XXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos
24 a 28 de maio de 2010
Gramado, RS

I Workshop de Sistemas Embarcados (WSE)

Editora
Sociedade Brasileira de Computação (SBC)

Organizadores
Flávio Rech Wagner (UFRGS)
Luigi Carro (UFRGS)
Leandro Buss Becker (UFSC)
Carlos Eduardo Pereira (UFRGS)
Antônio Jorge Gomes Abelém (UFPA)
Luciano Paschoal Gaspary (UFRGS)
Marinho Pilla Barcellos (UFRGS)

Realização
Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)

Promoção
Sociedade Brasileira de Computação (SBC)
Laboratório Nacional de Redes de Computadores (LARC)

Capa: Josué Klafke Sperb
Produção Editorial: Flávio Roberto Santos, Roben Castagna Lunardi, Matheus Lehmann, Rafael Santos Bezerra, Luciano Paschoal Gaspary e Marinho Pilla Barcellos.

Cópias Adicionais:
Sociedade Brasileira de Computação (SBC)
Av. Bento Gonçalves, 9500 - Setor 4 - Prédio 43.412 - Sala 219
Bairro Agronomia - CEP 91.509-900 - Porto Alegre - RS
Fone: (51) 3308-6835
E-mail: sbc@sbc.org.br

Dados Internacionais de Catalogação na Publicação (CIP)

Workshop de Sistemas Embarcados (1. : 2010 : Gramado, RS).

Anais / I Workshop de Sistemas Embarcados; organizadores Flávio Rech Wagner... et al. − Porto Alegre : SBC, c2010.

175 p.

ISSN 2177-496X

1. Redes de computadores. 2. Sistemas distribuídos. I. Wagner, Flávio Rech. II. Título.

Promoção

Sociedade Brasileira de Computação (SBC)

Diretoria

Presidente
José Carlos Maldonado (USP)

Vice-Presidente
Marcelo Walter (UFRGS)

Diretor Administrativo
Luciano Paschoal Gaspary (UFRGS)

Diretor de Finanças
Paulo Cesar Masiero (USP)

Diretor de Eventos e Comissões Especiais
Lisandro Zambenedetti Granville (UFRGS)

Diretora de Educação
Mirella Moura Moro (UFMG)

Diretora de Publicações
Karin Breitman (PUC-Rio)

Diretora de Planejamento e Programas Especiais
Ana Carolina Salgado (UFPE)

Diretora de Secretarias Regionais
Thais Vasconcelos Batista (UFRN)

Diretor de Divulgação e Marketing
Altigran Soares da Silva (UFAM)

Diretor de Regulamentação da Profissão
Ricardo de Oliveira Anido (UNICAMP)

Diretor de Eventos Especiais
Carlos Eduardo Ferreira (USP)

Diretor de Cooperação com Sociedades Científicas
Marcelo Walter (UFRGS)

Promoção

Conselho

Mandato 2009-2013
Virgílio Almeida (UFMG)
Flávio Rech Wagner (UFRGS)
Silvio Romero de Lemos Meira (UFPE)
Itana Maria de Souza Gimenes (UEM)
Jacques Wainer (UNICAMP)

Mandato 2007-2011
Cláudia Maria Bauzer Medeiros (UNICAMP)
Roberto da Silva Bigonha (UFMG)
Cláudio Leonardo Lucchesi (UNICAMP)
Daltro José Nunes (UFRGS)
André Ponce de Leon F. de Carvalho (USP)

Suplentes - Mandato 2009-2011
Geraldo B. Xexeo (UFRJ)
Taisy Silva Weber (UFRGS)
Marta Lima de Queiroz Mattoso (UFRJ)
Raul Sidnei Wazlawick (UFSC)
Renata Vieira (PUCRS)

Laboratório Nacional de Redes de Computadores (LARC)

Diretoria

Diretor do Conselho Técnico-Científico
Artur Ziviani (LNCC)

Diretor Executivo
Célio Vinicius Neves de Albuquerque (UFF)

Vice-Diretora do Conselho Técnico-Científico
Flávia Coimbra Delicato (UFRN)

Vice-Diretor Executivo
Luciano Paschoal Gaspary (UFRGS)

Membros Institucionais
CEFET-CE, CEFET-PR, IME, INPE/MCT, LNCC, PUCPR, PUC-RIO, SESU/MEC, UECE, UERJ, UFAM, UFBA, UFC, UFCG, UFES, UFF, UFMG, UFPA, UFPB, UFPE, UFPR, UFRGS, UFRJ, UFRN, UFSC, UFSCAR, UNICAMP, UNIFACS, USP.

Realização

Comitê de Organização

Coordenação Geral
Luciano Paschoal Gaspary (UFRGS)
Marinho Pilla Barcellos (UFRGS)

Coordenação do Comitê de Programa
Luci Pirmez (UFRJ)
Thaís Vasconcelos Batista (UFRN)

Coordenação de Palestras e Tutoriais
Lisandro Zambenedetti Granville (UFRGS)

Coordenação de Painéis e Debates
José Marcos Silva Nogueira (UFMG)

Coordenação de Minicursos
Carlos Alberto Kamienski (UFABC)

Coordenação de Workshops
Antônio Jorge Gomes Abelém (UFPA)

Coordenação do Salão de Ferramentas
Nazareno Andrade (UFCG)

Comitê Consultivo
Artur Ziviani (LNCC)
Carlos André Guimarães Ferraz (UFPE)
Célio Vinicius Neves de Albuquerque (UFF)
Francisco Vilar Brasileiro (UFCG)
Lisandro Zambenedetti Granville (UFRGS)
Luís Henrique Maciel Kosmalski Costa (UFRJ)
Marcelo Gonçalves Rubinstein (UERJ)
Nelson Luis Saldanha da Fonseca (UNICAMP)
Paulo André da Silva Gonçalves (UFPE)

Realização

Organização Local
Adler Hoff Schmidt (UFRGS)
Alan Mezzomo (UFRGS)
Alessandro Huber dos Santos (UFRGS)
Bruno Lopes Dalmazo (UFRGS)
Carlos Alberto da Silveira Junior (UFRGS)
Carlos Raniery Paula dos Santos (UFRGS)
Cristiano Bonato Both (UFRGS)
Flávio Roberto Santos (UFRGS)
Jair Santanna (UFRGS)
Jéferson Campos Nobre (UFRGS)
Juliano Wickboldt (UFRGS)
Leonardo Richter Bays (UFRGS)
Lourdes Tassinari (UFRGS)
Luís Armando Bianchin (UFRGS)
Luis Otávio Luz Soares (UFRGS)
Marcos Ennes Barreto (UFRGS)
Matheus Brenner Lehmann (UFRGS)
Pedro Arthur Pinheiro Rosa Duarte (UFRGS)
Pietro Biasuz (UFRGS)
Rafael Pereira Esteves (UFRGS)
Rafael Kunst (UFRGS)
Rafael Santos Bezerra (UFRGS)
Ricardo Luis dos Santos (UFRGS)
Roben Castagna Lunardi (UFRGS)
Rodolfo Stoffel Antunes (UFRGS)
Rodrigo Mansilha (UFRGS)
Weverton Luis da Costa Cordeiro (UFRGS)

Mensagem dos Coordenadores Gerais

Bem-vindo(a) ao XXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2010)! Esta edição do simpósio está sendo realizada de 24 a 28 de maio de 2010 na pitoresca cidade de Gramado, RS. Promovido pela Sociedade Brasileira de Computação (SBC) e pelo Laboratório Nacional de Redes de Computadores (LARC) desde 1983, o SBRC 2010 almeja não menos que honrar com uma tradição de quase 30 anos: ser reconhecido como o mais importante evento científico em redes de computadores e sistemas distribuídos do país, e um dos mais concorridos em Informática. Mais do que isso, pretende estimular intercâmbio de idéias e discussões qualificadas, aproximá-lo(a) de temas de pesquisa efervescentes e fomentar saudável aproximação entre estudantes, pesquisadores, professores e profissionais.

Para atingir os objetivos supracitados, reunimos um grupo muito especial de professores atuantes em nossa comunidade que, com o nosso apoio, executou com êxito a tarefa de construir um **Programa Técnico** de altíssima qualidade. O SBRC 2010 abrange as seguintes atividades: 20 sessões técnicas de artigos completos, cobrindo uma grande gama de problemas em redes de computadores e sistemas distribuídos; 2 sessões técnicas para apresentações de ferramentas; 5 minicursos ministrados de forma didática, por professores da área, sobre temas atuais; 3 palestras e 3 tutoriais sobre tópicos de pesquisa avançados, apresentados por especialistas nacionais e estrangeiros; e 3 painéis versando sobre assuntos de relevância no momento. Completa a programação técnica a realização de 8 *workshops* satélites em temas específicos: WRNP, WGRS, WTR, WSE, WTF, WCGA, WP2P e WPEIF. Não podemos deixar de ressaltar o **Programa Social**, organizado em torno da temática "*vinho*", simbolizando uma comunidade de pesquisa madura e que, com o passar dos anos, se aprimora e refina cada vez mais.

Além da ênfase na qualidade do programa técnico e social, o SBRC 2010 ambiciona deixar, como marca registrada, seu esforço na busca por excelência organizacional. Tal tem sido perseguido há mais de dois anos e exigido muita determinação, dedicação e esforço de uma equipe afinada de organização local, composta por estudantes, técnicos administrativos e professores. O efeito desse esforço pode ser percebido em elementos simples, mas diferenciais, tais como uniformização de datas de submissão de trabalhos, portal *sempre* atualizado com as últimas informações, comunicação sistemática com potenciais participantes e pronto atendimento a qualquer dúvida. O nosso principal objetivo com essa iniciativa foi e continua sendo oferecer uma elevada *qualidade de experiência* a você, colega participante!

Gostaríamos de agradecer aos membros do Comitê de Organização Geral e Local que, por conta de seu trabalho voluntário e incansável, ajudaram a construir um evento que julgamos de ótimo nível. Gostaríamos de agradecer, também, à SBC, pelo apoio prestado ao longo das muitas etapas da organização, e aos patrocinadores, pelo incentivo à divulgação de atividades de pesquisa conduzidas no País e pela confiança depositada neste fórum. Por fim, nossos agradecimentos ao Instituto de Informática da UFRGS, por viabilizar a realização, pela quarta vez, de um evento do porte do SBRC.

Sejam bem-vindos à Serra Gaúcha para o "SBRC do Vinho"! Desejamos que desfrutem de uma semana agradável e proveitosa!

Luciano Paschoal Gaspary
Marinho Pilla Barcellos
Coordenadores Gerais do SBRC 2010

Mensagem dos Coordenadores do WSE

A área de Sistemas Embarcados, que possui uma crescente importância no mercado mundial e também um enorme impacto na economia brasileira, já era tema de pesquisa em diversas universidades e centros de pesquisa brasileiros, porém ainda não tinha um evento próprio no Brasil. Visando reunir comunidades do Brasil e da América Latina que trabalham com diferentes aspectos de Sistemas Embarcados, mas que até hoje têm estado dispersas em diferentes eventos, estamos organizando, junto ao SBRC 2010, esta primeira edição do WSE - Workshop de Sistemas Embarcados. Neste primeiro evento dedicado exclusivamente a esta área, temos como objetivo não somente discutir aspectos científicos e tecnológicos importantes, mas também agregar a comunidade e encontrar novos meios de formação de massa crítica na área no Brasil e na América Latina.

O WSE, o WTR - Workshop de Tempo Real e Sistemas Embarcados e a 4th ARTIST School for Embedded Systems in South America, promovida pela rede europeia de excelência ARTIST, compõem a Primeira Semana Sul-Americana de Sistemas Embarcados e de Tempo Real, numa saudável sinergia entre estas comunidades.

Autores foram convidados a submeter trabalhos ao WSE reportando contribuições de pesquisa inovadoras, projetos em andamento e resultados experimentais relacionados a todos os aspectos da especificação, modelagem, análise, projeto, verificação, arquitetura, implementação e teste de hardware e software para Sistemas Embarcados, destinados a todas as áreas de aplicação, tais como comunicação, entretenimento, automação e controle, sistemas automotivos, aviônica, etc. Foram submetidos 29 trabalhos, dos quais 18 foram selecionados para apresentação oral durante o evento e publicação de artigo completo nos anais, que farão parte dos anais do SBRC 2010. Estes 18 trabalhos apresentam um mosaico bastante diversificado de atuação da comunidade nos diferentes aspectos de Sistemas Embarcados e certamente ensejarão novas interações e colaborações.

Desejamos a todos os participantes do WSE um excelente evento, na certeza de que nele encontrarão um espaço fértil para a discussão de suas ideias e avanço de sua pesquisa.

Flávio Rech Wagner
Luigi Carro
Coordenadores do WSE 2010

Comitê de Programa do WSE

Alexandre Trevisan, Trevisan Tecnologia
Alfredo Olivero, Universidad Nacional de San Martín
Antonio Augusto Fröhlich, Universidade Federal de Santa Catarina (UFSC)
Carlos Eduardo Pereira, Universidade Federal do Rio Grande do Sul (UFRGS)
Cesar Zeferino, Universidade do Vale do Itajaí (UNIVALI)
Diego Garbervetsky, Universidad de Buenos Aires
Dieter Schwanke, Sony-Ericsson
Diógenes Silva, Universidade Federal de Minas Gerais (UFMG)
Edna Natividade Silva Barros, Universidade Federal de Pernambuco (UFPE)
Fabiano Hessel, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Fernando Moraes, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Flávio Rech Wagner, Universidade Federal do Rio Grande do Sul (UFRGS)
Guido Araújo, Universidade Estadual de Campinas (UNICAMP)
Ivan Saraiva Silva, Universidade Federal do Rio Grande do Norte (UFRN)
Jean-Marie Farines, Universidade Federal de Santa Catarina (UFSC)
José Carlos Maldonado, Universidade de São Paulo (USP)
Julis Leite, Universidade Federal Fluminense (UFF)
Leandro Buss Becker, Universidade Federal de Santa Catarina (UFSC)
Leandro Soares Indrusiak, University of York
Lisane Brisolara, Universidade Federal de Polotas (UFPel)
Luigi Carro, Universidade Federal do Rio Grande do Sul (UFRGS)
Luiz Claudio Villar dos Santos, Universidade Federal de Santa Catarina (UFSC)
Paulo Romero Martins Maciel, Universidade Federal de Pernambuco (UFPE)
Ricardo Jacobi, Universidade de Brasília (UnB)
Rodolfo Jardim de Azevedo, Universidade Estadual de Campinas (UNICAMP)
Sergio Yovine, Universidad de Buenos Aires

Revisores do WSE

Alexandre Carissimi, Universidade Federal do Rio Grande do Sul (UFRGS)
Alexandre Trevisan, Trevisan Tecnologia
Alfredo Olivero, Universidad Nacional de San Martín
Antonio Augusto Fröhlich, Universidade Federal de Santa Catarina (UFSC)
Arliones Hoeller Junior, Universidade Federal de Santa Catarina (UFSC)
Carlos Eduardo Pereira, Universidade Federal do Rio Grande do Sul (UFRGS)
Cesar Zeferino, Universidade do Vale do Itajaí (UNIVALI)
Diego Garbervetsky, Universidad de Buenos Aires
Dieter Schwanke, Sony-Ericsson
Diógenes Silva, Universidade Federal de Minas Gerais (UFMG)
Edison Pignaton de Freitas, Universidade Federal do Rio Grande do Sul (UFRGS)
Edna Natividade Silva Barros, Universidade Federal de Pernambuco (UFPE)
Edson Moreira, Universidade de São Paulo (USP)
Elisa Yumi Nakagawa, Universidade de São Paulo (USP)
Fabiano Hessel, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Fernando Ataíde, CP Fiat
Fernando Moraes, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Flávio Rech Wagner, Universidade Federal do Rio Grande do Sul (UFRGS)
Giovani Gracioli, Universidade Federal de Santa Catarina (UFSC)
Guido Araújo, Universidade Estadual de Campinas (UNICAMP)
Ivan Saraiva Silva, Universidade Federal do Rio Grande do Norte (UFRN)
Jean-Marie Farines, Universidade Federal de Santa Catarina (UFSC)
José Carlos Maldonado, Universidade de São Paulo (USP)
Julis Leite, Universidade Federal Fluminense (UFF)
Leandro Buss Becker, Universidade Federal de Santa Catarina (UFSC)
Leandro Soares Indrusiak, University of York
Lisane Brisolara, Universidade Federal de Polotas (UFPel)
Luigi Carro, Universidade Federal do Rio Grande do Sul (UFRGS)
Luiz Claudio Villar dos Santos, Universidade Federal de Santa Catarina (UFSC)
Paulo Romero Martins Maciel, Universidade Federal de Pernambuco (UFPE)
Ricardo Jacobi, Universidade de Brasília (UnB)
Rodolfo Jardim de Azevedo, Universidade Estadual de Campinas (UNICAMP)
Rodrigo Allgayer, Universidade Federal do Rio Grande do Sul (UFRGS)
Sergio Yovine, Universidad de Buenos Aires
Victor Braberman, Universidad de Buenos Aires

Sumário

I Workshop de Sistemas Embarcados

♦

Sessão Técnica 1

Sistemas Operacionais, Virtualização e Tempo Real

Uma Estrutura de Reprogramação em Rede para Sistemas Operacionais Embarcados

Rodrigo Steiner, Giovani Gracioli e Antônio Augusto Fröhlich

[1]Laboratório de Integração Software e Hardware (LISHA)
Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476, 88049-900, Florianópolis, SC, Brasil

{rodrigo,giovani,guto}@lisha.ufsc.br

Resumo. *Este artigo apresenta uma estrutura de reprogramação de software em rede para sistemas embarcados. Tal estrutura foi desenvolvida no sistema operacional EPOS e é formada por um protocolo de disseminação de dados e um ambiente de suporte de sistema operacional que isola os componentes do sistema em unidades físicas independentes de posição de memória, permitindo que sejam atualizados em tempo de execução. A estrutura foi testada em nodos reais e avaliada em termos de consumo de memória, tempo de disseminação e de reprogramação.*

Abstract. *This paper presents a software network reprogramming structure for embedded systems. Such a structure was developed in the EPOS operating systems and it is composed by a data dissemination protocol and an operating system support environment that isolates the system components in memory position independent physical units, allowing their updating at execution time. The structure was tested in real nodes and evaluated in terms of memory consumption, dissemination and reprogramming time.*

1. Introdução

Reprogramar o software de um programa em execução é uma tarefa presente na grande maioria dos ambientes computacionais. A gama de aplicações que utiliza algum meio de reprogramação varia desde browsers para internet à sistemas dedicados, como controladores em um veículo, por exemplo. Devido às características especiais destes sistemas dedicados (e.g. limitação de recursos), a estrutura de reprogramação de software é diferente daquela presente nos ambientes computacionais convencionais. Além disso, alguns desses sistemas dedicados, como Redes de Sensores Sem Fio (RSSF), são formados por uma grande quantidade de nodos, onde coletar todos para reprogramá-los é impraticável.

A estrutura de reprogramação de software em um sistema embarcado deve ser composta por um protocolo de disseminação de dados e uma estrutura que organize os dados de forma consistente na memória do sistema. Uma maneira de oferecer isso às aplicações embarcadas é ocultar tal estrutura em um sistema operacional. Geralmente, a estrutura de reprogramação utilizada em um SO embarcado é composta por módulos atualizáveis em tempo de execução. Esses módulos são independentes de posição de memória para as aplicações e podem ser substituídos em tempo de execução [Han et al. 2005, Dunkels et al. 2004].

Não obstante, é essencial que a totalidade dos novos dados de um ou mais módulos cheguem corretamente a todos os nodos envolvidos na reprogramação. Por este motivo, um protocolo de disseminação deve ser usado juntamente com a estrutura do SO. De forma simplificada, um protocolo funciona da seguinte maneira: a disseminação começa através de uma estação base responsável por transmitir os dados aos seus nodos vizinhos. Uma vez que um nodo recebe os dados ele passa a ser capaz de retransmiti-los aos seus próprios vizinhos. Assim, se um nodo A tem B como vizinho, e o nodo B tem A e C como vizinhos, ao receber os dados de A o nodo B vai retransmiti-los para C. O processo se repete para todos os nodos, de forma que a rede inteira receba os dados [Thanos Stathopoulos 2003, Hui and Culler 2004].

