kanna 1989] é uma técnica utilizada para verificar se os resultados produzidos por um programa estão corretos ou não. Para que essa técnica seja viável, o mecanismo de verificação deve ser assintoticamente menor que o algoritmo sendo verificado. Do contrário, a verificação de programas equivale-se à recomputação dos resultados. os autores em [Prata and Silva 1999] mostraram, com o suporte de campanhas de injeção de falhas, que verificação de programas é superior à ABFT, incorrendo em acréscimos consideravelmente menores no tempo total de execução do programa.

Os autores em [Lisboa et al. 2007] propuseram utilizar o vetor da técnica de Freivalds [Motwani and Raghavan 1995] fixado em $r = \{1\}^n$, protegendo com essa técnica um multiplicador de matrizes implementado em hardware. Em [Argyrides et al. 2009], o esquema apresentado em [Lisboa et al. 2007] foi estendido para permitir a correção de erro em um elemento da matriz resultante. Este trabalho adota essas técnicas como mecanismo de correção e detecção de erros em matrizes, implementando-as em software.

A proteção de variáveis críticas de uma aplicação pode ser realizada através de análise estática durante a compilação do código-fonte [Pattabiraman et al. 2007]. Nesse método, realiza-se o particionamento do programa a partir dos blocos básicos dele extraídos, seguindo a análise de criticidade de uma variável realizada através da contagem de leituras e escritas de cada variável. Os autores em [Pattabiraman et al. 2007] obtiveram um acréscimo médio de 33% em tempo total de execução para proteger somente 5 variáveis. Comparada à nossa abordagem, essa taxa é altíssima. Considerando uma variável de 32 bits, através da análise de criticidade de cada variável, há um acréscimo médio de 33% para se proteger somente 160 bits, enquanto na nossa abordagem protegemos uma matriz composta de n^2 e $n \times n/2$ variáveis com menos penalidade ao desempenho, sendo de 5% para Huffman e de $\sim$ 30% para MDCT/IMDCT.

Outro problema causado nas aplicações devido a falhas transitórias é a inserção de falhas no fluxo de controle. Os autores em [Vemu et al. 2007] apresentam um mecanismo - batizado ACCE - baseado em software para a detecção e correção desse tipo de falha. Após a análise de dependência entre os blocos básicos, ACCE é capaz de detectar o nodo de controle que contém erro no grafo de execução, permitindo sua posterior

correção. Neste trabalho, assume-se que as falhas de controle são tratados em hardware, através da triplicação do contador de programa. Nesse caso, adicionam-se somente dois registrados, incorrendo em acréscimo em área negligenciável. Além disso, este trabalho propõe a descrição completa da aplicação com matrizes, o que acaba com as asserções de controle na aplicação, amortizando a ocorrência de falhas de controle. Esse mecanismo de correção de falhas de controle em matrizes é um dos nossos trabalhos futuros.

Outra técnica para detectar falhas transitórias é o uso de invariantes de software através da detecção automática de pré-, pós-condições e invariantes de laço. Esse método baseia-se em ferramentas estado-da-arte para a detecção de invariantes, tais como a Daikon [Ernst et al. 2007]. Infelizmente, sendo a detecção de invariantes de laço um problema indecidível, essas ferramentas empregam heurísticas para inferir alguns dos invariantes. Alguns autores reportam resultados bastante negativos ao se usar invariantes para detectar falhas [Krishnamurthi and Reiss 2003]. Pode-se melhorar a detecção de falhas com invariantes, mas, geralmente, necessitando de procedimentos de instrumentação de código [Cheynet et al. 2000]. Apesar do baixo acréscimo no tempo de execução incorrido pela técnica de invariantes, [Lisboa et al. 2009], a taxa de detecção de falhas é baixa comparada à técnica de matrizes [Lisboa et al. 2007]. Isso ocorre porque os detectores de invariantes não são cientes da semântica implementada na aplicação, pois elas trabalham sobre o código-fonte.

Os autores em [Chen and Kandemir 2005] propuseram uma Máquina Virtual Java (JVM) tolerante a falhas para o domínio de sistemas embarcados. Nessa JVM, há dois motores executando duas instâncias da aplicação, trabalhando de maneira similar à duplicação com comparação. Para amortizar o alto acréscimo no tamanho de programa imposto pela duplicação de objetos, os autores propuseram um mecanismo de compartilhamento de objetos entre as duas instâncias da aplicação em execução. Essa abordagem sempre incorre em um acréscimo maior que 100% no tempo total de execução, ao executar em ambientes monoprocessados. Na nossa abordagem, o acréscimo pode ser tão baixo quanto 5% em alguns casos. Existem linguagens de programação tolerantes a falhas com o objetivo de recuperar automaticamente a execução em sistemas distribuídos na camada de aplicação [Florio and Blondia 2008], mas nenhuma dessas é voltada ao domínio de sistemas embarcados, no qual o desenvolvimento de software possui restrições severas em recursos de hardware.

