UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Tiago de Albuquerque Reis

Suporte de Sistema Operacional Para Reconfiguração Dinâmica de Componentes de Hardware Para Sistemas Embarcados

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Antônio Augusto Medeiros Fröhlich, Dr. (Orientador)

Florianópolis, Março de 2010

Suporte de Sistema Operacional Para Reconfiguração Dinâmica de Componentes de Hardware Para Sistemas Embarcados

Tiago de Albuquerque Reis

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Sistemas Embarcados e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Mário Antônio Ribeiro Dantas, Dr. (Coordenador)

Banca Examinadora

Prof. Antônio Augusto Medeiros Fröhlich, Dr. (Orientador)

Prof. Flávio Rech Wagner, Dr. (UFRGS)

Prof. Mário Antônio Ribeiro Dantas, Dr. (UFSC)

Prof. Frank Augusto Siqueira, Dr. (UFSC)

"Somos adultos agora. E é a nossa vez de decidir o que isso significa."
Randall Munroe

Agradecimentos

Agradeço à minha família que sempre me apoiou e acreditou em mim, em especial à Thiely, minha esposa, namorada, amiga e companheira, à minha mãe Iara e meus avós Gladir e Nildo. Agradeço também a gurizada do apê, Cristiano, Mateus e Rodolfo, que foram a minha segunda família e cujo companheirismo e parceiria foram fundamentais para superar os desafios nessa jornada em pagos distantes.

Agradeço ao LISHA e seu integrantes (Carlinha, Hugo, Roberto, Giovani, Mateus, Alexandre, Rodrigo, Tiago e João) por proporcionar um ótimo ambiente para o desenvolvimento do meu trabalho e, em especial, ao Prof. Guto que, apesar dos desentendimentos e da minha teimosia, não desistiu de mim e me conduziu com muita transparência e dedicação. Agradeço também aos membros da banca, Professores Flávio Wagner, Mário Dantas e Frank Siqueira pelas contribuições ao meu trabalho. Também à Verinha que sempre me atendeu com simpatia e paciência. Um agradecimento especial ao CNPq, que me permitiu desenvolver esse trabalho com dedicação.

Agradeço também a todos os meus amigos que de uma forma ou de outra me ajudaram a chegar até aqui.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

ACPI *Advanced Configuration and Power Interface*

ADESD *Application-driven Embedded System Design*

ELUS *EPOS Live Update System*

EPOS *Embedded Parallel Operating System*

ETP *ELUS Transport Protocol*

FPGA *Field-Programmable Gate Array*

HDL *Hardware Description Language*

ICAP *Internal Configuration Access Port*

LUT *Lookup Table*

NoC *Network-on-a-Chip*

OS4RS *Operating System for Reconfigurable Systems*

RTOS *Real Time Operating System*

SAL *Software Abstraction Layer*

SDR *Software-defined Radio*

SoC *System-on-a-Chip*

VHDL *VHSIC Hardware Description Language*

VHSIC *Very-High-Speed Integrated Circuit*

Resumo

As vantagens que podem ser obtidas com a utilização de computação reconfigurável são largamente conhecidas. No escopo de sistemas embarcados, a utilização dessa tecnologia pode trazer boas respostas para duas questões recorrentes na área: desempenho e consumo de energia. Contudo, sua utilização ainda está longe das prateleiras, limitando-se a incontáveis projetos de pesquisa. Isso se dá, em parte, pelo aumento da complexidade de desenvolvimento de tais sistemas. Uma maneira de diminuir a complexidade de desenvolvimento de sistemas embarcados é através de sistemas operacionais, que provêem abstrações, tanto de *hardware* quanto de *software*, para o desenvolvimento da aplicação e ainda permitem alcançar um nível maior de portabilidade da solução. Estendendo-se sistemas operacionais para dar suporte a *hardware* reconfigurável, provendo esses dispositivos como uma abstração de alto nível, pode-se diminuir a complexidade de desenvolvimento de sistemas embarcados reconfiguráveis.

Com esse intuito, foi desenvolvido um suporte à reconfiguração de *hardware* no contexto do sistema operacional EPOS. Esse suporte estende o gerenciador de energia do EPOS, que permite a propagação de comandos para trocas de modo de operação pelos diversos componentes do sistema, para realizar as operações de hibernar e acordar o sistema como um todo. Essas operações são necessárias devido à utilização do método de diferenciação para gerar *bitstreams* parciais, o que produz resultados imprevisíveis que podem afetar o processador *softcore* sobre o qual o sistema operacional executa. Entretanto, tal abordagem possui vantagens que não podem ser ignoradas no contexto de sistemas embarcados, como simplificação significativa no projeto do *hardware* e melhor portabilidade da implementação do *hardware* entre diferentes modelos de FPGAs. A implementação presente neste trabalho mostra que pode-se chegar a uma plataforma arquiteturalmente independente, utilizando reconfiguração baseada em diferença, e sua viabilidade de utilização em sistemas embarcados reconfiguráveis.

Palavras chave: Sistemas Embarcados, Computação Reconfigurável, Reconfiguração Dinâmica.

Abstract

The advantages obtained by using reconfigurable computing are largely known. This technology can provide good answers for two recurring problems for embedded systems: performance and energy consumption. However, its utilization is happening almost exclusively in research projects, far away from the shelves. This is partially due to the increase of complexity for developing such systems. One way to diminish the development complexity of embedded systems is through operating systems, that provide software and hardware abstractions to the application development and improve the solution's portability. By extending operating systems to support reconfigurable hardware, providing it as an high level system abstraction, the complexity of developing reconfigurable embedded systems can be reduced.

With this intent, a reconfigurable hardware support was developed inside the EPOS operating system. This support extends EPOS' power manager, which propagates commands for operation mode changes through the system components, to allow the hibernation and wake up of the system. This operations are necessary due to the utilization of difference-based partial reconfiguration, which produces unpredictable bitstream results that may affect the softcore processor that runs the operating system. Nevertheless, this approach presents some advantages that cannot be ignored when developing reconfigurable embedded systems, such as simplifying the hardware design and improving its portability between different FPGA models. The implementation in this work shows that we can reach an architecturally independent platform, using difference-based partial reconfiguration, and its feasibility in reconfigurable embedded systems.

Keywords: Embedded Systems, Reconfigurable Computing, Dynamic Reconfiguration.

Capítulo 1

Introdução

Hardware reconfigurável, inicialmente visando prototipação rápida, rapidamente transformou o mercado na última década. Esses dispositivos permitem configuração e, em alguns casos, reconfiguração pós-fabricação, provendo assim um meio flexível para a implementação de uma variedade de circuitos sobre um *hardware*-base [Garcia et al. 2006]. Vários dispositivos eletrônicos podem se beneficiar com a possibilidade de atualizações em campo devido à evolução de padrões e necessidades em termos de performance. Multimídia e telecomunicações são um exemplo de ambas necessidades, pois atualizações somente de *software* podem não ser suficientes para que um mesmo *hardware* realize novas funções definidas por novos padrões, e a utilização de módulos para uma computação específica permite acelerar a execução de tarefas computacionalmente intensas [Durbano et al. 2004].

Enquanto isso, a comunidade acadêmica procura tirar proveito dessa tecnologia para criar sistemas camaleões, que se auto reconfiguram sob demanda, alterando sua estrutura e comportamento para se adaptar a mudanças no ambiente. Entre os dispositivos que atualmente utilizam essa tecnologia, podemos citar aparelhos pessoais multi-função, *switches* adaptativos, rádios cognitivos e um grande número de sistemas dedicados.

Do ponto de vista do *hardware*, módulos podem ser guardados em uma memória não volátil até que sejam necessários, devido a dispositivos reconfiguráveis modernos darem suporte à reconfiguração parcial e dinâmica. Essa reconfiguração se dá pela modificação da memória de configuração do dispositivo, o que pode ser feito parcialmente, para alterar apenas uma parte da funcionalidade. Essa reconfiguração parcial ainda pode ser dinâmica, permitindo que a alteração de uma parte da lógica ocorra enquanto o restante continua funcionando. Isso é projetado com o auxílio de ferramentas sofisticadas que mapeiam componentes lógicos para as estruturas programáveis da malha reconfigurável [Raghavan e Sutton 2002].

Entretanto, do ponto de vista do *software*, essa reconfiguração dinâmica do *hardware* muitas vezes fica sob a responsabilidade da aplicação [Gonzalez, Aguayo e Lopez-Buedo 2007] [Ullmann et al. 2004]. O suporte em sistema operacional existente é bastante limitado e foca principalmente na substituição de componentes, sem se preocupar em oferecer uma interface alto nível para o desenvolvedor [Majer et al. 2007] [Deng et al. 2005]. Já os sistemas que se provêem facilidades para o desenvolve-

dor acrescem algum tipo de sobrecarga ao sistema, seja ela de consumo de memória [Nollet et al. 2003], utilização de *hardware* [Walder e Platzner 2003] ou latência de reconfiguração [Santambrogio, Rana e Sciuto 2008], ou ainda diminuição de desempenho [Williams e Bergmann 2004].

Dentro desse contexto, esta dissertação propõe uma estratégia no nível do sistema operacional para dar suporte à reconfiguração de *hardware*. Essa estratégia se baseia em dois conceitos principais: objetos de *hardware* que podem ser criados e destruídos dinamicamente; e um mecanismo que, de forma consistente, salva e recupera o estado global do sistema para permitir a manipulação dos objetos de *hardware* de forma segura. Essa estratégia foi implementada utilizando mecanismos previamente concebidos no contexto do sistema operacional EPOS (*Embedded Parallel Operating System*) que permite, de forma eficiente, propagar mensagens de mudança de modo de operação pelo sistema.

A necessidade de salvar e recuperar o estado do sistema vem do fato deste trabalho focar em reconfiguração parcial baseada em diferença, uma alternativa pouco explorada no desenvolvimento de sistemas reconfiguráveis. Ao contrário da reconfiguração parcial baseada em módulos, baseando-se em diferença, a etapa de projeto de *hardware* é bastante simplificada, com a desvantagem de uma pior previsibilidade. Isso porque, ao invés de contar com módulos previamente projetados para se encaixarem em um sistema com lacunas pré-definidas, reconfiguração baseada em diferença simplesmente obtém a diferença entre duas *bitstreams* (arquivo binário utilizado para configurar o comportamento de um FPGA) e aplica essa diferença para realizar a reconfiguração. Mas, devido a essa diferenciação ocorrer após etapas de otimização na síntese desse *hardware*, uma pequena diferença na lógica pode significar uma imensa diferença na lógica otimizada, o que não pode ser previsto.

Como essas alterações podem ocorrer em qualquer parte do *hardware* reconfigurável, inclusive aonde está o processador, nenhum *software* pode estar sendo executado no momento da reconfiguração. Pode-se argumentar que, devido a essa necessidade, não se trata de uma reconfiguração e sim de uma reinicialização. Apesar dessa ser uma maneira válida de interpretar essa estratégia, o fato de o *hardware* mudar de uma configuração para outra de maneira transparente para aplicação, permite que consideremos a operação como um todo uma reconfiguração de *hardware*.

1.1 Objetivos

O principal objetivo dessa dissertação é o desenvolvimento de um suporte à reconfiguração de *hardware* no sistema operacional EPOS, para permitir que uma aplicação possa controlar reconfigurações de *hardware*

através de uma abstração de alto nível.

Partindo deste objetivo principal, os seguintes objetivos específicos são definidos:

- Estudar as técnicas utilizadas para realizar reconfigurações de *hardware*.
- Fazer um levantamento de abordagens para o suporte em sistema operacional para reconfiguração de *hardware* em sistemas embarcados.
- Estudar o funcionamento do gerenciador de energia do EPOS, base para o desenvolvimento do mecanismo de reconfiguração.
- Modelar e implementar a infraestrutura de reconfiguração, tendo como plataforma-alvo o processador Plasma.
- Verificar a necessidade de modificações no Plasma para melhor adaptá-lo ao sistema reconfigurável final.
- Realizar testes para avalizar a implementação e sua adequação no contexto de sistemas embarcados.
- Analisar os resultados obtidos e compará-los com os trabalhos relacionados.

1.2 Organização do Texto

O restante do texto está organizado da seguinte maneira: O Capitulo 2 faz uma revisão de conceitos básicos necessários para o entendimento do restante do texto. O Capítulo 3 apresenta abordagens de suporte em sistema operacional para reconfiguração de *hardware*. No Capítulo 4 é apresentado o suporte proposto juntamente com a descrição de sua implementação. No Capítulo 5 a implementação é avaliada e os resultados obtidos são apresentados. O Capítulo 6 apresenta conclusões e trabalhos futuros relativos à dissertação.

Capítulo 2

Conceitos Básicos

Este capítulo apresenta uma revisão de conceitos e tecnologias sobre computação reconfigurável. Inicialmente apresenta-se uma introdução a este paradigma e, em seguida, apresenta-se FPGAs e sua utilização no contexto da computação reconfigurável. Na seqüência, o sistema operacional EPOS é introduzido, ressaltando suas principais características. Por fim, alguns processadores *softcore* são apresentados.

2.1 Computação Reconfigurável

Computação reconfigurável é um paradigma de computação que procura unir o alto desempenho do *hardware* com a flexibilidade do *software* utilizando *hardware* reconfigurável [Todman et al. 2005]. Essa abordagem tem como objetivo preencher a lacuna entre o *hardware* e o *software*, alcançando assim um desempenho maior que o *software* enquanto mantém um nível de flexibilidade maior que o *hardware* [Compton e Hauck 2002]. Na computação de alto desempenho, a utilização de computação reconfigurável é vantajosa pela possibilidade de explorar múltiplos tipos e níveis de paralelismo [Peterson e Smith 2001].

O conceito de computação reconfigurável foi proposto na década de 1960 por Gerald Estrin, cuja idéia era ter um arranjo de processadores reconfiguráveis controlados por um processador tradicional. Nessa arquitetura, o processador configuraria o *hardware* reconfigurável para executar uma determinada tarefa e, assim que esse terminasse o processamento, o reconfiguraria para outra tarefa [Estrin et al. 1963].

Várias tentativas de se criar computadores reconfiguráveis apareceram nas décadas de 1980 e 1990, mas não obtiveram sucesso por limitações de tecnologia. Até que em 1991 surgiu o primeiro computador reconfigurável, o Algotronix CHS2X4 [Kean 2007]. Processadores reconfiguráveis somente se firmaram comercialmente com os FPGAs, inventado em 1984 por Ross Freeman.

No contexto de sistemas embarcados, com a crescente pressão para reduzir custos e *time-to-market*, as restrições de projeto desse tipo de sistema representa um sério desafio para seus desenvolvedores. Dispositivos reconfiguráveis podem prover uma plataforma eficiente e flexível para satisfazer as necessidades de área, performance, custo e consumo de energia de sistemas embarcados [Garcia et al. 2006].

As necessidades em termos de performance e consumo de energia,

levam desenvolvedores a usar componentes de *hardware* dedicados para realizar computações específicas, o que aumenta o paralelismo e evita sobrecargas impostas pela busca, decodificação e execução de instruções em *software* [Garcia et al. 2006]. Várias aplicações de sistemas embarcados possuem computações repetitivas que, se implementadas em *hardware*, executam em um fração do tempo necessário para executar a mesma em *software* [Lu et al. 1999, Kuzmanov, Gaydadjiev e Vassiliadis 2004]. Dentre essas aplicações podemos ressaltar multimídia, criptografia, comunicação sem fio, entre outras.

2.1.1 FPGA

Field-Programmable Gate Array é um dispositivo semi-condutor que possui blocos lógicos e interconexões programáveis. Esse dispositivo permite a implementação de circuitos lógicos relativamente grandes que consistem de um arranjo desses blocos lógicos e interconexões, contidos em um único circuito integrado. Assim, ele pode ser utilizado para computação híbrida [Mangione-Smith e Hutchings 1997], onde parte da execução se dá em um processador tradicional controlado por *software* e outra parte em um processador programado exclusivamente para realizar determinada tarefa.

Um FPGA é composto de três componentes básicos: blocos lógicos, chaves de interconexão e blocos de entrada e saída. Sua estrutura pode ser vista na figura 2.1, com exceção dos blocos de entrada e saída que se localizam nas extremidades da malha.

Os blocos lógicos são formados por uma tabela de busca, também chamada LUT (*Lookup Table*), um multiplexador e um *flip-flop*. Um exemplo de bloco lógico, como mostrado na Figura 2.2, possui quatro entradas, duas saídas e um sinal de relógio. A tabela de busca é utilizada para implementar a função lógica do bloco, visto que uma tabela de busca com *n bits* pode codificar uma função Booleana de *n* entradas através de tabela verdade.

A definição do comportamento de um FPGA, assim como qualquer outro dispositivo de *hardware*, pode ser feita através de uma linguagem de descrição de *hardware* (*Hardware Description Language* - HDL). O circuito projetado em HDL é compilado em um formato intermediário independente de plataforma (*netlist*), utilizando alguma ferramenta de automação de *design* eletrônico, que então será transformada em um arquivo binário que configura o FPGA (*bitstream*), utilizando ferramentas disponibilizadas pelos fabricantes do dispositivo.