Este artigo apresenta o projeto e implementação de uma estrutura de reprogramação de software eficiente para sistemas embarcados transparente para aplicações. A estrutura é implementada no EMBEDDED PARALLEL OPERATING SYSTEM (EPOS) [Fröhlich 2001] e é composta por um sistema que isola os componentes do sistema em unidades físicas da memória e um protocolo de disseminação de dados que garante a entrega dos novos dados para os nodos envolvidos na reprogramação. A estrutura foi avaliada em termos de consumo de memória, tempo disseminação e de reprogramação, apresentando bons resultados.

O restante deste artigo está organizado da seguinte maneira. A seção 2 apresenta os principais trabalhos relacionados a protocolos de disseminação e sistemas operacionais que suportam reprogramação. A seção 3 mostra o projeto do protocolo de disseminação usado neste trabalho e compara suas características com os protocolos relacionados. A seção 4 apresenta a integração do protocolo de disseminação com a estrutura do SO. A avaliação da estrutura de reprogramação completa é realizada na seção 5 e finalmente, a seção 6 conclui o artigo.

2. Trabalhos Relacionados

2.1. Protocolos de Disseminação

Multi-hop Over the Air Programming é um mecanismo de distribuição de código projetado priorizando o consumo de energia e memória em detrimento da latência [Thanos Stathopoulos 2003]. O MOAP utiliza um mecanismo de disseminação chamado *Ripple*, que distribui os dados de vizinhança em vizinhança. Em cada vizinhança apenas um pequeno subconjunto de nodos (de preferência apenas um) funcionam como emissores, enquanto os restantes são receptores. Quando os nodos recebem todos os dados eles podem se tornar emissores para seus próprios vizinhos (que estavam fora do alcance do emissor original). Para evitar que nodos se tornem emissores em uma vizinhança que já possua um, o mecanismo utiliza uma interface divulga/inscreve, nodos emissores divulgam sua versão e todos os interessados se inscrevem. Caso um emissor não receba inscrições ele fica em silêncio.

Deluge é um protocolo de disseminação projetado para propagar uma grande quantidade de dados de forma rápida e confiável. Ele compartilha várias ideias com o MOAP, como o uso de NACKs, pedidos de retransmissão *unicast* e transmissão de dados *broadcast* [Hui and Culler 2004]. Com o intuito de limitar a quantidade de informações que um receptor deve manter, possibilitar atualizações incrementais e permitir que os nodos continuem a disseminação antes de possuírem todos os dados, o protocolo utiliza o

conceito de páginas. Os dados são divididos em *P* páginas, sendo que uma página nada mais é do que um conjunto de *N* pacotes. Utilizando um vetor de idades para descrever a idade de cada página, os nodos são capazes de determinar quando uma página mudou e se necessitam ou não requisitá-la. Exigindo que os nodos recebam uma página por vez, pode-se utilizar um mapa de bits de apenas *N* bits para gerenciamento de segmentos, pois não é mais necessário manter registros de todos os pacotes ao mesmo tempo.

Multi-hop Network Programming (MNP) é um protocolo de reprogramação em rede cujas principais características incluem um mecanismo para seleção de emissor e uma abordagem para reduzir o uso da memória RAM [Wang 2004]. Assim como no MOAP a disseminação ocorre de vizinhança em vizinhança e um nodo só pode se tornar emissor após receber todos os dados. Através do algoritmo para seleção de emissor, um nodo decide se deve transmitir o código ou não. O objetivo deste algoritmo é o de garantir que a qualquer momento apenas um nodo esteja transmitindo os dados por vez, e que este transmissor seja o que vai causar maior impacto, em outras palavras, o que tiver um maior número de receptores. É importante ressaltar que o algoritmo não garante encontrar o emissor ideal, todavia, ele seleciona "bons" emissores e reduz o número de colisões.

Infuse é um protocolo de disseminação de dados baseado em uma comunicação sem colisões devido ao uso do MAC (*Medium Access Control*) TDMA (*Time Division Multiple Access*) [Arumugam 2004]. Este protocolo requer que os nodos conheçam tanto a sua localização como a de seus vizinhos, classificando-os em predecessores e sucessores. Assim um nodo ouve durante a faixa de tempo de seus predecessores para receber os dados e transmite durante a sua. A Tabela 1 apresenta as prioridades dos protocolos analisados.

2.2. Sistemas Operacionais Embarcados

Alguns SOs embarcados são projetados com uma abstração para atualização de software. Através desta abstração é possível realizar reprogramações em tempo de execução sem a necessidade de reiniciar o sistema, desta forma, evitando perda de dados.

TINYOS é um sistema operacional constituído de componentes reutilizáveis que são usados em conjunto formando uma aplicação específica [Levis et al. 2005]. Este SO suporta uma ampla gama de plataformas de hardware e tem sido utilizado em várias gerações de nodos sensores, podendo ser considerado o SO mais usado na área de RSSF. Ele apresenta um modelo de concorrência orientado a eventos e originalmente não suporta reconfiguração de software. Contudo, todos os protocolos apresentados anteriormente foram implementados utilizando o TINYOS, desta forma possibilitando a reconfiguração.

SOS é um sistema operacional constituído por módulos dinamicamente carregáveis e um *kernel* [Han et al. 2005]. Esses módulos enviam mensagens e se comunicam com o *kernel* através de uma tabela do sistema que contém *jumps* relativos. Desta forma o código em cada módulo torna-se independente de posição da memória, possibilitando alterações no software de maneira mais eficiente.

CONTIKI é um SO que possui uma estrutura de reconfiguração semelhante a do SOS. Ele implementa processos especiais, chamados *serviços*, responsáveis por prover funcionalidades a outros processos [Dunkels et al. 2004]. Esses serviços podem ser substituídos em tempo de execução através de uma *interface stub* responsável por redirecionar as chamadas das funções para uma *interface de serviço*, que possui ponteiros para as

implementações atuais das funções do serviço correspondente. A Tabela 2 resume o processo de reconfiguração nos SOs analisados.

Tabela 1. Prioridades dos protocolos analisados.

Protocolo	Energia	Latência	Memória
MOAP	1º	3º	2º
Deluge	3º	1º	2º
MNP	2º	3º	1º
Infuse	1º	2º	3º

Tabela 2. Processo de reconfiguração nos SOs analisados.

SO	Processo de Reconfiguração
TINYOS	Sem suporte direto.
SOS	Módulos reconfiguráveis.
CONTIKI	Módulos reconfiguráveis.

3. Reprogramação em Rede

Em geral o processo de reprogramação em rede é dividido em três etapas, como ilustrado na Figura 1. A primeira é responsável pela preparação dos dados a serem disseminados. A segunda etapa engloba todo o processo de disseminação, onde os dados são enviados e armazenados pelos nodos pertencentes à rede. Por fim, o mecanismo de reconfiguração do SO interpreta os dados recebidos e os utiliza para atualizar a memória de programa.

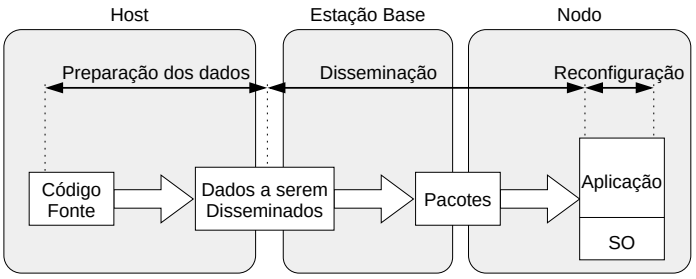


Figura 1. Processo de programação em rede.

3.1. Protocolo de Disseminação

Protocolos de disseminação de dados são utilizados para propagar dados pela rede utilizando seus próprios nodos para isso. Em especial, um protocolo utilizado por um mecanismo de reprogramação em rede deve ser confiável, ou seja, garantir a entrega correta de todos os dados a todos os nodos. Abaixo, as propriedades que devem ser levadas em conta ao se projetar um protocolo de disseminação [Lanigan et al. 2005]:

Baixa Latência: como a atualização é um serviço secundário o protocolo não deve interromper a aplicação principal por muito tempo.

Baixo Consumo de Memória: os dados necessários para a atualização devem ser armazenados até que a transmissão termine, entretanto o protocolo deve requisitar pouco espaço de armazenamento de forma a não restringir a quantidade de memória disponível para a aplicação principal.

Confiabilidade: ao contrário de algumas aplicações tradicionais onde a perda de um pacote é tolerável devido ao fato de que os dados são redundantes e correlacionados, na reprogramação cada pacote é crucial e todos devem ser recebidos para que a atualização possa ocorrer. Sendo assim o protocolo deve possuir uma política de retransmissão permitindo a recuperação de pacotes perdidos.

Eficiência Energética: o protocolo deve minimizar seu consumo de energia de forma a não diminuir severamente o tempo de vida do nodo.

Tolerância a Inclusão/Remoção de nodos: é possível que um nodo falhe durante um período de tempo e depois volte a funcionar, ou até mesmo que novos nodos sejam incluídos na rede. Desta forma a disseminação não deve ser severamente afetada pela inclusão ou remoção de nodos.

Uniformidade: para garantir que a rede inteira seja atualizada, todos os dados devem ser entregues a todos os nodos da rede. Nodos incluídos na rede durante ou depois de uma atualização também devem ser capazes de receber os dados da atualização.

As propriedades de **confiabilidade** e **uniformidade** são obrigatórias, uma vez que garantem o funcionamento correto do protocolo. Já as propriedades de baixa latência, baixo consumo de memória, eficiência energética e tolerância a inclusão ou remoção de nodos são apenas desejáveis, pois não garantem corretude. Entretanto um protocolo que as ignore seria de pouca utilidade na prática [Thanos Stathopoulos 2003].

3.1.1. Características de um Protocolo

A Figura 2 apresenta o diagrama de características de um protocolo de disseminação de dados. Este tipo de diagrama possibilita caracterizar as propriedades de um determinado conceito, apresentando seus pontos de variação [Czarnecki and Eisenecker 2000]. As características são representadas como nodos de uma árvore, cuja raiz é o conceito, e só estão presentes se seu nodo pai está presente. Características obrigatórias e opcionais são representadas por um círculo no final de suas arestas, preenchido e vazio respectivamente. Características alternativas são conjuntos do qual apenas uma característica pode estar presente e são representadas por um arco ligando suas arestas.

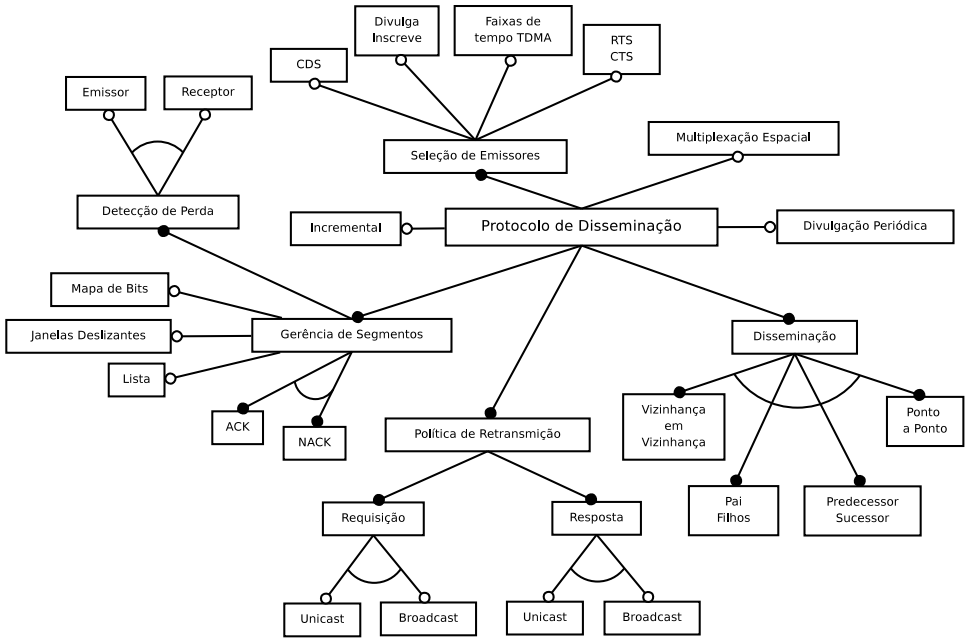


Figura 2. Diagrama de características.

Incremental: um protocolo com esta característica envia somente as diferenças entre os novos dados e os antigos. Desta forma diminui-se a quantidade total de dados a serem transmitidos, consequentemente diminuindo o consumo de energia.
Disseminação: a forma pela qual os dados são propagados pela rede.
Gerência de Segmentos: mecanismo utilizado para detectar perdas de pacotes.
Seleção de emissor: a forma como o protocolo decide quais nodos se tornarão emissores pode diminuir tanto o número total de colisões quanto o de mensagens transmitidas.
Política de Retransmissão: a forma em que são feitas as requisições por pacotes perdidos e retransmissões.
Divulgação Periódica: um protocolo com esta característica requer que todos os nodos divulguem suas versões periodicamente, possibilitando a nodos que de alguma forma perderam a operação de disseminação receberem os dados necessários para a reprogramação.
Multiplexação Espacial: um protocolo com esta característica não exige que os nodos recebam todos os dados para tornarem-se emissores, desta forma possibilitando que os dados sejam transmitidos em paralelo por toda a rede.

3.1.2. Escolhas de Projeto

Como algumas propriedades desejáveis entram em conflito com outras, os protocolos existentes realizam escolhas de projetos dando preferência a umas em detrimento de outras. Quanto as escolhas realizadas no protocolo desenvolvido neste trabalho:

1. A propriedade não obrigatória considerada mais importante foi a de eficiência energética, uma vez que todas as operações realizadas necessitam de energia e, em muitos sistemas embarcados, há apenas uma quantidade finita disponível.
2. Consumo de memória foi a segunda propriedade considerada mais importante, visto que o protocolo de disseminação não é a aplicação principal do nodo, mas apenas um serviço oferecido pelo sistema operacional. Desta forma, não se deve limitar a quantidade de memória disponível para as aplicações.
3. Por fim, a latência. Para poder otimizar o consumo de energia e memória algumas propriedades que diminuiriam a latência não foram utilizadas (e.g. multiplexação espacial).

3.1.3. Implementação

A Figura 3 apresenta a máquina de estados do protocolo desenvolvido. Ele distribui os dados de vizinhança em vizinhança, utiliza um mecanismo de seleção de emissores baseado no MNP (divulga / inscreve), responsabiliza os receptores por detectar perdas, realiza requisições *unicast* e retransmissões *broadcast* e utiliza o mecanismo de janelas deslizantes para gerência de segmentos.

Nodos divulgam suas versões, periodicamente, e todos os interessados a requisitam. Um potencial emissor mantêm uma variável *ReqCtr*, inicializada com zero, e a incrementa para cada nova requisição recebida, destinada a ele, vinda de um nodo ainda não computado. As mensagens de divulgação tem duas funções: anunciar uma nova versão e

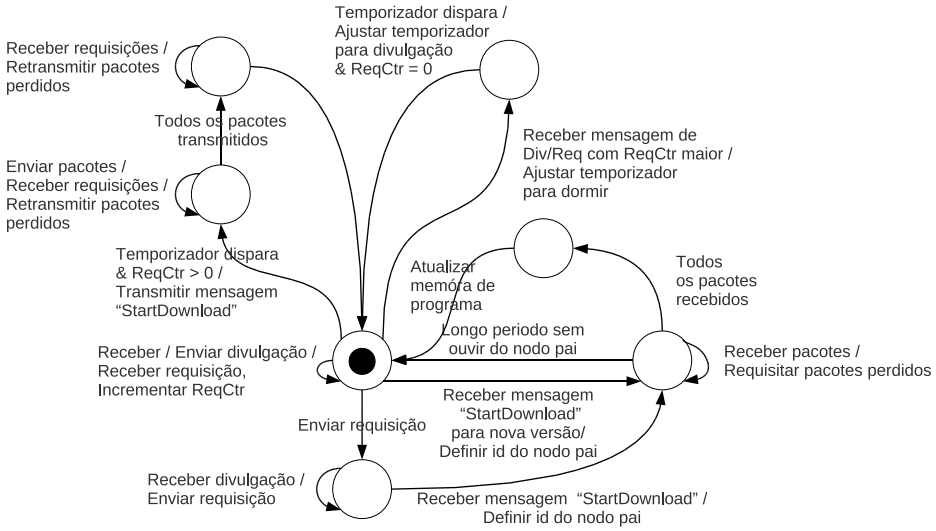


Figura 3. Máquina de estados do protocolo desenvolvido.

prevenir que nodos com menos requisições virem emissores; elas possuem o número da versão, o *id* do emissor e sua variável *ReqCtr*. Quando um nodo recebe uma mensagem de divulgação que contenha uma nova versão, ele irá enviar uma requisição *broadcast* contendo seu *id*, o do transmissor e o valor da *ReqCtr* recebida. Como as divulgações e requisições são *broadcasts* outros nodos que estão na disputa para se tornarem emissores também as recebem e caso possuam um *ReqCtr* menor vão para o estado *sleep*. Como critério de desempate é utilizado o *id* dos nodos.

Ao virar emissor um nodo transmite uma mensagem *broadcast “StartDownload”* e passa a enviar os dados, pacote por pacote. Os receptores definem este nodo como seu “pai” e só aceitam pacotes vindo dele. Cada pacote possui um identificador único sequencial e os receptores mantêm o número do último pacote recebido. Assim, ao receber um novo pacote é verificado se há uma lacuna entre estes dois números e os pacotes intermediários são considerados perdidos. Ao detectar uma perda, o receptor envia um pedido de retransmissão para o emissor, utilizando um pacote *unicast*. Os pedidos de retransmissão possuem uma prioridade maior que pacotes normais, então um emissor irá primeiro responder a todas as requisições antes de continuar com a transmissão.

4. Integração com o Sistema Operacional

EPOS é um sistema operacional multi-plataforma baseado em componentes, onde serviços tradicionais do SO são implementados através de componentes do sistema independentes de plataforma [Fröhlich 2001]. O suporte a serviços específicos de plataforma é realizado através de mediadores de hardware [Polpeta and Fröhlich 2004]. Mediadores são funcionalidades equivalentes a drivers de dispositivos em plataformas UNIX, mas não são camadas de abstração de hardware tradicionais. Ao contrário, os mediadores fazem uso de interfaces independentes de plataforma para sustentar suas interfaces de contrato entre componentes de hardware. Devido ao uso de metaprogramação estática em C++ e funções *inlining*, o código do mediador é dissolvido nos componentes em tempo de compilação.

Reprogramação no EPOS é suportada através do EPOS LIVE UPDATE SYSTEM

(ELUS) [Gracioli 2009]. O ELUS modificou o framework de componentes do EPOS, mais especificamente, o aspecto de invocação remota [Fröhlich 2001] para suportar também reconfiguração do software. A Figura 4 demonstra a nova estrutura do framework. Ao invés dos elementos `Proxy` e `Agent` estarem em diferentes espaços de endereçamento (e.g. diferentes nodos), ambos estão presentes no mesmo nodo. Desta forma, somente o `Agent` tem conhecimento sobre a posição de memória de um componente, podendo assim atualizar o código e dados deste componente. A Figura também mostra como o processo de reconfiguração é habilitado ou não para um componente. Isso é realizado através da classe *Trait* do componente. Habilitando a opção de reconfiguração (*reconfiguration*) irá adicionar ao sistema em tempo de compilação o suporte à reprogramação ao componente. Somente os componentes habilitados suportam reprogramação. O elemento `Adapter` é usado para adaptar o componente aos diferentes cenários de execuções, aplicando os correspondentes aspectos suportados pelo `Scenario` antes e depois da chamada real do método do componente [Fröhlich and Schröder-Preikschat 2000].

As invocações dos métodos entre o `Proxy` e o `Agent` acontece através de uma função que possui uma tabela de métodos, chamada de `Dispatcher`, que contém os endereços dos métodos do `Agent`. Essa função garante que não aconteça chamadas ao componente enquanto este estiver sendo atualizado utilizando um Semáforo. O `Agent` armazena os objetos do componente em uma tabela hash, e usa a tabela de métodos virtuais do objeto para fazer a chamada ao método real. Ao se atualizar métodos de um componente, basta atualizar os métodos dentro da tabela virtual.