3. Fingerprinting de Matrizes

Fingerprinting consiste em verificar se x e y são iguais dado um universo U. Considerando um mapeamento aleatório de U em um universo V, onde $|V| \ll |U|$, pode-se verificar eficientemente que $x = y$ se e somente se suas imagens são as mesmas em V. Esse mapeamento em um outro universo de menor cardinalidade reduz o espaço de busca, tornando o processo de verificação mais eficiente do que recomputar a operação em U [Motwani and Raghavan 1995].

A *técnica de Freivalds* tal como apresentada em [Motwani and Raghavan 1995] verifica se a multiplicação de matrizes $XY = Z$ em tempo $O(n^2)$, sendo muito mais eficiente do que o melhor algoritmo conhecido para a multiplicação de matrizes, o qual é $O(n^{2,376})$ [Coppersmith and Winograd 1987]. Apesar da existência desse algoritmo, a maioria das implementações adotam a solução trivial de tempo $O(n^3)$ devido a sua

clareza e concisão de codificação. Na realidade, a técnica de Freivalds é capaz de verificar qualquer identidade entre matrizes $X \bullet Y = Z$, mas é equivalente à recomputar $X \bullet Y$ quando a operação $\bullet$ é $O(n^2)$, o que é o caso da adição e subtração de matrizes. A técnica de Freivalds se caracteriza pelo seguinte teorema provado em [Motwani and Raghavan 1995]:

Teorema 1 *Seja $XY \neq Z$ e sejam X e Y matrizes $n \times n$. Escolha aleatoriamente de maneira uniforme um vetor $r \in \{0,1\}^n$. Tem-se $X(Yr) = Zr$ com probabilidade $p \leq 1/2$.*

A computação de $X(Yr)$ e Zr requer $O(n^2)$ para cada operação. Portanto, a verificação da identidade de matrizes reduz-se à verificação se os dois vetores calculados são iguais, e essa operação pode ser realizada em tempo $O(n)$. Assim, a operação total requer $O(n^2)$. Os autores em [Lisboa et al. 2007] adotaram a técnica de Freivalds fixando $r = \{1\}^n$. De acordo com o Teorema 1, esse vetor é uma das possibilidades que pode ser escolhida aleatoriamente. Ao fixar r como proposto, tem-se o seguinte corolário provado em [Lisboa et al. 2007]:

Corolário 1 *Seja $XY \neq Z$ e sejam X e Y matrizes $n \times n$. Fixe o vetor $r = \{1\}^n$. Nesse caso, tem-se $X(Yr) \neq Zr$ com probabilidade 1.*

Fixando r como apresentado no Corolário 1 nos permite utilizar a técnica de Freivalds como um verificador eficiente de operações sobre matrizes. Em [Argyrides et al. 2009], estendeu-se Freivalds para permitir a correção e detecção do elemento errôneo da matriz resultante Z com somente duas adições, o que requer tempo $O(1)$. Neste trabalho, adotamos a técnica de Freivalds como apresentado no Corolário 1, aplicando-a para todas as operações sobre matrizes ocorrendo nos programas, incluindo as que requerem tempo $O(n^2)$ para executar. Também adotamos o mecanismo de correção proposto em [Argyrides et al. 2009], oferecendo-nos um *framework* de correção extremamente eficiente no caso de se detectar um erro após a computação de Freivalds e da checagem da identidade $X \bullet Y = Z$. No caso de erro, evita-se a recomputação de $X \bullet Y$, reduzindo a complexidade de $O(n^3)$ e $O(n^2)$ para $O(1)$, caso a operação $\bullet$ seja $O(n^3)$ e $O(n^2)$ respectivamente.

4. Estudos de Caso

Apresentamos nessa seção os dois algoritmos utilizados como estudo de caso nos experimentos de injeção e proteção contra falhas. Huffman e MDCT/IMDCT são algoritmos importantes, sendo componentes centrais de aplicações multimídia, portanto, são claramente exemplos representativos de aplicações embarcadas reais.