Dentre essas linguagens, VHDL é atualmente, juntamente com Verilog, a linguagem de descrição de *hardware* mais utilizada. Esta lingua-

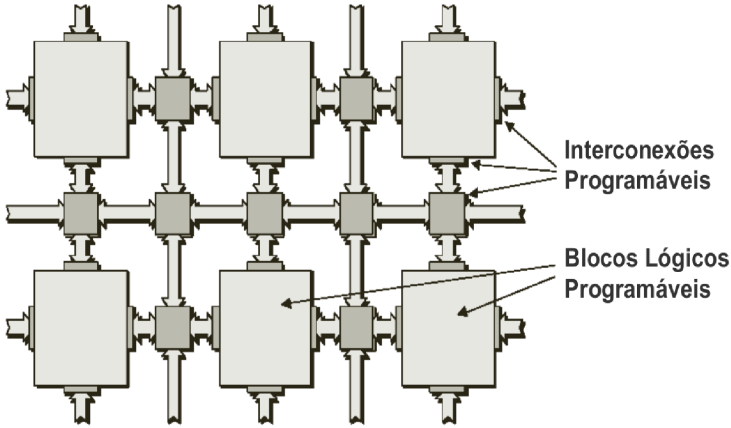

Figura 2.1: Arquitetura de FPGA genérica [Maxfield 2004]

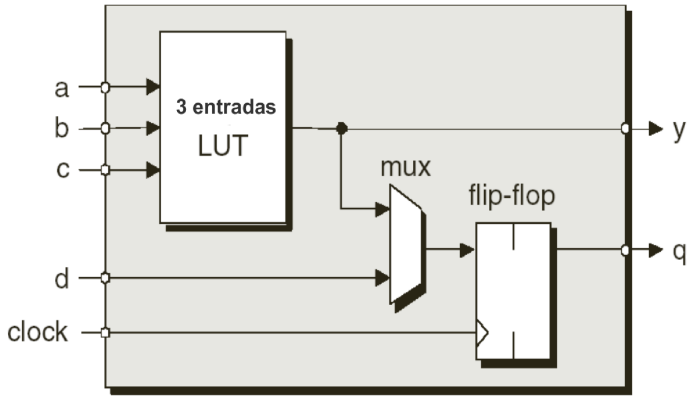

Figura 2.2: Elementos de um bloco lógico programável [Maxfield 2004]

gem surgiu na década de 1980 como linguagem de descrição de *hardware* VHSIC (*Very High Speed Integrated Circuits*) e em 1987 foi aprovado como padrão da IEEE (*Institute of Electrical and Electronic Engineering*) [Marra et al. 2001].

Do ponto de vista do desenvolvedor de *hardware*, VHDL é de grande

utilidade, pois permite definir o comportamento do *hardware* de maneira parecida com linguagens de alto nível e permite também definir portas de entrada e saída do componente de maneira intuitiva.

2.1.2 Reconfiguração Parcial

Reconfiguração parcial é o processo que procura otimizar o tempo de reconfiguração, modificando apenas alguns blocos lógicos do FPGA e, portanto, necessita que ele já esteja configurado. Pode ser estático, necessitando que o sistema pare durante a reconfiguração, ou dinâmico, que permite que o restante do sistema continue ativo. Exitem duas metodologias para geração de *bitstreams* parciais: baseada em módulos e em diferença.

Reconfiguração parcial baseada em módulos permite a utilização de abordagens análogas às de computadores convencionais devido à sua estruturação modular. Assim, como placas se encaixam para formar um sistema computacional convencional, os diversos módulos se encaixam logicamente dentro do FPGA. Isso se dá através da pré-definição de regiões reconfiguráveis que podem ser estruturadas em barramentos ou redes de interconexão.

Baseando-se nessa idéia, algumas arquiteturas, como a Erlangen Slot Machine [Majer et al. 2007], propuseram uma organização do espaço e troca de módulos análogas à uma placa-mãe de um computador pessoal. Esse tipo de organização torna a utilização de um conceito complicado, como atualização parcial de *hardware*, em algo intuitivo.

Tais sistemas têm a vantagem de controlar a inserção, remoção e substituição de módulos via *software* e de forma simples: apontando para o local aonde a *bitstream* parcial está, mecanismos se encarregam de encontrar o local mais apropriado para colocá-la [Gonzalez, Aguayo e Lopez-Buedo 2007, Ullmann et al. 2004, Majer et al. 2007].

Essa simplicidade de entendimento e utilização tem seu preço. Para utilizar módulos parciais o projeto precisa de uma etapa de planejamento da área (*floor-planning*), que especifica fisicamente o lugar exato dentro do FPGA em que um módulo pode ser encaixado. Isso faz com que o projeto fique focado em apenas um modelo de FPGA, comprometendo totalmente a portabilidade da solução. Essa especificação manual também tende a resultados não otimizados além de ser uma operação tediosa, mas para essa etapa já existem ferramentas que encontram o melhor lugar para as regiões reconfiguráveis automaticamente [Carver et al. 2008].

Além disso, o módulo precisa ser ligado a um barramento ou rede de interconexão que, por não estarem em uma área reconfigurável, são estáticos, limitando o número máximo de módulos que podem ser inseridos no sistema. Isso sem contar que esse barramento (ou rede) divide o

mesmo espaço do FPGA com os módulos que realizam as tarefas, reduzindo o número de blocos lógicos que podem ser gastos com o que realmente importa: a implementação das funcionalidades necessárias para a aplicação. É importante também ressaltar que essa especificação física de regiões reconfiguráveis pré-define o tamanho do módulo, o que pode causar fragmentação do espaço [Steiger, Walder e Platzner 2004] (a diferença entre o tamanho da região e o tamanho do módulo fica inutilizável), ou ainda pior, um módulo não previsto no início do projeto pode não caber.

A outra maneira de se obter *bitstreams* parciais é através da diferença entre *bitstreams*, que tem seu uso recomendado apenas para alterações pontuais, como mudança na equação de uma ou mais LUTs, conteúdo de *Block* RAM ou padrão de entrada e saída [Xilinx 2006]. Isso porque a maneira como o resultado desta diferenciação afetará o sistema é imprevisível. Essa imprevisibilidade está associada ao fato de que a diferenciação é feita somente após a síntese do *hardware* que, por sua vez, passa por algoritmos de otimização. Assim, uma pequena alteração no *design* pode fazer com que o sistema como um todo seja otimizado de forma diferente e, conseqüentemente, a diferença entre os *designs* poderá ser desproporcional ao tamanho da alteração.

Apesar de ser indicada apenas para pequenas mudanças, esse método pode ser usado para reconfigurar maiores quantidades de lógica. E nesse sentido, possui a vantagem de não necessitar de um planejamento prévio (*floor-planning*) das regiões reconfiguráveis, bastando apenas modificar o projeto original e obter a diferença. Uma importante conseqüência disso é a portabilidade, pois não fica-se amarrado a um modelo de FPGA pela definição física de regiões reconfiguráveis. Em contrapartida, não se pode garantir que as *bitstreams* parciais sejam muito menores que a *bitstream* completa, pelo mesmo motivo da imprevisibilidade mencionado anteriormente.

Usando reconfiguração parcial baseada em módulos, sabemos exatamente onde estão os módulos e as regiões nas quais eles podem ser encaixados. Assim, o gerenciamento deles pode facilmente ficar a cargo do *software* que executa no processador (Figura 2.3). Com reconfiguração parcial baseada em diferença, as mudanças imprevisíveis no *hardware* podem acontecer até mesmo dentro da área do processador, impossibilitando que algum *software* continue executando durante uma reconfiguração. Um exemplo disso pode ser a configuração da Figura 2.4a se tornar a da Figura 2.4b. Note que a localização do processador dentro do FPGA pode, inclusive, ser alterada para melhor acomodar o sistema dentro do FPGA. Isso é uma decisão tomada pelo sintetizador do *hardware*.

Talvez por isso que os próprios fabricantes não recomendem seu uso

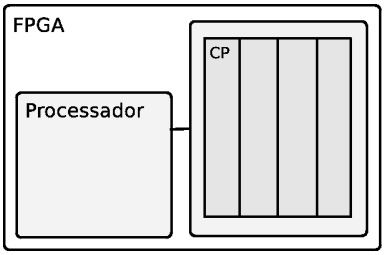

Figura 2.3: Reconfiguração baseada em módulos

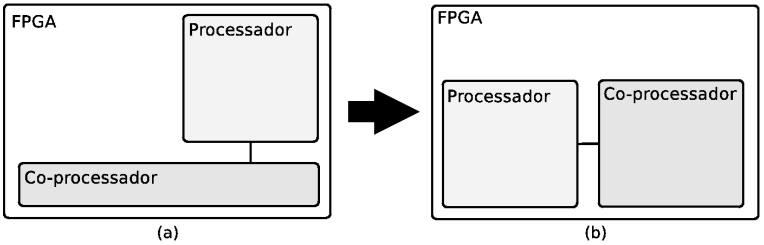

Figura 2.4: Reconfiguração baseada em diferença

para alterações não pontuais. Mas um mecanismo que permita pausar a execução de um programa e, após a reconfiguração, retomá-la, adaptando-se às mudanças no *hardware*, pode permitir que um sistema se beneficie das vantagens já citadas de se gerar *bitstreams* parciais através de diferenciação.

2.2 EPOS

Como base para a implementação do suporte para reconfiguração foi utilizado o sistema operacional *Embedded Parallel Operating System*. EPOS é indicado para tal projeto por ser um sistema operacional para sistemas embarcados sofisticado e de baixo consumo de memória, com suporte para diversas arquiteturas de *hardware* que variam desde simples microcontroladores de 8 *bits* até complexas arquiteturas como IA32 [Marcondes et al. 2006].

Projeto de Sistemas Embarcados Dirigidos à Aplicação (ADESD - *Application-driven Embedded System Design*) propõe a analise do domínio do problema, identificando as entidades significativas, e organizando-

as em famílias de abstrações (Figura 2.5). Interfaces infladas criam uma visão de cada família enquanto adaptadores de cenário adequam abstrações a dados cenários, simplificando a programação, pois só existe uma interface por família e o membro que será utilizado só será escolhido durante a geração do sistema. Seguindo essa metodologia foi concebido o *Embedded Parallel Operating System* (EPOS) [Fröhlich 2001].

EPOS é um *framework* para geração de sistemas de suporte a aplicações dedicadas, permitindo o desenvolvimento de aplicações independentes de plataforma [Wanner e Fröhlich 2008]. Sendo gerados após um processo de engenharia de domínio, os componentes formam a base do sistema, que é composto somente pela aplicação e seu suporte necessário tanto em *software* quanto em *hardware*. Essa engenharia de domínio consiste em um desenvolvimento sistemático de um modelo de domínio e sua implementação [Marcondes et al. 2006].

Esse suporte vem das diversas variações e semelhanças identificadas no domínio de sistemas operacionais para sistemas embarcados: políticas de escalonamento, sincronização, temporização, gerência de memória, tratamento de interrupções e tratamento de entrada e saída [Marcondes et al. 2006]. Como essas características são configuráveis, um certo nível de portabilidade é alcançado, permitindo uma melhor adaptação para diferentes tipos de *hardware* e aplicações [Marcondes et al. 2006]. Assim, EPOS procura ser o mais completo possível, considerando-se o domínio de sistemas operacionais, provendo essas funcionalidades até mesmo para computadores embarcados muito pequenos.

Para fazer a interface com o *hardware*, EPOS usa Mediadores de *Hardware* [Polpeta e Fröhlich 2004]. Estes artefatos sustentam um contrato de interface entre componentes de *hardware* e *software* mas, diferentemente de camadas de abstração de *hardware* comuns, nenhum código desnecessário é gerado. São na sua maior parte meta-programados, portanto dissolvem-se dentro dos componentes o que resulta em sobrecusto nulo. Mediadores exportam as funcionalidades necessárias para as abstrações do sistema de nível mais alto, através de uma interface *software/hardware* (Figura 2.6). Essas abstrações representam os serviços padrão de sistemas operacionais citados anteriormente.

2.2.1 Gerência de Energia no EPOS

Sistemas embarcados, em muitos casos, são alimentados por baterias e o tempo de vida dessas baterias é um fator determinante para disponibilidade do sistema [Wiedenhoft, Jr e Fröhlich]. Para lidar eficientemente com gerência de energia em sistemas com limitações de processamento e memória, um gerente hierárquico foi proposto [Júnior 2007]. Essa abor-

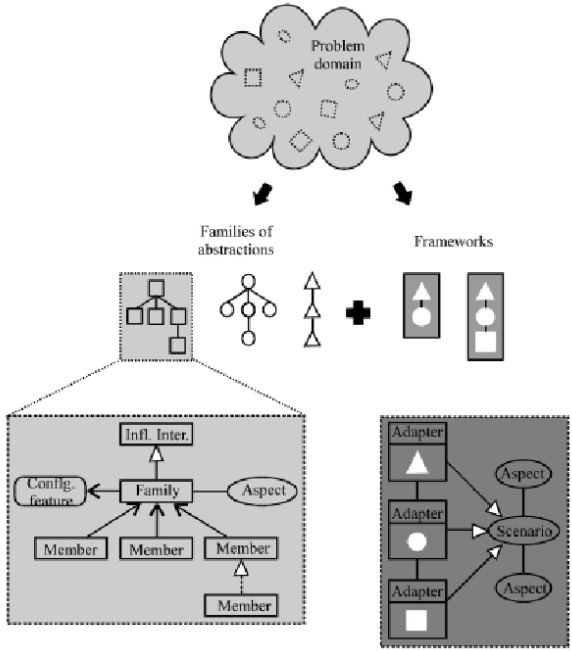

Figura 2.5: Decomposição de domínio

dagem propõe que a aplicação, em tempo de execução, se responsabilize pelas trocas de modos de operação em componentes do sistema. Isso porque a aplicação é centro de um sistema embarcado e, portanto, é quem melhor sabe como o sistema deve se comportar. Abordagens para gerência de energia como ACPI, em que o sistema operacional atua ativamente, não são adequadas para sistemas embarcados devido aos requisitos acerca de poder de processamento e consumo de memória [Júnior 2007].

Quatro estados universais foram definidos (*Full*, *Light*, *Standby* e *Off*), estes estados podem ser aplicados ao sistema como um todo, apenas subsistemas ou a componentes específicos. O desenvolvedor também pode definir outros estados para outros componentes nos quais os estados pré-definidos não sejam adequados ou suficientes.

Como enviar sinais de troca de modo de operação para vários componentes não seria uma solução "prática", o gerente de energia pode propagar esses sinais para subsistemas formados por componentes semelhantes

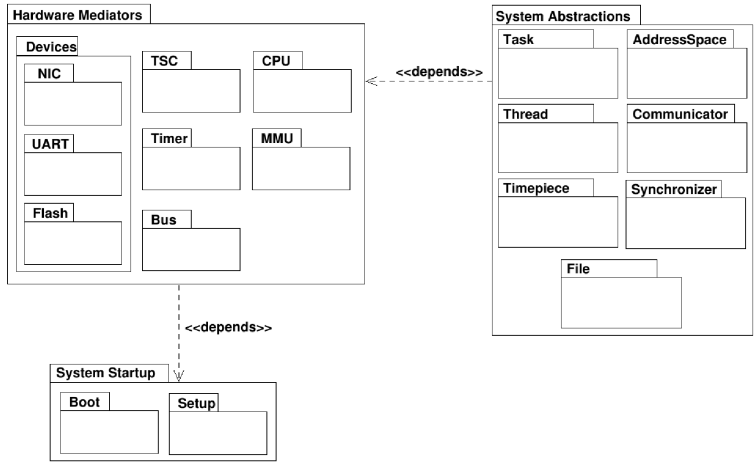

Figura 2.6: Visão geral de componentes

ou o sistema como um todo. Como mostrado na Figura 2.7, a aplicação pode trocar o modo de operação somente do processador, de toda a abstração *Communicator* (incluindo os dispositivos de *hardware*) ou de todo o sistema, com apenas uma invocação da operação `power`.

Com esse esquema para trocas de modo de operação, pode-se implementar um mecanismo para hibernar e acordar o sistema. Para hibernação, durante a propagação do sinal *hibernar*, cada componente do sistema salva um conjunto de informações julgadas necessárias para futuramente acordar o componente. Para acordar, é feita a operação inversa durante a propagação do sinal *acordar*. Assim, tem-se um mecanismo para hibernar e acordar o sistema facilmente personalizável para diversas arquiteturas de *hardware* e controlado pela aplicação.

2.2.2 EPOS Live Update System

EPOS *Live Update System* (ELUS) é um sistema para atualização de *software* proposto por Gracioli [Gracioli 2009]. ELUS usa a estrutura de invocação remota do EPOS que provê uma camada de isolamento aos componentes do sistema. Essa estrutura foi estendida para permitir atualizações de *software* sem a necessidade de reiniciar o sistema.

A Figura 2.8 mostra como invocações de método são feitas com o ELUS. A chamada passa pelo *Proxy*, que envia uma mensagem para o *OS*

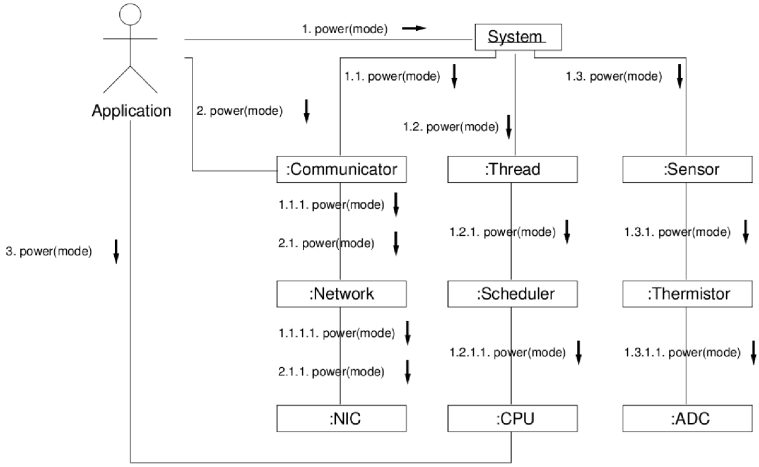

Figura 2.7: Propagação de estado com o gerente de energia [Júnior 2007]

Box, um controlador de acessos que evita a invocação de um método que esteja sendo atualizado. Uma vez dentro do *Agent* a chamada é passada para o componente que, depois de terminada a execução, repassa o valor de retorno ao *Agent*. Assim um nível de indireção é criado e apenas o *Agent* sabe aonde estão os componentes dentro da memória do sistema. Esse conceito de componente pode também ser aplicado aos mediadores do sistema, permitindo que eles sejam inseridos e removidos do sistema em tempo de execução.