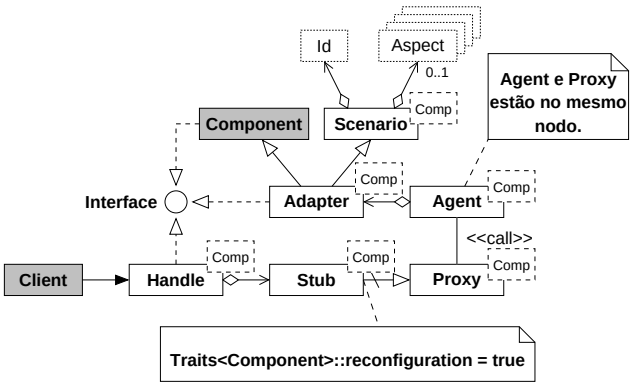


Figura 4. Framework do EPOS modificado para reprogramação de software.

O ELUS recebe requisições de reprogramação de um componente através de um protocolo, chamado de ELUS TRANSPORT PROTOCOL (ETP). A Figura 5 apresenta as mensagens disponíveis pelo ETP. Os 4 bits menos significativos do campo de controle definem o tipo da mensagem e os 4 bits mais significativos definem a quantidade de campos que a mensagem contém, pois isso varia de acordo com o tipo de mensagem. Uma *Thread* criada na inicialização do sistema, chamada de RECONFIGURATOR, cria uma instância do protocolo de disseminação e após o recebimento dos dados, inicia uma reprogramação. A escrita dos dados na memória de programa é abstraída por um gerenciador de código (*Code Manager*).

A mensagem (a) informa uma adição de método a um componente. A mensagem (b) informa que um método está sendo removido de um componente. A mensagem (c) requisita a atualização de todos os métodos do componente. Para isso, além do novo

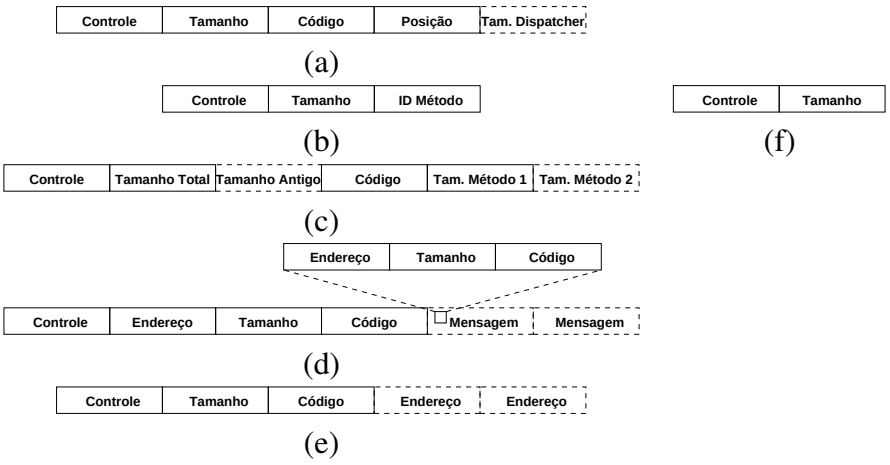


Figura 5. Mensagens de reprogramação do ETP. (a) Adição de método (b) Remoção de método (c) Atualização do componente (d) Atualização de endereço (e) Atualização da aplicação (f) Adição de atributos.

tamanho do código, são informados o tamanho antigo, o novo código e o novo tamanho dos métodos do componente. Todas essas informações são utilizadas pelo `Agent` para decidir se há necessidade de alocar novo espaço de memória ou se o novo código cabe no espaço antigo. O ETP ainda permite atualizar um endereço específico (d). Várias mensagens podem ser concatenadas, e o número de mensagens é controlado pelo campo de controle. A mensagem (e) informa a atualização de uma aplicação e a mensagem (f) requisita a adição de atributos, enviando para o `Agent` o tamanho do objeto que deve ser criado somando os tamanhos dos atributos antigos com os novos. O `Agent` irá alocar espaço para o novo objeto contando com os novos atributos, transferir o estado (dados) do objeto antigo para o novo e apagar o objeto antigo. Os atributos só podem ser acessados através dos métodos `set` e `get`, por isso uma mensagem de adição de atributos também deve ser seguida por uma mensagem de adição de métodos.

A estrutura de mensagens criada pelo ELUS permite que um protocolo de disseminação de dados seja facilmente integrado ao sistema operacional. O protocolo de disseminação deve, portanto, criar mensagens no formato ETP e realizar uma chamada ao `Agent` informando uma atualização. Essa simples estrutura é um diferencial não encontrado nos trabalhos relacionados, o que torna o processo de atualização simples e abstrai do desenvolvedor detalhes de como a reprogramação acontece efetivamente. Além disso, não é necessário reinicializar o sistema, evitando perda de dados, diferentemente dos protocolos de disseminação apresentados nos trabalhos relacionados.

A Figura 6 apresenta o diagrama de sequência do processo de atualização, demonstrando como é realizada a integração entre o protocolo de disseminação de dados e a estrutura do ELUS. O RECONFIGURATOR inicia o protocolo através da chamada ao método *run*. Este método fica bloqueado até que uma atualização (nova versão) requisitada pelo nodo seja recebida. Após o recebimento dos novos dados, o RECONFIGURATOR cria uma mensagem no formato ETP e passa os dados para o `Agent` através da chamada ao *trapAgent*. Por fim, o `Agent` realiza a escrita dos dados na memória de código.

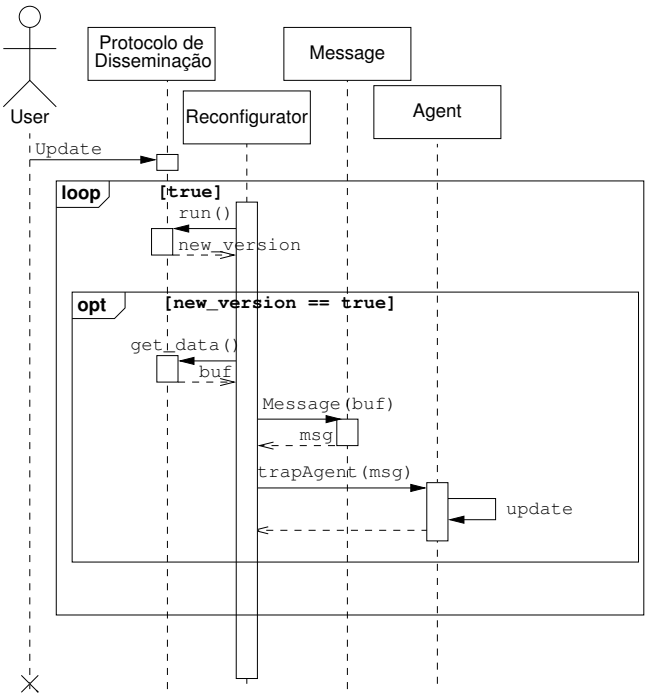


Figura 6. Integração do protocolo de disseminação com a estrutura do ELUS.

5. Avaliação

A estrutura de reprogramação formada pelo ELUS e o protocolo de disseminação foi avaliada pelo consumo de memória, latência de envio de dados para toda a rede e tempo de reconfiguração, este considerando o tempo de recebimento dos dados pela rede e o tempo de escrita na memória de programa. Estes testes foram realizados utilizando-se nodos sensores Mica2 [Crossbow Technology Inc.], que possuem um microcontrolador ATMega128, 4kB de memória ram, 128kB de memória flash, 4kB de EEPROM e comunicação via rádio. O sistema foi gerado com o compilador GNU g++ 4.0.2 e o consumo de memória medido utilizando-se a ferramenta GNU *objdump* 2.16.1. A latência e tempo de reconfiguração foram medidos através do temporizador do microcontrolador.

5.1. Memória

A Tabela 3 apresenta o consumo de memória de todos os elementos da estrutura. Para este teste, o suporte a reconfiguração foi habilitado para um componente que possui 4 métodos. O protocolo de disseminação ocupa 2534 bytes na área de código e 21 bytes na área de dados não inicializados. Vale-se ressaltar que um buffer é criado dinamicamente na recepção do novo código pelo protocolo, sendo assim dependente do tamanho do código a ser enviado. A estrutura do framework do ELUS consome 2076 bytes de código, 94 bytes de dados e 99 bytes de não inicializados devido às tabelas e variáveis necessárias para armazenar os objetos e métodos.

Além disso, quando um novo componente tem o suporte à reprogramação habilitado, este deve ter seus métodos incluídos no framework do ELUS. A Tabela 4 mostra o consumo de memória necessário pelo ELUS quando um novo componente é adicionado no sistema. Os métodos `Create`, `Destroy` e `Update` representam o construtor, destrutor do objeto do componente e o método de atualização e devem, portanto, sempre

Tabela 3. Consumo de memória da estrutura: ELUS e protocolo de disseminação.

Elementos da Estrutura	Tamanho da Seção (bytes)			
	.text	.data	.bss	.bootloader
Protocolo de Disseminação	2536	0	21	0
RECONFIGURATOR	166	0	4	0
AVR Code Manager	36	0	0	416
ELUS	2076	94	74	0
Total	4814	94	99	416

Tabela 4. Consumo de memória do ELUS ao habilitar o suporte à reconfiguração em um componente.

Método do Framework	Tamanho da Seção (bytes)	
	.text	.data
Create	180	0
Destory	138	0
Método sem parâmetro e valor de retorno	94	0
Método com um parâmetro e sem valor de retorno	98	0
Método sem parâmetro e com valor de retorno	112	0
Método com um parâmetro e valor de retorno	126	0
Update	1250	0
Dispatcher	0	2 X (n. de métodos)
Semáforo	0	18
Tamanho mínimo	1662	26

estar presentes. Para cada componente também é necessário um semáforo para controlar o acesso exclusivo ao componente e impedir que uma atualização ocorra ao mesmo tempo que seu código esteja executando. O total de consumo mínimo para um novo componente adicionado, composto pelo construtor, destrutor, método `Update` e um método que não receba parâmetros e não tenha nenhum valor de retorno é de 1662 bytes de código e 26 bytes de dados.

A Equação 5.1 sumariza o sobrecusto de memória. O tamanho de um componente "C" é o somatório dos tamanhos de todos os métodos conforme a Tabela 4 somados com o tamanho da implementação dos métodos pelo próprio componente. Ainda, somam-se a este valor, o tamanho dos métodos `Create`, `Destroy` e `Update`. Já o tamanho dos dados é a soma dos dados do componente com os valores do `Dispatcher` e `Semáforo` que são utilizados pelo ELUS.

$$Tamanho_c = \sum_{i=1}^{n}(Método_i) + Create + Destroy + Update \quad (1)$$

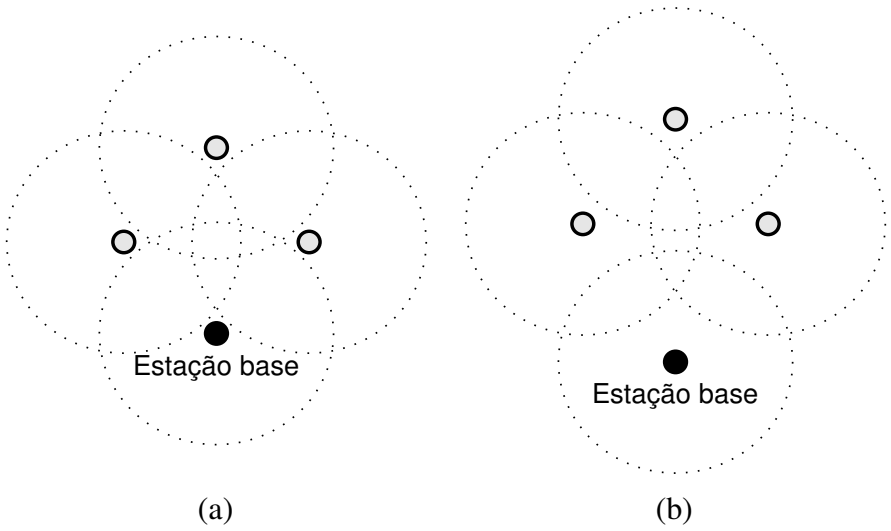


Figura 7. Topologias utilizadas para medição de latência. (a) Estação base possui comunicação com todos os nodos (b) Estação base não possui comunicação com todos os nodos.

5.2. Latência

Foram utilizadas duas topologias para medir a latência do protocolo de disseminação, ilustradas na Figura 7: (a) a estação base possui comunicação com todos os nodos, (b) a estação base não possui comunicação com um nodo, que está ao alcance dos outros. Em ambas topologias repetiu-se o processo de disseminação vinte vezes, de forma a atualizar um método de um componente do sistema, propagando 10 bytes de dados (utilizados na atualização do método) e 6 bytes de informações de controle (utilizadas pelo protocolo).

A Figura 8 apresenta a média do tempo que a estação base leva para propagar os dados aos nodos a sua volta. Foi observado um desvio padrão de 0,0233 segundos. Este tempo não mudou alterando o número de receptores entre um e três, isto porque pacotes perdidos estão altamente correlacionados, ou seja, vários receptores perdem o mesmo conjunto de pacotes [Wang 2004].

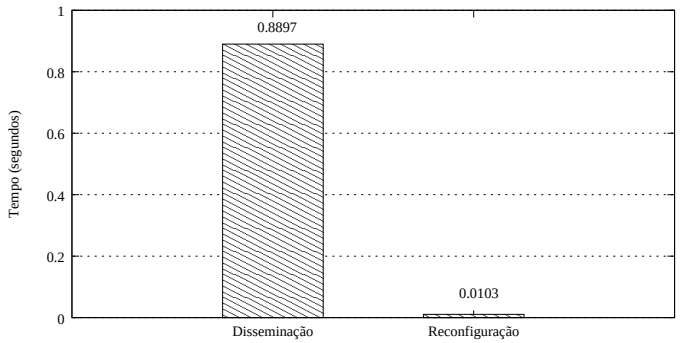


Figura 8. Tempo de disseminação e reconfiguração.

A Figura 9 mostra a média de tempo que os dados levam para serem propagados da estação base para nodos intermediários, e destes para o nodo fora de alcance da

estação base (segunda topologia). É possível perceber que o tempo necessário para propagar dados entre nodos normais da rede é aproximadamente quatro vezes maior que o tempo gasto pela estação base, isto se deve ao fato que a estação base não executa a etapa de seleção de emissor, ou seja, não perde tempo divulgando sua versão e recebendo requisições para enfim se tornar um emissor e começar a disseminar os dados. O tempo de disseminação dos nodos intermediários apresentou um desvio padrão de 1,1288 segundos.

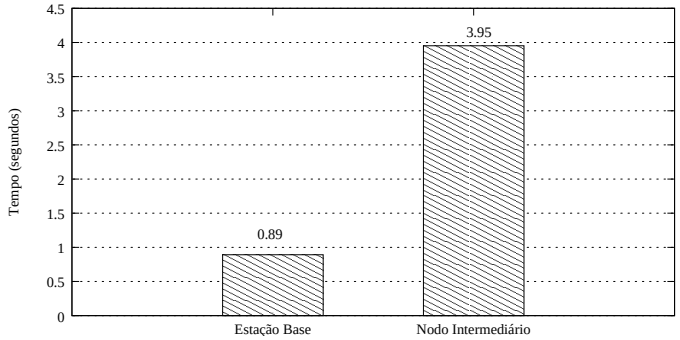


Figura 9. Tempo de disseminação.

5.3. Tempo de Reconfiguração

Foi considerado como tempo de reconfiguração, o tempo que o nodo leva para atualizar sua memória de programa após ter recebido todos os dados necessários, via o protocolo de disseminação. Na estrutura proposta este tempo engloba: a chamada do RECONFIGURATOR, os métodos `p` e `v` do semáforo, a chamada para o método `Update`, a recuperação dos argumentos passados na mensagem ETP, a recuperação do objeto a ser atualizado em uma tabela hash, descobrir o endereço da `vtable` e a escrita dos dados na flash. A Figura 8 mostra a média do tempo de reconfiguração obtido, sendo que o mesmo apresentou um desvio padrão de 0,0103 segundos.

Uma característica da arquitetura utilizada é que não é possível mudar apenas um byte por vez em sua memória flash. Esta memória só permite a escrita em páginas, cujo tamanho é de 256 bytes, e antes de reescrever uma página é necessário apagar seu conteúdo. Desta forma para atualizarmos uma parte da memória é necessário ler o conteúdo da página e armazená-lo em um buffer temporário, modificar apenas a parte desejada para enfim escrever na flash.

6. Conclusões

Este artigo apresentou uma estrutura de reprogramação em rede para sistemas operacionais embarcados que permite reconfiguração do software em tempo de execução. Esta estrutura é composta por um protocolo de disseminação de dados e um ambiente de suporte de sistema operacional que isola os componentes do sistema em unidades independentes de posição de memória. O protocolo garante a entrega correta de todos os dados para todos os nodos da rede e o ambiente de suporte permite que o sistema seja reconfigurado em tempo de execução sem a necessidade de reinicialização. A estrutura foi testada em uma RSSF utilizando nodos sensores reais, Mica2, e avaliada em termos de consumo de memória, tempo de disseminação e de reprogramação.

Referências

Arumugam, M. U. (2004). Infuse: a tdma based reprogramming service for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 281–282, New York, NY, USA. ACM.

Crossbow Technology Inc. *MICA2 Datasheet*.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Dunkels, A., Grönvall, B., and Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA.

Fröhlich, A. (2001). Application-Oriented Operating Systems. *Sankt Augustin: GMD-Forschungszentrum Informationstechnik*, 1.

Fröhlich, A. A. and Schröder-Preikschat, W. (2000). Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th WMSCI*, Orlando, U.S.A.

Gracioli, G. (2009). Reconfiguração dinâmica de software em sistemas profundamente embarcados.

Han, C.-C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. (2005). A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA. ACM Press.

Hui, J. W. and Culler, D. (2004). The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA. ACM.

Lanigan, P., Gandhi, R., and Narasimhan, P. (2005). Disseminating Code Updates in Sensor Networks: Survey of Protocols and Security Issues. Technical report, School of Computer Science, Carnegie Mellon University, PA.

Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. (2005). Tinyos: An operating system for sensor networks. pages 115–148.

Polpeta, F. V. and Fröhlich, A. A. (2004). Hardware mediators: a portability artifact for component-based systems. In *International Conference on Embedded and Ubiquitous Computing*, volume 3207 of Lecture Notes in Computer Science, pages 271–280, Aizu, Japan. Springer.

Thanos Stathopoulos, John Heidemann, D. E. (2003). A remote code update mechanism for wireless sensor networks. Technical report.

Wang, L. (2004). Mnp: multihop network reprogramming service for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 285–286, New York, NY, USA. ACM.

Virtualização em sistemas embarcados: é o futuro?

Alexandra Aguiar, Fabiano Hessel

Faculdade de Informática – PUCRS – Av. Ipiranga 6681, Porto Alegre, Brasil

alexandra.aguiar@pucrs.br, fabiano.hessel@pucrs.br

Abstract. *Traditionally, virtualization has been adopted by enterprise industry aiming to make better use of general purpose multi-core processors and its use in embedded systems (ES) seemed to be both a distant and unnecessary reality. However, with the rise of each more powerful multiprocessed ESs, virtualization brings an opportunity to use simultaneously several operating systems (OS) besides offering more secure systems and even an easier way to reuse legacy software. Although ESs have increasingly bigger computational power, they are still far more restricted than general purpose computers, especially in terms of area, memory and power consumption. Therefore, this paper studies the possibilities of using virtualization - a technique that typically demands robust systems - in powerful yet restricted current embedded systems.*

Resumo. *Tradicionalmente, virtualização tem sido adotada por diversas empresas como uma forma de aproveitar melhor o grande poder computacional oferecido por arquiteturas com vários processadores e seu uso em sistemas embarcados parece desnecessário e distante da realidade. Entretanto, com a utilização de sistemas embarcados multiprocessados, a virtualização oferece uma oportunidade para a utilização de diferentes sistemas operacionais simultaneamente em um único processador, possibilitando o desenvolvimento de aplicações onde a segurança dos dados é importante e possibilitando a reutilização de software de forma mais fácil. Atualmente, os sistemas embarcados são mais abrangentes, multifuncionais, com maior poder de processamento e convergentes, porém, ainda possuem muitas restrições comparadas aos sistemas de propósito geral. Nesse contexto, o presente artigo aborda os motivos que tornam a intersecção entre virtualização e sistemas embarcados um grande desafio a ser superado.*

1. Introdução

Virtualização de sistemas computacionais consiste em criar um grupo lógico de recursos que se assemelham aos recursos físicos oferecidos por um ambiente computacional [Popek e Goldberg, 1974]. Essa técnica tem sido adotada amplamente no mundo empresarial, especialmente para explorar o potencial de sistemas multiprocessados, além de oferecer outras vantagens, tais como:

- permitir que vários sistemas operacionais (SO) sejam executados em uma única máquina;
- prover isolamento de uma máquina virtual para outra, aumentando a segurança;
- aumentar a flexibilidade do sistema;
- melhorar o gerenciamento da carga de trabalho, e:

- permitir a independência de hardware.