4.1. Código de Huffman

Essa seção apresenta o algoritmo que calcula a árvore ótima de Huffman baseado inteiramente em multiplicações e adições de matrizes definidas sobre um semi-anel. Esse e outros algoritmos foram apresentados em [Atallah et al. 1989], onde problemas de programação dinâmica foram reduzidos à operações de matrizes definidas em semi-anéis e, como apresentado, essa abordagem é aplicável a uma ampla classe de problemas de programação dinâmica.

Sejam $(p_1, \ldots, p_n)$ o vetor ordenado de freqüências do alfabeto, $p_{i,j} = \sum_{k=i}^{j} p_k$ a freqüência acumulada entre as palavras i e j do alfabeto, e S uma matriz $n \times n$ contendo

freqüências acumuladas como segue:

$$S_{i,j} = \begin{cases} p_{i+1,j} & \text{se } i < j \\ +\infty & \text{se } i \geq j \end{cases} \quad (1)$$

Esse algoritmo é definido sobre o semi-anel {min, +} (conhecido como semi-anel *tropical*). Esse semi-anel possui $\mathbb{R} \cup \{+\infty\}$ como seu domínio e $\forall a, b \in \mathbb{R} \cup \{+\infty\}, a \oplus b = min(a, b)$ e $a \otimes b = a + b$. No semi-anel tropical, $\forall a \in \mathbb{R} \cup \{+\infty\}, a + \infty = (+\infty) + a = +\infty$ e $min(a, +\infty) = a$. Seja A_h a matriz contendo todos os comprimentos de árvores de Huffman, na qual a entrada $(A_h)_{i,j}$ contém o comprimento de caminho das árvores $(p_{i+1}, \ldots, p_n)$ de tamanho no máximo h, com $0 \leq h \leq \lceil \log n \rceil$. Com essas definições e sendo $0 < i < j \leq n$, A_h é uma matriz $n \times n$ definida recursivamente da seguinte maneira:

$$(A_0)_{i,j} = \begin{cases} +\infty & \text{se } i \geq j \text{ ou } (j - i) > 1 \\ 0 & \text{caso contrário} \end{cases} \quad (2)$$

$$A_h = A_{h-1} \oplus \Big(A_{h-1} \otimes (A_{h-1} + S) \Big) \quad (3)$$

Onde as operações tropicais $X \otimes Y$ e $X \oplus Y$ são:

$$(X \otimes Y)_{i,j} = \bigoplus_{k=1}^{n} X_{i,k} \otimes Y_{k,j} \quad (4)$$

$$(X \oplus Y)_{i,j} = X_{i,j} \oplus Y_{i,j} \quad (5)$$

Na Equação 2, o 0 vem do fato que o comprimento de caminho da árvore até a altura $h = 0$ é igual a 0. Note que na Equação 3 a operação + é a adição tradicional de matrizes, não a definida pelo semi-anel tropical. Note também que a Equação 4 é similar à multiplicação tradicional de matrizes, requerendo tempo $O(n^3)$, e a Equação 5 é similar à adição tradicional de matrizes, requerendo tempo $O(n^2)$. A entrada $(A_{\lceil \log n \rceil})_{1,n}$ contém o comprimento de caminho ótimo da árvore de Huffman de n folhas. Isso conclui a apresentação da geração do código de Huffman implementado totalmente com adição e multiplicação de matrizes. Ressalta-se que essa redução de programação dinâmica em operações sobre o semi-anel tropical aplica-se a qualquer algoritmo baseado em árvores, não somente para Huffman [Atallah et al. 1989].

4.2. Transformadas MDCT e IMDCT

Duas funções importantes no domínio de aplicações multimídia são as transformadas MDCT e IMDC. Essas funções são amplamente utilizadas em aplicações multimídia de tempo-real em padrões de áudio e vídeo, tais como MPEG e MP3. Tanto a MDCT quanto a IMDCT são calculadas de maneira bastante simples, sendo implementadas inteiramente com multiplicação de matrizes representando os dados em compressão. Os algoritmos usados neste artigo foram extraídos de [Cheng and Hsu 2003].