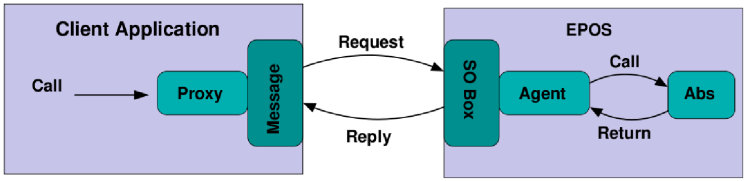

Figura 2.8: Invocação de métodos com ELUS [Gracioli 2009]

Uma *thread* para receber requisições de atualização e código executável é criada durante a inicialização do sistema. Essa *thread* usa o protocolo ETP (ELUS *Transport Protocol*) para se comunicar com o *Agent*,

o responsável por fazer a interface com o componente. Para atualizar um componente, essa *thread* passa a requisição para o *Agent* que aloca memória para o novo código, copia o código recebido para essa nova posição de memória, atualiza os endereços dos novos métodos recebidos, destrói o objeto antigo, cria o novo objeto e adiciona o novo objeto à tabela de objetos [Gracioli e Frohlich 2008].

Essa infraestrutura apresentou [Gracioli e Frohlich 2008] pouca sobrecarga em termos de memória e processamento devido à utilização de técnicas de meta-programação estática que resolvem as questões referentes aos componentes marcados como atualizáveis em tempo de compilação. Portanto, tal infraestrutura pode ser utilizada sem comprometer a funcionalidade de sistemas embarcados.

2.3 Processadores *Softcore*

Para implementação desse trabalho, decidiu-se pela utilização de processadores *softcore* pela sua facilidade de obtenção em relação a núcleos *hardcore* dentro de FPGAs ou sistemas com processador físico externo ao FPGA. Acredita-se que assim o sistema possa abranger um número maior de plataformas-alvo, pois basta ter um FPGA e um dos diversos processadores sintetizáveis disponíveis para formar um sistema embarcado reconfigurável.

Processadores *softcore* são processadores implementados em linguagens de descrição de *hardware* que podem ser sintetizados para dispositivos programáveis. Esses processadores são mais simples que processadores em *hardware* e alcançam apenas entre 30% a 50% da velocidade, mas possuem a vantagem de serem adaptáveis às necessidades da aplicação, o que pode gerar ganhos imensos de performance [Maxfield 2004].

Entre os diversos existentes, o processador escolhido foi o Plasma devido ao seu pequeno tamanho e simplicidade e por já existir para ele um porte funcional do EPOS. Apesar no número restrito de funcionalidades, o Plasma consegue rodar o EPOS sem qualquer tipo de perda de funcionalidade, descartando a necessidade de se utilizar um processador *softcore* mais sofisticado.

2.3.1 Plasma

Plasma é uma máquina RISC de 32 *bits* que suporta todas as instruções da arquitetura MIPS I com exceção de *load* e *store* desalinhadas, por serem patenteadas [Rhoads 2008]. Está atualmente em sua terceira versão e é considerado estável. Por ser uma máquina MIPS, pode ser usado para executar código gerado através do compilador C da GNU, que normalmente não gera as instruções não suportadas.

Por ter seu código aberto, é adequado para projetos de pesquisa pela possibilidade de adaptar seu funcionamento para suprir necessidades e/ou adicionar/remover funcionalidades. Possui 2 ou 3 estágios de *pipeline*, controlador de interrupções, multiplicador em *hardware*, *timer*, GPIO e controladores para UART, SRAM, DDR SDRAM, Ethernet e Flash e executa a 25 MHz.

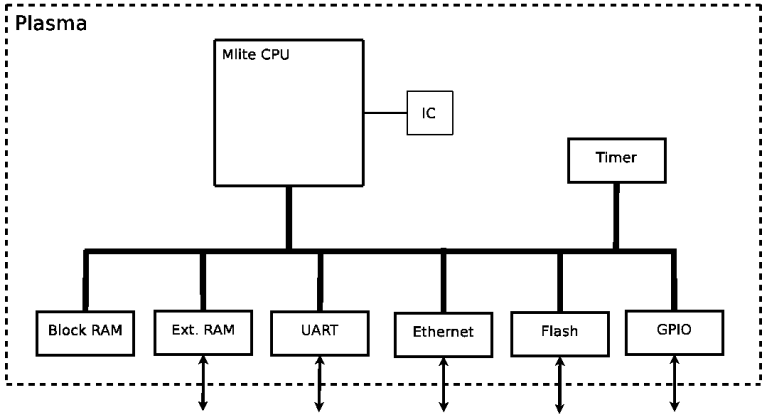

Figura 2.9: Diagrama de blocos do Plasma

O Plasma possui duas memórias, uma interna que utiliza a *Block* RAM do FPGA e é usada para armazenar e executar o *boot loader* e outra externa que é utilizada pelo sistema operacional. A Tabela 2.1 mostra como a memória do Plasma é mapeada, incluindo também os registradores que servem para propósitos diversos. Os registradores marcados com um * foram adicionados ao Plasma como parte deste trabalho.

2.3.2 Outros Processadores

Existem diversos processadores softcore disponíveis atualmente entre os quais se destacam o LEON e o OpenRISC 1200, bastante comuns na literatura.

LEON2 é uma máquina SPARC V8 [SPARC International Inc. 1992] de 32 bits, que possui cache de dados e instruções separadas (arquitetura Harvard) [Gaisler Research 2005]. É modular e possui diversos IPs que podem ser adicionados ou retirados do sistema, possui uma interface para co-processadores e modo *Power Down*. Possui as seguintes características: multiplicador e divisor em *hardware*, controlador de interrupções,

Tabela 2.1: Mapa de memória do Plasma

Descrição	Endereço
RAM interna (*Block RAM*)	0x00000000 - 0x0000FFFF (8 kB)
RAM externa (física)	0x10000000 - 0x100FFFFF (1 MB)
UART (leitura e escrita)	0x20000000
Máscara de IRQ	0x20000010
Estado de IRQ	0x20000020
Saída GPIO	0x20000030
Saída GPIO (negada)	0x20000040
Entrada GPIO	0x20000050
Contador de ciclos (leitura)	0x20000060
Contador de ciclos (escrita)*	0x20000070
Reconfiguração*	0x20000080

dois *timers* de 24 bits, modo *Power Down*, *watchdog*, GPIO de 16 bits, *Ethernet* MAC e interface PCI, além de controladores para PROM, SRAM, SDRAM e UART e uma interface para co-processadores. Executou a 54 MHz.

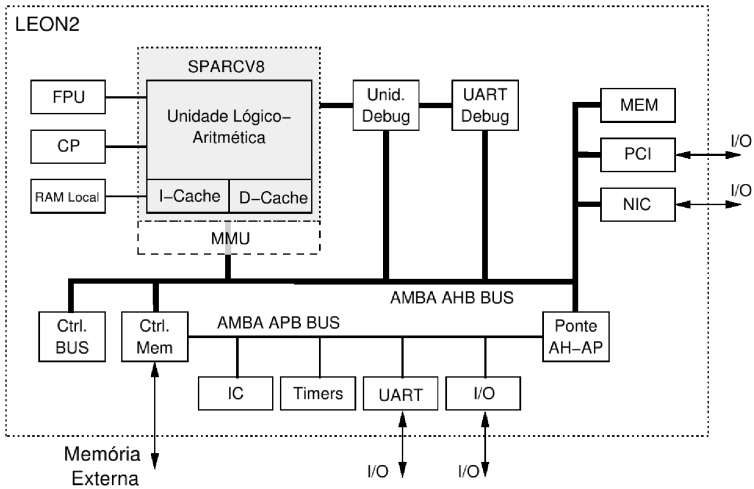

Figura 2.10: Diagrama de blocos do LEON2

OpenRISC 1200 é um implementação de código aberto da família

de processadores OpenRISC 1000, é uma máquina RISC de 32 bits baseada em MIPS com arquitetura Harvard. Tem como características um *pipeline* de 5 estágios, suporte a memória virtual (MMU), unidade de depuração, temporizador de alta resolução, controlador de interrupção programável e suporte a gerenciamento de energia [Erlandsson 2009].

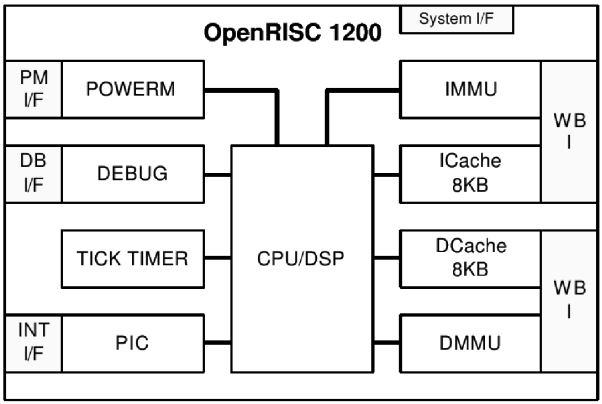

Figura 2.11: Diagrama de blocos do OpenRISC 1200 [Erlandsson 2009]

Capítulo 3

Estado da Arte

Nesse capítulo serão apresentadas algumas abordagens em relação a suporte de sistema operacional para sistemas embarcados reconfiguráveis. O foco dessa revisão de bibliografia não é analisar sistemas operacionais (ou camadas de *software*) pela habilidade em gerenciar *bitstreams* ou velocidade de configuração, pois gerenciamento de módulos de *hardware* é a tarefa mais básica que um sistema desse tipo deve fazer. Essa revisão visa o suporte em termos de abstrações oferecidas pelos sistemas ao desenvolvedor da aplicação.

Focado em automação automotiva, Ullmann et al. [Ullmann et al. 2004] propõem um ambiente de execução que preza pelo tamanho e simplicidade. Esse sistema executa sobre um processador *softcore* (Microblaze) e é responsável pelo gerenciamento de reconfiguração de módulos de *hardware* e pelas trocas de mensagens entre esses módulos e deles com o exterior. Por limitar-se a apenas essas tarefas, esse sistema não pode ser considerado um sistema operacional, pois até mesmo o escalonamento das tarefas é responsabilidade do desenvolvedor do sistema. A infraestrutura de *hardware* é composta, além do Microblaze, por um árbitro de barramento que gerencia a comunicação com os quatro *slots* reconfiguráveis e com o exterior. A organização desse sistema pode ser vista na Figura 3.1.

O sistema não apresenta nenhuma facilidade de integração de *software* para o desenvolvedor da aplicação, e a integração da aplicação com ambiente de execução é feita inteiramente pelo desenvolvedor. Os autores defendem uma plataforma de software pequena e simples para algumas aplicações automotivas e de automação doméstica, considerando sistemas mais complexos inadequados para essas aplicações.

Deng et al. [Deng et al. 2005] apresentam um sistema operacional de tempo real desenvolvido com suporte a *hardware* reconfigurável em mente. Possui um *kernel* que implementa funcionalidades básicas como gerenciamento de recursos (particionamento da área reconfigurável, gerenciamento de memória e mapeamento de dispositivos de E/S), escalonamento de tarefas e comunicação entre tarefas. As tarefas do sistema podem ser tanto de *software* quanto de *hardware*, pois ambas possuem a mesma interface com o sistema operacional, e são gerenciadas sem distinção pelo escalonador. Essas tarefas se comunicam entre si através de uma interface do sistema operacional para comunicação entre tarefas. A arquitetura do sistema pode ser vista na Figura 3.2.

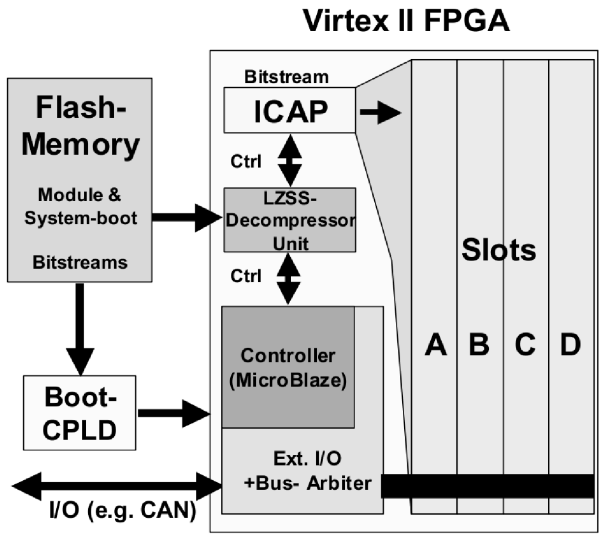

Figura 3.1: Sistema proposto por Ullmann et al. [Ullmann et al. 2004]

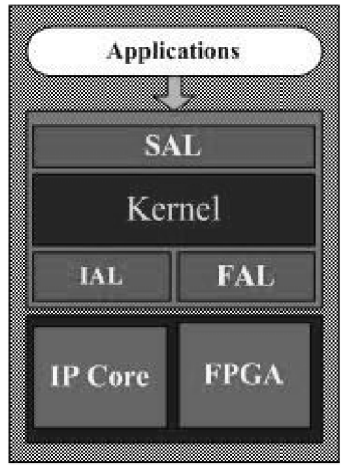

Figura 3.2: Arquitetura do sistema proposto por Deng et al. [Deng et al. 2005]

Uma aplicação, que é constituída de várias tarefas de *hardware* ou *software*, é integrada ao sistema através de uma camada de abstração de *software*. Essa SAL (*Software Abstraction Layer*) faz a interface entre a aplicação e o *kernel*. A parte de *hardware* é dividida em módulos e o *hardware* como um todo é gerenciado por uma camada de abstração. Essa camada se divide em IAL (*IP-core Abstraction Layer*) e FAL (*FPGA Abstraction Layer*) que tem por objetivo fornecer uma interface padrão ao *kernel* para facilitar o porte do sistema para diferentes modelos de FPGA e diferentes implementações de uma mesma função em *hardware*.

Walder e Platzner [Walder e Platzner 2003] apresentam um sistema operacional para sistemas embarcados reconfiguráveis que possuem UCP externa ao FPGA. Nessa abordagem, o sistema operacional está parte em *software* rodando na UCP e parte em *hardware*, configurado no FPGA. A parte em *software* é responsável pelos serviços comuns de sistema operacional como escalonamento de tarefas, gerenciamento de recursos e de tempo. Enquanto isso, a parte em *hardware* cuida do gerenciamento de memória (MMU - *Memory Management Unit*) e gerenciamento dos módulos de *hardware*, além de estabelecer uma infraestrutura de comunicação entre os módulos e destes com o processador central. Essa arquitetura pode ser visualizada na Figura 3.3. Diferencia-se de outras abordagens por ser um sistema operacional mais completo, com características não mencionadas nos projetos já citados, como semáforos, caixas de mensagens, gatilhos de tempo e gerenciamento de memória através de uma MMU própria.

Essa abordagem apresenta uma infraestrutura de comunicação inteligente que interliga os *slots* reconfiguráveis. Essa infraestrutura permite que módulos de tamanho múltiplo do tamanho padrão do *slot* sejam inseridos, o que aumenta a flexibilidade do sistema. Por outro lado, esse barramento, somado à parte de *hardware* do sistema operacional, implica em um sobrecusto de espaço, fazendo com que essa solução seja muito pouco eficiente em FPGAs pequenos. Outra desvantagem é a necessidade de se ter um processador externo ao FPGA, o que limita sua utilização em sistemas embarcados.

Nesse sistema, a aplicação é composta de tarefas de *hardware* do usuário e objetos do sistema operacional [Walder, Steinegger e Platzner 2004], que se comunicam através de uma interface padronizada. Infelizmente, não é possível definir e, conseqüentemente, escalonar tarefas somente de *software*. Isso limita, de certa forma, as possibilidades de uso do sistema, pois vários tipos de tarefas são muito simples para serem implementadas em *hardware*.

Williams e Bergmann [Williams e Bergmann 2004] apresentam a

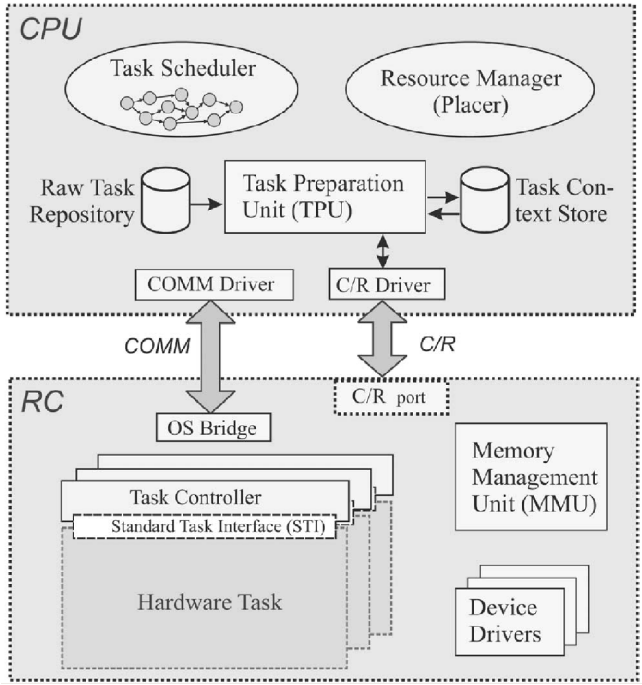

Figura 3.3: Arquitetura proposta por Walder e Platzner [Steiger, Walder e Platzner 2004]

utilização de *uClinux* como plataforma de software para reconfiguração de hardware. O *uClinux* foi portado para executar no processador *softcore* MicroBlaze da Xilinx e para ele foi criado um *driver* para ICAP que o enxerga como um dispositivo de caracteres. Assim, um programa pode controlar uma reconfiguração através de chamadas como *open*, *read* e *write* que acessam diretamente o dispositivo.

