Técnica de Proteção de Bytecodes para Processador Java em Tecnologia CMOS

Jardel Silveira (jardel@lesc.ufc.br), David Viana (david@lesc.ufc.br)
Helano Castro (helano@lesc.ufc.br), Alexandre Coelho (alexandre@lesc.ufc.br)
Jarbas Silveira (jarbas@lesc.ufc.br)
Universidade Federal do Ceará
LESC (http://www.lesc.ufc.br)
Fortaleza, CE, Brasil

Resumo

O soft core *JOP (*Java Optimized Processor*) para FPGAs (*Field Programmable Gate Array*) é uma implementação otimizada da máquina virtual Java em* hardware*, para aplicações de tempo real. No entanto, este processador não contempla em sua arquitetura técnicas de tolerância a falhas. O trabalho descrito neste artigo é parte de um esforço maior para tornar o processador JOP um processador tolerante a falhas. Neste artigo, apresentamos os resultados da aplicação de uma técnica de tolerância a falhas, proteção de memória através de ECC (*Error Correction Code*), no* soft core *JOP, que detecta e corrige erros na área destinada ao código da memória SRAM (*Static Random Access Memory*). A ocorrência da falha é percebida no nível sistêmico através de uma exceção, característica esta disponível na linguagem Java. Este artigo apresenta resultados inovadores na medida em que não existem registrados na literatura outro processador Java de tempo real e tolerante a falhas.*

1 Introdução

Os sistemas eletrônicos de tempo real embarcados em missões espaciais estão sujeitos aos elevados níveis de radiação existentes no espaço. Por isso, tais sistemas estão sujeitos a falhas causadas pelas colisões de partículas altamente energizadas contra estruturas nanométricas de silício presentes nos circuitos integrados modernos.

Particularmente para circuitos integrados contendo blocos de memória SRAM, a principal conseqüência destas colisões é a ocorrência de SEUs (*single event upsets*), que são a inversão permanente do valor de um bit. Existem fábricas especializadas na produção de circuitos integrados tolerantes à radiação. No entanto, devido ao pequeno volume de produção desses circuitos integrados, os mesmos têm preços elevados. Portanto, é muito importante garantir tolerância à radiação no âmbito do projeto do circuito integrado, independentemente do processo de fabricação, pois isto reduz os custos do sistema e permite utilizar os mais modernos processos de fabricação existentes.

Devido ao tamanho reduzido e a alta frequência de operação dos circuitos eletrônicos digitais modernos, os mesmos estão cada vez mais sucetíveis a ruído. Por isso, problemas antes somente encontrados em sistemas submetidos a radiações com intensidade similar à espacial, hoje são enfrentados em sistemas operando em ambiente terrestre. Aplicações seguras constituem outro exemplo em ambiente terrestre que requerem técnicas de tolerância a falhas, pois falhas em sistemas seguros podem ser exploradas por *crackers* para descobrir chaves secretas armazenadas na memória interna de um circuito integrado [13].

Diante desse cenário, percebe-se cada vez mais a necessidade da utilização de técnicas de tolerância a falhas não somente em sistemas embarcados em missões espaciais, mas também nos sistemas terrestres. Dentre alguns exemplos dessas aplicações terrestres podemos citar a indústria automobilística, bancos e outras aplicações cujos requisitos temporais e de alta disponibilidade são prioritários para o correto funcionamento do sistema.

Notavelmente, a linguagem de programação C é atualmente a mais utilizada para desenvolvimento de *software* para sistemas embarcados, tanto para o sistema operacional quanto para a aplicação. Isto pode ser facilmente demonstrado por uma análise dos compiladores comerciais disponíveis para os processadores modernos de sistemas embarcados.

Usualmente, o sistema operacional é responsável por funções de suporte a tempo real, gerenciamento de memória e comunicação inter-processos. Tais recursos são disponibilizados para a aplicação por meio de chamadas de sistema (*system calls*) [21] e de uma API (*Application Program Interface*).