Sejam x^T e $\hat{x}^T$ dois vetores transpostos de comprimento n, os quais representam os valores originais e os produzidos pela IMDCT, respectivamente. Sejam M uma matriz

$n/2 \times n$ contendo os coeficientes da transformada e X um vetor de comprimento $n/2$ contendo o resultado da MDCT. A MDCT e a IMDCT podem ser calculadas como segue:

$$X = Mx \tag{6}$$

$$\hat{x} = M^T X \tag{7}$$

Onde $M_{i,j}$, com $0 \leq i \leq n/2$ e $0 \leq j \leq n-1$, é:

$$M_{i,j} = \cos\left[\frac{\pi}{2n}\left(2j + 1 + \frac{n}{2}\right)(2i+1)\right] \tag{8}$$

As Equações 6 e 7 são computadas e verificadas em tempo $O(n^2)$. Nesse caso, a MDCT e a IMDCT se beneficiarão de alguma forma da técnica de Freivalds ao se proteger o resultado da multiplicação e ao recomputar a matriz em caso de erro. Entretanto, o acréscimo no tempo de execução ao proteger as transformadas não será tão eficientemente amortizado ao aumentar o tamanho do espaço do problema como em Huffman. A próxima seção discute esses aspectos de amortização da proteção e espaço do problema.

5. Experimentos e Resultados

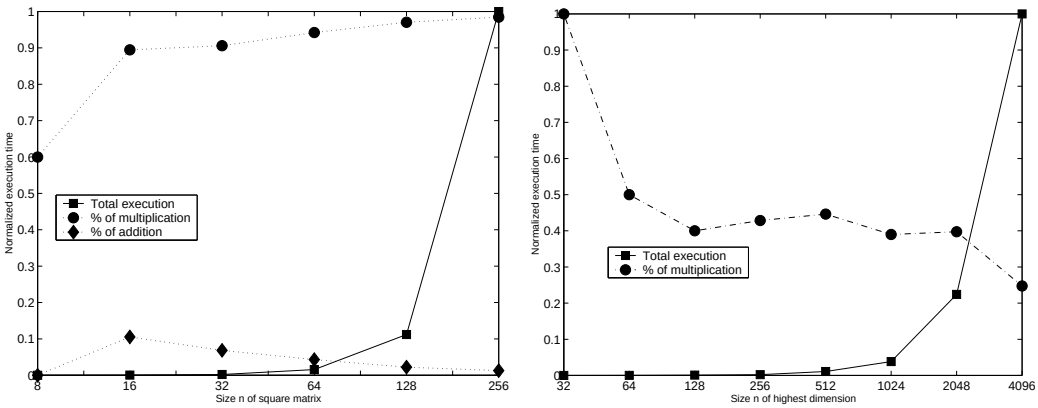
5.1. Metodologia

Utilizamos um desktop Intel Dual Core 1.6 GHz com 1 GB de memória RAM, executando o sistema operacional Windows XP com Service Pack 3 e Matlab R11.1. O Matlab foi configurado para executar em somente um processador, permitindo extrair resultados mais apurados. Mensuramos o tempo de execução para cada estudo de caso através do mecanismo interno do Matlab de *profiling*. A carga do processador foi mantida a mais constante possível entre as mensuráveis. As entradas foram geradas com a função interna *rand* do Matlab, evitando enviesar a execução e alterar o desempenho real devido a entradas pré-determinadas.

Codificamos as seguintes operações sobre matrizes: multiplicações e adições tradicional e tropical. Não utilizamos as funções internas de adição e multiplicação de matrizes do Matlab para evitar viés de execução e otimizações, permitindo uma comparação justa com as operações definidas sobre o semi-anel tropical. Nos experimentos consideramos o tamanho mínimo da matriz aquele para o qual o tempo de execução calculado pelo Matlab tenha sido maior que 0, com precisão de 0,016 segundos. Realizamos injeção de falhas através da troca de valores de um dos elementos de cada matriz computada durante as iterações dos algoritmos, sendo necessária a detecção e a correção desse elemento. Por exemplo, em Huffman para cada altura de árvore h sendo computada, são calculadas três matrizes, sendo inseridos, portanto, três falhas a cada iteração.