Por outro lado, a virtualização pode ser considerada uma técnica que demanda alto poder computacional, já que, normalmente, requer um grande espaço em disco e muito uso de memória RAM, além de inserir uma camada extra de gerenciamento: o monitor de máquinas virtuais (do inglês, *Virtual Machine Monitor* – VMM), também conhecido como *hypervisor*, camada essa responsável por permitir que instruções executadas pela máquina virtual sejam executadas normalmente pela máquina hospedeira.

Em servidores comerciais, a virtualização permite que um único servidor físico sirva como múltiplos servidores lógicos além de prover múltiplas instâncias de diferentes Sistemas Operacionais (SO), como Windows, Linux e outros. Frequentemente, esses sistemas são empregados em processadores *multi-core* da Intel e AMD, tendência essa que é adotada atualmente pela maioria dos fabricantes de processadores, cujos projetos ultrapassam os quatro *cores* para um futuro próximo.

Já no mundo dos Sistemas Embarcados (SE), assim como na computação de propósito geral, o uso de plataformas multiprocessadas tem se mostrado uma forte tendência nos últimos anos [Jerraya *et.al.*2005]. Dispositivos que utilizam tais plataformas forçam uma mudança na maneira pela qual os desenvolvedores de SE's realizam a concepção de seus sistemas, já que técnicas antes vistas na computação multiprocessada de propósito geral precisam ser reavaliadas antes de ser empregadas em SE's [Martin, 2006].

Enquanto a virtualização possibilita a execução de múltiplas instâncias de sistemas operacionais em um único processador (*mono-* ou *multi-core*), a sua utilização em sistemas embarcados não é trivial, pois são muito diferentes de sistemas empresariais [Waldspurger, 2002]. Desse modo, para que a virtualização possa ser empregada de maneira vantajosa em sistemas embarcados, muito esforço deve ser realizado para que se entenda como deve-se adaptá-la às necessidades e características dos SE's, sistemas normalmente restritos com relação ao consumo de energia, quantidade de memória, restrições temporais e tamanho de área.

Adicionalmente, o uso de técnicas previamente otimizadas para sistemas comerciais Windows e Linux não são as mais adequadas em SE. Ao invés disso, técnicas existentes de virtualização devem ser adaptadas aos sistemas embarcados, verificando a relação custo benefício de seu uso. Além disso, deve-se verificar qual o tipo de sistema embarcado teria mais vantagens ao se utilizar a virtualização.

Nesse contexto, o presente trabalho visa discutir os principais conceitos envolvendo virtualização, alem de explorar as possibilidades de uso dessa técnica em sistemas embarcados. Para isso, são descritas as maneiras mais adequadas de se desfrutar da virtualização em sistemas embarcados de acordo com a meta a ser atingida. Ainda, apresentam-se as principais desvantagens e problemas esperados durante a aplicação da técnica em SE além de possíveis soluções ou formas para contornar esses problemas.

O artigo está organizado como segue. A próxima seção exibe alguns conceitos básicos adotados durante o artigo. Já a Seção 3 discute as vantagens da virtualização em SE seguida da seção 4 que mostra os principais problemas esperados nessa adaptação.

Na Seção 5 discute-se como se pode aplicar virtualização em SE e, finalmente, a Seção 6 conclui o trabalho.

2. Conceitos básicos

Nesta seção são apresentados os conceitos básicos sobre virtualização e sistemas embarcados. Destacam-se aqueles conceitos necessários para melhor entender o restante do artigo.

2.1 Sistemas Embarcados

Nos últimos anos, pode-se afirmar que sistemas embarcados eram comumente caracterizados por dispositivos simples, com severas restrições, tais como uso de memória, poder de processamento e vida útil da bateria. Sua funcionalidade era determinada, tipicamente, via hardware, sendo que seu software usualmente era composto por *drivers* de dispositivos, escalonador de tarefas e alguma lógica de controle, causando uma baixa dinamicidade das tarefas ofertadas por um dado sistema ao longo do seu tempo de vida [Hansson *et.al.* 2005].

No entanto, este cenário encontra-se diferente. Cada vez mais sistemas híbridos, convergentes e não críticos, com inúmeras características dos sistemas de propósito geral estão surgindo. A maior mudança, sem dúvidas, é o fato de se oferecer um número antes inimaginável de funções que afeta e aumenta, dramaticamente, a complexidade do software desses sistemas. Também, é muito comum que aplicações de propósito geral necessitem de ser executadas nessas plataformas. Aplicações as quais nem sempre foram desenvolvidas por pessoas com o conhecimento adequado acerca da programação para dispositivos embarcados [Wolf, 2003].

Nesse contexto, API's de alto nível, fornecidas pelos sistemas operacionais (SO's) são cada vez mais importantes. Além disso, torna-se muito comum a disponibilização dessas API's pela própria indústria, como incentivo ao desenvolvimento de aplicações embarcadas, como no caso das API's para desenvolvimento de aplicações para telefones celulares. Esse fato, porém, traz à tona o problema da segurança. No passado, somente os fabricantes tinham acesso a determinadas partes do sistema. Atualmente, com o fornecimento das API's, os desenvolvedores passam a ter acesso a partes sensíveis do sistema, o que pode torná-lo vulnerável a ataques visando à captura de dados confidenciais como, por exemplo, senhas bancárias.

Apesar de todo avanço conquistado, os sistemas embarcados continuam sendo muito diferentes dos sistemas de propósito geral: ainda possuem requisitos de tempo real estritos, além de necessitar customizações que visem à economia no consumo de energia, já que a vida útil da bateria deve ser o mais longa possível [Lavagno e Passerone, 2005]. Isso impacta na frequência de processador possível de ser adotada: normalmente frequências mais baixas são obrigatórias para que se possam atingir determinadas metas de consumo de energia.

Outra restrição comum diz respeito ao uso de memória, já que os usuários de dispositivos modernos anseiam por mais memória apesar de, tipicamente, exigirem dispositivos a preços acessíveis e com baixo consumo de energia (condição que pode facilmente ser violada pelo uso de mais memória) [Hohmuth *et.al.*2005].

Finalmente, é possível afirmar que sistemas embarcados são, cada vez mais, parte do cotidiano das pessoas, além de ser usado em sistemas especificamente críticos [Tanenbaum, 2007]. Apesar de essa ser uma classe de SE com muito mais restrições do que a classe *moderna* de SE's, o uso da virtualização em seus sistemas poderia trazer vantagens, como aumento da segurança e confiabilidade.

Nesse contexto, a virtualização pode ajudar os sistemas embarcados em muitos aspectos: aumentar o poder de processamento, aumentar a segurança além de permitir que usuários carreguem suas aplicações sem comprometer o sistema como um todo. Apesar disso, é mandatório que se estude como realizar a virtualização sem violar as diversas restrições existentes nos sistemas embarcados.

2.2 Virtualização

Como dito anteriormente, a virtualização permite que um único computador hospede múltiplas máquinas virtuais, sendo que existe isolamento entre elas com a possibilidade de que executem sistemas operacionais diferentes. A principal vantagem é que, se uma máquina virtual falha, as outras são mantidas funcionando a um custo razoável [Heiser, 2008].

No âmbito empresarial, apesar de isso significar que o sistema possui um único ponto de falha, uma vez que muitos servidores podem ser colocados em um hardware único, pode ser considerado que essa é uma abordagem mais segura porque a maioria das interrupções de serviço não é causada por falhas no hardware, mas sim, por problemas do software, que tipicamente é grande e complexo demais, o que atrapalha na sua manutenibilidade. Além disso, o software que normalmente contém grande parte desses erros é o sistema operacional. Nesse sentido, a virtualização utiliza um único elemento de software que é executado no modo *kernel*: o *hypervisor*, que tipicamente é – pelo menos – duas ordens de magnitude menor do que um sistema operacional e, portanto, menos suscetível a erros [Tanenbaum, 2007].

Adicionalmente, deve-se observar o funcionamento desse componente. De acordo com [Popek e Goldberg, 1974], existem dois tipos diferentes de *hypervisor*:

- tipo 1, conhecido como virtualização no nível de hardware, onde se considera que o *hypervisor* é um sistema operacional por si só, já que somente ele opera em modo *kernel*, como pode ser observado no lado esquerdo da Figura 1. Sua principal tarefa, além de controlar a máquina real, é prover a noção de máquinas virtuais, e;
- tipo 2, ou virtualização no nível de SO, onde o *hypervisor* é como qualquer outra aplicação de usuário e não tem acesso direto ao hardware (deve passar antes pelo sistema operacional da máquina). Nesse caso, perde-se uma das principais vantagens da virtualização que é justamente o uso de SO's diferentes. Por esse motivo, considera-se, neste trabalho, somente a virtualização no nível de hardware.

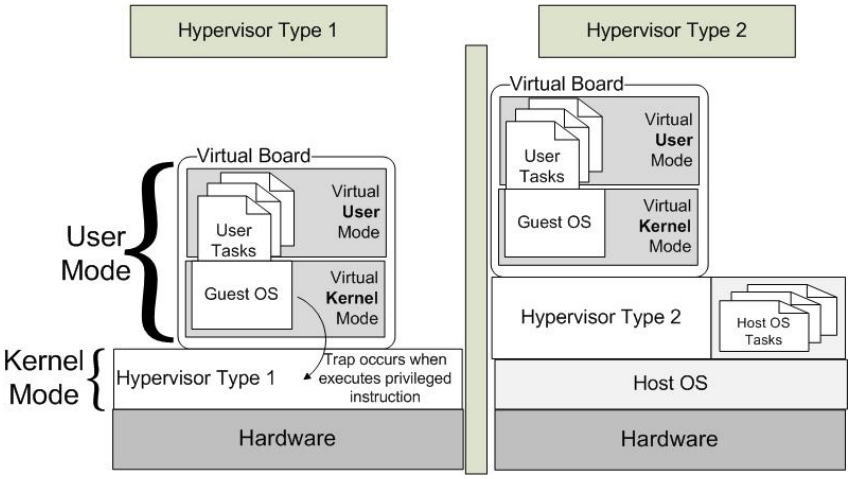


Figura 1 – Hypervisors tipos 1 e 2

Em ambos os casos a máquina virtual deve ter o mesmo comportamento do hardware verdadeiro. Mais especificamente, deve ser possível iniciá-la e reiniciá-la como um computador real, bem como instalar diferentes sistemas operacionais. A criação deste cenário é responsabilidade do hypervisor.

O hypervisor (independente de seu tipo) juntamente com o hardware, é responsável por lidar com as instruções vindas da máquina virtual. É importante destacar que uma vez que a máquina virtual "imita" o hardware real, também deve separar a execução nos modos *kernel* e usuário.

Nesse sentido, estudos clássicos de Popek e Goldberg [1974] introduzem uma classificação dos ISA (*Instruction Set Architecture*) em três grupos diferentes:

1. *instruções privilegiadas:* aquelas que causam em uma *trap* quando executadas em modo usuários mas que não causam *trap* se empregadas no modo *kernel*;
2. *instruções sensitivas de controle:* aquelas que tentam modificar a configuração dos recursos no sistema, e;
3. *instruções sensitivas de comportamento:* aquelas cujo comportamento ou resultado depende da configuração de recursos (o conteúdo do registrador de relocação ou o modo do processador).

Assim sendo, de acordo com os Popek e Goldberg [1974], para que a virtualização de uma dada máquina seja possível, as instruções sensitivas (de controle e comportamento) devem ser um subconjunto das instruções privilegiadas. Isso não é realidade em muitos processadores, como os da família Intel x86, e, nesse caso, a solução comumente perpassa por adotar suporte em nível de hardware por parte do processador. A Intel possui o *IntelVT (Virtualization Technology)* e a AMD possui o SVM (*Secure Virtual Machine*). O suporte pelo hardware pode não ser a melhor solução no caso dos sistemas embarcados, já que é interessante que a virtualização consiga lidar com o hardware existente, especialmente para acelerar o restrito *time-to-market.*

Conforme dito anteriormente, neste trabalho considera-se apenas a virtualização em nível de hardware. Detalhando a maneira pela qual essa técnica é empregada sem o suporte de hardware (como aquele fornecido pela Intel e pela AMD), é possível

observar que, inicialmente, sempre que a máquina virtual tentar executar uma instrução privilegiada (requisição de E/S, escrita em memória etc.), ocorre uma *trap* para o *hypervisor*. Isso é conhecido como virtualização pura e é normalmente uma forma muito ineficiente de se aplicar essa técnica [Waldspurger, 2002].

Outra opção no nível de hardware é conhecida como virtualização impura e requer que as instruções sensíveis (aquelas que causam uma *trap* no *hypervisor*) sejam removidas do código a ser executado na máquina virtual. Isso pode ser feito tanto em tempo de compilação (através de pré-virtualização) ou por reescrita de código binário, em tempo de execução, onde o código é vistoriado com o intuito de substituir tais instruções. O problema de ambas essas abordagens é que causam perda de desempenho.

Alternativamente, a técnica de *para-virtualização* pode ser empregada para substituir as instruções sensitivas do código original por chamadas explícitas ao *hypervisor* (*hypercalls*). Na verdade, o sistema operacional da máquina virtual está agindo como uma aplicação normal de usuário sendo executada sobre um sistema operacional normal[1], com a diferença que o sistema operacional convidado está sendo executado sobre o *hypervisor*. Quando a para-virtualização é adotada, o *hypervisor* deve definir uma interface composta por chamadas de sistemas que possam ser usadas pelo sistema operacional convidado. Ainda, é possível remover todas as instruções sensitivas do SO convidado, forçando-o a usar comente as *hypercalls* o que torna o *hypervisor* mais parecido com um *microkernel*, o que pode aumentar o desempenho da virtualização.

As diferenças entre virtualização pura e para-virtualização são mostradas na Figura 2. Na parte A dessa figura, a virtualização pura é mostrada. Nesse caso, sempre que o SO convidado utiliza uma instrução sensitiva, uma *trap* é causada no hypervisor, o que emula o comportamento de uma instrução com os resultados apropriados. Na parte B, a para virtualização é mostrada. O SO convidado deve ser modificado para efetuar as *hypercalls* ao invés de conter instruções sensitivas. Nesse caso, a *trap* é similar ao que ocorre em sistemas não virtualizados quando uma aplicação de usuário faz uma chamada de sistema do SO.

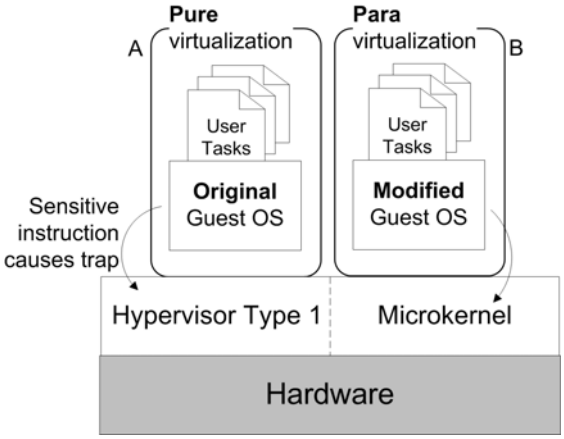


Figura 2 – Controle do *hypervisor* na virtualização pura (parte A) e na para-virtualização (parte B)

[1] Neste contexto, normal significa não virtualizado.

Através dos conceitos apresentados nesta seção, são discutidas a seguir as vantagens, desvantagens e maneiras para se usar a virtualização em sistemas embarcados.

3. Vantagens da virtualização em sistemas embarcados

Neste trabalho, considera-se que os sistemas embarcados são multiprocessados, não críticos, mas contêm restrições temporais. Sendo assim, nesta seção destacam-se possíveis casos de uso para a virtualização em tais sistemas.

Uma das vantagens mais apropriadas e diretas da virtualização em sistemas embarcados consiste em permitir que SO's diferentes possam coexistir na mesma máquina. Nesse caso, pode-se atacar dois problemas diferentes:

- o uso de software legado, já que é possível a criação (ou manutenção) de um sistema operacional compatível com esse software e outro mais moderno, que permita que novos recursos sejam explorados, e;
- dividir o sistema em uma parte onde o usuário tem acesso, com chamadas específicas conhecidas por ele, separadas da parte crítica, responsável por manter o dispositivo funcionando. Nesse caso, dois sistemas operacionais, um de usuário e outro de sistema, podem ser empregados simultaneamente.

Quando a virtualização for empregada com esses objetivos, o *hypervisor* deve ter controle total do hardware além de criar diferentes máquinas virtuais, uma por SO. Como pode ser observado na Figura 3, essa abordagem pode ser usada tanto em máquinas mono- ou multi-core. Ainda, permite que se aumente a qualidade de desenvolvimento de software, uma vez que o projetista pode escolher entre diversos SO's aquele mais adequado à sua aplicação. Além disso, o tempo requerido para desenvolver uma aplicação pode ser reduzido drasticamente, já que a reusabilidade das aplicações cresce sensivelmente [Shen e Petrot, 2009].

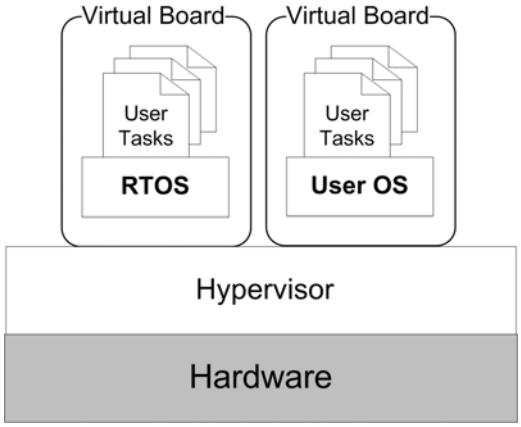


Figura 3 – Hypervisor para separação de máquinas com vários SO's

Além disso, essa abordagem oferece a vantagem de se alcançar uma arquitetura de software unificada que pode ser executada em múltiplas plataformas de hardware. Nesse caso, um problema atual e recorrente nos sistemas embarcados – a portabilidade de software – pode ser amplamente afetado e os projetistas têm o potencial de satisfazer mais rapidamente o *time-to-market* cada vez mais restrito.

Adicionalmente, a segurança do sistema embarcado também é um forte apelo para a virtualização, já que através dela pode se prevenir que ataques ocorridos ao sistema atinjam o SO principal, como pode ser visto na Figura 4. Nesse caso, o código malicioso fica restrito à maquina virtual, sem contaminar o resto do sistema, pois não possui o conhecimento necessário do *hypervisor* para poder explorar os seus pontos fracos. Ainda, o *hypervisor* pode detectar a ocorrência de um ataque e reinicializar a maquina virtual, sem prejudicar o resto do sistema.

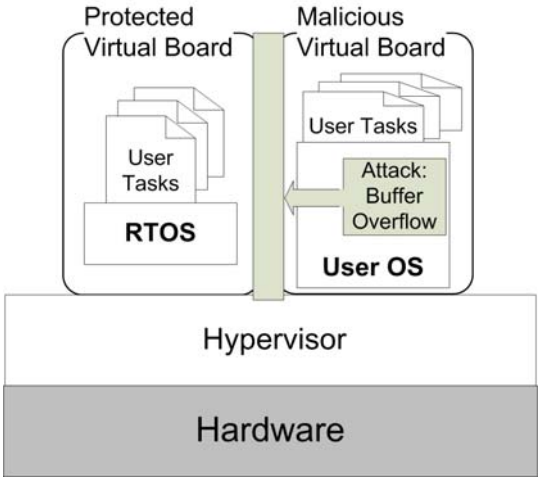


Figura 4 – Ataque de usuário bloqueado através do isolamento das máquinas virtuais

Consequentemente, o uso dos *hypervisors* previamente conhecidos, exatamente como aqueles adotados em sistemas de propósito geral, provavelmente não será possível. Com o objetivo de manter o hypervisor pequeno (para que ataques e falhas sejam mais difíceis), restrições específicas dos sistemas embarcados devem ser levadas em consideração, o que afeta todo o ciclo de desenvolvimento do software.