O uso de uma linguagem de alto nível de abstração traz benefícios do ponto de vista do desenvolvimento do

sistema, tais como diminuir a probabilidade de erros de codificação e a redução do tempo de desenvolvimento de um sistema [2]. Java é uma linguagem de alto nível muito utilizada e com grande suporte para o desenvolvimento de sistemas *standalone*. Além do alto nível de abstração, a linguagem Java traz no seu núcleo (Máquina Virtual Java) recursos comumente implementados em nível de sistema operacional, tais como comunicação inter-processo e escalonamento de tarefas.

Em uma implementação tradicional de sistemas embarcados de tempo real, baseada em um processador de uso geral e um sistema operacional de tempo real, essas vantagens têm um custo elevado em termos de recursos computacionais, o que é incompatível com as severas restrições de recursos computacionais em sistemas embarcados. Esta incompatibilidade pode ser resolvida pelo uso de um processador específico para linguagem Java, como proposto por Schoeberl em [19] e vários outros[1][9][10][15]. No entanto, para nenhum destes processadores, esses autores discutem a garantia de funcionamento (*dependability*) do processador. Quando comparado com os demais processadores Java, o JOP [19] se destaca, por exemplo, em relação a sua característica de tempo real, porém também desconsiderando requisitos de garantia de funcionamento. Por isso, escolhemos o processador JOP como plataforma base para este trabalho.. Note que existem outros processadores de uso geral tolerantes a falhas [8][7][16][4][6][11], mas nenhum deles é um processador Java tolerante a falhas e de tempo real.

Neste trabalho, propomos uma técnica de tolerância a falhas para proteger a memória cache SRAM de código interna ao JOP contra SEUs. Não se pretende com a aplicação desta técnica esgotar a discussão sobre tolerância a falhas no JOP, no entanto, por proteger uma região extremamente crítica, no caso a memória de código, e que não é viável de ser protegida via técnicas de software, consideramos a mesma de extrema importância. Informações sobre o JOP necessárias para a compreensão do problema, bem como os detalhes desta e os resultados obtidos são apresentados nas seções que se seguem.

2 Processador JOP

Um processador Java é uma implementação da máquina virtual Java. Essa implementação não é necessariamente completa em *hardware*, pois uma Máquina Virtual Java contém funções complexas como, por exemplo, escalonamento, gerenciamento e intercomunicação de processos. O custo de implementar todos esses recursos em *hardware* pode tornar a implementação não viável. Portanto, o conceito de um processador Java difere de um processador comum, no qual apenas elementos de *hardware* estão envolvidos. Dessa forma, um processador Java é uma implementação baseada em *hardware* e, possivelmente, em algum *software*.

Para um processador Java de tempo real, sua característica de tempo real deve permear tanto o seu *software*, como o seu *hardware*.

O JOP (Java Optimized Processor) é uma implementação em *hardware* e *software* de uma máquina virtual Java de tempo real, baseada no perfil J2ME (*Java 2 Micro Edition*), CLDC (*Connected Limited Device Configuration*) e na especificação SCJ (*Safety Critical Java*). Este processador é implementado como *soft core* em FPGAs Xilinx ou Altera e, diferentemente da JVM (*Java Virtual Machine*), que é uma máquina CISC (*Complex Instruction Set Computer*)[14], o JOP é, internamente, uma máquina RISC (*Reduced Instruction Set Computer*) [14] e contém seu próprio conjunto de instruções.

2.1 Requisitos de tempo real

Aplicações de tempo real para o JOP são explicitamente separadas em duas partes: Fase de Inicialização e Fase de Missão. Na fase de inicialização são criados todos os objetos que serão usados durante toda a execução da aplicação e, portanto, áreas de memórias são alocadas e inicializadas. Nesta fase não existe garantia de tempo real. Na fase de missão, as *threads* são executadas concorrentemente de acordo com o algoritmo de escalonamento.