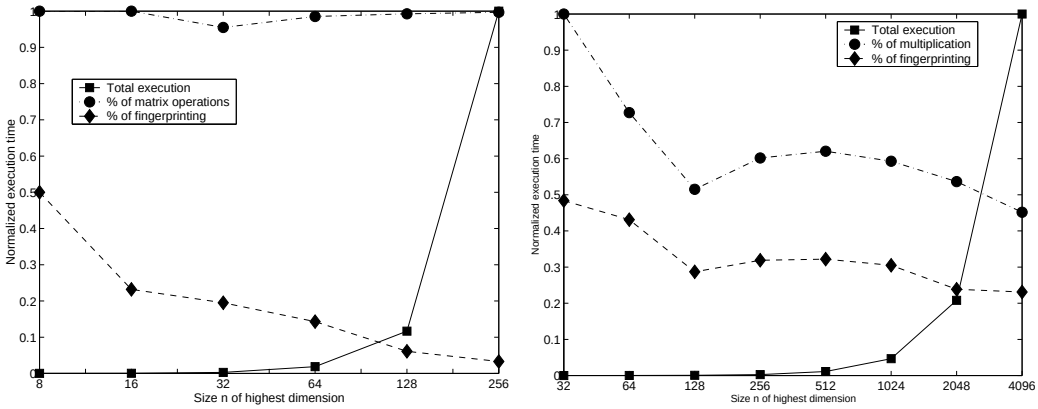
5.2. Resultados

Mensuramos a influência das operações sobre matrizes no tempo total de execução das aplicações, *i.e.* a porcentagem do tempo de execução exclusivamente gasto com a computação de adições e multiplicações de matrizes, tanto as tradicionais quanto as tropicais. Fez-se também a comparação do ganho em desempenho da nossa abordagem em relação à duplicação com comparação. As Figuras 1, 2 e 3 apresentam os resultados.



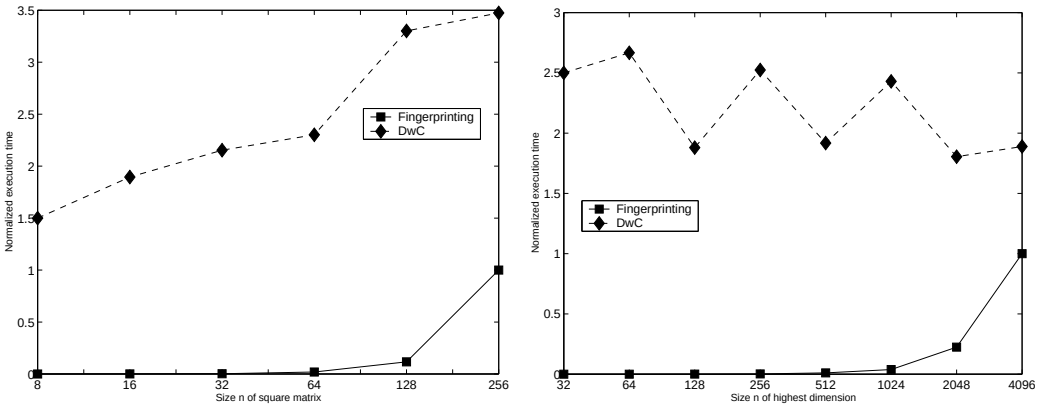
(a) Huffman (b) MDCT/IMDCT

Figura 1. Influência de operações sobre matrizes no tempo total de execução para (a) Huffman e (b) MDCT/IMDCT



(a) Huffman (b) MDCT/IMDCT

Figura 2. Acréscimo no tempo de execução devido a fingerprinting para (a) Huffman e (b) MDCT/IMDCT.



(a) Huffman (b) MDCT/IMDCT

Figura 3. Acréscimo no tempo de execução de duplicação com comparação (DwC) relativo a fingerprinting para (a) Huffman e (b) MDCT/IMDCT.

5.2.1. Influência das Operações em Matrizes sobre o Tempo Total de Execução

A Figura 1 apresenta a influência das operações de matrizes sobre o tempo total de execução das aplicações. A Figura 1(a) apresenta que, como esperado para o algoritmo de Huffman introduzido na seção 4.1, $\sim 100\%$ do tempo total de execução é dedicado à operações de adição e multiplicação de matrizes. Quando $n \geq 16$, a mulplicação de matrizes compreende mais de 90% do tempo total de execução; esses resultados mostram que Huffman se beneficiará muito da técnica de Freivalds. Quando $n = 8$, a porcentagem de multiplicação de matrizes é $\sim 60\%$. Nesse caso, a multiplicação de matrizes não é amortizada eficientemente em relação ao tempo necessário para calcular as matrizes S e A, como apresentado nas Equações 1, 2, e 3. Portanto, para o caso do algoritmo de Huffman (e qualquer outro descrito com o semi-anel tropical), a influência das operações sobre matrizes aumenta significativamente com o tamanho do espaço de problema.