Em arquiteturas multi-core, existem maneiras diferentes de se utilizar os diversos processadores do sistema. Uma forma bastante comum e usual de se fazer isso é executar um único SO sobre todos os processadores, criando o que se conhece como configuração multiprocessada simétrica (SMP). Essa abordagem traz a vantagem principal de oferecer balanceamento de carga entre os processadores de forma direta. Entretanto, não permite o uso de SO's diferentes em um mesmo MPSoC, o que já foi argumentado anteriormente como sendo uma atrativa opção. Além disso, faz com que exista somente um ponto único de falha no sistema inteiro e – sempre que houver uma falha no sistema – todos os *cores* devem ser reiniciados.

Nesse caso, pode-se adotar uma configuração multiprocessada assimétrica (AMP) onde cada processador possui seu próprio SO, responsável por escalonar suas próprias tarefas. A configuração AMP consegue aproveitar diversas vantagens oferecidas pela virtualização, uma vez que provê a arbitragem do uso de recursos entre as máquinas virtuais, evitando que o SO de usuário cause um comportamento inesperado no RTOS [4]. Se nenhuma técnica de virtualização é empregada, a única forma de se conseguir tal separação é fazendo isso manualmente, o que pode ser mais complicado e mais propenso a erros. O *hypervisor* pode mapear cada máquina virtual em um *core* do sistema multi-procesado ou até mesmo mapear um único SO em múltiplos *cores*, criando um subconjunto SMP de *cores* [11].

Para o mundo empresarial, a redução de custos trazida pela virtualização além das possibilidades de atualizações facilitadas são vantagens promissoras que encorajam seu uso.

Sumarizando, as principais vantagens da utilização da virtualização em sistemas embarcados são:

- o uso de diferentes SO's;
- segurança e confiabilidade do sistema são aumentadas, e;
- diferentes alternativas para configuração de um ambiente multi-core.

4. Barreiras para o uso de virtualização em sistemas embarcados

Enquanto a virtualização embarcada pode trazer inúmeras vantagens, é importante que se esclareça a que custo esses benefícios podem ser alcançados. Algumas das limitações já estão presentes na virtualização de propósito geral, enquanto que outras surgem do seu uso em ambientes tão severamente restritos, como os sistemas embarcados.

Um dos principais problemas a ser atacado está relacionado com o escalonamento das tarefas realizado pelo *hypervisor*. Sistemas embarcados tipicamente têm restrições temporais e, por isso, qualquer deslize do *hypervisor* pode comprometer o sistema.

Pode-se ainda considerar o caso onde um dado multi-core apresenta um comportamento multi-processado assimétrico, com dois SO's: um de usuário e um RTOS. Nesse caso, cada SO é tratado como uma máquina virtual separada e, em sistemas embarcados, é desejável que o RTOS seja priorizado em relação ao SO de usuário, assim como tarefas de tempo-real que eventualmente sejam executadas no SO de usuário (como aplicações multimídia) também devem ter preferência. Esse escalonamento com prioridades vai de encontro com os princípios das máquinas virtuais, nos quais todas as máquinas virtuais devem dividir o hardware real em proporções iguais.

Além disso, a heterogeneidade típica de sistemas embarcados pode representar um grande desafio, já que o *hypervisor* tem de, teoricamente, conseguir comunicar-se com o maior número possível de arquiteturas. Enquanto que na computação de propósito geral a arquitetura Intel x86 é amplamente usada, por exemplo, em sistemas embarcados existe uma grande variedade de arquiteturas empregadas, desde DSP's a processadores ARM, passando ainda por arquiteturas PowerPC e MIPS.

Ainda, o isolamento excessivo e absoluto trazido pelas máquinas virtuais – que aumenta os níveis de segurança e confiabilidade – podem causar dificuldades para que os diversos subsistemas embarcados cooperem entre si, o que altamente desejável em sistemas embarcados.

5. Possíveis casos de uso para virtualização embarcada

Após considerar os conceitos previamente explanados, descrevem-se, nesta seção, alguns cenários onde a virtualização embarcada pode ser adotada.

No *Cenário 1* pretende-se reduzir o número total de processadores em um sistema, colocando-os em diversas máquinas virtuais sobre um único processador (seja

ele mono- ou multi-core). Na Figura 5 pode-se observar uma possibilidade de tal configuração.

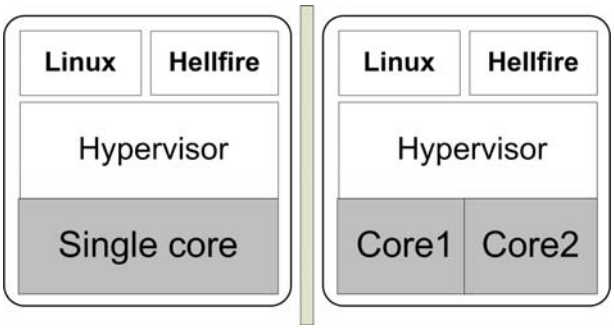


Figura 5 – Cenário 1: redução do número total de processadores

Mesmo no caso onde o processador oferece suporte de hardware à virtualização (como VT ou SVM), é desejável se adotar a para-virtualização, o que permite que o hypervisor torne-se independente de suporte além de trazer um aumento de desempenho.

No *Cenário 2* a confiabilidade de sistemas AMP pode ser aumentada através da separação dos recursos, com a capacidade de se reiniciar as máquinas virtuais, como pode ser visto na Figura 6.

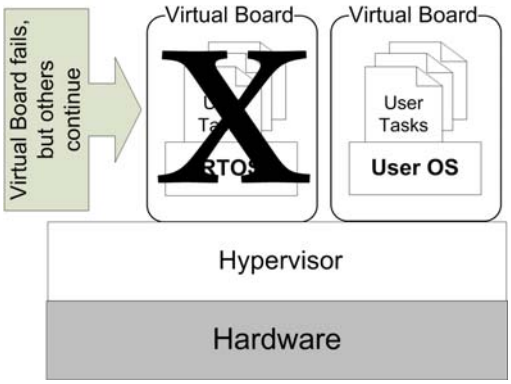


Figura 6 – Cenário 2: confiabilidade aumentada

No *Cenário 3* é possível migrar sistemas existentes em uma máquina virtual e adicionar novas funcionalidades a elas, provendo assim, oportunidade para reuso e inovação. Além disso, a migração de tarefas entre máquinas virtuais é facilitada. Esse cenário pode ser visualizado na Figura 7. É válido lembrar que as vantagens da migração em sistemas embarcados tem sido objeto amplo de estudo ao longo dos anos [Shen e Petrot, 2009], [Bertozzi *et. al.*, 2006], [Nollet *et. al.*, 2006].

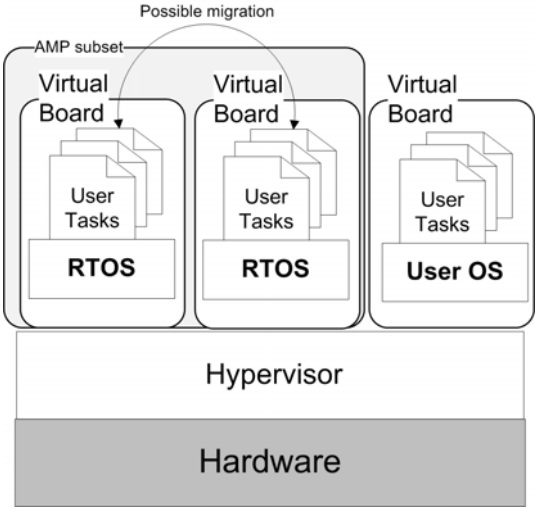


Figura 7 – Cenário 3: migração entre máquinas virtuais

No *Cenário 4*, a combinação de sistemas de tempo-real, legados e SO's de propósito geral no mesmo dispositivo é alcançada através da virtualização, como mostra a Figura 8.

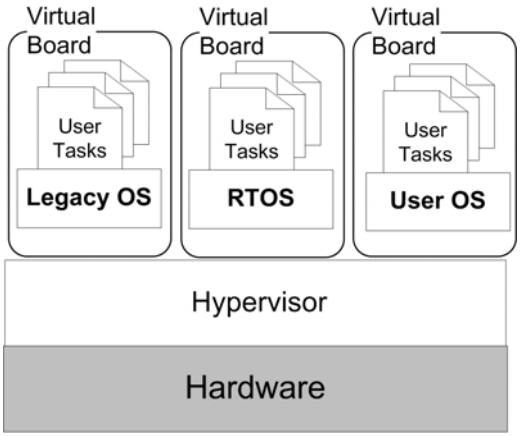


Figura 8 – Cenário 4: uso de software legado com novas aplicações de usuário

É válido lembrar que todos esses cenários são bastante adequados para sistemas embarcados não críticos, como dispositivos de entretenimento móveis. Especialmente, uma vez que a demanda dos usuários continua a crescer, cada vez mais o *time-to-market* está restrito e a virtualização pode ajudar no seu cumprimento.

6. Considerações finais

Virtualização tem sido amplamente usada em sistemas corporativos como forma de aproveitar melhor o poder de processamento das máquinas multi-core. Enquanto isso, sistemas embarcados costumavam ser extremamente restritos e de propósito específico. Esse cenário, no entanto, tem se modificado. Sistemas embarcados cada vez mais fazem parte da vida das pessoas e suas múltiplas funcionalidades levam a um crescimento não linear da complexidade de software. Nesse contexto, muitas soluções têm sido estudadas, entre elas, a virtualização. Os principais apelos para se usar a virtualização em sistemas embarcados são: (i) permitir que diversos SO's sejam executados em um

mesmo processador; (ii) reduzir o custo de fabricação, através do aumento da utilização dos processadores do sistema; (iii) aumentar a confiabilidade e a segurança, e; (iv) ajudar a diminuir a complexidade inerente do desenvolvimento de software em sistemas embarcados.

O desafio, portanto, é adaptar as abordagens nas quais o hypervisor aproxima-se a implementações de um microkernel, com o objetivo de se desenvolver técnicas de virtualização mais leves, adequadas para os sistemas embarcados atuais e futuros.

Referências

Bertozzi, S., Acquaviva, A., Bertozzi, D. e Poggiali, A. Supporting task migration in multi-processor systems-on-chip: A feasibility study. In *Design, Automation and Test in Europe, 2006.* DATE'06. *Proceedings*, pages 1-6, 2006.

Hansson, H., Nolin, M. e Nolte, T. Real-Time in Embedded Systems. In *Richard Zurawski, editors, Embedded Systems Handbook, chapter 2*. CRC press, 2005.

Heiser, G. The role of virtualization in embedded systems. *IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 11--16, New York, NY, USA, 2008. ACM.

Hohmuth, M., Peter, M., Härtig H. e Shapiro, J. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop, pages 22*, New York, NY, USA, 2004. ACM.

Jerraya, A., Tenhunen, H. e Wolf, W. (2005). Multiprocessor systems-on-chips. In *Computer*, 38 (Issue 7), pages 36 – 40.

Lavagno, L. e Passerone, C. (2005). Design of embedded systems. In *Embedded Systems Handbook, chapter 3*, R. Zurawski (editor). CRC press.

Martin, G. (2006). Overview of the mpsoc design challenge. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 274–279, New York, NY, USA. ACM Press.

Nollet, V., Avasare, P., Mignolet, J-Y. e Verkest, D. Low cost task migration initiation in a heterogeneous MPSoC. In *DATE'05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 252-253, Washington, DC, USA, 2005. IEEE Computer Society.

Popek, G. e Goldberg, R. (1974). Formal requirements for virtualizable third generation architectures. In *Communication*. 17(7). pages 412–421. ACM press.

Shen, H e Petrot, F. Novel task migration framework on configurable heterogeneous MPSoC platforms. In *Design Automation Conference, 2009.*ASP-DAC 2009. *Asia and South Pacific*, pages 733-738, Jan. 2009.

Tanenbaum, A. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

Waldspurger, C. (2002). Memory resource management in VMWARE ESX server. In: *SIGOPS Operating Systems Review*, 36(SI). pages 181–194.

Wolf, W (2003). A decade of hardware/software co-design. In: *Computer*, 36(4). pages 38–43.

A real-time system based on FPGA to measure the transition time between tasks in a RTOS

Lidia H. Shibuya, Sandro S. Sato, Osamu Saotome, Fernando G. Nicodemos

Instituto Tecnológico de Aeronáutica – ITA
Praça Marechal Eduardo Gomes, 50 - Vila das Acácias - CEP 12.228-900
São José dos Campos – SP – Brasil

lishibuya@uol.com.br, shoiti@ita.br, osaotome@ita.br,
fgnicodemos@terra.com.br

Abstract. *This paper describes the conception, the design, the development, the implementation and the experimental results of a real-time electronic instrumentation system that measures accurately the transition time between tasks running under a Real Time Operating System (RTOS) supervision.*

1. Introduction

The use of Real Time Operating System (RTOS) has been increasing over the years in embedded system design. A RTOS is designed to meet rigorous time constraints. Nevertheless, selecting a RTOS for an embedded application can be a complex process especially when it is to be used in critical applications. For these critical applications, e.g. space applications or medical applications, the use of hard-real time system is a requirement [Burns, 1991]. Using Kopetz's definition (2002): *"A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced."*

One important feature of a RTOS is the fact that the scheduler performs more deterministic scheduling of tasks. The scheduling policy used by the RTOS scheduler gets impacts in the overall system performance. The use of an RTOS does not guarantee that the system will always meet the deadlines, once it also depends on how the overall system is designed.

It means that every single task shall have a well defined time to be processed and to output a result. It also suggests that the inherent transition time to switch among tasks should be considered during the development of the software for hard-real time systems. Due to the several possible internal processors architectures, including for embedded applications, that implements for example, different levels of cache memory and depths of pipeline, the performance of the processor can change. In addition, the performance of different RTOS can vary enormously.

Many RTOS have presented some specific software components to measure their characteristics. These components are usually software routines attached to the system, as for example the debug capacity of the RTEMS gdb compiler [OAR, August 2008]. When the performance is measured using these attached software routines, the results can differ from each RTOS, because each RTOS usually implements different

functions, as a result, the data gathered from different RTOS components cannot be compared.

Efforts have been made to measure the performance of RTOS as shown in Anh and Tan (2009), Martinez et. al (1996), Kar and Porter (1989) and Sacha (1995). These RTOS performance analyzers are based on benchmarking method of frequently used system services. The performance of an RTOS is measured verifying if the task complies with its deadline. As there are various types of applications with each having very different requirements, benchmarking against any generic applications will not be reflective of the RTOS strengths and weaknesses.

Another approach is presented by Lamie and Carbone (2007), where they provide some APIs which can be used in different RTOS to verify the performance. But in this proposal, there is software interference on the performance measurements. According to the target, some changes also may be required. The JEWEL tool proposed by Lange et. al (1992) was developed to analyze the performance of distributed systems. Also Wybranietz and Haban (1988) proposed integrated tool for monitoring distributed systems continuously during operation.

The main objective of the test environment proposed in this paper is to measure absolute time characteristics of a RTOS, that influences performance, for embedded applications, without the insertion of overhead, neither including changes on the embedded software. This approach allows time comparisons that may be helpful to analyze different RTOS performances in different processors. To show the proposed real-time system based on FPGA to measure the transition time between tasks in a RTOS, this work contains 5 sections. In Section 2 an overview of the proposed solution is presented, introducing the elements used on the implementation. Section 3 presents in details the system implementation. In Section 4, experimental results are described. The paper ends with a conclusion, acknowledgment, and references.

2. Proposed Solution

This paper proposes a real-time system based on FPGA to measure the transition time between tasks in a RTOS. The use of FPGAs allows many processes to start at the same time [D'Amore, 2007] which creates an efficient tool to analyze the performance of a device running RTOS, and there is no overhead due to software initialization or software modifications that usually occur on the debuggers.

In order to determine the transition time between the tasks in a RTOS, the first step was to decide which device should be chosen as the target. In this case, the Device Under Test (DUT) is an ERC32 development kit [ATMEL, August 2003]. The ERC32 is a radiation-tolerant 32-bit RISC processor developed for space applications, [ATMEL, August 2004] and [ATMEL, March 2005].

The second step was to choose a RTOS. The chosen DUT accepts the real time operating system RTEMS (Real-Time Executive for Multiprocessor Systems) [OAR, 2006].

The third step regards to the choice of the task model to be implemented on the DUT. The task should produce stimulus for the data acquisition module, where these stimulus could be read and stored in a temporary memory. Once the data acquisition

module reads the stimulus, this data were acquired by dedicated software, in order to analyze the data and show the results in graphical format.

The general architecture of the Project is shown in Figure 1. The dedicated software, running on the PC, collects and provides the analysis. The FPGA was configured to monitor the GPI (General Purpose Interface) collecting data and sending it to the Personal Computer (PC). The DUT was configured to run the RTOS RTEMS to generate signals to the GPI.

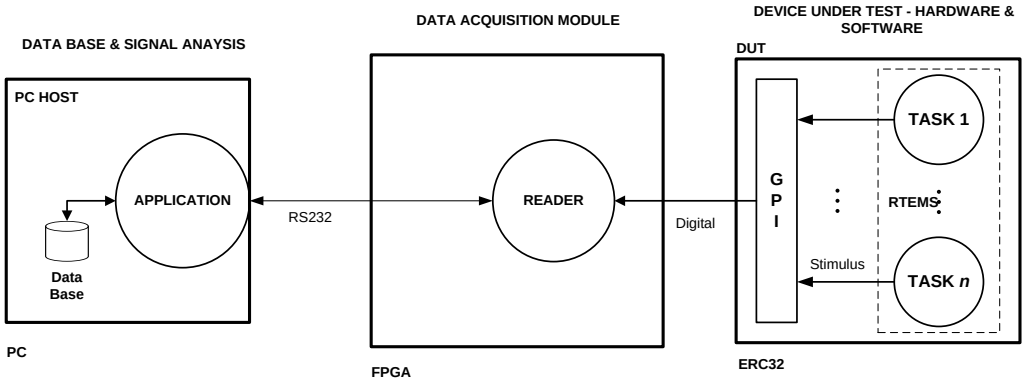


Figure 1. Hardware General Architecture

3. System Implementation

The proposed solution shown in section 2 was organized and implemented in three components: System Under Test, Data Acquisition Module and Software Analysis Module.

The system under test is composed by an ERC32 processor (SPARC V7 architecture) running the RTOS RTEMS. The RTEMS was configured with four tasks that run equivalent routines.

The data acquisition module was implemented on a FPGA, which continuously monitors the signals from the DUT and when it detects that the task changed the Data Acquisition Module, stores the information that will be processed after the software analysis module.

The analysis software was developed in Visual Studio. It receives all the data collected from the acquisition module and decodes them in order to generate graphics that simplifies the view and the analysis process.

3.1 Device Under Test - DUT

The Device Under Test (DUT) is an ERC32 processor. This processor is used mostly in aerospace application. It is developed based on SPARC V7 architecture. The tools required to develop RTEMS codes and to connect to the development kit (DUT) are based on Linux. This project uses Fedora 7 platform.

The ERC32 processor has a GPI (General Purpose Interface) that is used to signalize when a task is preempted. This GPI interface is composed by a register of 8 bit which can be configured as inputs or outputs. To identify the GPI, the GPI[0~7] notation is used, and the GPI[0], GPI[1] through GPI[7] is used to identify only one interface. For this work the entire GPI was configured as outputs.

The running software was developed in C language. The software was build using RTEMS development tools. The main characteristics of the implemented tasks and the RTEMS configuration are:

- Four equivalent tasks with the same priority level were implemented using RTEMS.
- Each task was implemented as an infinite loop that produces a periodic square wave in the GPI.
- The RTEMS was configured to use round-robin scheduling with a fixed time slice.

Figure 2 shows the general idea of the task model used in the software architecture. The cooperative context switching task model proposed by Lamie and Carbone (2007) was chosen.

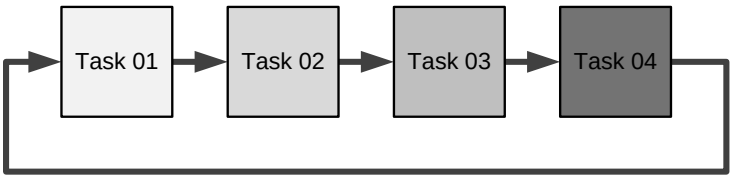


Figure 2: Task model

Figure 3 shows an overview of the system under test and the output expected on the GPI.