2.1.1 Análise de WCET no JOP

Por ter sido desenvolvida para ser usada em sistemas embarcados com aplicações de tempo real, a arquitetura do processador JOP permite calcular com facilidade o WCET (*Worst Case Execution Time*) de uma tarefa.

A máquina virtual Java do JOP implementa classes que permitem desenvolver aplicações de tempo real. Essas classes não são compatíveis com o padrão RTSJ (*Real Time Specification for Java*) [12], pois apenas um subconjunto deste padrão é implementado. Embora as áreas de código e de pilha do JOP utilizem memória de *cache*, o modelamento do comportamento desta no JOP é perfeitamente previsível no tempo. Pois, diferentemente do que acontece em outros processadores, no JOP não ocorrem "*cache misses*", ou seja, cada instrução solicitada pelo processador à *cache* estará sempre previamente carregada na *cache* [17].

2.1.2 Garantia de funcionamento

O JOP não implementa em sua arquitetura técnicas de garantia de funcionamento. Portanto, para aplicações de tempo real do tipo *hard*, ou seja, que envolvem risco para vidas humanas, o projetista do sistema deverá assegurar a garantia de funcionamento em nível sistêmico. No JOP, uma falha de *hardware*, por exemplo um *opcode* ilegal ou

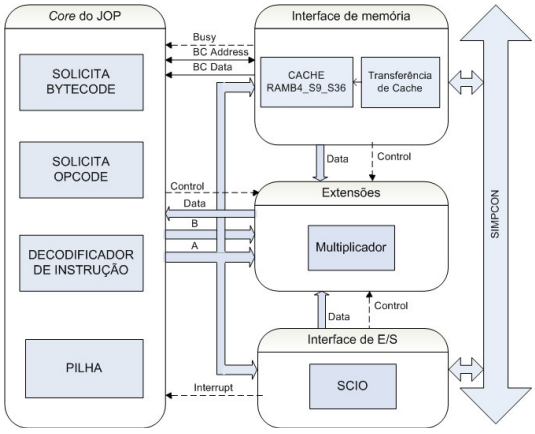

Figura 1. Diagrama de Blocos do JOP

um erro de paridade de memória, levará o sistema a um shutdown [18].

2.2 Arquitetura do JOP

O JOP é composto de quatro blocos principais (ver Figura 1): Interface de memória, *core do JOP*, interface de E/S (**scio**) e "extensões". O bloco de interface de memória comunica-se com os controladores de memórias externas através do barramento de comunicação **simpcon**. Os controladores de memória, por sua vez, comunicam-se, através dos pinos do processador com as memórias SRAM e *flash*. O bloco "*core* do JOP" é responsável por decodificar e executar os *bytecodes* fornecidos pela interface de memória e comandar as demais partes do processador. O bloco interface de E/S comunica-se com controladores de E/S, tais como porta USB e Serial RS232, através do barramento **simpcon**. Estes controladores de E/S, por sua vez, comunicam-se com os dispositivos do ambiente externo através dos pinos do processador. O bloco de "extensões" serve para agregar funções de co-processadores matemáticos sem realizar modificações no núcleo do processador.

2.3 Documentação e portabilidade

Além das características técnicas do JOP, descritas anteriormente, podemos destacar também a ampla documentação disponível, a sua portabilidade, pois já foi implementado em diversas placas de desenvolvimento de FPGA (Xilinx e Altera) disponíveis no mercado, e por último, mas não menos importante, a disponibilidade do código fonte do JOP para *download* pelo *site* `http://www.opencores.org` e licenciado sob a GPL (*Gnu Public License*) versão 3. Atualmente o JOP é utilizado em dois sistemas comerciais, tendo um deles a característica de tempo real, e em vários sistemas de pesquisa [19]. Portanto, o JOP se posiciona como uma excelente alternativa para plataforma base, de pesquisa e desenvolvimento, de novas técnicas de sistemas de tempo real.