Por outro lado, a Figura 1(b) mostra que para as aplicações MDCT/IMDCT, a influência de operações sobre matrizes no tempo total de execução diminui com o aumento do tamanho do espaço de problema. Quando esse aumenta, o tempo dedicado à computação do cosseno na Equação 8 é consideravelmente maior que a computação das Equações 6 e 7. Entretanto, a multiplicação de matrizes compreende $\sim 40\%$ do tempo total de execução quando $128 \leq n \leq 2048$, estando longe de ser negligenciável. Note que nas transformadas MDCT/IMDCT a única operação de matrizes existente é a multiplicação.

5.2.2. Diminuição do Desempenho devido ao Fingerprinting de Operações em Matrizes

Mensuramos o acréscimo incorrido ao proteger as operações sobre matrizes com fingerprinting, relacionando o tempo de execução gasto em fingerprinting com o tempo gasto em operações sobre matrizes. A Figura 2 apresenta os resultados. No caso de Huffman, implementamos a técnica de Freivalds usando a multiplicação e a adição tropical sem nenhuma alteração em Freivalds. Nesses experimentos, introduzimos um valor errôneo em cada matriz computada durante a execução de Huffman. Lembre que fingerprinting consiste em detectar a existência do valor errôneo e prosseguir com a sua correção. Essas operações são realizadas em tempo $O(n^2)$ e $O(1)$, respectivamente.

A Figura 2(a) apresenta a mensuração para Huffman. A linha pontilhada superior com círculos é a porcentagem do tempo total de execução (o qual é representado pela linha sólida com quadrados) gasto com operações sobre matrizes para uma dada dimensão n. A linha tracejada com diamantes representa a porcentagem do tempo total de execução gasto com fingerprinting. Esses resultados mostram que os custos de se realizar fingerprinting são amortizados com o aumento do tamanho da matriz, sendo de 5% quando $n = 256$ e $\sim 20\%$ quando $16 \leq n \leq 64$.

Esses resultados são muito promissores, dado que a indústria atualmente enfrenta o rápido acréscimo no volume de dados tratados em sistemas embarcados comuns e em eletrônica de consumo. Nesse sentido, pode-se esperar que as matrizes manipuladas possuirão tamanhos grandes, o que amortiza o custo de realizar fingerprinting na aplicação. Note que é possível realizar fingerprinting de maneira mais eficiente: o algoritmo de multiplicação de matrizes utilizado neste trabalho possui complexidade $O(n^3)$, mas, como

apresentando anteriormente, há uma alternativa de complexidade $O(n^{2,376})$.

Obtemos resultados similares para as transformadas MDCT/IMDCT, como apresentado na Figura 2(b). O percentual de acréscimo no tempo total de execução obtido foi de $\sim 30\%$ para $128 \leq n \leq 2048$. Como discutido anteriormente, o cálculo do cosseno demanda $\sim 60\%$ do tempo total de execução, o que faz que a técnica de fingerprinting tenha gasto de tempo percentual linear com o aumento de n. Isso é explicado pelo fato de as Equações 6 e 7 operarem sobre vetores. Assim, o custo de execução aumenta quadraticamente em relação a n, não cubicamente como em Huffman.

5.2.3. Desempenho de Fingerprinting e de Duplicação com Comparação

Mensuramos o acréscimo em tempo de execução de Duplicação com Comparação (DwC) relativamente a fingerprinting. Em ambas as técnicas, injetamos um elemento errôneo em cada matriz computada pelos algoritmos, sendo necessário corrigir esse elemento. Para DwC, calculou-se duas vezes a função, comparou-se seus valores de retorno e, se diferentes, computou-se essa função pela terceira vez. A Figura 3 apresenta os resultados.

Devido à alta influência das operações de matrizes no tempo total de execução de Huffman, DwC é muito inferior em relação a desempenho quando comparado a Freivalds, como pode ser observado na Figura 3(a). Quando $16 \leq n \leq 64$, o acréscimo no tempo de execução incorrido por DwC é $\sim 200\%$, podendo alcançar até 300% para $n \geq 128$ quando comparado proporcionalmente a fingerprinting.

Apesar do alto acréscimo no tempo total de execução ao se proteger as transformadas MDCT/IMDCT com DwC, esse permaneceu constante com o acréscimo em tamanho do problema, como pode ser visto na Figura 3(b). O acréscimo oscilou entre 200% e 250% do tempo total de execução relativamente a fingerprinting, o que nos mostra a superioridade da técnica de fingerprinting em termos de desempenho, dando suporte à abordagem proposta neste trabalho.