Para utilizar essa interface de baixo nível, é necessário entender o funcionamento do módulo ICAP e detalhes sobre a mecânica envolvida em uma reconfiguração, o que dificulta a implementação. Mas como estamos em um sistema *Unix-like*, podemos acessar o dispositivo, e assim realizar reconfigurações, de uma maneira mais alto nível através de *Shell Scripts*. Por exemplo, executando o comando *cat*, com sua saída redirecionada para o dispositivo ICAP, sobre um *bitstream* resulta em uma reconfiguração. Seguindo o raciocínio, podemos utilizar diversas outras ferramentas como

gunzip para compressão de *bitstreams* ou *wget* para obter um *bitstream* de um servidor remoto através de uma rede.

Obviamente, a utilização de *Shell Scripts* para realizar reconfigurações, apesar de simplificar o processo, implica em uma diminuição na performance do sistema, ainda mais se esse for embarcado. Assim, o desenvolvedor tem que escolher entre utilizar ferramentas e agregar sobrecusto de execução ou acessar o dispositivo diretamente e controlar sua configuração, *byte* por *byte*, para garantir performance. É interessante notar também que, por ter sua mecânica de reconfiguração atrelada ao módulo ICAP, FPGAs menores ou mais simples, que não possuem tal recurso, estão fora do escopo de utilização desse sistema.

Esse trabalho foca em como a reconfiguração acontece e não se preocupa em mostrar como a aplicação se comunica com os módulos. Assim, não se pode avaliar de forma completa o suporte para o desenvolvedor.

OS4RS [Nollet et al. 2003] (*Operating System for Reconfigurable Systems*) é uma extensão para o RTAI (*Real Time Application Interface*). RTAI, por sua vez, é uma extensão de tempo real para Linux composta de uma HAL própria e de serviços para facilitar o desenvolvimento de aplicações de tempo real baseadas em Linux.

OS4RS possui uma camada de abstração de *hardware* responsável por gerenciar uma NoC (*Network-on-a-Chip*) e um gerenciador de tarefas. Essa NoC é responsável pela organização da área reconfigurável do FPGA e pela abstração da comunicação entre módulos de *hardware* e desses com o processador principal. Também é utilizada para gerenciar reconfiguração de módulos através de uma API. A Figura 3.4 mostra um exemplo dessa NoC (chamada na figura de ICN) com quatro módulos reconfiguráveis. Esses módulos são vistos pelo sistema operacional como tarefas de *hardware* que, por possuírem a mesma interface com o sistema operacional que tarefas de *software*, são tratadas e escalonadas sem distinção pelo gerenciador de tarefas, como representado na Figura 3.5. Esse tratamento igual de tarefas, aliado a um mecanismo de salvamento de estado das tarefas, permite a migração de tarefas de *software* para *hardware* e vice-versa. Para realizar a comunicação entre essas tarefas, o sistema operacional provê uma interface de comunicação entre tarefas.

Por utilizar uma NoC, esse sistema apresenta maior fragmentação interna e redução do espaço utilizável que sistemas baseados em *slots* e pode não ser indicada para FPGAs de menor porte. Outra desvantagem é o tamanho de *software* do sistema, além de se basear em Linux, as extensões de reconfiguração aumentam consideravelmente o tamanho do sistema como um todo.

No trabalho desenvolvido por Santambrogio, Rana e Sciuto [San-

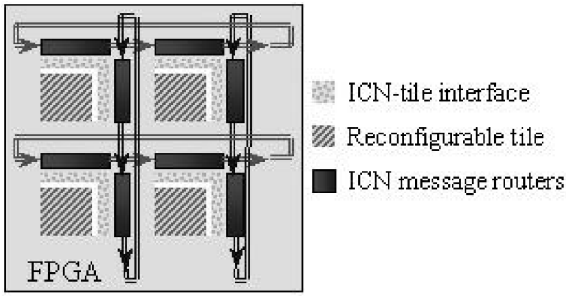

Figura 3.4: Rede em *Chip* utilizada pelo OS4RS [Nollet et al. 2003]

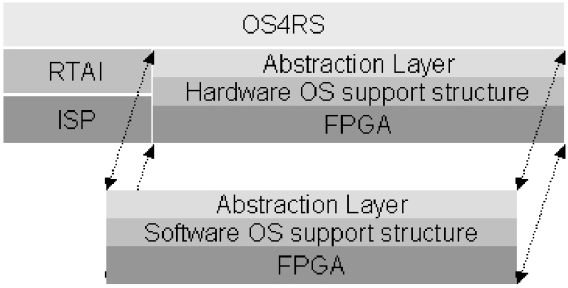

Figura 3.5: Pilha de abstrações do OS4RS com interface igual para tarefas de *software* e de *hardware* [Nollet et al. 2003]

tambrogio, Rana e Sciuto 2008], o sistema operacional Linux foi estendido para gerenciar reconfigurações de hardware. Um gerenciador central de reconfiguração foi implementado na forma de *Daemon* e cabe a ele encontrar lugar na malha reconfigurável para encaixar novos módulos e remover módulos fisicamente, apagando sua memória de configuração, ou logicamente, apenas marcando no sistema operacional o espaço como livre. Também é responsabilidade do gerenciador inicializar *drivers* para realizar a comunicação com os módulos, que são vistos pelo sistema como dispositivos de hardware.

Do ponto de vista do programador, a tarefa fica bastante simplificada pois existem somente quatro operações: requisição de um novo módulo, liberação de módulo, remoção de módulo (que difere da liberação por remover fisicamente) e listagem de módulos atualmente configurados.

Esse ganho em simplicidade de utilização implica em um sobrecusto de execução, o carregamento de um módulo de *hardware*, por exemplo, pode ter sobrecusto de *software* de até 60% do tempo de reconfiguração. Um dos motivos para essa grande sobrecarga é a sofisticação do gerenciador de reconfiguração, que a cada inserção de módulo, procura em toda a malha do FPGA pelo melhor lugar para encaixá-lo. Por outro lado, isso reduz a fragmentação da área reconfigurável.

Um resumo das características encontradas pode ser visto na Tabela 3.1. Esses sistemas serão abordados quantitativamente no Capítulo 5, juntamente com uma comparação com a abordagem proposta.

A maioria das soluções estudadas baseiam-se em sistemas operacionais, o que é necessário para facilitar o desenvolvimento de aplicações através de abstrações, além de melhorar a portabilidade da solução. Alguns desses sistemas operacionais possuem características não necessárias, mas bastante úteis, como escalonamento em tempo real e possibilidade de migração de tarefas de *software* para *hardware* e vice-versa. No que diz respeito à comunicação do *software* com tarefas de *hardware*, os sistemas que possuem tarefas tanto de *hardware* quanto de *software* utilizam uma interface de comunicação entre tarefas providas pelo sistema operacional. Já os sistemas que possuem somente tarefas de *software* tendem a se comunicar através de *drivers*, que acessam diretamente os módulos ou que abstraem um barramento de comunicação, dependendo da organização do *hardware*. Notou-se que alguns dos sistemas apresentaram-se com um tamanho binário grande, em comparação com o EPOS, o que não se justifica pelas funcionalidades apresentadas.

Várias abordagens foram adotadas para realizar reconfigurações. A mais simples delas, do ponto de vista do desenvolvedor, foi a proposta por Santambrogio, Rana e Sciuto que define apenas quatro operações de alto nível para inserir, remover e listar módulos. Outras abordagens tratam o FPGA como um dispositivo e o acessam através de *drivers* que fornecem meios para realizar configurações, geralmente através de operações de nível mais baixo, o que dificulta o trabalho do desenvolvedor.

Outro ponto interessante é a organização de *hardware* dos sistemas. Em sua maioria, são compostos por um processador *softcore* e por *slots* reconfiguráveis acessados através de um barramento. OS4RS utiliza uma NoC ao invés de um barramento simples, o que acaba fragmentando e utilizando mais recursos de *hardware*, enquanto que a proposta de Walder e Platzner necessita de uma UCP externa ao FPGA, além de possuir partes do sistema operacional implementadas em *hardware*. Ambas abordagens, devido a essas características, possuem uma variedade de utilização menor do que sistemas com menos exigências em relação ao *hardware*.

Tabela 3.1: Resumo de características dos trabalhos relacionados

Abordagem	**API para Reconfiguração**	**Comunicação com módulos**	**Organização do *Hardware***	***Software***
Ullmann et al.	Iniciado pela aplicação, gerenciado pelo *hardware*	Definido na aplicação, controlado por árbitro de barramento	Processador *softcore*, 4 *slots*	Sistema gerenciador de reconfiguração e troca de mensagens
Deng et al.	Dentro do SO	Entre tarefas de *software* e de *hardware*	Modular	RTOS com tarefas de *software* e de *hardware*
Walder e Platzner	*Driver* para FPGA controlado pelo SO	Barramento com interface para o SO	Módulos com barramento e processador externo. Parte do SO em *hardware*	SO somente com tarefas de *hardware*
Williams e Bergmann	Baixo nível (C/C++) e alto nível (Bash)	Não especificado	Processador *softcore* e módulos	μClinux
OS4RS	API que abstrai NoC	Intermódulo, gerenciado pela NoC	NoC com módulos	RTOS baseado em Linux com tarefas de *hardware* e *software*
Santambrogio, Rana e Sciuto	Somente 4 operações (Inserir, Listar, Liberar e Remover)	*Drivers*	Modular	Linux

Capítulo 4

Suporte de Sistema Operacional para Reconfiguração de Hardware

Com a crescente demanda por maior poder de processamento e menor consumo de energia em sistemas embarcados, o *hardware* reconfigurável surge como uma opção capaz de atender a tais características quase contraditórias [Lodi, Toma e Campi 2003, Abnous et al. 1998, Mencer, Morf e Flynn 2000, Tessier e Burleson 2001]. No entanto, gerenciar consistentemente mudanças de *hardware* e sua integração com o *software* aumenta consideravelmente o esforço de desenvolvimento do sistema, dificultando a adoção comercial da tecnologia, apesar de todas as vantagens associadas a ela.

Por isso, torna-se necessário uma infraestrutura de gerenciamento que facilite o desenvolvimento da aplicação, afastando o programador da complexidade do sistema e provendo uma API simples e clara [Nollet et al. 2003]. Um sistema operacional forma um abstração que esconde do desenvolvedor os detalhes da tecnologia envolvida e diminuem *time-to-market* e custos não recorrentes de engenharia, além de aumentar o reuso do *software* e a flexibilidade em relação ao *hardware* [Deng et al. 2005]. Portanto, um sistema operacional com infraestrutura de gerenciamento de reconfiguração de *hardware* é um caminho quase intuitivo a seguir.

Abstrações de sistema operacional facilitam o reuso de código e circuitos testados e confiáveis, o que pode acelerar consideravelmente os ciclos de desenvolvimento e diminuir *time-to-market* [Walder e Platzner 2003]. Além disso, o processo de portar aplicações fica bastante simplificado se o sistema operacional suportar diversas arquiteturas, e recompilação/re-síntese geralmente resulta em versões portadas funcionalmente corretas, permitindo que o desenvolvedor se concentre em, por exemplo, desempenho [Walder e Platzner 2003].

Nesse contexto, optou-se por desenvolver um suporte para reconfiguração de *hardware* no sistema operacional EPOS. Como o foco são sistemas embarcados, esse suporte tem a preocupação de adicionar o menor sobrecusto possível, tanto de tempo de execução quanto de tamanho do sistema, mantendo a filosofia do EPOS de fazer o máximo com um mínimo.

Primeiramente, após um estudo sobre reconfiguração parcial, descobriu-se que para habilitar um FPGA a realizar tal tarefa pelo modo comumente utilizado, era necessário um conjunto especial de ferramentas de

software fornecidos pelo fabricante. Sem essas ferramentas não se consegue especificar os pontos da malha do FPGA que delimitarão as áreas reconfiguráveis e, portanto, fazem com que o projeto fique dependente delas.

Não concordando com essa necessidade, foi procurada uma alternativa para a obtenção de *bitstreams* parciais que não necessitasse de ferramentas que não sejam as mais comumente utilizadas, no caso da Xilinx, a ferramenta ISE Foundation. Além disso, reforçam a motivação de se procurar tal alternativa, as desvantagens de reconfiguração modular ressaltadas da Seção 2.1.2 juntamente com o fato de reconfiguração parcial baseada em módulos ser pouco utilizável em FPGAs de menor porte.

Essa alternativa foi encontrada na reconfiguração parcial baseada em diferença, discutida na próxima seção. Diferentemente da reconfiguração baseada em módulos, essa estratégia tem como característica a imprevisibilidade na geração de *bitstreams* parciais, resultando em uma reconfiguração desestruturada. Desse modo, o esforço de desenvolvimento que seria gasto com estruturação do *hardware* (*floor-planning*) passa para a resolução do problema de manter o *software* executando sobre um processador que pode mudar de posição de um instante para outro.

Como realizar alterações no processador enquanto o mesmo executa pode causar perda ou corrupção de dados, não se pode manter um processador executando em uma situação dessas, o que se pode é simular tal comportamento. Assim, a solução encontrada foi pausar a execução do *software* enquanto o *hardware* é reconfigurado. Essa pausa deve salvar todo o estado do *hardware* e depois da alteração, recuperar essas informações para seguir com a execução do *software*. Nesse momento, tem-se duas questões importantes para resolver: como salvar eficientemente as informações necessárias (recuperar é apenas a operação inversa) e como fazer o *software* saber quando deve voltar à ativa.

A resolução da primeira questão foi baseada em como sistemas computacionais comuns a resolvem, utilizando um processo de suspensão ou hibernação, controlado por um gerenciador de energia. Felizmente, o EPOS dispõe de um gerente de energia que, apesar de não possuir a capacidade de hibernar o sistema, pode ser modificado para tal fim. E assim foi feito, criando-se dois novos estados (hibernado e acordado) que percorrem, através do esquema de propagação de troca de estados do gerente de energia, todos os componentes do sistema que precisam saber que uma reconfiguração vai acontecer ou acabou de acontecer. A definição dos dados que precisam ser salvos depende de um estudo da parte de *hardware* do sistema para identificar componentes que têm seu estado interno modificado durante a execução. Dentre esses componentes podemos citar o processa-

dor, que precisa salvar seus registradores, o temporizador do sistema, que precisa salvar o valor atual do contador e o controlador de interrupções, que precisa salvar o valor atual da máscara de interrupções ativas. O salvamento dessas informações acontece em uma área da memória principal do sistema reservada para esse fim.

Com o mecanismo de salvamento e recuperação resolvido, focou-se no problema de como avisar um programa que não está mais executando que ele deve voltar à ativa. O processador *softcore* utilizado possui uma memória interna (que utiliza Block RAM do FPGA) para armazenar um *boot loader*, que é executado toda vez que um sinal de *reset* é enviado para dentro do FPGA. Originalmente, esse *boot loader* abre uma conexão na porta serial e espera o recebimento de um binário para executá-lo. Modificando-o para, através de um "aperto de mão", descobrir se há um EPOS em estado de espera e fazê-lo voltar a executar, pode-se resolver o problema. Em caso afirmativo, o *boot loader* recupera o contexto do processador e com isso faz o fluxo de execução pular para o lugar de onde a execução anterior parou, permitindo iniciar o processo de recuperação das outras informações previamente salvas.

Esse mecanismo permite que o *software* seja pausado e retorne a executar para que o *hardware* seja modificado. Mas o processo não acaba nesse ponto, se o *hardware* foi modificado é porque algum componente foi inserido, removido ou alterado e isso precisa ser repassado ao *software*. Como se trata de sistemas embarcados, a aplicação é o ponto central e, portanto, sabe o que está acontecendo com o sistema. Como é ela quem pede por um componente novo, ou para excluir um existente, deve ficar a cargo dela instanciar ou remover mediadores para esses componentes. No caso do componente não ser acessado diretamente pela aplicação e sim por uma abstração do sistema, essa abstração fica responsável pelo gerenciamento dos componentes subordinados a ela.

Para facilitar o trabalho do programador, foi criada uma classe C++ que serve como base para o desenvolvimento desses mediadores. Essa classe define atributos e operações básicas que podem ser especializadas para melhor suportar o componente de *hardware*, o que deve ser feito em tempo de desenvolvimento. Para o programador, no desenvolvimento da aplicação ou de uma abstração, basta instanciar um novo objeto do tipo do componente quando um componente for inserido, ou apagar um objeto já instanciado de um componente que será removido. Futuramente, a tarefa de adequar o *software* à nova realidade do *hardware* será responsabilidade do ELUS, apresentado no Capítulo 2. ELUS possui uma solução muito mais sofisticada para atualização de *software*, que poderá ser feita de uma maneira mais automatizada, diminuindo a necessidade de intervenção do

desenvolvedor.

Assim como algumas das outras abordagens, o sistema proposto possui uma interface de alto nível e simples para que o trabalho do programador seja facilitado ao máximo. Limita-se a três operações: iniciar reconfiguração (escolhendo qual configuração utilizar), instanciar mediador e remover mediador.

Uma diferença para outras implementações é a não classificação das tarefas em *software* ou *hardware*. O EPOS não foi modificado para entender, e conseqüentemente escalonar, componentes de *hardware* como tarefas isoladas. Isso dá mais liberdade ao desenvolvedor, pois nem sempre vai-se querer utilizar esses componentes dessa maneira. Deste modo, um componente de *hardware* pode realizar suas atividades sem a ciência do sistema operacional, sendo acessado para enviar dados ou obter resultados diretamente pela aplicação ou por uma abstração através do mediador. Mas pode-se facilmente criar uma tarefa de *hardware* designando uma *thread* para gerenciar exclusivamente um componente de *hardware*.