3 JOP Tolerante a Falhas

Erros na memória de código de um sistema computacional são críticos por serem armazenados permanentemente e podem, portanto, causar sucessivos erros no processo de computação que fizer uso dos dados errôneos. Para detectar e corrigir erros na memória de dados, existem técnicas bastante efetivas implementadas em *software*. No entanto, as técnicas aplicadas no âmbito de *software* para detectar e corrigir erros na memória de código não são eficazes, pois o próprio *software* pode estar corrompido. Nesse sentido, propomos o uso de uma técnica no âmbito de *hardware* para detectar e corrigir erros nas memórias RAM interna (*cache*) ao processador JOP, de forma a aumentar a confiabilidade do sistema de *cache* de métodos descritos na seção anterior.

Durante a inicialização do processador JOP, todo o código é transferido da memória *flash* para a memória RAM. Ao fim da transferência de código para a RAM, o sistema de *cache* transferirá, sob demanda, métodos inteiros da memória externa para a memória de *cache* interna. Por último, após o método estar completamente carregado na memória interna, o *core* irá solicitar e executar os *bytecodes*, um a um.

A Figura 2 mostra o diagrama de blocos do JOP modificado. Quando comparado ao diagrama de blocos do JOP original (ver Figura 1), nota-se que três novos blocos foram adicionados ao bloco de interface de memória da arquitetura original do JOP. Estes blocos se referem à implementação da técnica de proteção de instruções:

1. Codificador de Hamming;
2. Detector e corretor de erros em bytecodes (Decodificador de Hamming);
3. Redundância de cache (RAMB4_S4_S18);

3.1 Técnica de proteção de instruções

Esta técnica detecta e corrige erros ocorridos nos *bytecodes*, desde o armazenamento destes na *cache* até o início da execução destes pelo *core* do JOP. O erro é detectado e corrigido imediatamente antes de o *core* iniciar a execução do *bytecode*. Portanto, essa é uma técnica de verificação de último instante (*last minute check*)[5].

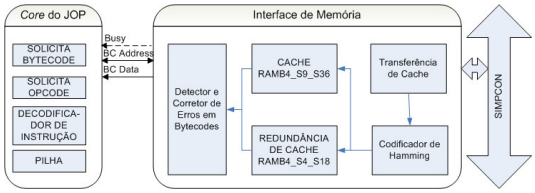

Figura 2. Diagrama de Blocos do JOP modificado

Simultaneamente à escrita de um *bytecode* (tamanho de 8 bits) na memória *cache*, 4 bits extras de redundância (bits de Hamming) são calculados e armazenados em uma memória *cache* de redundância (ver Figura 2), que se posiciona em paralelo com a *cache* original. Esses bits extras são calculados por um *core* escrito em RTL (*Register Transfer Level*), que implementa um codificador de Hamming.

Imediatamente após a memória *cache* fornecer um *bytecode* para o *core* do JOP, porém antes de ser executado, um *core* decodificador de Hamming lê os 4 bits armazenados na *cache* de redundância (Ver Figura 2). Com base nesses 12 (doze) bits (oito bits do *bytecode* mais quatro bits de Hamming), esse *core* irá verificar se houve alguma inversão de bit em algum dos bits do *bytecode*. Em caso afirmativo, isto significa que houve uma falha. Neste caso, o *core* decodificador de Hamming irá corrigir automaticamente o *bytecode*, desde que apenas um bit tenha sido invertido. Finalmente, o *bytecode* correto será entregue para a execução por parte do *core* do JOP.

3.2 Percepção da falha no âmbito sistêmico

Uma falha de *hardware* no JOP original, como por exemplo, um *opcode* ilegal, leva o sistema a um *shutdown* [18]. Neste trabalho, o processador foi modificado para que, na ocorrência de uma falha de *hardware*, uma exceção seja gerada.