5.2.4. Análise do Tamanho Total de Programa

Mensuramos o acréscimo em tamanho de programa imposto pelo uso de matrizes e de Freivalds. Para tal, codificamos Huffman em C e o compilamos com o *gcc* com a opção de otimização de código *-O3*, tendo como alvo a arquitetura x86. Para mensurar o tamanho de cada função codificada, criamos um arquivo .c para cada função compondo cada algoritmo, e utilizamos a ferramenta do Unix *size* para contar o tamanho em bytes de cada função.

A Tabela 1 apresenta os resultados para Huffman. A biblioteca de matrizes causou o acréscimo mais significativo em tamanho de programa, tanto para a versão protegida quanto para a desprotegida (27% e 37%, respectivamente). A biblioteca de Freivalds incorre em aumento de somente 8% em tamanho de programa. Note que o tamanho em bytes de cada função das bibliotecas é constante, independente do tamanho da aplicação. Assim, se a aplicação fosse maior, os tamanhos das bibliotecas seriam eficientemente amortizados. Na realidade, a biblioteca de Freivalds foi codificada com somente 246 bytes, sendo perfeitamente implementável em qualquer sistema embarcado com fortes

Tabela 1. Impacto das Operações sobre Matrizes e da Técnica de Freivalds em Tamanho de Programa para Huffman

Função		
Nome	Tamanho (bytes)	% do tamanho total[a]
Main	132	4% (6%)
Huffman Protegido	1838	61% (n/a)
Huffman Desprotegido	1249	n/a (57%)
Tamanho Total de Biblioteca		
Nome	Tamanho (bytes)	% do tamanho total[a]
Biblioteca de Matrizes	806	27% (37%)
Biblioteca de Freivalds	248	8% (n/a)
Tamanho Total de Programa		
Versão	Tamanho (bytes)	
Huffman Protegido	3024	
Huffman Desprotegido	2187	

[a]% da versão protegida (% da versão desprotegida).

restrições de recursos, seja ele crítico ou não. O incremento em tamanho de programa obedece a mesma tendência (pois as bibliotecas possuem tamanho constante independentemente do algoritmo que as utilizem) para qualquer algoritmo e é corroborado pelos resultados das transformadas MDCT/IMDCT.

6. Conclusões e Trabalhos Futuros

Neste artigo, propusemos descrever programas completamente com matrizes e realizar a correção e a detecção de falhas transitórias utilizando a técnica de Freivalds. Utilizamos Huffman e as transformadas MDCT/IMDCT completamente escritos com operações sobre matrizes, mensuramos os tempos de execução e calculamos a porcentagem das operações sobre matrizes no tempo de execução. Mostramos que as operações sobre matrizes são um componente importante nesses estudos de caso (sendo a maioria em Huffman), demonstrando a importância e a adequação da abordagem proposta para a realização de tolerância a falhas em nível de sistema.

No caso de Huffman, a implementação baseada em matrizes não é a mais comum, a qual é baseada em programação procedural sobre árvores. Essa implementação com matrizes é orientada a fluxo de dados, permitindo a aplicação da técnica de Freivalds de maneira extremamente eficiente para a proteção contra falhas transitórias. Isso é claramente benéfico, dado que a proteção de aplicações orientadas a controle utilizando a análise estática e de criticidade das variáveis é consideravelmente menos eficiente que Freivalds, como discutido nos trabalhos relacionados.

Demonstramos que o acréscimo no tempo de execução incorrido devido a Freivalds decresce com o aumento do tamanho do problema para Huffman, sendo de 5% quando $n = 256$. No caso das transformadas MDCT/IMDCT, como as operações sobre

matrizes compreendem $\sim 40\%$ do tempo total de execução, o custo de Freivalds não é tão eficientemente amortizado quanto em Huffman, mas ainda é extremamente eficiente para detecção e correção, sendo muito melhor que técnicas clássicas de tolerância a falhas como Duplicação com Comparação, fato corroborado pelos experimentos realizados. Apresentou-se que fingerprinting baseado em Freivalds supera significativamente em desempenho a DwC, demonstrando a viabilidade de implantar fingerprinting em sistemas embarcados com fortes restrições de recursos. Além disso, demonstramos que a proteção por Freivalds e as operações sobre matrizes demandam um acréscimo ínfimo em memória de programa, sendo de 248 e 806 bytes, respectivamente, independente da aplicação. Esses resultados mostram que essas técnicas podem ser facilmente empregadas em sistemas embarcados com restrição de tamanho de memória de programa.