Em termos de tamanho do sistema, utilizar EPOS como ponto de partida é uma enorme vantagem em relação a outros sistemas, devido ao seu tamanho e número de funcionalidades. Principalmente em relação a Linux embarcado, bastante utilizado para esse tipo de suporte, como pôde ser visto no capítulo anterior. Apesar de ter tamanho bem menor que uClinux, EPOS não deixa a desejar em termos de funcionalidades, diferentemente de ambientes de execução utilizados como plataforma de *software* para reconfiguração, que possuem apenas um escalonador, ou sequer isso.

Por tamanho do sistema também entende-se o tamanho da parte de *hardware*, que em geral envolve definição dos módulos reconfiguráveis e barramentos de interconexão ou redes em *chip*. Quanto mais sofisticada a parte de *hardware*, apesar de permitir operações mais sofisticadas pelo *software*, também representa uma redução no espaço utilizável do FPGA. Como deseja-se poder utilizar esse sistema no maior número possível de modelos, inclusive os de menor capacidade, optou-se por uma abordagem não modular, que reduz a zero o sobrecusto de *hardware*.

Em relação à sobrecarga de tempo de execução, devido à estratégia de reconfiguração adotada, não há a necessidade de um processo de *software* que encontre a melhor maneira de realizá-la. Isso deixa mais simples o processo do lado do *software*, porém aumenta o tempo gasto para reconfigurar o *hardware*, pois as *bitstreams* parciais geradas tendem a ser maiores. A sobrecarga em termos de *software* nesse caso se dá pelo tempo gasto pelo gerente de energia para percorrer o sistema para salvar e posteriormente recuperar informações. Esse valor varia com a quantidade de componentes que precisam ser salvos.

Outra característica importante para um sistema embarcado é a portabilidade. Foi observado que a maioria dos sistemas estudados possui uma implementação focada em um determinado modelo de FPGA, devido à necessidade de especificar fisicamente o local dos módulos reconfiguráveis e/ou utilizar recursos especiais, como ICAP. Devido à utilização do processo de diferenciação para gerar *bitstreams*, o processo de adaptação do projeto para outro modelo é trivial, e também não necessita ICAP. Portar uma solução para outro modelo de FPGA também é dificultado pela existência de barramentos complexos ou redes em *chip*, que podem ter que mudar drasticamente devido a limitações de tamanho e geometria.

Apesar de utilizar uma estratégia não muito comum para reconfiguração de *hardware* em sistemas embarcados, o suporte de sistema operacional apresentado neste trabalho pode ser usado para o desenvolvimento de sistemas embarcados reconfiguráveis, trazendo algumas vantagens e desvantagens em relação a outras abordagens.

4.1 Reconfiguração Não Modular

Em um sistema comum de reconfiguração de *hardware*, reconfiguração baseada em módulos é utilizada para realizar alterações em grandes porções de lógica. Componentes de *hardware* são encapsulados em módulos que podem ser encaixados nos espaços pré-definidos da malha do FPGA. Durante o projeto do *hardware* é definida a parte estática, que em geral consiste de um processador e um barramento ou rede para fazer a conexão com as partes dinâmicas. Essas partes dinâmicas, por sua vez, são espaços definidos fisicamente na malha da FPGA. O projeto de um sistema desse tipo é feito utilizando ferramentas especiais providas pelo fabricante.

Para alimentar os componentes que futuramente serão encaixados nesses espaços, o barramento ou rede de interconexão é composto de diversas linhas de sinais, tanto de dados quanto de controle. Esse conjunto estático de sinais é escolhido, durante o projeto, para melhor atender as necessidades dos componentes. Um exemplo de sistema desse tipo pode ser visto na Figura 2.3. Apesar de ser um modelo consistente a ser seguido, algumas questões podem ser levantadas.

Depois do projeto terminado, pode-se descobrir que outro modelo de FPGA é mais indicado devido a questões de custo ou consumo de energia. Como a definição dos módulos é feita fisicamente, o projeto fica estrito a um determinado modelo de um determinado fabricante. Portar a solução pode ter algumas dificuldades, principalmente em relação à geometria dos módulos e localização de pinos de entrada e saída específicos de um dispositivo.

Ou ainda, após o projeto terminado, um novo componente não previsto deverá ser criado e o mesmo não é compatível com o conjunto de sinais que o conectam ao restante do sistema. Como a interconexão do módulo com o resto do sistema é estática, um novo componente que necessite de outros ou mais sinais simplesmente não poderá ser utilizado sem realizar alterações no barramento.

Além disso, esse novo módulo pode não caber nos espaços pré-definidos. A definição física dos módulos também implica em restrições de tamanho. Enquanto todos os componentes são previstos durante o projeto, isso não é um problema. Mas se um novo componente não previsto for maior, não há o que fazer além de reestruturar todo o projeto.

Seria interessante possuir uma alternativa de projeto de sistemas reconfiguráveis que permitisse um maior dinamismo no desenvolvimento, fosse menos atrelado a ferramentas e permitisse uma maior portabilidade. Felizmente, isso existe mas é ignorado pela comunidade acadêmica. Isso porque gerando-se *bitstreams* parciais através de diferença, não se pode prever como essa *bitstream* afetará o *hardware* já configurado, resultando em uma reconfiguração não estruturada. Mas se for possível contornar o problema de se ter uma reconfiguração imprevisível e desestruturada, pode-se ter boas respostas para as questões anteriores e ainda algumas outras vantagens.

Assim como em projetos sem preocupação com reconfiguração, apenas o mapeamento dos sinais de entrada e saída para os pinos externalizados do FPGA é específicos para cada modelo. Assim, se for necessário adaptar o projeto para um outro modelo de FPGA, basta remapear esses sinais, o que não prejudica a portabilidade do projeto.

Como todo o *design* pode ser alterado, caso um componente novo não se encaixe nos sinais do barramento, basta alterar o barramento. O problema de um componente não caber inexiste, pois durante a síntese todas as partes do sistema podem ser realocadas para melhor adequação ao espaço disponível.

Além de solucionar os principais problemas do outro método, também podemos ressaltar algumas vantagens deste. Talvez a principal delas seja a simplificação da fase de projeto. Nenhuma etapa de preparação do *design* é necessária, basta criar o original e a(s) nova(s) configuração(ões) que as ferramentas comuns de geração de *bitstreams* se encarregam de gerar as diferenças entre elas. Uma desvantagem aqui é que é necessário um *bitstream* parcial para cada troca. Por exemplo, se tivermos três configurações possíveis e seja necessário trocar de qualquer umas delas para qualquer outra, serão geradas seis *bitstreams* parciais, e esse problema tende a piorar exponencialmente. Mas cabe ressaltar que nem todos os siste-

mas precisam trocar de qualquer configuração para qualquer outra. Nesses casos, otimiza-se a quantidade de *bitstreams* necessárias adotando uma estratégia incremental, onde cada nova configuração baseia-se na anterior. É interessante notar que passamos a pensar em termos de configuração como um todo e não apenas de componentes, com vários componentes podendo fazer parte de uma mesma reconfiguração, tanto para entrar quanto para sair.

Outra vantagem é que não há sobrecusto em termos de *hardware*, pois não existe uma infraestrutura estática de comunicação. O *hardware* sintetizado contém apenas o que é realmente útil para o funcionamento do sistema. Também não existe o problema de fragmentação interna do espaço de lógica, como não existem áreas pré-definidas, espaços não ficam inacessíveis quando um componente pequeno é inserido no sistema.

Uma desvantagem desse método é o tamanho das *bitstreams* geradas, pois não se pode garantir que elas sejam muito menores do que a original e muito provavelmente serão maiores do que *bitstreams* parciais que contêm um componente. Como tem-se observado uma contínua redução do custo de tecnologias de memória não-volátil, o impacto no custo do produto final pode ser compensado pelas vantagens obtidas. Obviamente, um aumento no tamanho da *bitstream* aumenta também seu tempo de configuração. Esse aumento pode reduzir a abrangência de adoção do método proposto e precisa ser medido para se ter uma noção do impacto no sistema.

Então, decidiu-se implementar um suporte no sistema operacional EPOS para esse tipo de reconfiguração e, assim, investigar mais a fundo sua viabilidade para sistemas embarcados reconfiguráveis.

4.2 Implementação

Para dar suporte à reconfiguração de *hardware* sem reiniciar a execução da aplicação, este trabalho propõe um mecanismo de salvamento e recuperação de estado. Ele é responsável por salvar informações relevantes para a execução do *software* e restaurar essas informações após uma reconfiguração. Para colher essas informações do sistema e depois regravá-las, foi utilizado o gerenciador de energia do EPOS, que permite que mensagens contendo informações sobre troca de modos de energia se propaguem pelas diversas partes do sistema.

O objetivo é chegar em um sistema como mostrado na Figura 4.1. O *hardware* é um FPGA com um processador *softcore* sintetizado e coprocessadores específicos para a aplicação, que podem estar integrando o sistema ou guardados em memória. A qualquer momento, a aplicação pode pedir por uma nova configuração de *hardware*, resultando em uma

reconfiguração. O *software* é um sistema operacional que quando recebe uma requisição de reconfiguração da aplicação, toma as medidas necessárias para que ela não perceba que o sistema terá que ser parado.

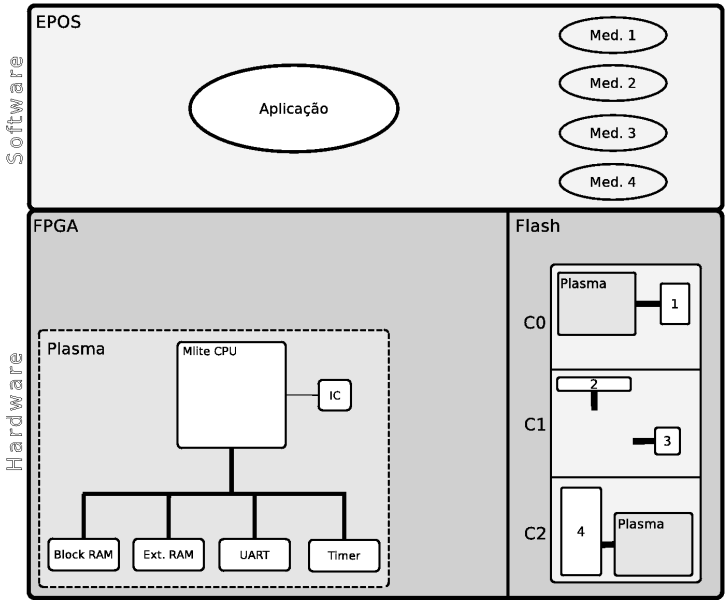

Figura 4.1: Modelo do sistema proposto

Inicialmente, imaginando-se as dificuldades atreladas a esse mecanismo, foi escolhida uma plataforma que pudesse ser facilmente modificada, permitindo ajustes caso assim fossem necessário para a implementação. Com isso em mente, foi escolhido o processador Plasma que, apesar de ser arquiteturalmente simples, permite a execução do sistema operacional EPOS sem nenhum tipo de limitação. Apesar disso, essa solução pode ser portada para qualquer outro processador *softcore*, modificando apenas os mediadores de *hardware* para se adequar ao novo processador.

Para permitir a reconfiguração de *hardware* com EPOS, a abstração *Reconfigurator* foi proposta. Essa abstração possui as seguintes operações: `select_config` e `reconfig`. A primeira seleciona a configuração que será utilizada mas, por limitações no ambiente de desenvolvimento, na implementação atual só permite selecionar uma única configuração. Essa limitação vem do fato da memória não-volátil do ambiente de desenvolvimento permitir o armazenamento de somente uma *bitstream*, seja ela total

ou parcial.

A segunda operação basicamente usa o gerente de energia para enviar dois sinais para todo o sistema. Primeiramente um sinal para iniciar o processo de hibernação (*Hibernate*) e, em seguida, um sinal para acordar o sistema (*Wake Up*).

O sinal *Hibernate* propaga pelo sistema, salvando informações de componentes que necessitem ser salvos para completar corretamente o processo de hibernação. Esse processo salva informações como os registradores do processador, o ponteiro da pilha, máscara de interrupção, valor do contador de ciclos de relógio e informações sobre o estado interno de componentes de *hardware* que estejam no sistema. Por fim, esse processo envia um sinal de reconfiguração para o FPGA, que imediatamente começa o processo de cópia do novo *bitstream*.

Quando o carregamento do *bitstream* termina, o processador *soft-core* precisa ser resetado para que o *boot loader* seja recarregado e recupere o contexto da UCP previamente salvo. Isso faz com que o fluxo de execução salte para dentro do EPOS logo antes do envio do sinal para acordar o sistema.

O processo de acordar envia um sinal *Wake Up* para todos os componentes que tiveram dados salvos e recupera essas informações. Assim, o sistema fica exatamente no mesmo estado de antes da reconfiguração, exceto obviamente, pelas mudanças no *hardware*. Esse processo como um todo pode ser visto na Figura 4.2.

Para lidar com componentes de *hardware* recentemente incluídos ou excluídos do sistema, instancia-se ou deleta-se mediadores que fazem a interface com esses componentes, respectivamente.

Nas próximas sub-seções será mostrada a implementação proposta.

4.2.1 Hibernar e Acordar

Como o processador deixará momentaneamente de existir, o programa que nele executa precisa ser guardado em outro lugar. Por programa entendemos toda a memória endereçada pelo EPOS e o contexto de execução do processador.

Primeiramente, foi reservado espaço na memória RAM do sistema suficiente para replicar toda a memória endereçada pelo EPOS e salvar os 34 registradores da UCP, o ponteiro da pilha, informações provenientes de mediadores e uma palavra de 32 *bits* que sinaliza um salvamento bem sucedido. Assim, na memória de 1 MB contida no *kit* de desenvolvimento utilizado, pode-se alocar 511,9 kB para o EPOS e o restante servirá como espaço de salvamento. O novo mapa de memória do EPOS pode ser visto na Figura 4.3.

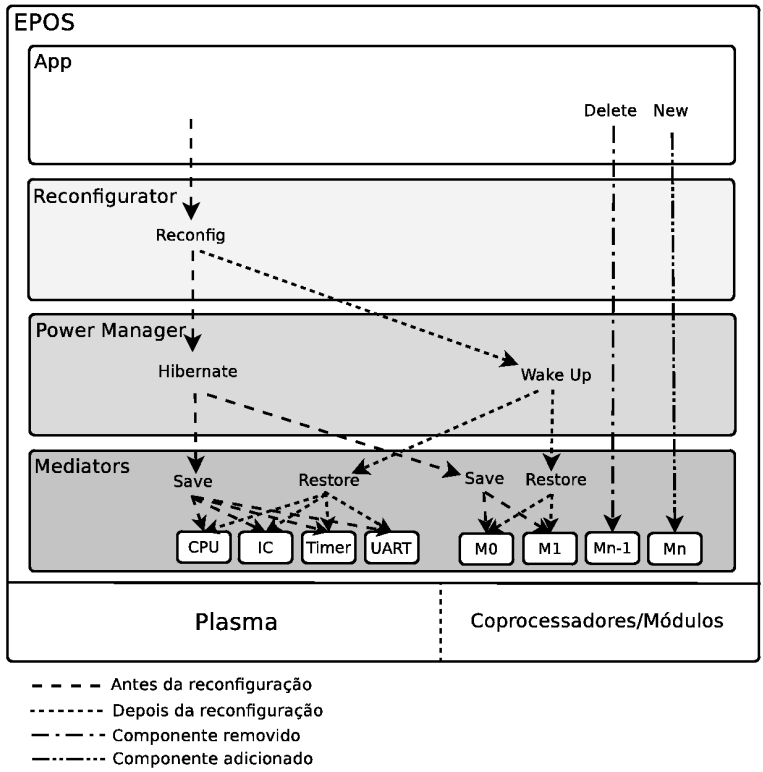

Figura 4.2: Fluxo de reconfiguração

Com a reserva de espaço concluída, primeiramente foi escrito o método CPU::save() que contém uma rotina em linguagem *assembly* responsável por salvar o contexto da UCP. Essa rotina é baseada na rotina de salvamento de contexto do MIPS e foi modificada para salvar em um endereço fora do EPOS, definido nas configurações do sistema. Logo após salvar os registradores, esse método copia todo o conteúdo da memória do sistema operacional para o espaço reservado e escreve 'EPOS' na palavra reservada que sinaliza um salvamento bem sucedido. Por fim, o método desliga o *bit* menos significativo do registrador de reconfiguração o que inicia imediatamente a cópia da *bitstream* contida na memória de configuração.

O *software* então entra em um estado latente, esperando que alguém

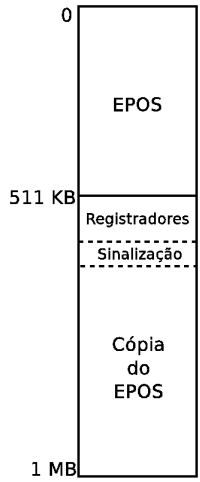

Figura 4.3: Mapa de memória

o acorde. Como, por causa da reconfiguração, não existe mais *software* executando, o Plasma então é reiniciado, forçando o *boot loader* a entrar em ação. Cabe a ele a tarefa de acordar o EPOS.

4.2.2 *Boot Loader*

Originalmente, o *boot loader* do Plasma ficava apenas escutando a porta serial, esperando um executável ser transferido e, em seguida, faz uma chamada ao ponto de entrada desse executável. Nesse novo caso, não quer-se receber outra imagem e sim recuperar uma execução suspensa. Para isso, no início de sua execução, verifica-se o endereço de memória no qual o EPOS sinaliza um salvamento e, em caso positivo, chama o método de recuperação do EPOS (Figura 4.4).

Esse método sinaliza que iniciou uma recuperação, sobrescrevendo esse endereço de memória e então, copia toda a memória previamente salva para o lugar original. Para finalizar, recupera os registradores da UCP, o que faz com que o fluxo de execução retorne para exatamente após o método de salvamento da UCP, entrando então na etapa de recuperação de informação dos mediadores.