4 Resultados

O JOP modificado pelo uso da técnica de proteção de instruções foi simulado utilizando a ferramenta NC-VHDL para avaliar se houve regressão de seu funcionamento. Para avaliar a eficácia da técnica foi desenvolvido um módulo injetor de falhas escrito em VHDL. Esse módulo gera aleatoriamente eventos do tipo SEU nos BYTECODES de um programa em execução pelo JOP, sendo que não deve existir mais que um dos bits invertidos em um mesmo BYTECODE. A técnica mostrou-se eficaz, tendo em vista que todos os bytecodes corrompidos com somente um bit invertido, foram automaticamente corrigidos. Para aqueles com dois bits invertidos, uma exceção foi gerada para ser tratada por rotina específica para este fim. Além da simulação, o JOP modificado foi embarcado na placa Spartan-3 starter kit board [22] da Digilent. Essa placa contém uma FPGA Spartan 2 XC3S200 e 1MB de memória SRAM. Neste caso, a injeção de falhas foi realizada através de mudanças do nível lógico dos pinos da FPGA. Dois tipos de falhas foram injetadas: stuck-at (1 ou 0) e inversão de bits aleatórios. Para todas as falhas inseridas, de um tipo ou de outro, houve detecção e correção automática pelo uso da técnica de proteção de instruções. Em termos numéricos, a possibilidade de detecção de falhas pode ser facilmente caculada com base no algoritimo de hamming aplicado a 8 bits de dados úteis e 4 bits de redundância [20].

Tabela 1. Tabela Comparativa entre o JOP Modificado e o JOP Original

	FPGA Slices	RAM (Kbits)	Freq (MHz)
JOP	3150	16	100
JOPFT	3201	24	100

A Tabela 1 compara a área de FPGA, em termos de *slices* e de memória RAM para o JOP original e o JOP modificado (JOPFT) pela técnica proposta.

5 Conclusão

De acordo com a pesquisa bibliográfica realizada, não foi encontrado nenhum processador Java que tenha simultaneamente garantia de tempo real e de funcionamento. Portanto, este trabalho é único no sentido de conceber, a partir do JOP, um processador Java de tempo real tolerante a falhas. Estas características são importantíssimas para sistemas embarcados de tempo real para aplicações que envolvem risco de vidas humanas.

A técnica proposta utiliza algoritmo de hamming implementado em hardware para aumentar a confiabilidade do processador Java. 4 bits de Hamming são armazenados em conjunto com cada bytecode Java (palavra de 8 bits) do processador JOP. Estes 4 bits são usados para detectar e corrigir erros na memória de código. Desta forma, agregamos ao processador JOP a capacidade de detectar e corrigir erros na memória de código. Isto permite manufaturar o JOPFT utilizando-se os mesmos processos de fabricação de silício, que são utilizados para fabricar chips comerciais, ao invés de usar processos de fabricação específicos para chips tolerantes a radiação. Logo, reduz-se drasticamente o custo por área de silício.

Finalmente é importante destacar que proteger a

memória de código do JOP é um passo importante para aumentar sua confiabilidade. Outras técnicas, como a técnica de votação de CRC de frames, anteriomente proposta pelos autores em [3] e também as técnicas propostas em [8] estão sendo aplicadas ao JOP e avaliadas em termos de confiabilidade, custo adicional em termos de área de silício, desempenho e implicações na características de tempo real do JOP, assim como foi feito para a técnica descrita neste trabalho. Portanto, como trabalho futuro, JOP modificado será testado com esta e outras técnicas de tolerância a falhas. Destacamos ainda que esse trabalho foi aprovado para ser manufaturado na tecnologia CMOS IBM 130 nm gratuitamente no service Mosis. Portanto, o JOP modificado será prototipado no processo IBM 130 nm, para então ser submetido a testes de funcionamento quando sob bombardeamento de partículas altamente energizadas. Estes testes permitirão avaliar a confiabilidade do processador e poderão ser realizados, por exemplo, no LIN - Laboratório de Instrumentação Nuclear da UFRJ.