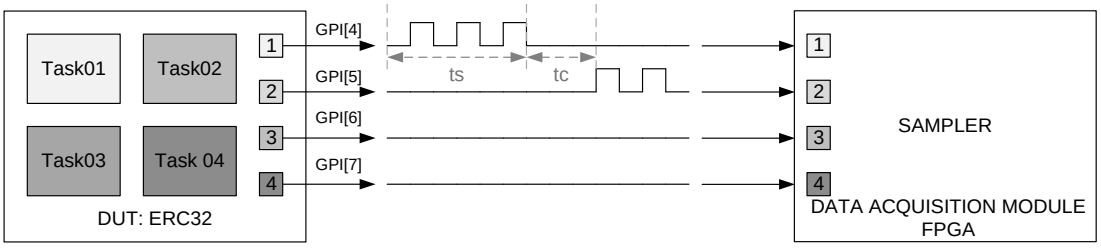


Figure 3: General overview of the device under test and the output on GPI[4~7]

In Figure 3, on the left, each task produces a square wave in one GPI output. On the right, the output expected on the GPI, when the scheduler achieves the time set on the time slice, represented by (ts). There is transition time, represented by (tc), between the Task01 and Task02. Along the time, after each time slice, there is a transition time (tc) between the tasks. It is important to emphasize that each task produces a square wave on a specific GPI output and keeps all the other GPI interfaces in zero, e.g, task01 produces a square wave on GPI[4] output, task02 produces a square wave on GPI[5], task03 outputs to GPI[6] and task04 outputs to the GPI[7]. The period of the square wave can be changed through an internal counter on the task routine. The RTEMS configuration parameters were configured as follows.

```
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_MAXIMUM_TASKS             5
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_EXTRA_TASK_STACKS
(4*RTEMS_MINIMUM_STACK_SIZE)
#define CONFIGURE_USE_MINIIMFS_AS_BASE_FILESYSTEM
#define CONFIGURE_MICROSECONDS_PER_TICK     10000 // the default is 10000
#define CONFIGURE_TICKS_PER_TIMESLICE       50    // the default is 50
```

The piece of code shows task01 implementation. It is important to remember that the four tasks have the same priority level and perform the same code. The only difference among the tasks is the fact that each task produces a square wave in one specific GPI, from GPI[4] through GPI[7].

```
/* Task 01 - GPIO 4 */
rtems_task Task_GPIO4(rtems_task_argument unused) {
  unsigned char count = 0;
  for (;;) {
    for (count=0; count<16;) {
      *leds = 0x10; /* GP4 <= '1' */
      count++;
    }
    for (count=0; count<16;) {
      *leds = 0x00; /* GP4 <= '0' */
      count++;
    }
  }
}
```

Figure 4 depicts the GPI output on GPI[4] and GPI [5] after the implementation on the DUT (ERC32). The measured transition time among the tasks on the scope was about 135us.

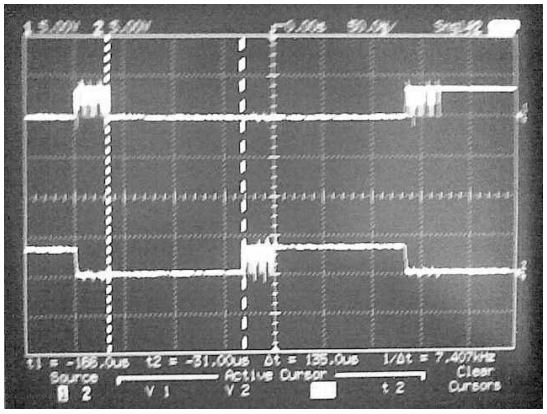


Figure 4: Output on GPI measured on the scope

3.2 Data Acquisition Module

The Data Acquisition Module makes use of an ALTERA FPGA. The development environment for the FPGA is composed by the Quartus® II software [ALTERA, October 2007] and a Stratix II development kit [ALTERA, May 2007] and [ALTERA, January 2007].

The FPGA was programmed with VHDL and ALTERA Mega-Function Blocks. The Data Acquisition Module can be divided in two functional blocks: the Sampler and the Communication block.

Figure 5 shows the block diagram of the Data Acquisition Module implemented in FPGA. The communication controller transfers all the data storage to the PC.

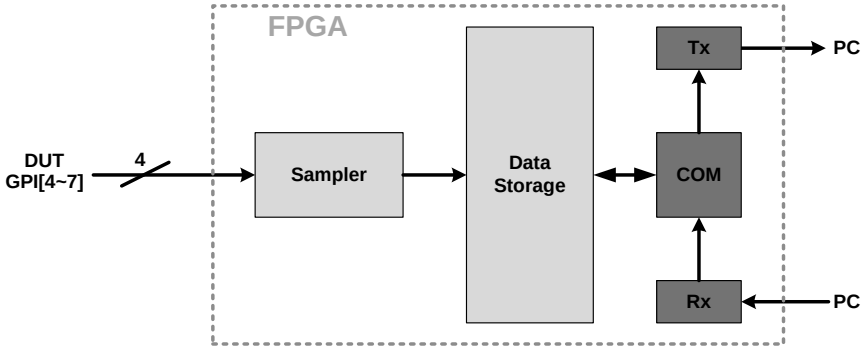


Figure 5: Overview of FPGA implementation

The Sampler Block performs the monitoring and analyzing function. This block has a free-running counter (ticks counter). This free-counter tick value is stored when the hardware detects a task transition on the GPI. At this moment, the minimum historical data are stored, to permit infer the transition time between the tasks. To monitor the GPI, the data acquisition module reads the GPI as an 8 bits register. The four low significant bits (GPI[0~3]) are always read as zero.

Figure 6 represents the moment of a transition time. The samples occur on the rising edge of the data acquisition module clock. A First In First Out (FIFO) register stores the three transition in the GPI that represents the transition time, ticks t_5, t_7 and t_{14}. The data are stored in the FPGA memory and when the FPGA receives a command from PC, the FPGA dumps the memory to PC.

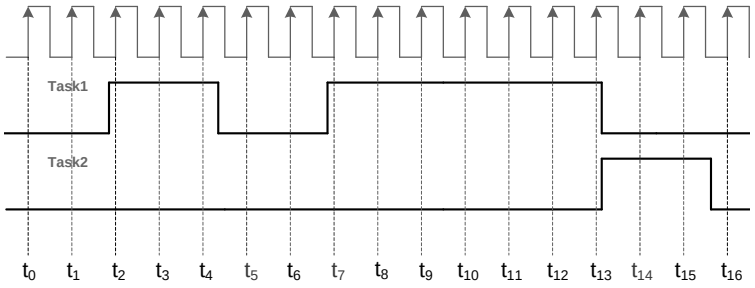


Figure 6: Example of a GPI output and the sample times for transition time analysis

The Communication Block was developed to send data to PC and to receive commands from PC. This block is basically a serial interface with configurable baud rate.

The data acquisition module presents some inherent characteristics due to the hardware description used in the implementation and also due to the task behavior, a periodic square wave. These characteristics contribute in a non-desirable manner on the transition time.

Figure 7 shows the condition where the non-desirable additional contribution has the minimum value. In this case, the data acquisition module stores the time presented by tm (ts – time slice, tc – transition time between the task). At the moment that the square wave toggles its status from zero to one, the transition time (tc) starts. The next task starts the square wave from zero to one. In this case, the transition time measured (tm) is almost the same as the transition time between the tasks (tc).

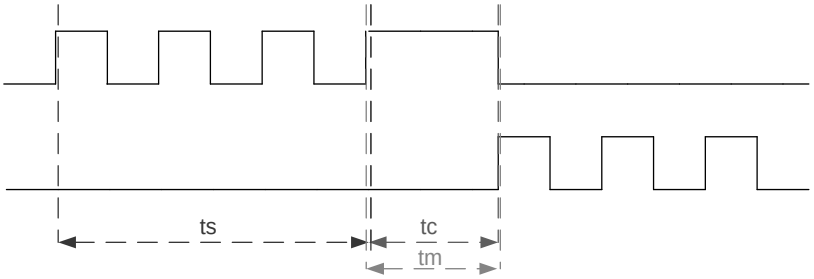


Figure 7. Minimum non-desirable contribution on the transition time measured by the data acquisition module

There is, although, one condition where the non-desirable contribution has the maximum value. This condition is presented on Figure 8. In this case, the task toggles its GPI from one to zero and on the moment before of switching again to one the transition time (tc) starts. If the next task starts from zero, the data acquisition module is not able to store the transition time until the task switches to one. The picture shows the transition time measured by the data acquisition module. As one can see, the non-desirable contribution in this case is about one period of the square wave.

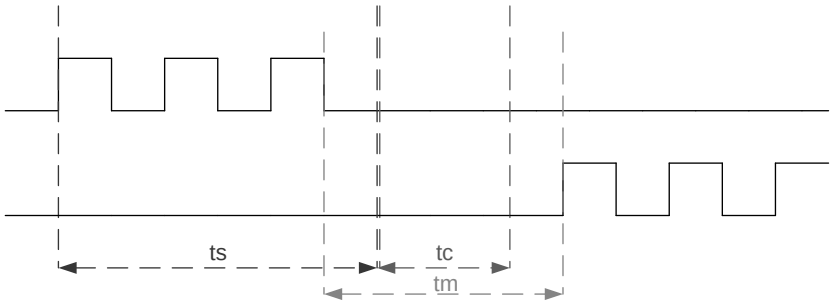


Figure 8. Maximum non-desirable contribution on the transition time measured by the data acquisition module

The non-desirable contribution can be reduced increasing the frequency of the square wave for each task, reducing consequently the period, as shown in Figure 9. Once the data acquisition module works with a clock more than 1000 times faster than the frequency of the square waves on GPI, it's possible to increase the square wave frequency without loss of accuracy of the measurements.

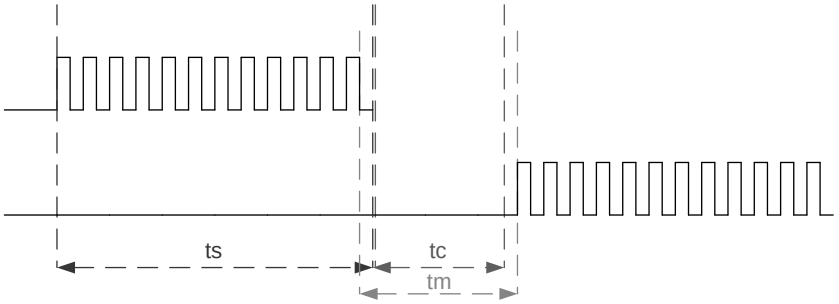


Figure 9. Square wave with higher frequency reduces the worst condition for the data acquisition module.

3.3 Analysis Module

The main function of the Analysis software is to receive, to verify and to generate the graphs. The development tools used for analysis module are composed by the Visual Studio .NET Framework to acquire the data and the Dundas software to the graphical interface [Dundas, 2009].

The PC host, with the developed analyzer software uses the serial interface. The transfer rate, frame format (start bit, parity and stop bit) must be configured, using also the developed software. The Figure 10 shows the developed software window.

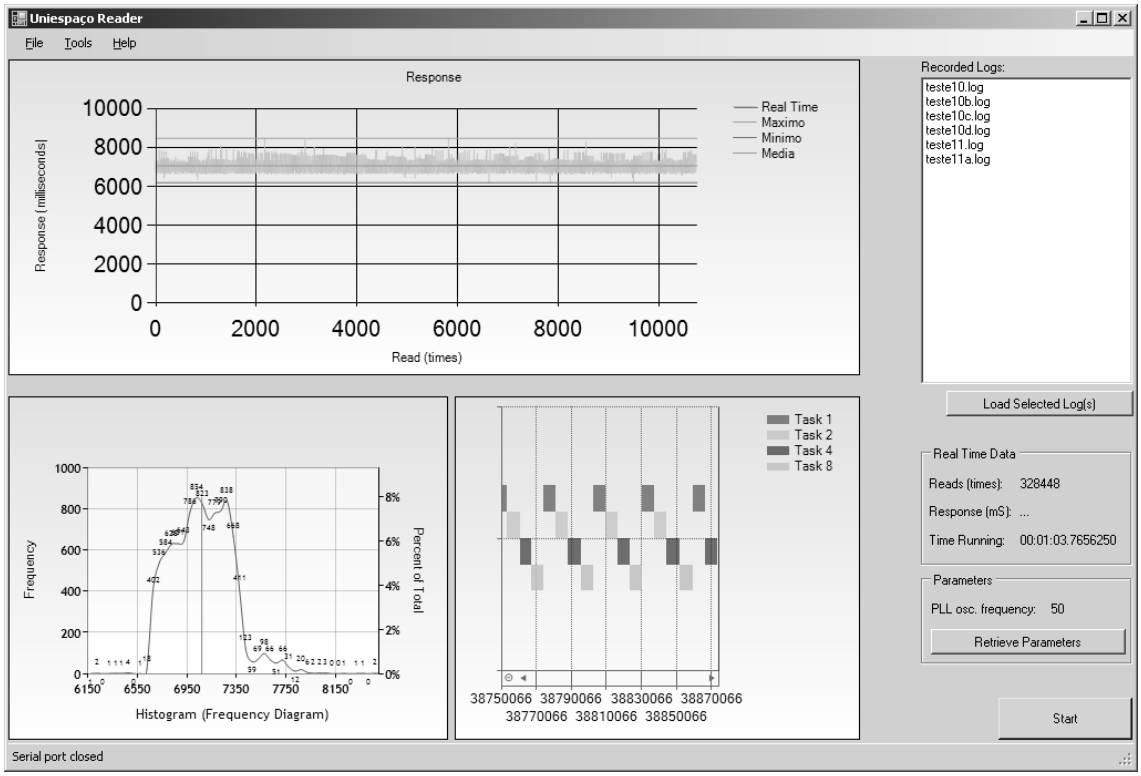


Figure 10. Data Acquisition and Analysis software

Figure 10 shows the developed software window. On the top, the x-axis represents the number of samples acquired and the y-axis represents the measured transition time (number of ticks) for each cooperative context switching. In this graph there are also three lines representing the maximum value, the minimum value and the average value resulting from the analysis. On bottom-left, a histogram is presented to show that the measure has an average behavior. On bottom-right the Gantt graph represents for how long one task had been running on the processor. The purpose of this graph was to show the round-robin scheduler.

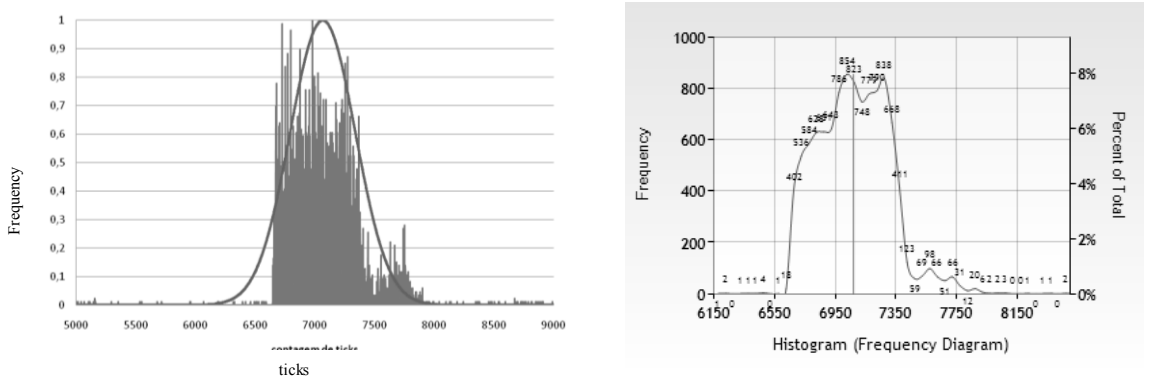
4. Experimental Results

The results depicted in this section were acquired using the developed software. There is also a comparison among the data acquired by the developed software and a manual analysis of the acquired data. The data were acquired using the parameters:

#define CONFIGURE_MICROSECONDS_PER_TICK 1000

#define CONFIGURE_TICKS_PER_TIMESLICE 5

The FPGA was configured to work with a clock of 50MHz, the number of ticks (in average) measured using the analysis software was about 7000 ticks, which results in a transition time of about 140us (7000(ticks)/50000000Hz).



(a) Ticks (b) Histrogram

Figure 11. Results one acquisition: (a) ticks and (b) Histogram

Figure 11a, on the right, depicts the results of one acquisition from the PC host software and the Figure 11b, on the left, depicts the results of the same acquisition, but analyzed manually, using the Excel. As one can see, both pictures have the same shape. The attenuation on the small picture can be explained once the developed software for the PC host considers a range of values (ticks) to plot the graph and the manual analysis is plotted using the ticks (x axis) incremented one by one. These transition time values were compared with the expected value of 135us, observed on the oscilloscope in the lab.

As it was explained on Section 3.2, the results presented in both graphs in Figure 11 show the measurements considering all the possibilities of additional contributions due to the task implementation. The difference between the maximum and minimum value taking into account the average value is about ±395 ticks, which represents a range of ±7,9us.

5. Conclusion

The results of this implementation show that it is possible, using the proposed real-time system based on FPGA, to measure the transition time between tasks in a running RTOS. These measurements can be helpful to model more precisely the tasks that one RTOS shall perform and to estimate with a better level of accuracy the real time performance accuracy of the software, particularly, contributing to the study of safe critical applications.

6. Acknowledgements

We want to thanks AEB and UNIESPAÇO program for the financial support of the project.

7. References

ALTERA (January, 2008) “Stratix II Device Handbook, Volume 2 – SII5V2-4.4”,January.

ALTERA (May, 2007) “Stratix II Device Handbook, Volume 1 – SII5V1-4.3”, May.

ALTERA (October, 2007) “Introduction to the Quartus® II Software – Version 7.2”, October.

ATMEL (August, 2003), “Evaluation Board TSC695 – User Guide – Rev. 4139F–AERO–08/03”, http://www.atmel.com, April.

ATMEL (August, 2004), “Low-Voltage Rad-Hard 32-bit SPARC Embedded Processor – Datasheet – Rev. 4204C–AERO–08/04”, http://www.atmel.com, April.

ATMEL (March, 2005), “TSC695 SPARC V7 Processor (ERC32) – Development Tools – Rev. 7503A-AERO–03/05”, http://www.atmel.com, April.

Burns, A, (1991) “Scheduling hard real-time systems: a review”, Software Engineering Journal, IEEE. p.p 116-128

D’amore, R. (2005), “Vhdl – Descrição e Síntese de Circuitos Digitais”, LTC, 275 pages.

Dundas (2010), “Dundas Dashboard Documentation”, http://support.dundas.com/Dashboard1.Documentation.ashx, April.

Kar, R. P., Porter, K., (1989) “Rhealstone: A real-time benchmarking proposal”, Dr. Dobb's Journal

Lamie, W., Carbone, J. (2007) “MEASURE YOUR RTOS'S REAL-TIME PERFORMANCE”, www.embedded.com

Lange, F., Kröger, R., Gergeleit, M., (1992) "JEWEL: Design and Implementation of a Distributed Measurement System", IEEE Trans. on Parallel and Distributed Systems, Vol. 3, No. 6, pp. 657-671

Martinez, A., Conde, J. F., Vina, A., (1996) “A comprehensive approach in performance evaluation for modern real-time operating systems”, Proceedings of the 22nd EUROMICRO Conference, pp. 61,

OAR (August, 2008), “Getting Started with RTEMS - Edition 4.8.1, for 4.8.1”.

OAR, (2006), “RTEMS 4.8.1 On-Line Library”, http://www.rtems.org/onlinedocs/releases/rtemsdocs-4.8.1/share/rtems/html/, April.