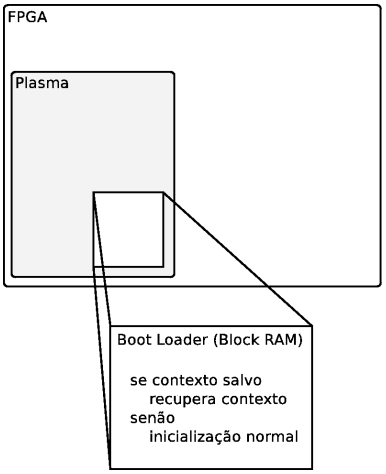

Figura 4.4: *Boot loader*

4.2.3 Integração com o Gerenciador de Energia

Como o processador não é o único *hardware* do sistema, os outros componentes de *hardware* também foram resetados e por isso também precisam voltar ao estado que se encontravam antes. No caso do Plasma, que possui um *hardware* bem simples, somente duas informações foram identificadas como necessárias para salvar: a máscara que define as interrupções ativas do controlador de interrupções e o valor atual do contador de ciclos de relógio. No caso de existirem outros componentes de *hardware*, eles também poderão ter informações salvas.

Apesar de não ser o caso do Plasma, mas já pensando em sistemas mais complexos utilizando esse mecanismo, procurou-se uma forma eficiente de percorrer os mediadores de *hardware* para efetuar o salvamento e recuperação de informações. E foi encontrado dentro do próprio EPOS, seu gerente de energia. Como visto no Capítulo 2, o gerente de energia do EPOS permite que sinais de troca de modo de operação sejam propagados pelo sistema. Para melhor servir ao propósito desse trabalho, ele foi adaptado para enviar sinais de hibernar e acordar para o sistema. Quando o método `reconfig` é invocado, duas chamadas para o método *power* que abrangem todo o sistema são feitas (Figura 4.5).

Para cada mediador que precisa ser hibernado, é criada uma entrada no gerente de energia, a Figura 4.6 mostra a entrada relativa ao *Timer* do

```
void Reconf::reconfig(int config_id)
{
    CPU::int_disable();
    Reconf::select_config(config_id);
    System.power(HIBERNATE);
    System.power(WAKEUP);
    CPU::int_enable();
}
```

Figura 4.5: Implementação do método de reconfiguração

Plasma. Esse método decide, baseado no modo passado como parâmetro, se invoca o método de salvamento ou de recuperação, ambos implementados no mediador do dispositivo. Seguindo com o exemplo do *Timer*, as Figuras 4.7 e 4.8 mostram como esses métodos foram implementados. Esses métodos fazem, apenas, as devidas cópias do valor do contador.

```
template<>
void Power_Manager<PLASMA_Timer>::power(int mode)
{
    switch(mode) {
        case Traits<PLASMA_Timer>::E0://HIBERNATE
            power_off();
            break;
        case Traits<PLASMA_Timer>::E3://WAKE UP
            power_full();
            break;
    }
}
```

Figura 4.6: Método `power` do temporizador

4.2.4 Ajustes no Plasma

Originalmente, o contador de ciclos de relógio do Plasma possuía apenas um registrador de leitura. Como se deseja que o estado da máquina

```
static void power_off() {
    unsigned int *timestamp =
              Traits<PLASMA_Reconf>::TIMESTAMP;
    volatile unsigned int *time =
              Traits<PLASMA_Timer>::DATA_ADDRESS;

    *timestamp = *time;
}
```

Figura 4.7: Método de salvamento do temporizador

```
static void power_full() {
    unsigned int *timestamp =
             Traits<PLASMA_Reconf>::TIMESTAMP;
    unsigned int *set_timer =
             Traits<PLASMA_Timer>::WRITE_ADDRESS;

     *set_timer = *timestamp;
}
```

Figura 4.8: Método de recuperação do temporizador

seja exatamente o mesmo antes e depois da reconfiguração, é necessário poder sobrescrever esse registrador com o valor antigo.

Para isso, um registrador de escrita foi adicionado para, a qualquer momento, poder modificar o valor atual do contador e fazê-lo seguir contando a partir desse valor. Isso foi implementado na etapa de decodificação de endereços do Plasma, caso o endereço de escrita do contador seja identificado, ao invés de incrementar o contador, o valor contido no endereço de escrita é armazenado. O novo *traits* do temporizador do Plasma no EPOS pode ser visto na Figura 4.9. Foram adicionadas também as informações referentes ao gerente de energia.

Para poder controlar o início da reconfiguração, FPGAs em geral possuem um pino externalizado que, quando setado, inicia o processo. Mas esse estímulo precisa vir de fora, não pode ser controlado interna-

```
template <>
struct Traits<PLASMA_Timer>:
                      public Traits<PLASMA_Common>
{
    //Power Management
    enum {
        HIBERNATE = 0,
        WAKEUP    = 1
    };
    static const char E0 = HIBERNATE;
    static const char E1 = HIBERNATE;
    static const char E2 = HIBERNATE;
    static const char E3 = WAKEUP;
    static const bool power = true;

    unsigned int FREQUENCY = CLOCK / (1<<18);
    unsigned int BASE_ADDRESS = 0x20000060;
    unsigned int DATA_ADDRESS = BASE_ADDRESS;
    unsigned int WRITE_ADDRESS = 0x20000070;
};
```

Figura 4.9: Configuração do temporizador do Plasma

mente. Para fazer o EPOS ter acesso a esse pino, foi criado um registrador que liga seu *bit* menos significativo a um pino de entrada e saída não utilizado pelo Plasma. Para fechar o circuito, um fio externo foi usado para conectar os dois pinos. Uma representação dessa ligação pode ser vista na Figura 4.10.

4.2.5 Inserção e Remoção de Mediadores

Alterações no *hardware* afetam o *software* quando um componente de *hardware* é inserido ou removido do sistema. No caso de modificações em um módulo já existente, desde que não seja na sua interface, talvez não precise de alterações no *software*, mas esse nem sempre é o caso.

Como estamos pensando em sistemas embarcados, a aplicação é a parte mais importante do sistema e sendo assim, considerou-se que cabe a ela controlar as mudanças no *software*. Isso é feito através da inserção ou remoção de mediadores que fazem a interface com esses componentes

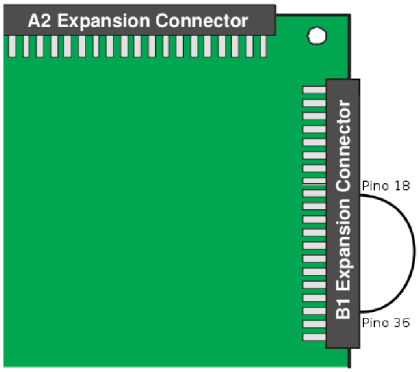

Figura 4.10: Ligação externa para controle de reconfiguração

de *hardware*. Esse gerenciamento de mediadores também pode ser feito por abstrações do sistema, no caso delas acessarem o componente e não a aplicação.

Quando acontece uma inserção de componente, o sistema operacional precisa de um mediador, uma classe C++ que estabelece uma interface entre o sistema operacional e o componente de *hardware*, para se comunicar com ele. Isso porque é o mediador quem sabe onde e como acessar o componente. Para isso, a aplicação (ou abstração) instancia um mediador para um componente adicionado e, em após uma remoção, apaga esse mediador para evitar acessos indevidos. Deixar as operações de inserção e remoção a cargo da aplicação é plausível devido a ela ser a usuária dos componentes, portanto ela é quem melhor sabe, dentro do sistema, o que alterar no *software*. Além disso, como vários componentes podem entrar e sair do sistema em uma mesma reconfiguração, reconhecer todos esses componentes seria uma tarefa muito onerosa para um sistema operacional embarcado.

Esse mediador precisa ser implementado em tempo de desenvolvimento do sistema para formar um par juntamente com o componente de *hardware*. Para facilitar essa implementação, um mediador de referência é fornecido. Esse mediador implementa operações básicas para envio e recebimento de dados que são herdadas pelo mediador específico, sendo necessário somente especificar o endereço pelo qual se acessa o componente, ou adicionar novas operações caso seja necessário. Isso é possível devido ao Plasma se comunicar com os componentes exclusivamente por

entrada e saída mapeada em memória.

O termo mediador é usado para se referir a um artefato de *software* mais parecido com um *driver*. Mas diferentemente de *drivers*, mediadores simplesmente estabelecem uma interface *software/hardware* e, por serem meta-programados, se dissolvem no sistema em tempo de compilação. A semelhança desses mediadores dinâmicos com os mediadores estáticos do EPOS está no fato de serem apenas classes C++ que são agregadas à aplicação. Assim, mediadores estáticos são compilados juntamente com o sistema e podem ser inseridos e removidos utilizando o esquema de indireção provido pelo ELUS [Gracioli e Frohlich 2008]. O código do mediador de referência pode ser visto na Figura 4.11.

```
class Dynamic_Mediator
{
private:
    static unsigned int io_address;
public:
    Dynamic_Mediator(unsigned int addr)
    {
        io_address = addr;
    }
    ~Dynamic_Mediator() {}
    unsigned int get_data()
    {
        unsigned int *mem =
                        (unsigned int *)io_address;
        return *mem;
    }
    void put_data(unsigned int data)
    {
        unsigned int *mem =
                        (unsigned int *)io_address;
        *mem = data;
    }
};
```

Figura 4.11: Mediador de referência

Capítulo 5

Avaliação e Resultados

Este capítulo apresenta uma avaliação da implementação proposta em termos de aumento no tamanho final do binário gerado e tempo necessário para efetuar uma reconfiguração do sistema. Para a avaliação foi utilizado o processador *softcore* Plasma sintetizado para o FPGA Spartan-3 da Xilinx.

5.1 O FPGA Spartan-3 da Xilinx

Neste trabalho o FPGA Spartan-3 da Xilinx foi utilizado para implementar e testar a solução proposta. Essa família foi desenvolvida em 2003 para dar seqüência às famílias de baixo custo da Xilinx. Foi o primeiro FPGA com processo de fabricação de 90 nm e, quando lançado, era o FPGA de menor custo por porta lógica [Xilinx 2006].

O modelo utilizado foi o XC3S200, que equivale em tamanho a 200.000 portas lógicas, segundo menor dentre os modelos disponíveis. Possui 4320 células lógicas, 1920 *Slices*, 12 *Block* RAMs de 18 k*bit* cada, 12 multiplicadores em *hardware* de 18x18 *bits* e 173 sinais para entrada e saída. O *kit* utilizado (Figura 5.1) também possui 1 M*bit* de memória SRAM e 2 M*bit* de memória *Flash* para armazenar *bitstreams*, como pode ser visto na Figura 5.1. Para efeito de comparação, um FPGA da última geração da Família (Spartan-6), possui 147.443 células lógicas, 23.038 *Slices* e 268 *Block* RAMs de 18 k*bit* cada.

Sintetizando o Plasma para o modelo de Spartan-3 utilizado, obtemos as estatísticas apresentadas na Tabela 5.1.

Tabela 5.1: Utilização da Spartan-3 pelo Plasma

Característica	Usado	Disponível	Utilização
Número de *Slice Flip Flops*	393	3840	10%
Número de LUTs de 4 entradas	2573	3840	67%
Número de *Slices* ocupados	1465	1920	76%
Total de LUTs de 4 entradas	2843	3840	74%
Equivalente em *gates*	298714	-	-

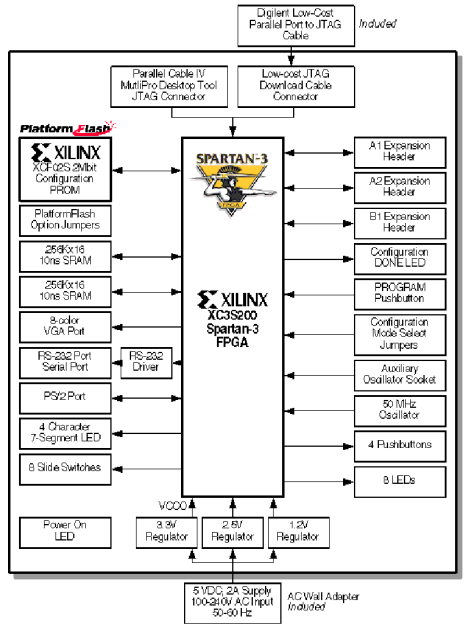

Figura 5.1: Diagrama de blocos do *kit* de desenvolvimento usado

5.2 Tamanho do *Software*

Primeiramente, foi avaliado o aumento do tamanho do executável gerado com a inserção do suporte para reconfiguração. Essa avaliação se dá através da comparação com o tamanho do EPOS original. As duas versões do EPOS foram geradas para o processador Plasma utilizando o compilador *mips-gcc* versão 4.0.2.

Como a aplicação é gerada juntamente com o restante do sistema, uma aplicação sintética que apenas mapeia os botões para os LEDs presentes na placa foi utilizada para interferir o menos possível no tamanho final do sistema. Para forçar o gerador do EPOS a incluir o suporte para reconfiguração, uma chamada para o método que inicializa uma reconfiguração foi incluído logo antes do código de mapeamento dos botões. Os valores obtidos podem ser vistos na Tabela 5.2. Nota-se que a inserção do suporte aumentou o tamanho do executável em 22,3%, como era esperado, mas esse acréscimo de forma alguma inviabiliza a utilização da técnica em um sistema embarcado.

Nesses valores não está incluso o tamanho do *boot loader* do Plasma.

Tabela 5.2: Tamanho do binário gerado (bytes)

Versão	.text	.data	.bss	Total
EPOS	16692	68	221	16981
EPOS Reconfigurável	20532	68	181	20781

Isso porque o Plasma possui um *boot loader* estático que é armazenado dentro do próprio Plasma durante o processo de síntese. Cada modificação no *boot loader* acarreta uma nova síntese do processador.

Devido a essa exigência, o *boot loader* fica à parte e, até o presente momento, não foi integrado ao EPOS e por isso é apresentado separadamente. Os tamanhos do *boot loader* original e com suporte são apresentados na Figura 5.2.

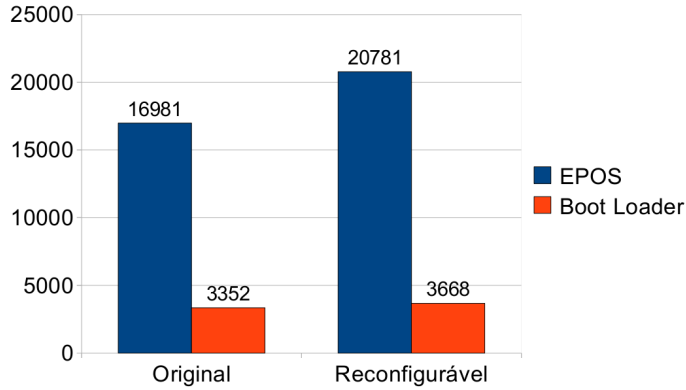

Figura 5.2: Tamanho do EPOS e *boot loader* (bytes)

5.3 Tempo de Reconfiguração

O aspecto mais crítico para ser avaliado é o tempo de reconfiguração, que corresponde ao tempo em que a aplicação precisa ficar parada para completar o processo. Dependendo da aplicação, um tempo muito alto implica na impossibilidade de utilizar o mecanismo proposto.

Para essa medição, primeiramente foram sintetizados dois Plasmas, um com esforço máximo de otimização em relação à área durante a síntese e outro focando em velocidade. Isso foi feito para ressaltar que apesar ser

o mesmo processador, as *bitstreams* geradas são completamente diferentes e nada do outro processador pode ser aproveitado para facilitar o processo. Para se ter uma idéia dessa diferença, o tamanho máximo de uma *bitstream* para o Spartan-3 é de 128 kB, e a *bitstream* parcial gerada através da diferença dessas duas versões do Plasma é de 127,9 kB.

O tempo de reconfiguração pode ser dividido em tempo de *software* e de *hardware*. O tempo de *software* é o tempo necessário para realizar o salvamento e recuperação do estado de execução, ou seja, é a sobrecarga imposta pelo mecanismo. O tempo de *hardware* é o tempo de carregar uma *bitstream* da memória *Flash* de configuração para o FPGA. Esse tempo de *hardware* não é influenciado diretamente pelo mecanismo proposto, mas o é indiretamente pelo fato de se utilizar o processo de diferenciação na geração de *bitstreams* parciais, o que não garante *bistreams* menores do que as geradas pelo outro processo.

O tempo de *software* foi medido separando-o em três partes, contexto do processador, mediadores e conteúdo da memória RAM. O contexto do processador consiste de seus 34 registradores e o ponteiro da pilha de execução. Os mediadores inclusos no processo são: o controlador de interrupções, de onde se salva a máscara de interrupção e o contador de ciclos de relógio, de onde se salva o valor atual do contador. Em relação à memória RAM, todo espaço gerenciado pelo EPOS é salvo, o que totaliza 376 kB. A Figura 5.3 mostra os valores obtidos com a média de mil medições.

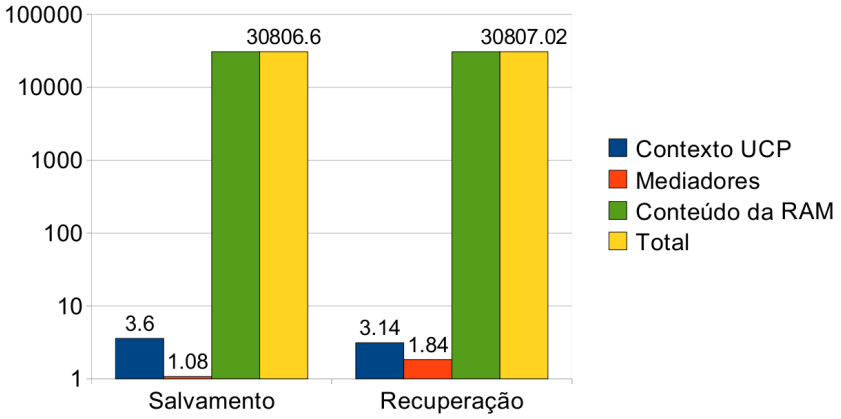

Figura 5.3: Tempo de salvamento e recuperação (em μs)

Nota-se que a cópia da memória RAM é o fator que mais influencia no valor final e depende diretamente da quantidade de memória salva. Cabe ressaltar que o EPOS não utiliza completamente esses 376 kB, essa é a memória que ele mapeia e existem espaços vazios entre as seções criadas. Esse valor pode ser otimizado de duas maneiras, diminuindo o mapa de memória do EPOS caso a aplicação permita ou descobrindo o quê, dentro do tamanho total, está realmente sendo utilizado. Essa última alternativa pode ser custosa pois essa descoberta deverá ser feita a cada salvamento, o que pode não compensar o tempo gasto salvando toda a memória.