Agradecimentos

Os autores gostariam de agradecer a Funcap pelo suporte financeiro através do seu programa de bolsas de mestrado (Processo BMD0008-00052.01.05). Agradecemos também o Ministério da Ciência e Tecnologia (MCT) do Brasil, a Sociedade Brasileira de Microeletrônica (SBMICRO), a Cadence e a Anacom que viabilizaram o programa universitário da Cadence no Brasil. Agradecer ainda a Xilinx pelo apoio fornecido através de seu programa universitário. Agradecer também o Mosis, IBM e ARM que viabilizarão a manufatura do circuito integrado fruto deste trabalho. E ainda o DETI, FCPC, LESC, Flextronics Institute of Technology (FIT) e Flextronics por apoiarem este trabalho. E por último, mas não menos importante, a Martin Schoeberl por distribuir livremente o código fonte do JOP sob a licença GPL.

Referências

[1] A. C. Beck and L. Carro. Low power java processor for embedded applications. In *Proceedings of the 12th IFIP International Conference on Very Large Scale Integration*, pages 213–228, December 2003.

[2] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages*. Addison-Wesley, 1991.

[3] H. Castro, A. A. Coelho, and R. J. Silveira. Fault-tolerance in fpga's through crc voting. In *SBCCI '08: Proceedings of the 21st Annual Symposium on Integrated Circuits And System Design*, pages 188–192, New York, NY, USA, 2008. ACM.

[4] M. A. Check and T. J. Slegel. Custom s/390 g5 and g6 microprocessors. *IBM J. RES. DEVELOP*, 43(5/6):671–680, SEPTEMBER/NOVEMBER 1999.

[5] H. de Sousa Castro. *Fault Tolerance Through Reconfigurability: Applications In Space Instrumentation*. Phd thesis, The University of Sussex, June 1992.

[6] Érika Cota, F. Lima, S. Rezgui, L. Carro, R. Velazco, M. Lubaszewski, and R. Reis. Synthesis of an 8051-like micro-controller tolerant to transient faults. *J. Electron. Test.*, 17(2):149–161, 2001.

[7] J. Gaisler. Concurrent error-detection and modular fault-tolerance in a 32-bit processing core for embedded space flight applications. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 128–130, June 1994.

[8] J. Gaisler. A portable and fault-tolerant microprocessor based on the sparc v8 architecture. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 409–415, Washington, DC, USA, 2002. IEEE Computer Society.

[9] T. R. Halfhill. Imsys hedges bets on Java. *Microprocessor Report*, pages 1–4, August 2000.

[10] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.

[11] C. Meinhardt, R. Reis, M. Violante, and M. Reorda. Recovery scheme for hardening system on programmable chips. In *Test Workshop, 2009. LATW '09. 10th Latin American*, pages 1–6, March 2009.

[12] P. Mikhalenko. Real-time java: An introduction, September 2008.

[13] C. R. Moratelli, E. Cota, and M. S. Lubaszewski. A cryptography core tolerant to dfa fault attacks. *Journal Integrated Circuits and Systems*, 2(1):14–21, 2007.

[14] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 3 edition, 2007.

[15] W. Puffitsch and M. Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 213–221, Vienna, Austria, September 2007. ACM Press.

[16] N. Quach. High availability and reliability in the itanium processor. *IEEE Micro*, 20(5):61–69, 2000.

[17] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. Phd thesis, Vienna University of Technology, 2005.

[18] M. Schoeberl. *Jop Reference Handbook*. 1 edition, 2007.

[19] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, doi:10.1016/j.sysarc.2007.06.001, 2008.

[20] M. L. Shooman. *Reliability of Computer Systems and Networks*. Wiley, 1 edition, 2002.

[21] A. S. Tanenbaum. *Modern Operating Systems*. Pearson Education, 3 edition, 2008.

[22] Xilinx, Inc. *Spartan-3 Starter Kit Board User Guide*, May 2005. v1.2.