Como trabalhos futuros, estamos projetando um mecanismo de descrição de porções orientadas a controle usando-o para induzir um anel parcialmente ordenado sobre matrizes. Com essa construção algébrica, teremos tanto as porções orientadas a controle quanto as orientadas a dados protegidas contra falhas transitórias, bastando aplicar a eficiente técnica de Freivalds em ambos os casos. Por fim, estamos escolhendo algumas funções relevantes para o domínio de sistemas embarcados aeroespaciais, funções as quais usaremos para validar nossa proposta através de um estudo de caso de grande porte.

Referências

Argyrides, C., Lisboa, C. A. L., Pradhan, D. K., and Carro, L. (2009). A fast error correction technique for matrix multiplication algorithms. In *IOLTS '09: Proc. of the 15th IEEE International On-Line Testing Symposium*, pages 133–137, Los Alamitos, CA, USA. IEEE.

Atallah, M. J., Kosaraju, S. R., Larmore, L. L., Miller, G. L., and Teng, S.-H. (1989). Constructing trees in parallel. In *SPAA '89: Proc. of the 1st Ann. ACM Symposium on Parallel Algorithms and Architectures*, pages 421–431, New York, NY, USA. ACM.

Blum, L., Shube, M., and Smale, S. (1989). On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin (New Series) of the American Mathematical Society*, 21(1):1–46.

Blum, M. and Kanna, S. (1989). Designing programs that check their work. In *STOC '89: Proc. of the 21st Annual ACM Symposium on Theory of Computing*, pages 86–97, New York, NY, USA. ACM.

Chen, G. and Kandemir, M. (2005). Improving java virtual machine reliability for memory-constrained embedded systems. In *DAC '05: Proc. of the 42nd Annual Design Automation Conference*, pages 690–695, New York, NY, USA. ACM.

Cheng, M.-H. and Hsu, Y.-H. (2003). Fast imdct and mdct algorithms - a matrix approach. *IEEE Transactions on Signal Processing*, 51(1):221–229.

Cheynet, P., Nicolescu, B., Velazco, R., Rebaudengo, M., Sonza Reorda, M., and Violante, M. (2000). Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. *IEEE Transactions on Nuclear Science*, 47(6):2231–2236.

Coppersmith, D. and Winograd, S. (1987). Matrix multiplication via arithmetic progressions. In *STOC '87: Proc. of the 19th Annual ACM Symposium on Theory of Computing*, pages 1–6, New York, NY, USA. ACM.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. (2007). The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45.

Florio, V. D. and Blondia, C. (2008). A survey of linguistic structures for application-level fault tolerance. *ACM Computing Surveys*, 40(2):1–37.

Huang, K.-H. and Abraham, J. A. (1984). Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518–528.

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.

Krishnamurthi, S. and Reiss, S. P. (2003). Automated fault localization using potential invariants. In *AADEBUG '03: Proceedings of the International Workshop on Automated and Algorithmic Debugging*, pages 1–4.

Lisboa, C., Erigson, M., and Carro, L. (2007). System level approaches for mitigation of long duration transient faults in future technologies. In *ETS '07: Proc. of the 12th IEEE European Test Symposium*, pages 165–172, Los Alamitos, CA, USA. IEEE.

Lisboa, C. A., Grando, C. N., Moreira, A. F., and Carro, L. (2009). Building robust software using invariant checkers to detect run-time transient errors. In *DFR '09: Proceedings of the 1st Workshop on Design for Reliability*, pages 48–54.

Motwani, R. and Raghavan, P. (1995). *Randomized algorithms*. Cambridge University Press, New York, NY, USA.

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

Pattabiraman, K., Kalbarczyk, Z., and Iyer, R. K. (2007). Automated derivation of application-aware error detectors using static analysis. In *IOLTS '07: Proceedings of the 13th IEEE International On-Line Testing Symposium*, pages 211–216, Washington, DC, USA. IEEE.

Prata, P. and Silva, J. G. (1999). Algorithm based fault tolerance versus result-checking for matrix computations. In *FCTS '99: Proc. of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 4–11, Los Alamitos, CA, USA. IEEE.

Princen, J., Johnson, A., and Bradley, A. (1987). Subband/transform coding using filter bank designs based on time domain aliasing cancellation. In *ICASSP '87: Proc. of the Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 12, pages 2161–2164, Los Alamitos, CA, USA. IEEE.

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