No caso de aplicações que demandem muita memória RAM, como no caso de aplicações multimídia, não é necessário copiar de toda a aplicação, o que inviabilizaria esse tipo de reconfiguração. Basta fazer o salvamento da área de memória do sistema operacional; a área correspondente a aplicação permanecerá intacta e poderá ser acessada novamente ao término da reconfiguração. Isso é possível porque uma reconfiguração não altera a maneira como o *software* se organiza dentro da memória, sendo apenas necessário que o sistema operacional seja atualizado em relação aos estados anteriores dos componentes de *hardware* que foram reiniciados. Assim, o tempo de salvamento obtido com os 376 kB do EPOS pode ser considerado válido mesmo quando utilizado com aplicações mais complexas.

O tempo gasto com o salvamento de mediadores foi pequeno devido também à simplicidade dos mediadores necessários para o Plasma. No caso de sistemas com *hardware* mais complexo, mais informações terão de ser armazenadas, mas como se trata de sistemas embarcados, a complexidade dos componente de *hardware* dificilmente influenciará de maneira que impossibilite a utilização desse mecanismo.

O processo de reconfiguração do *hardware* é completamente à parte do sistema operacional. Como não existem locais pré-definidos para se encaixar módulos, o processo fica simplificado, bastando aplicar a *bitstream* contida na memória não-volátil. Assim, o EPOS entra em ação somente sinalizando o início do processo.

Para obter o tempo de *hardware* foi medido, com o auxílio de um osciloscópio, o tempo decorrido entre o sinal enviado pelo EPOS e o sinal interno do FPGA que indica uma reconfiguração bem sucedida. Essa medição não necessitou muitas repetições pois o tempo manteve-se constante. A reconfiguração realizada foi do Plasma otimizado para área para o Plasma otimizado para velocidade. Como o tamanho máximo do *bitstream* é de 128 kB e o *bitstream* utilizado tem 127,9 kB, o tempo obtido é bastante próximo do pior caso para o Spartan-3. O tempo obtido pode ser visto, juntamente com o tempo de *software*, na Figura 5.4.

Analisando o impacto da parte de *software* e de *hardware* no tempo

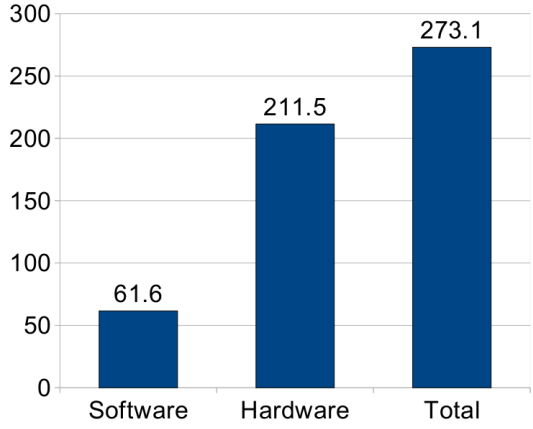

Figura 5.4: Tempo de reconfiguração (em ms)

total, nota-se que o tempo de reconfiguração de *hardware* influi mais, pelo menos no seu pior caso. Como é impossível prever o tamanho de uma *bitstream* gerada através de diferenciação, é prudente levar em conta sempre o pior caso.

Considerando os 273 milissegundos, pode-se dizer que nos casos em que as atualizações de *hardware* são interativas ou esporádicas, é perfeitamente plausível esperar esse tempo para ter a aplicação funcionando novamente. Em aplicações mais exigentes, que necessitem modificar o *hardware* diversas vezes para realizar um processamento, é necessário fazer uma análise mais profunda. Por exemplo, no caso de Rádio Definido por *Software* (SDR - *Software-defined Radio*), parar por 273 milissegundos significa ficar esse tempo armazenando amostras. Como sistemas embarcados possuem recursos limitados, dependendo da taxa de amostragem, a memória necessária pode ser incompatível com a realidade do sistema.

Em uma aplicação de SDR utilizada no laboratório para recepção de sinais FM, são geradas 6 milhões de amostras de 32 bits por segundo, o que gera um fluxo de 23 MB/s. Em 273 milissegundos, são gerados 6,2 MB de dados, um valor nada absurdo para sistemas embarcados. Mas também é necessário analisar se esse tempo parado não vai interferir em um possível *pipeline* de execução da aplicação.

Considerando-se aplicações com requisitos de tempo real, as chances de se poder utilizar o modelo proposto com sucesso são bem menores.

A sobrecarga presente em cada reconfiguração dificultaria a capacidade do sistema operacional em honrar os prazos se, para isso, for necessário efetuar uma reconfiguração. Como essa abordagem é bastante dependente do processador e da aplicação, é necessário analisar caso a caso sua viabilidade, não pode-se chegar a uma estimativa de pior caso genérica.

Extrapolando os resultados obtidos para um FPGA de maior porte, como o Virtex-II utilizado no OS4RS [Nollet et al. 2003], podemos ter uma noção de como essa abordagem se comportaria em um sistema real. Se obtivermos o pior caso para uma *bitstream* parcial, essa *bitstream* teria 2667 kB (tamanho total desse FPGA) e demoraria 499,4 ms para ser configurada. Pode-se também chegar ao caso ótimo, pegando a menor *bitstream* parcial possível para determinado módulo. No exemplo utilizado, OS4RS insere um módulo, cuja *bitstream* parcial possui 534 kB, em 100 ms. Como essa *bitstream* parcial representa a menor diferença possível, o tempo para transformar uma configuração somente com um Plasma para uma com o Plasma conectado ao referido módulo ficaria entre 161,6 e 561 ms, já contando o tempo necessário para tratar a parte de *software* do sistema. Como já explicado, esse tempo teria um pequeno acréscimo caso informações de estado de execução internas desse módulo precisem ser salvas, diferentemente de possíveis dados que ele utilize e que estejam residentes na memória.

5.4 Sobrecusto de *Hardware*

Na abordagem proposta, não existe sobrecusto de *hardware* em termos de infraestrutura de comunicação, já que toda a lógica contida no sistema é realmente necessária para a configuração atual. O que pode ser considerado um sobrecusto de *hardware* são as modificações efetuadas no Plasma: adição do contador programável e do registrador de reconfiguração.

Como pode ser visto na Figura 5.5, a adição de características para habilitar o Plasma a suportar reconfiguração o aumentou em 2,3% o que, em termos de utilização da Spartan-3, representa 1%. Esse acréscimo pode ser considerado mínimo e não interfere significativamente em sua utilização, mesmo em FPGAs de pequeno porte.

5.5 Estudo de Caso: Criptografia em *Hardware*

Para testar a reconfiguração de um módulo de *hardware*, foi desenvolvido um módulo de criptografia. Esse módulo instancia e gerencia as 8 *Block* RAMs restantes no FPGA como se fosse uma memória única de 16 kB, para armazenar os dados criptografados. Esse módulo possui um registrador que armazena a chave e é acessado através de um endereço

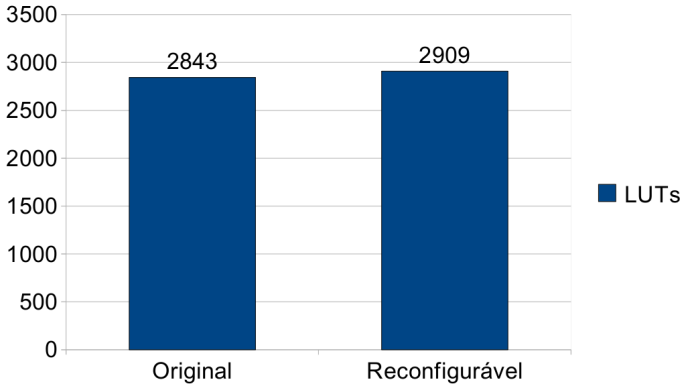

Figura 5.5: Tamanho do Plasma

pré-definido. Quando um dado chega no módulo, é feita uma operação de ou-exclusivo entre o dado e a chave, e em seguida, o resultado é armazenado na memória. A Figura 5.6 mostra o diagrama de blocos do sistema.

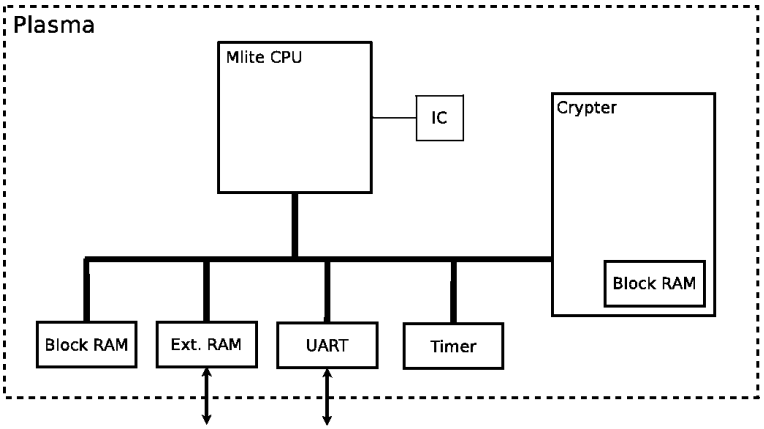

Figura 5.6: Diagrama de blocos do Plasma com criptografia

Com o módulo de criptografia, o tamanho total do sistema aumentou 7,9%, totalizando 3141 LUTs e utilizando 81% do FPGA. Este exemplo, apesar de pequeno e não refletir uma aplicação real de um sistema

reconfigurável, permite mostrar que a abordagem proposta é funcional e merece um estudo mais aprofundado. Para reconfiguração, o tamanho da *bitstream* obtida foi 127,9 kB que, como visto no teste anterior, precisa de 211,5 ms para reconfigurar o FPGA. O fato dessa *bitstream* parcial possuir o mesmo tamanho da *bitstream* parcial obtida pela diferenciação de duas versões do Plasma é meramente coincidência, mas reforça que imprevisibilidade da diferenciação tende a produzir *bitstreams* parciais próximas ao pior caso.

Para controlar o módulo pela aplicação, um mediador foi desenvolvido. Esse mediador possui 6 operações: ler e escrever chave, ler e escrever um dado de 32 bits, ler e escrever um vetor de dados de 32 bits. As implementações desses métodos são bastante simples, pois acessam o módulo como se fosse uma memória comum. Essa memória é mapeada a partir do endereço 0x30000000 e estende-se até o endereço 0x30000FFF, permitindo a criptografia de um vetor de até 4096 posições de cada vez. Já chave de criptografia fica mapeada no endereço 0x38000000. Um exemplo de utilização dessas operações pode ser visto na Figura 5.7.

```
Crypter crypt;
// especifica chave de criptografia
crypt.put_key(0x42424242);
// criptografa 4096 elementos do vetor data
crypt.put_data(data, 4096);
// recupera 4096 elementos criptografados
crypt.get_data_vector(data, 4096);
```

Figura 5.7: Exemplo de uso do criptografador

A adição desse mediador, juntamente com uma aplicação que realiza a criptografia de um vetor de 4096 elementos logo após reconfigurar o Plasma por um Plasma com módulo de criptografia, fez o tamanho do EPOS aumentar 2,3% em relação à sua versão reconfigurável, totalizando 21265 bytes, sendo 21012 em *.text*, 72 em *.data* e 181 em *.bss*.

Traçando-se um paralelo com reconfiguração parcial modular, a única diferença substancial estaria no tempo de reconfiguração. Enquanto a abordagem proposta reconfigurou praticamente todo o FPGA, demorando 211,5 ms, a inserção somente do módulo de criptografia precisaria de 16,7 ms para ser configurado. Além disso, o tempo para salvar e recuperar o *software* também não seria necessário, reduzindo o tempo de 273,1

ms para 16,7 ms.

5.6 Comparação com Trabalhos Relacionados

Esta seção visa comparar os resultados obtidos com o a implementação desse trabalho com os trabalhos apresentados no Capítulo 3. Essa comparação foi dividida em tamanho da parte de *software*, tempo de reconfiguração e sobrecusto de utilização do *hardware*. Cabe ressaltar que nem todos os trabalhos apresentaram números como resultado, ou não apresentaram números de uma das partes do sistema.

5.6.1 Tamanho do *Software*

Os trabalhos que apresentaram tamanhos de *software* o fizeram de forma genérica, sem especificar uma aplicação. Como o EPOS precisa de uma aplicação para ser gerado, uma aplicação com pequeno impacto no tamanho final foi utilizada na seção 5.2 e, portanto, é um valor válido para comparar com os trabalhos a seguir.

O trabalho proposto por Ullmann et al. [Ullmann et al. 2004] tem como tamanho do ambiente de execução 6,3 kB e necessita de mais 1,8 kB para executar, totalizando um *footprint* de 8,1 kB. Vale lembrar que esse sistema não é um sistema operacional e tem como função apenas gerenciar a reconfiguração do *hardware* e trocas de mensagem.

OS4RS, um extensão reconfigurável e tempo real para Linux, apresenta como tamanho total do sistema 500 kB. Apesar de possuir funcionalidades não presentes no EPOS, como permitir que uma tarefa migre de *software* para *hardware* e vice-versa, ter um tamanho quase 25 vezes maior que o EPOS não é justificado. Os outros sistemas baseados em Linux não apresentam o tamanho do sistema, mas por terem o mesmo sistema operacional como base, devem possuir um tamanho parecido com OS4RS. Esses valores podem ser visualizados na Figura 5.8.

5.6.2 Tempo de Reconfiguração

Apenas dois trabalhos apresentaram resultados tanto para o *hardware* quanto para o *software*. O que obteve os melhores resultados foi o trabalho proposto por Santambrogio, Rana e Sciuto [Santambrogio, Rana e Sciuto 2008], utilizando um FPGA Virtex-II da Xilinx. O tempo necessário para reconfigurar um módulo variou entre 1,47 e 5,82 milissegundos com um adicional de 1,05 milissegundos para cada BlockRAM utilizada. A sobrecarga adicionada pela camada de *software* totalizou 4,6 milissegundos, sendo dividido em 0,5 milissegundos para iniciar o *Daemon*, 0,65 milissegundos para iniciar o *driver* e 3,45 milissegundos para carregar o módulo. Como o *Daemon* só precisa ser iniciado uma vez, a partir da

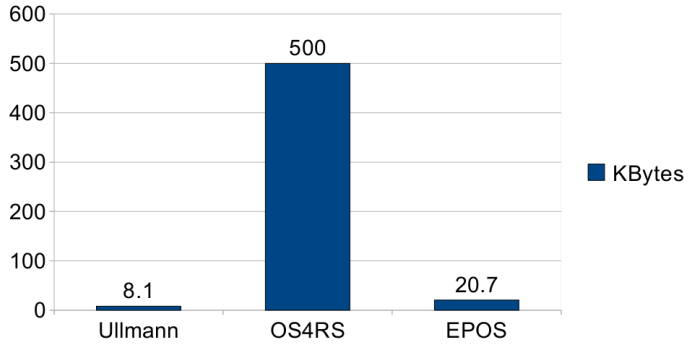

Figura 5.8: Comparação de tamanho do software com trabalhos relacionados

segunda reconfiguração, a sobrecarga cai para 4,1 milissegundos. Se somarmos os piores casos de *hardware* e *software* obtemos um tempo de reconfiguração total de 10,42 milissegundos.

Outro trabalho que também apresentou as sobrecargas separadamente foi o OS4RS [Nollet et al. 2003]. Também utilizando um FPGA Virtex-II, o tempo necessário para encaixar um módulo na NoC fica entre 10 e 100 milissegundos, sendo que o tamanho máximo da *bitstream* parcial é 534 kB (de um total de 2667 kB). A sobrecarga imposta pelo sistema operacional é de 0,1 milissegundo. O tempo total de reconfiguração apresentado pelo artigo é de 108 milissegundos, mas não é explicado de onde surge essa diferença com soma dos tempos de *hardware* e *software*.

No trabalho de Walder e Platzner [Walder, Steinegger e Platzner 2004], que também utilizou um FPGA Virtex-II, o tempo de reconfiguração do *hardware* ficou entre 2,83 e 14,1 milissegundos para tarefas de tamanho 1 e tarefas de tamanho 5, respectivamente. O tamanho da tarefa corresponde ao número de *slots* que ela ocupa, sendo que cada *slot* tem 2048 *slices*. Para efeito de comparação, o Plasma reconfigurável possui 1500 *slices*. Já o trabalho de Deng et al. [Deng et al. 2005], que utiliza um Virtex-4, possui *slots* que acomodam módulos com *bitstream* parcial de 350 kB e para reconfigurá-los são necessários 70 milissegundos. Infelizmente, esses trabalhos não apresentam a sobrecarga imposta pela camada de *software*. Esses valores podem ser vistos na Figura 5.9.