Sacha, K. M., (1995) “Measuring the real-time operating system performance”, Seventh Euromicro workshop on Real-time systems proceedings, Odense, Denmark, pp. 34–40

Wybranietz, D., Haban, D. (1988) "MONITORING AND PERFORMANCE MEASURING DISTRIBUTED SYSTEMS DURING OPERATION", ACM SIGMETRICS Performance Evaluation Review, Volume 16, Issue 1, pp 197 – 206

I Workshop de Sistemas Embarcados

♦

Sessão Técnica 2

Arquiteturas I

Avaliação do Custo de Comunicação com a Memória Externa de uma Arquitetura em Hardware para Estimação de Movimento H.264

Alba S. B. Lopes[1], Ivan Saraiva Silva[2]

[1]Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte (UFRN) – Natal, RN – Brasil

[2]Departamento de Informática e Estatística
Universidade Federal do Piauí (UFPI) – Teresina, PI – Brasil
alba@lasic.ufrn.br, ivan@ufpi.edu.br

Abstract. *This paper presents an architecture for H.264 Motion Estimation. The proposed architecture uses a fixed block size 8x8 pixels and a search area 32x32. This paper also presents a cost evaluation of the communication of the proposed architecture with external memory using the Altera DE2 prototyping kit. The impact of this cost in architecture global performance is analyzed and some ideas to improve this performance are given.*

Resumo. *Este trabalho apresenta uma proposta de arquitetura para um núcleo de Estimação de Movimento segundo o padrão H.264. A arquitetura proposta utiliza um tamanho de bloco fixo 8x8 pixels e uma área de pesquisa 32x32. Neste trabalho é também apresentando um estudo realizado sobre a comunicação da arquitetura proposta com a memória externa utilizando o kit de prototipagem Altera DE2. É feita uma avaliação do impacto desta comunicação no desempenho final da arquitetura e são apresentadas propostas para melhorar este desempenho.*

1. Introdução

Atualmente, diversos aparelhos eletrônicos como celulares, DVD *players*, filmadoras, câmeras e TVs digitais são capazes de manipular vídeos digitais [Agostini 2007]. Um vídeo tal como foi capturado exige uma quantidade muito elevada de bits para a sua representação. Entretanto, uma característica intrínseca destes dados é a redundância que eles apresentam [Ghanbari 2003]. Dessa forma, grande parte dos dados utilizados para a representação da informação é desnecessária. A compressão de vídeos torna-se, então, essencial para possibilitar o armazenamento e principalmente a transmissão desses dados.

O padrão H.264 [ITU-T 2007] é o mais novo padrão de compressão de vídeo que atingiu seu objetivo ao reduzir em aproximadamente 50% a quantidade de bits necessária para a representação da informação ao ser comparado aos demais padrões existentes na atualidade. O ganho foi em conseqüência de um alto grau de complexidade computacional inserido na codificação.

Um vídeo é formado por uma seqüência de imagens, chamadas quadros, que costumam apresentar certo grau de similaridade entre eles. Esta similaridade é

conhecida como redundância temporal [Shi e Sun 1999]. No codificador de vídeo, o módulo responsável por tratar desta redundância é a Estimação de Movimento (do inglês *Motion Estimation - ME*). A ME é o módulo que possui a mais elevada complexidade computacional do codificador do padrão H.264 [Deng, Gao, Hu e Ji 2005]

Neste trabalho é apresentada uma arquitetura para um núcleo de ME e é feita uma avaliação do impacto da comunicação do núcleo com a memória externa à arquitetura. Na memória externa são armazenados os quadros do vídeo que devem ser codificados. Este é um assunto pouco abordado quando se trata de núcleos para a ME.

Este artigo está organizado da seguinte forma: a seção 2 apresenta uma breve descrição do funcionamento da ME. Na seção 3 é apresentado o núcleo para a estimação de movimento proposto. A seção 4 traz uma avaliação do desempenho do núcleo. A seção 5 apresenta um estudo de caso realizado com um kit de prototipagem que possui duas memórias externas disponíveis e apresenta o impacto no desempenho da arquitetura ao acrescentar os acessos à memória externa. Por fim, a seção 6 apresenta as conclusões e trabalhos futuros.

2. Estimação de Movimento

Para a estimação de movimento são fornecidos um quadro atual e quadros de referência (o padrão H.264 permite a utilização de múltiplos quadros de referência). Cada quadro atual é dividido em regiões não sobrepostas chamadas blocos [Verma e Akoglu 2008]. Para cada bloco, a ME deve realizar uma busca em uma região do quadro de referência denominada área de pesquisa, como apresentado na Figura 1. Esta busca deve resultar na geração de um vetor de movimento que indique o deslocamento relativo do bloco do quadro atual para a região do quadro de referência.

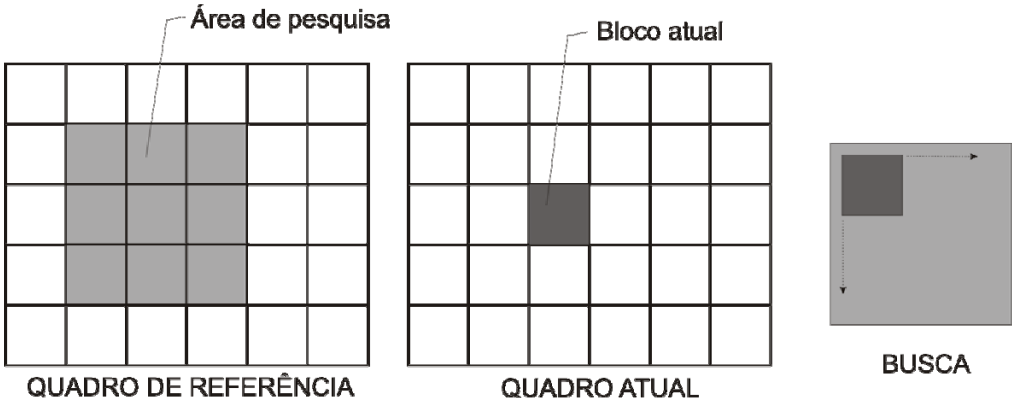


Figura 1. Representação da Estimação de Movimento

Existem diversos algoritmos para a busca que visam balancear desempenho e qualidade do vetor gerado. A Busca Completa (do inglês *Full Search*) prioriza a qualidade em relação ao desempenho [Bhaskaran e Konstantinides 1997]. Este algoritmo é chamado algoritmo ótimo, pois sempre encontra o melhor vetor existente. Algoritmos subótimos, como a Busca em Diamante (do inglês *Diamond Search*), fornecem um bom desempenho e um vetor aceitável, mas não o melhor [Yi e Ling 2005].

A avaliação da diferença entre dois blocos é feita através um critério de similaridade. Existem diferentes funções que servem para quantificar esta similaridade. Na ME, a medida mais utilizada é o SAD (do inglês *Sum of Absolute Differences*). [Vanne 2006] que calcula a diferença em módulo entre os pixels do bloco do quadro atual e seu respectivo no quadro de referência.

A arquitetura para a ME proposta neste trabalho utiliza tamanho de bloco 8x8 pixels, área de pesquisa 32x32, algoritmo da Busca Completa para a varredura da área de pesquisa e o SAD como critério de similaridade.

3. Arquitetura de um núcleo para a Estimação de Movimento

A arquitetura apresentada na Figura 2 é formada por uma memória local, três módulos de processamento, um banco de registradores de bloco e um comparador. Mais detalhes sobre cada um desses módulos serão apresentados nas subseções seguintes.

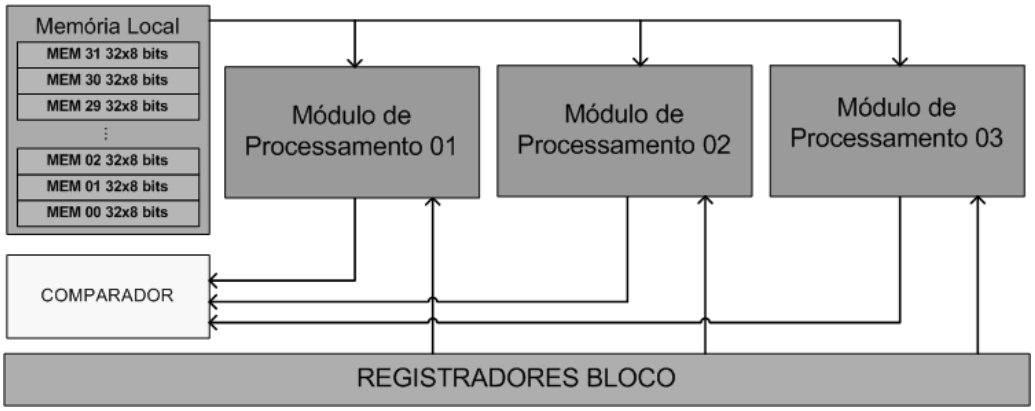


Figura 2. Arquitetura proposta de um núcleo para a Estimação de Movimento

3.1 Banco de Registradores de Bloco

Este módulo possui 64 registradores (dispostos em forma de matriz quadrada na Figura 3) para armazenar os 64 valores referentes a um bloco 8x8. Cada registrador do módulo possui capacidade de armazenamento de 8 bits, referentes a 1 pixel do bloco. Oito pixels são passados como entrada para o módulo, que através de deslocamentos sucessivos realiza o preenchimento completo da matriz 8x8. Só há um banco de registradores de bloco no núcleo e todos os módulos de processamento utilizam os dados nele contidos.

3.2 Módulo de Processamento

Cada Módulo de Processamento (MP) é formado por quatro Unidades de Processamento (UPs) e um banco de registradores de referência, como ilustra a Figura 4.a.

No banco de registradores de referência (Figura 4. b) são armazenados os valores de parte da área de pesquisa com a qual no momento será realizado o casamento com o bloco atual. Este banco é capaz de realizar deslocamento dos dados em todas as direções. Como exemplo, seja o dado que se encontra no registrador REF 00 representado na Figura 4.b. Após ser realizado um deslocamento para baixo, este dado irá para o registrador REF 10. Ao realizar um deslocamento para a direita, o dado

passará para o registrador REF 01. Em deslocamentos para cima e para a esquerda, o dado será descartado. O procedimento é análogo para os demais registradores do módulo.

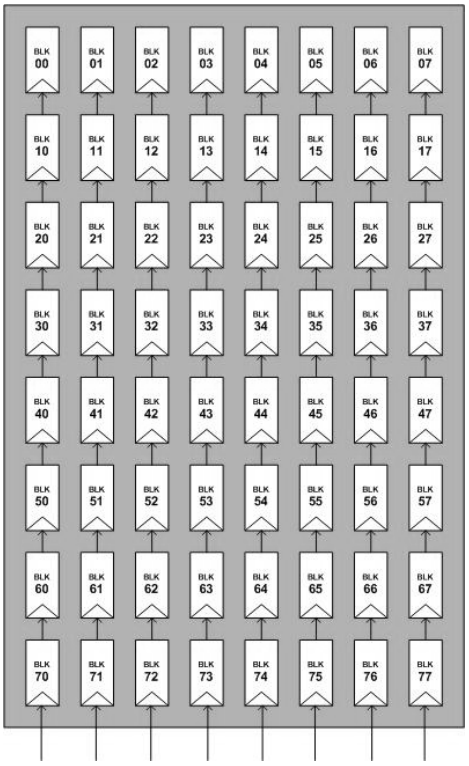


Figura 3. Representação do banco de registradores de bloco

Tal mecanismo foi idealizado para que os dados contidos nos registradores possam ser reaproveitados em outro casamento, pois na Busca Completa, o bloco é deslocado pixel em pixel na área de pesquisa, varrendo todos os blocos candidatos [Rosa 2007]. Dessa forma, a diferença entre um casamento e outro é de apenas uma coluna/linha de pixel. Sendo assim, apenas os dados que ainda não estão disponíveis para o próximo casamento são lidos da memória e carregados nos registradores. Este procedimento é controlado por meio de um seletor que indica qual deslocamento os dados devem realizar em cada momento.

A arquitetura conta com 3 Módulos de Processamento. Esta quantidade de módulos foi escolhida para realizar o melhor reaproveitamento de dados que arquitetura permite e reduzir a quantidade de área necessária. Os casamentos entre o bloco e a área de pesquisa são distribuídos entre os 3 MPs. Neste caso, dado que a área de pesquisa possui tamanho 32x32 e o bloco 8x8, são necessários 625 casamentos para determinar o melhor vetor de movimento, utilizando a Busca Completa.

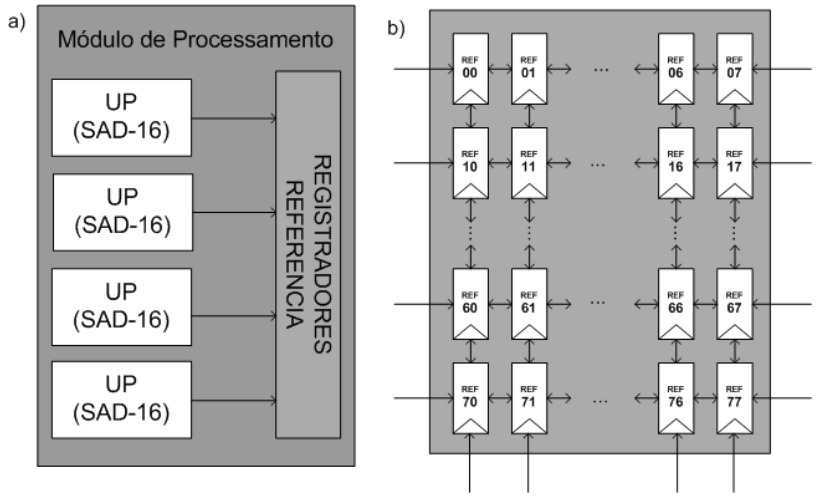


Figura 4. a) Detalhamento do Módulo de Processamento. b) Banco de registradores de referência.

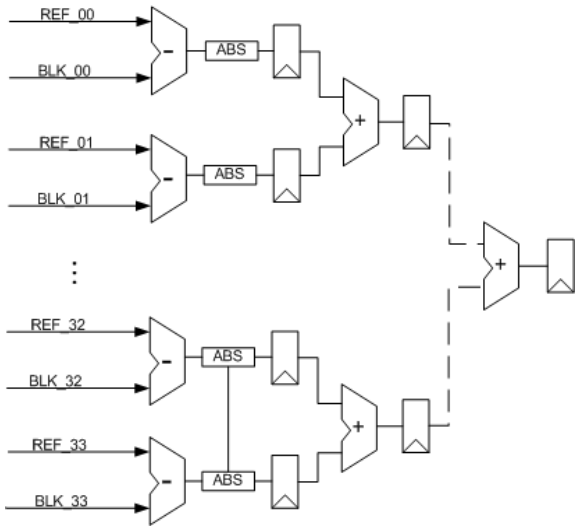


Figura 5. Detalhamento arquitetural da Unidade de Processamento

Destes 625 casamentos, o MP1 realiza 200, o MP2 realiza outros 200 e o MP3 realiza 225. Esta distribuição foi determinada tendo em vista o melhor reuso de dados que a arquitetura permite. Optou-se por deixar a MP3 com um pequeno número a mais de casamentos visando economia de área ocasionada pelo não acréscimo de outro MP à arquitetura para a realização dos 25 casamentos excedentes. Uma vez que cada casamento é realizado em um ciclo, isto acarreta uma ociosidade de 25 ciclos das duas outras MPs. Entretanto, esta ociosidade não se apresenta como um fator crítico tendo em vista a necessidade de preenchimento das memórias para o início de uma nova etapa de cálculos de similaridade. Desta forma, enquanto este último MP está processando, tem-se início o preenchimento da memória local para o próximo processamento.

3.3. Comparador

O comparador do núcleo é capaz de realizar a comparação dos resultados de SAD dos 3 MPs paralelamente e dar como resultado o menor valor de SAD e seu respectivo vetor

de movimento. Este componente, detalhado na Figura 6, opera em um *pipeline* de 3 estágios. Dessa forma, leva-se 3 ciclos para obter o primeiro resultado e após cada ciclo, uma nova comparação já foi efetuada. Além dos resultados de SAD, são passados como entrada para o componente as coordenadas (x, y) do casamento atual, para que seja possível ser identificado o vetor de movimento relativo ao melhor casamento.

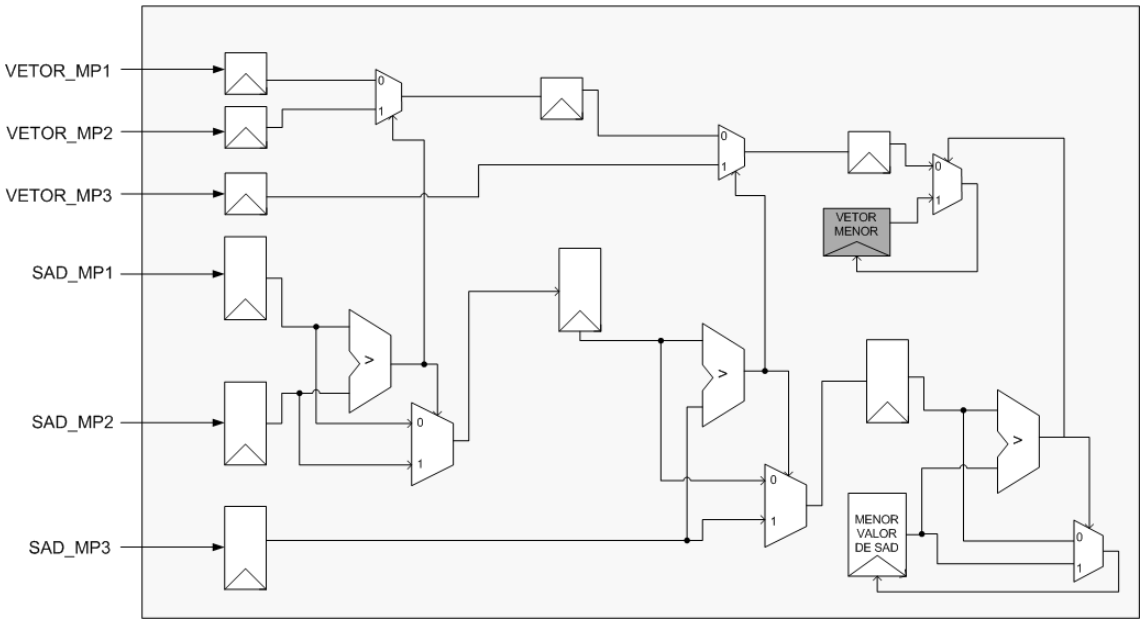


Figura 6. Detalhamento arquitetural do componente Comparador

3.4 Memórias

São 32 módulos de memória que constituem o núcleo. Cada um é responsável por armazenar 1/32 (uma linha) da área de pesquisa 32x32. As memórias foram idealizadas para funcionarem com o mecanismo de *dual-port*. Na escrita de dados, cada memória é interpretada como tendo 8 palavras de 32 bits. Já na leitura de dados, cada memória foi definida como possuindo 32 palavras de 8 bits (cada palavra representando um pixel).

Adotando esta forma, torna-se possível que a escrita nas memórias locais seja compatível com a leitura que deve ser feita na memória externa à arquitetura, onde estão armazenados os quadros da imagem. Nesta memória externa, a leitura do quadro deve ser realizada da esquerda para a direita e de cima para baixo. Enquanto torna a escrita compatível, este procedimento permite a varredura em zig-zag idealizada para este núcleo, que realiza um reaproveitamento dos dados de um casamento atual para a realização do próximo casamento. Como citado anteriormente, na Busca Completa a diferença de pixels de um casamento para outro é de apenas uma linha/coluna.

A varredura em zig-zag está apresentada na Figura 7. Esta figura apresenta parte da memória local inicializada com um valor qualquer. O quadrado 8x8 representa os dados que se encontram no banco de registradores de referência do MP. Estes dados são referentes ao primeiro casamento da Busca Completa. A primeira seta da linha da Memória 0 representa o deslocamento que deve ser realizado para que seja efetuado o segundo casamento da busca. Neste caso, para a realização do segundo casamento, o seletor do banco de registradores de referência sinaliza que os dados devem sofrer um deslocamento para a esquerda e novos dados lidos da memória são gravados no banco.

As setas seguintes representadas na figura são referentes aos demais casamentos/deslocamentos a serem realizados.

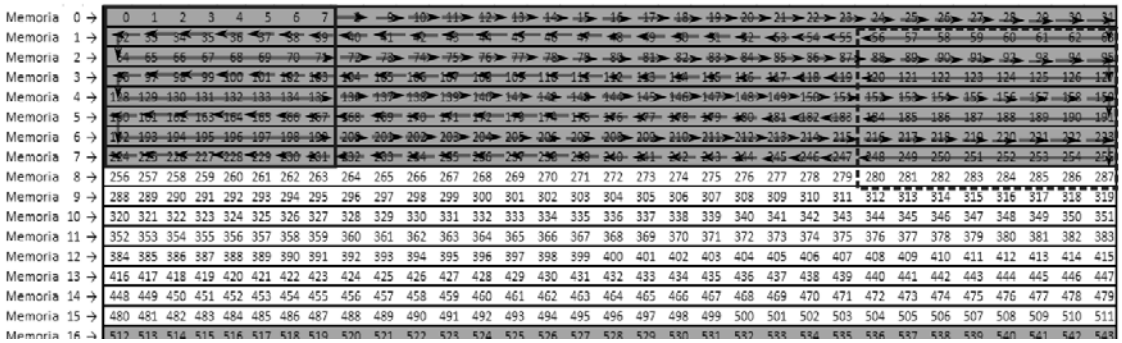


Figura 7. Varredura da área de pesquisa em zig-zag

A cada linha, são realizados 25 casamentos entre o bloco atual e o bloco de referência. Ao chegar ao 25º casamento, é feito um deslocamento para baixo no banco de registradores de referência, contando mais um casamento, e mais 24 casamentos são realizados para a segunda linha. Na segunda linha, os deslocamentos são realizados para a esquerda. Este procedimento é realizado até finalizar os casamentos destinados a cada MP.