Acredita-se que a grande diferença entre o tempo de reconfiguração desses trabalhos e do trabalho proposto seja devido à utilização de FPGAs mais modernos e com freqüências de relógio maiores. Enquanto a Spartan-3 trabalha a 50 MHz, as Virtex-II e 4 possuem uma freqüência

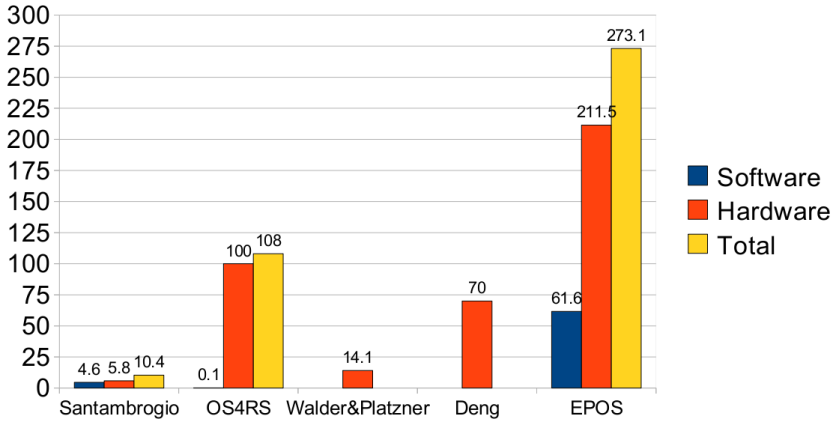

Figura 5.9: Comparação de tempo de reconfiguração com trabalhos relacionados

de 100 MHz, o que dobra a velocidade de configuração. Como esperado, a sobrecarga do sistema operacional na reconfiguração é muito menor em sistemas baseados em módulos pois não há a necessidade de realizar salvamento e recuperação de contexto.

5.6.3 Sobrecusto de *Hardware*

Talvez por não considerarem uma desvantagem ou considerarem um mal necessário, os autores não se preocuparam em apresentar os tamanhos das infraestruturas de *hardware* necessárias para o funcionamento do sistema. Exceto por Ullmann et al., que obteve um sistema com 1754 *slices*, mas como nesse número está incluso o processador *softcore*, não se tem como saber o quanto disso é sobrecarga de *hardware*. No trabalho de Walder e Platzner, apenas o tamanho da MMU é apresentado e ocupa 1262 *slices*, mas sabendo que somente uma parte tem quase o tamanho do Plasma inteiro, imagina-se que esse sistema necessita FPGAs com grande capacidade.

5.7 Possíveis Otimizações

Como foi visto, o tempo de reconfiguração obtido ficou bem acima dos tempos mostrados nos trabalhos relacionados. Buscou-se, então, otimizações que poderiam ser efetuadas de modo a diminuir essa diferença.

Considerando a parte de *software* do tempo de reconfiguração, uma otimização possível é diminuir a quantidade de informações que precisa

ser copiada em cada reconfiguração. Na implementação atual, toda a área de memória reconhecida pelo EPOS é salva, totalizando 376 kB. Mas dentro desse espaço podem existir espaços vazios, como por exemplo, memória *Heap* que ainda não foi alocada. O EPOS reconfigurável possui aproximadamente de 20 kB, somando-se hipotéticos 5 kB utilizados por variáveis obtém-se um total de 25 kB que necessitaria de apenas 4,1 milissegundos para ser copiado. Assim, otimizando-se o mecanismo de salvamento e recuperação para ignorar os espaços vazios, pode-se diminuir em torno de 55 milissegundos o tempo gasto para salvar a parte de *software* do sistema.

Já pela parte de *hardware*, o simples fato de substituir o FPGA por um modelo mais moderno melhoraria o resultado. Considerando os números apresentados nos trabalhos relacionados, a velocidade de reconfiguração obtida com os FPGAs Virtex-2 e Virtex-4 foram de aproximadamente 5 kB/ms enquanto que a velocidade obtida com o Spartan-3 foi de 0,6 kB/ms. Se utilizarmos a mesma velocidade obtida por outros trabalhos no sistema proposto, a *bitstream* parcial de 127,9 kB demoraria 25,6 milissegundos, o que representa uma redução de 88%.

Com apenas uma otimização de *software* e com a utilização de equipamento mais moderno pode-se diminuir o tempo de reconfiguração em mais de 89%, reduzindo de 273,1 para 29,7 milissegundos. Isso faria a implementação se equiparar com os trabalhos relacionados e assim poderia servir como base para uma maior variedade de sistemas.

Capítulo 6

Conclusões e Trabalhos Futuros

Reconfiguração dinâmica de *hardware* permite que módulos com atividades específicas entrem e saiam do sistema a mando da aplicação. Mas para tirar proveito dessas possibilidades sem inviabilizar o projeto devido a custos e tempo de desenvolvimento, é necessária a utilização de um sistema que promova o reuso de componentes de *software* e de *hardware* e que facilite o desenvolvimento afastando o desenvolvedor da complexidade de um sistema desse tipo.

Este trabalho propõe um suporte de sistema operacional para reconfiguração de *hardware* como uma abstração de sistema, permitindo que essa operação seja controlada pela aplicação de forma simples. Isso permite que o desenvolvedor do sistema se concentre no desenvolvimento de componentes de *hardware* e seus respectivos mediadores. Esses mediadores são implementados através da especificação de operações, já implementadas em um mediador de referência, para o componente em questão.

Esse suporte baseia-se no gerenciador de energia do EPOS, que permite a propagação de mensagens de troca de estado de componentes do sistema, utilizado para realizar salvamento e recuperação do estado do sistema. Isso é necessário pois, para simplificar a fase de projeto do *hardware*, não se utiliza reconfiguração modular. Assim, este trabalho propõe também que reconfiguração dinâmica baseada em diferenciação de *bitstreams* é viável para desenvolvimento de sistemas embarcados reconfiguráveis.

Isso é possível utilizando esse mecanismo de salvamento e recuperação de estado que evita os problemas oriundos da imprevisibilidade da geração de *bitstreams*. Dessa maneira, sistemas embarcados reconfiguráveis podem ser projetados sem preocupação com estruturação da comunicação inter-módulo, como barramentos ou redes de interconexão no FPGA. Isso simplifica o projeto, economiza espaço e faz com que a implementação seja portável, não dependendo do modelo do FPGA no qual será aplicado. Em contrapartida, as *bitstreams* parciais obtidas são em geral maiores do que *bitstreams* que possuem somente um módulo e, como visto nos testes, muitas vezes se equiparam em tamanho com *bitstreams* completas, problema que pode piorar à medida que se use FPGAs de maior capacidade.

Assim, utilizando-se do reuso de componentes e da portabilidade de *software* providos pelo sistema operacional juntamente com a simpli-

ficação da etapa de projeto e a portabilidade de *hardware* obtidos com a reconfiguração parcial baseada em diferença, é possível diminuir tempo e custos de desenvolvimento. Com isso, pode-se facilitar o estabelecimento de sistemas embarcados reconfiguráveis como solução viável para suprir as necessidades de sistemas embarcados convencionais atualmente disponíveis.

Este trabalho iniciou com uma investigação sobre a possibilidade de utilizar reconfiguração baseada e diferença [Reis e Fröhlich 2009] e originou-se de um projeto com o objetivo de transformar o EPOS em sistema operacional para sistemas embarcados reconfiguráveis que permitisse que componentes pudessem ser implementados tanto em *software* quanto em *hardware*. Desse modo, o sistema poderia ser modificado em tempo de execução permitindo que tarefas migrem entre *software* e *hardware*, alcançando uma melhor adaptação às necessidades da aplicação. Esse projeto teve como ponto de partida a arquitetura de componentes híbridos desenvolvida por Marcondes [Marcondes 2009]. Dando seqüência, Gracioli propôs um suporte para reconfiguração dinâmica de componentes de *software* [Gracioli 2009] para o EPOS. Juntamente com o presente trabalho, têm-se para o sistema operacional EPOS as três peças fundamentais para se chegar a um sistema completamente reconfigurável, independente de arquitetura e que permite a reconfiguração da aplicação em múltiplas plataformas. Esse trabalho também utilizou-se do trabalho desenvolvido por Júnior [Júnior 2007], como explicado na Seção 2.2.1.

Além de estar inserido nesse contexto, imagina-se que outra possível continuidade para este trabalho seja a criação de sistemas capazes de realizar multiprocessamento heterogêneo não-simultâneo. Em um sistema desse tipo, uma mesma aplicação pode ser executada por dois ou mais processadores diferentes que se alternam durante sua execução para melhor atender às necessidades de processamento e utilização da malha configurável. Ou seja, no caso da aplicação necessitar um componente que, por exemplo, não caiba juntamente com determinado processador, esse pode dar lugar a um processador menor e mais simples que seja capaz de gerenciar tal componente. A comunicação entre os processadores ficaria por conta do mecanismo de hibernação que faz persistir informações durante uma troca de processador. Outra possível utilização para um sistema desse tipo seria a troca entre processadores com velocidades de operação diferentes para diminuir o consumo de energia em períodos de menor demanda de poder de processamento.

Referências Bibliográficas

[Abnous et al. 1998]ABNOUS, A. et al. Evaluation of a Low-Power Reconfigurable DSP Architecture. *Lecture Notes in Computer Science*, Springer, v. 1388, p. 55–60, 1998.

[Carver et al. 2008]CARVER, J. et al. *Relocation and Automatic Floorplanning of FPGA Partial Configuration Bit-Streams*. [S.l.], 2008.

[Compton e Hauck 2002]COMPTON, K.; HAUCK, S. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, v. 34, n. 2, p. 171–210, 2002.

[Deng et al. 2005]DENG, Q. et al. A reconfigurable RTOS with HW/SW co-scheduling for SOPC. In: IEEE COMPUTER SOCIETY. *Proceedings of the Second International Conference on Embedded Software and Systems*. [S.l.], 2005. p. 121.

[Durbano et al. 2004]DURBANO, J. et al. FPGA-Based Acceleration of the 3D Finite-Difference Time-Domain Method. In: IEEE COMPUTER SOCIETY WASHINGTON, DC, USA. *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. [S.l.], 2004. p. 156–163.

[Erlandsson 2009]ERLANDSSON, M. *OpenRISC 1000: Openrisc 1200*. [S.l.]: Open Cores, 2009. `http://www.opencores.org/?do=project\&who=or1k\&page=openrisc%201200`.

[Estrin et al. 1963]ESTRIN, G. et al. Parallel Processing in a Restructurable Computer System. *IEEE Transactions on Electronic Computers*, v. 12, n. 5, p. 747–755, 1963.

[Fröhlich 2001]FRöHLICH, A. A. *Application-Oriented Operating Systems*. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001. 200 p. ISBN 3-88457-400-0.

[Gaisler Research 2005]Gaisler Research. *LEON2 XST User's Manual*. [S.l.: s.n.], 2005.

[Garcia et al. 2006]GARCIA, P. et al. An Overview of Reconfigurable Hardware in Embedded Systems. *EURASIP Journal on Embedded Systems*, v. 3, p. 4, 2006.

[Gonzalez, Aguayo e Lopez-Buedo 2007]GONZALEZ, I.; AGUAYO, E.; LOPEZ-BUEDO, S. Self-Reconfigurable Embedded Systems on Low-Cost FPGAs. *IEEE Micro*, IEEE Computer Society, p. 49–57, 2007.

[Gracioli 2009]GRACIOLI, G. *Reconfiguração Dinâmica de Software em Sistemas Profundamente Embarcados*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2009.

[Gracioli e Frohlich 2008]GRACIOLI, G.; FROHLICH, A. An Operating System Infrastructure for Remote Code Update in Deeply Embedded Systems. In: *Proceedings of the First ACM Workshop on Hot Topics in Software Upgrades (HotSWUp)*. [S.l.: s.n.], 2008.

[Júnior 2007]JúNIOR, A. *Gerência do Consumo de Energia Dirigida pela Aplicação em Sistemas Embarcados*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2007.

[Kean 2007]KEAN, T. *Algotronix History*. [S.l.]: Algotronix, 2007. http://www.algotronix.com/people/tom/album.html.

[Kuzmanov, Gaydadjiev e Vassiliadis 2004]KUZMANOV, G.; GAYDADJIEV, G.; VASSILIADIS, S. The Molen Processor Prototype. In: *Proceedings of 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. [S.l.: s.n.], 2004. p. 296–299.

[Lodi, Toma e Campi 2003]LODI, A.; TOMA, M.; CAMPI, F. A Pipelined Configurable Gate Array for Embedded Processors. In: ACM NEW YORK, NY, USA. *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*. [S.l.], 2003. p. 21–30.

[Lu et al. 1999]LU, G. et al. The Morphosys Parallel Reconfigurable System. *Lecture Notes in Computer Science*, Springer, p. 727–734, 1999.

[Majer et al. 2007]MAJER, M. et al. The Erlangen Slot Machine: A dynamically reconfigurable FPGA-based computer. *The Journal of VLSI Signal Processing*, Springer, v. 47, n. 1, p. 15–31, 2007.

[Mangione-Smith e Hutchings 1997]MANGIONE-SMITH, W.; HUTCHINGS, B. Configurable Computing: The Road Ahead. *Reconfigurable Architectures: High Performance by Configware*, p. 81–96, 1997.

[Marcondes 2009]MARCONDES, H. *Uma Arquitetura de Componentes Híbridos de Hardware e Software para Sistemas Embarcados*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2009.

[Marcondes et al. 2006]MARCONDES, H. et al. EPOS: Um Sistema Operacional Portável para Sistemas Profundamente Embarcados. *Workshop de Sistemas Operacionais*, 2006.

[Marcondes et al. 2006]MARCONDES, H. et al. Operating Systems Portability: 8 bits and beyond. *11th IEEE ETFA*, p. 124–130, 2006.

[Marra et al. 2001]MARRA, F. et al. Tradutor de C para VHDL–C2VHDL. *Revista Eletrônica de Iniciação Científica, Ano I, Vol. I, No. I*, 2001.

[Maxfield 2004]MAXFIELD, C. M. *The Design Warrior's Guide to FPGAs*. [S.l.]: Elsevier, 2004.

[Mencer, Morf e Flynn 2000]MENCER, O.; MORF, M.; FLYNN, M. Hardware Software Tri-design of Encryption for Mobile Communication Units. *Technology*, p. 2, 2000.

[Nollet et al. 2003]NOLLET, V. et al. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In: *Reconfigurable Architectures Workshop (RAW), Proceedings of the International Parallel and Distributed Processing Symposium*. [S.l.: s.n.], 2003.

[Peterson e Smith 2001]PETERSON, G.; SMITH, M. Programming High Performance Reconfigurable Computers. *SSGRR 2001*, 2001.

[Polpeta e Fröhlich 2004]POLPETA, F. V.; FRöHLICH, A. A. Hardware Mediators: a Portability Artifact for Component-Based Systems. In: *International Conference on Embedded and Ubiquitous Computing*. Aizu, Japan: Springer, 2004. (Lecture Notes in Computer Science, v. 3207), p. 271–280.

[Raghavan e Sutton 2002]RAGHAVAN, A.; SUTTON, P. JPG-a partial bitstream generation tool to support partial reconfiguration in virtex FPGAs. In: *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*. [S.l.: s.n.], 2002. p. 155–160.

[Reis e Fröhlich 2009]REIS, T.; FRöHLICH, A. Operating System Support for Difference-Based Partial Hardware Reconfiguration. In: IEEE COMPUTER SOCIETY. *Proceedings of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping-Volume 00*. [S.l.], 2009. p. 75–80.

[Rhoads 2008]RHOADS, S. *Plasma - most MIPS I opcodes*. [S.l.]: Open Cores, 2008. `http://www.opencores.org/projects.cgi/web/mips`.

[Santambrogio, Rana e Sciuto 2008]SANTAMBROGIO, M.; RANA, V.; SCIUTO, D. Operating System Support for Online Partial Dynamic Reconfiguration Management. In: *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. [S.l.: s.n.], 2008. p. 455–458.

[SPARC International Inc. 1992]SPARC International Inc. *The SPARC Architecture Manual*. [S.l.: s.n.], 1992.

[Steiger, Walder e Platzner 2004]STEIGER, C.; WALDER, H.; PLATZNER, M. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks. *IEEE TRANSACTIONS ON COMPUTERS*, IEEE Computer Society, p. 1393–1407, 2004.

[Tessier e Burleson 2001]TESSIER, R.; BURLESON, W. Reconfigurable computing for digital signal processing: A survey. *The Journal of VLSI Signal Processing*, Springer, v. 28, n. 1, p. 7–27, 2001.

[Todman et al. 2005]TODMAN, T. et al. Reconfigurable Computing: Architectures and Design Methods. *IEEE Proceedings: Computer & Digital Techniques, Vol. 152, No. 2, March 2005*, p. 193–208, 2005.

[Ullmann et al. 2004]ULLMANN, M. et al. An FPGA run-time system for dynamical on-demand reconfiguration. In: *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. [S.l.: s.n.], 2004.

[Walder e Platzner 2003]WALDER, H.; PLATZNER, M. Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations. In: *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*. [S.l.: s.n.], 2003. p. 284–287.

[Walder, Steinegger e Platzner 2004]WALDER, H.; STEINEGGER, S.; PLATZNER, M. Implementation of a Runtime Environment for Reconfigurable Hardware Operating Systems. *Swiss Federal Institute of Technology Zurich (ETH) Computer Engineering and Networks Laboratory, TIK Report*, v. 195, 2004.

[Wanner e Fröhlich 2008]WANNER, L.; FRöHLICH, A. Operating System Support for Wireless Sensor Networks. *Journal of Computer Science*, 2008.

[Wiedenhoft, Jr e Fröhlich]WIEDENHOFT, G.; JR, A. H.; FRöHLICH, A. A Power Manager for Deeply Embedded Systems. *12th IEEE ETFA*, p. 748–751.

[Williams e Bergmann 2004]WILLIAMS, J.; BERGMANN, N. Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-on-Chip. In: *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms*. [S.l.: s.n.], 2004.

[Xilinx 2006]XILINX. *Spartan-3 Generation FPGA User Guide*. [S.l.]: Xilinx, 2006.

[Xilinx 2006]XILINX. *Xilinx Development System Reference Guide*. 2006.