Esta figura apresenta os casamentos a serem realizados pela MP1. O quadrado tracejado na imagem representa os dados que devem estar no banco de registradores de referencia quando a MP1 estiver realizando o 26º casamento, ou ainda, o 1o casamento da segunda linha da área de pesquisa.

4. Avaliação de Desempenho da Arquitetura

Uma vez feito o primeiro preenchimento das memórias e do banco de registradores de bloco, a arquitetura é capaz de processar 1 vetor de movimento a cada 241 ciclos. Operando a uma freqüência máxima 180 MHz, são processados em média 51 quadros por segundo de uma imagem HDTV (1280x720). Com este resultado percebe-se que a arquitetura, tal como ela foi proposta, é capaz de processar imagens HDTV (1280x720) em tempo real com uma folga considerável. A Tabela 1 mostra a taxa de processamento para outros tamanhos de imagem, bem como a freqüência necessária para processamento em tempo real (30 quadros/s).

Tabela 1. Taxa de processamento de quadros por resolução de imagem

Resolução	Quadros/s	Freqüência mínima para tempo real (MHz)
QCIF (176x144)	1886,08	2,86
QVGA (320x240)	622,40	8,67
SD (720x480)	138,31	39,04
720p (1280x720)	51,86	104,11
1080p HD(1920x1080)	23,05	234,25

Como os dados apresentam, ainda não é possível atingir taxa de processamento para 1080p. Dessa forma, ainda é preciso reavaliar a arquitetura de modo a cogitar a questão do aumento de área para melhorar o desempenho obtido.

Embora o núcleo seja capaz de processar imagens 720p HDTV em tempo real, um fator muito importante foi desconsiderando para o cálculo deste tempo: os acessos à memória externa. Alguns trabalhos relacionados apresentam suas arquiteturas para estimação de movimento, entretanto desconsideram este gargalo de acesso. Para fazer a verificação deste impacto no desempenho foi feito um estudo referente ao tempo necessário para o preenchimento das memórias locais que constituem a arquitetura apresentada neste trabalho. Este estudo será apresentado na próxima seção.

4. Estudo sobre o impacto do acesso à memória externa

Para este estudo foi utilizado o kit de prototipagem Altera DE2 que possui um FPGA da família Cyclone II (EP2C35F672C6) e conta com duas memórias externas disponíveis: uma memória SRAM de 512 Kb e uma SDRAM de 8 Mb. Estas memórias possuem palavras de 16 bits. Para avaliar o custo de comunicação com as duas memórias externas foi desenvolvido um sistema composto por um Módulo de Acesso à memória, e um controlador de memória, ligado à memória externa, como apresentado na Figura 8.

Este sistema foi integrado utilizando o SOPC Builder [Altera Corporation 2009], que possibilita a fácil integração de módulos para criar sistemas em chip rapidamente. Como mecanismo de interconexão nesta ferramenta é utilizado o barramento Avalon. Os controladores utilizados para as memórias SRAM e SDRAM são módulos disponibilizados pela Altera para a utilização com seus dispositivos compatíveis.

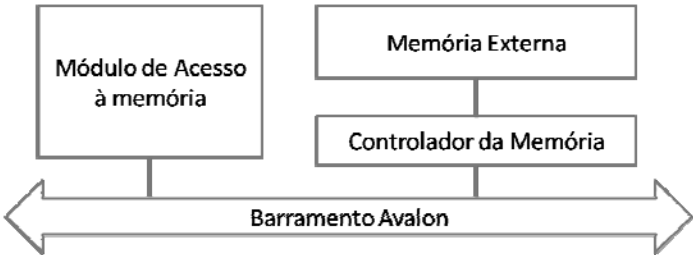


Figura 8. Sistema para análise do custo de comunicação com a memória externa

O Módulo de Acesso à Memória realiza as requisições necessárias para o preenchimento das memórias locais que constituem a arquitetura apresentada na seção 3. Para medir o tempo de acesso foi inserido um contador neste módulo que computa o tempo gasto para realizar as leituras na memória e obter o custo de realização desta tarefa. A Tabela 2 apresenta os resultados obtidos**.** A

Tabela 3 re-apresenta algumas das informações da Tabela 1, considerando a freqüência de 180Mhz e incluindo o tempo gasto com acesso à memória externa.

Tabela 2. Tempo em ciclos para preenchimento das memórias locais

Memória Externa	Tempo em ciclos
SRAM	2044
SDRAM	8701

Tabela 3. Taxa de processamento de quadros com acesso à memória externa

Resolução	Quadros/s SRAM	Quadros/s SDRAM
QCIF (176x144)	165,77	42,36
SD (720x480)	12,15	3,01
720p (1280x720)	4,55	1,16
1080p HD(1920x1080)	3,03	0,77

Pela tabela apresentada é possível verificar que o acesso à memória externa não é um fator a ser simplesmente desconsiderado, quando se trata de um módulo de estimação de movimento. A estimação de movimento é conhecida por sua grande complexidade computacional, que necessita realizar uma grande quantidade de cálculos para retornar o vetor de movimento. No estudo de caso apresentado, esta quantidade de cálculos chega a ser apenas 11,7% do custo da comunicação com a memória externa SRAM e pouco mais de 2% no caso da SDRAM. Vale salientar que o estudo de caso foi realizado no kit de prototipagem disponível no grupo e se trata de um equipamento lento.

Como estratégia para minimizar a necessidade de acessos à memória externa, esta arquitetura utilizará o reaproveitamento dos dados já existentes na memória local quando da realização da busca de um bloco em sua respectiva área de pesquisa. Na realização da busca do próximo bloco 8x8 em uma região 32x32, 75% dos dados que serão utilizados no próximo processamento já se encontram disponíveis na memória local. Isto porque um quadro é dividido em blocos não sobrepostos, mas as áreas de pesquisa de dois blocos possuem dados em comum. Este esquema está representado na Figura 9. Apenas no caso em que o bloco se encontra na extremidade do quadro é que não é possível realizar reaproveitamento para o próximo processamento.

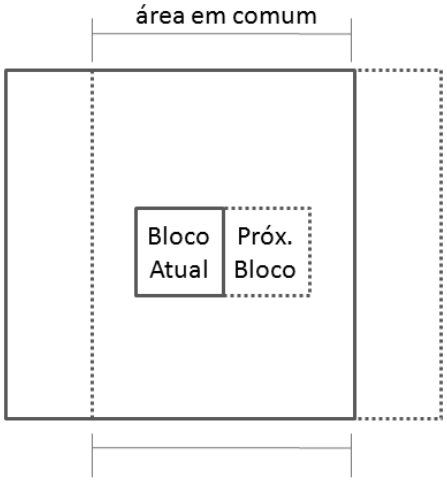


Figura 9. Sobreposição de área de pesquisa de blocos vizinhos

5. Conclusões e Trabalhos Futuros

Este artigo apresentou uma proposta de arquitetura para um núcleo de estimação de movimento segundo o padrão H.264. Este núcleo utiliza o algoritmo da Busca Completa

para varredura da área de pesquisa, tamanho de bloco 8x8 e área de pesquisa 32x32. A arquitetura é formada por 3 módulos de processamento (MPs), 1 banco de registradores de bloco, um comparador e memória local.

Foi apresentando também um estudo referente ao impacto do acesso à memória externa no desempenho da arquitetura. Para este estudo foi utilizado o *kit* de prototipagem Altera DE2. Dos experimentos concluiu-se que novas estratégias de acesso a memória devem ser utilizadas de modo a permitir que o poder de processamento da arquitetura seja bem utilizado. Comprova-se também que, muitos dos artigos que advogam o uso de algoritmos subótimos, como o da Busca em Diamante (*Diamond Search*) desconsideram o tempo de acesso a memórias externas para aquisição dos dados. Como estes algoritmos realizam um número inferior de cálculos para encontrar um casamento subótimo, o problema da espera dos dados tende a se agravar.

Trabalhos futuros incluem a utilização de um *kit* Xilinx (XUP V2P), para o qual está disponível um controlador de acesso a memória com melhor desempenho [Bonatto, Soares e Susin 2008]. Outro estudo importante a ser realizado é referente ao reuso de dados em comum entre áreas de pesquisa de blocos vizinhos e criação de uma hierarquia interna de memória na arquitetura, com o objetivo de reduzir os custos de acessos a dados. O aumento do paralelismo da arquitetura do ME, com a utilização de cinco módulos de processamento também é uma alternativa a ser estudada. Com este paralelismo, cada MP seria responsável pela execução 125 casamentos, possibilitando processamento em tempo real de imagens 1080p. Através desses estudos será possível analisar a possibilidade de redução de quantidade de acessos à memória externa e por conseqüência, aumento no desempenho global do módulo de estimação de movimento.

Referências

Agostini, L. V. (2007). “Desenvolvimento de Arquiteturas de Alto Desempenho Dedicadas à Compressão de Vídeo Segundo o Padrão H.264/AVC**”**. 172f. Programa de Pós Graduação em Computação, UFRGS, Porto Alegre.

Altera Corporation, (2009). “Introdution to SOPC Builder”, http://www.altera.com/literature/hb/qts/qts_qii54001.pdf, November.

Bhaskaran, V. and Konstantinides, K. (1997). “Image and Video Compression Standars: Algorighms and Architectures”. 2nd edition. Kluwer Academic Publishers: Boston.

Bonatto, A. C., Soares, A. B., Susin, A. A. (2008). “DDR-SDRAM Memory Controller Validation for FPGA Synthesis”. In: 9th IEEE Latin-American Test Workshop, 2008, Puebla. 9th IEEE Latin-American Test Workshop. p. 177-182.

Deng, L., Gao. W, Hu, M. Z. and Ji, Z. Z. (2005) “An efficient hardware implementation for motion estimation of AVC standard”, Consumer Electronics, IEEE Transactions on , vol.51, no.4, pages 1360-1366.

Ghanbari, M. (2003). “Standard Codecs: Image Compression to Advanced Video Coding (Telecommunications)”. Institution Electrical Engineers.

ITU-T – International Telecommunication Union. (2007). “ITU-T Recommendation H.264/AVC (05/03): advanced video coding for generic audiovisual services**”**.

Rosa, L. Z. P. (2007). “Investigação sobre Algoritmos para a Estimação de Movimento na Compressão de Vídeos Digitais de Alta Definição: Uma Análise Quantitativa”. 155f. Departamento de Informática, Universidade Federal de Pelotas, Pelotas.

Shi, Y. Q., Sun, H. (1999). “Image and Video Compression for Multimedia Engineering: Fundamental, Algorithms and Standards**”**. CRC Press.

Vanne, J., et al. (2006). “A High-Performance Sum of Absolute Difference Implementation for Motion Estimation.” Circuits and Systems for Video Technology, IEEE Transactions on**.** v. 16, n. 7, p. 876-883.

Verma, R., Akoglu, A. (2008). “A coarse grained and hybrid reconfigurable architecture with flexible NoC router for variable block size motion estimation”. Parallel and Distributed Processing. IPDPS 2008. IEEE International Symposium, p. 1-8.

Yi, X., Ling, N. (2005). “Rapid block-matching motion estimation using modified diamond search algorithm**”.** Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium, p. 5489-5492 Vol. 5486

Arquitetura Hardware/Software de um núcleo NCAP Segundo o Padrão IEEE 1451.1: Uma Prova de Conceito

José de Anchieta G. dos Santos[1], **Ivan Saraiva Silva**[2]

[1]Departamento de Informática e Matemática Aplicada – Universidade Federal do Rio Grande do Norte (UFRN)– Natal – RN – Brasil

[2]Departamento de Informática e Estatística – Universidade Federal do Piauí (UFPI) Teresina–PI–Brasil

{anchieta}@ppgsc.ufrn.br

{ivan}@.ufpi.edu.br

Abstract. *The IEEE 1451 standard is an IEEE and NIST initiative that aims standardize the software to communicate TIM and NCAP modules, which must to communicate among them using the standardized interface TII. This work presents a hardware/software architecture for the prototyping of a NCAP module in FPGA according the IEEE 1451.1. The NCAP is prototyped on DE2 Altera's board and the NIOS II embedded processor is used to support the software which is developed in C. The hardware behavior is described using VHDL language.*

Resumo. *O padrão IEEE 1451 é uma iniciativa do IEEE e do NIST que visa padronizar o software de comunicação entre os módulos TIM e NCAP, os quais devem se comunicar através da interface padronizada TII. Este trabalho apresenta uma arquitetura hardware/software para a prototipação de um módulo NCAP em FPGA de acordo com o padrão IEEE 1451.1. O NCAP é prototipado na placa DE2 da Altera e o processador embarcado NIOS II é utilizado para dar suporte ao software escrito em C. O comportamento do hardware é descrito usando a linguagem VHDL.*

1. Introdução

Os avanços na tecnologia dos sistemas micro-eletromecânicos (MEMS) fizeram com que os sensores pudessem realizar atividades além daquelas necessárias para gerar apenas uma correta representação dos dados. Esses sensores passaram a fazer parte de uma nova classe conhecida como sensores inteligentes [Oliveira 2004]. Eles se diferenciam do sensor comum pela capacidade de processamento *on-board* sobre os dados monitorados.

Nesse contexto o sensor inteligente está dividido em módulos TIM (*Transducer Independent Module*) e NCAP (*Network Capable Application Processor*). O TIM contém os transdutores (sensores e atuadores), o equipamento de conversão de sinal e de condicionamento do mesmo. O NCAP contém a parte de processamento de dados e comunicação do sensor em rede com o usuário final, sendo considerado o responsável por atribuir característica de inteligência ao sensor.

Em 1993, o IEEE e o NIST criaram uma iniciativa de padronização que é considerada como um significativo avanço na tecnologia de sensores inteligentes

[Wobschall 2002],[Viegas et al. 2005]. Ela deu origem ao padrão IEEE 1451 que é um meio padronizado para comunicação em redes de sensores e atuadores. Seu grande benefício é unificar diversos padrões existentes que variam dependendo do fabricante.

Apesar de ser considerado como um avanço o padrão IEEE 1451 possui dificuldades de ser aderido devido à falta de NCAPs disponíveis seguindo o mesmo [Viegas et al. 2005]. Embora já tenha se passado um certo tempo desde que essa informação foi notificada ainda há casos como o da *Smart Sensor System*. Essa empresa disponibiliza um módulo TIM segundo o IEEE 1451, porém não disponibiliza um módulo NCAP, mas apenas um software que permite o acesso às informações do TIM [System 2010].

O objetivo deste trabalho é o desenvolvimento de um módulo NCAP segundo o padrão IEEE 1451.1. Esse módulo é prototipado em FPGA e deve possuir capacidade de comunicação em rede tanto com outro NCAP através de uma interface RS-232 e com o módulo TIM através de uma interface serial padronizada de acordo com o IEEE 1451.2.

2. Padrão IEEE 1451.1

O padrão IEEE 1451.1 se baseia na linguagem orientada a objetos para especificar o software do NCAP. O seu objetivo é o desenvolvimento de um modelo de objeto comum para que módulos NCAPs possam trocar dados entre si. Apesar do padrão ser baseado na orientação a objetos ele deixa o desenvolvedor livre à utilização de outras linguagens.

Nele é definido um modelo de informação neutro que é composto por um conjunto hierárquico de classes que representam os blocos de objetos do NCAP. O modelo é neutro porque especifica a forma geral que deve ser feita a troca de informação entre NCAPs, independentemente da tecnologia de rede que será utilizada. Ele também define um modelo de dados, onde são especificados o tipo e a forma dos dados que devem ser utilizados pelo software de aplicação do NCAP [IEEE-Std-1451.1-1999 1999].

Existem 4 tipos especiais de classes que devem comportar todas as classes definidas por este padrão. A classe *Block* contém classes de suporte à configuração do sistema, bem como comunicação entre o software da aplicação e os transdutores. A classe *Component* contém construções comuns da aplicação, tais como informações estruturadas ou coleções de objetos específicos da aplicação. A classe *Service* contém classes que dão suporte a comunicação entre NCAPs distintos. A classe *Non-IEEE 1451* contém classes específicas do fabricante do NCAP e que não fazem parte do padrão. Essa divisão visa dar maior modularidade ao software.

De acordo com o padrão na classe *Block* existe uma classe chamada *BaseTransducerBlock* que contém operações de *IORead()* e *IOWrite()* para dar suporte a comunicação entre o TIM e o NCAP. A classe *Block* também contém as classes *FunctionBlock* que contém operações específicas da aplicação e *NCAPBlock* que contém funções de configuração do NCAP e que devem ser utilizadas durante a fase de inicialização do mesmo.

A classe *Service* contém classes como a *BasePublisherPort* que possui operações de configuração dos parâmetros utilizados na comunicação *Publish/Subscriber*, a classe *PublisherPort* que realiza a operação de publicação, a classe *SubscriberPort* que realiza operações de *AddSubscriber()* e *CallBack()*. A primeira operação é utilizada para

requisitar interesses através da rede e a segunda utilizada para notificar aos demais *subscribers* que uma publicação foi aceita. Existem também duas classes que dão suporte à comunicação Cliente/Servidor. A *BaseClientPort* que possui os parâmetros a serem configurados para esse tipo de comunicação e a *ClientPort* que contém as operações de *Perform()* e *Execute()* para que um NCAP possa solicitar que uma operação seja executada por outro NCAP remotamente.

A classe *Component* é responsável por dar suporte ao conjunto de estruturas que são utilizadas pelo NCAP. Ela possui classes como *Parameter* que define os tipos de parâmetros utilizados por um determinado NCAP e a classe *ParameterWithUpdate* que da suporte à atualização dos parâmetros do NCAP.

Como pode ser visto na descrição da classe *Service* o padrão IEEE 1451.1 define dois tipos de comunicação entre NCAPs, também chamada de comunicação de alto nível. Uma delas é a comunicação Cliente/Servidor e a outra é a comunicação *Publish/Subscriber*.

No tipo de comunicação Cliente/Servidor um NCAP funciona como cliente e o outro como servidor. Nesse tipo de comunicação o cliente deve interromper suas atividades até que receba o resultado da requisição feita ao servidor. De acordo com esse padrão o cliente deve invocar os serviços do servidor através da operação de *Execute*. Essa operação possui como argumentos o status do modo de execução, o endereço do NCAP servidor, os argumentos de entrada para o NCAP servidor e os argumentos de saída desejados. O NCAP servidor executa o serviço solicitado através da operação *Perform* e devolve o resultado através da rede.

Na comunicação *Publisher/Subscriber* os NCAPs não necessitam interromper suas atividades. O *Publisher* e o *Subscriber* não precisam saber da existência um do outro. O *Subscriber* publica seu interesse na *Subscriber Port* e o *Publisher* publica um determinado interesse que ele possui para todos os *Subscribers* da rede. Ao receber um interesse publicado pelo *Publisher* a *Subscriber Port* compara o mesmo com o interesse publicado pelo Subscriber utilizando o domínio, a chave e o qualificador de subscrição. Caso o interesse requisitado coincida com o interesse recebido, este é aceito. Caso contrário é descartado. Este é o tipo de comunicação utilizado pelo NCAP desenvolvido no presente trabalho.

3. Trabalhos Relacionados

Trabalhos interessantes na área de sensores inteligentes são os de [SAUSEN et al. 2007] e [Song and Lee 2008]. Neles é mostrado o estado da arte do padrão IEEE 1451, servindo como base para outros trabalhos da área. Dentre os trabalhos relacionados à este merece destaque o de [Kochan et al. 2004] que desenvolve um NCAP remotamente reprogramável usando microcontroladores. O trabalho de [Turchenko et al. 2005] é uma evolução do anterior que usa FPGA para dar suporte à interfaces de alta velocidade. Em [Mayikiv et al. 2007], é feita uma comparação entre algumas abordagens de desenvolvimento existentes para o módulo NCAP, onde são destacadas suas vantagens e desvantagens, nele também é dado destaque para aquelas usando microcontroladores.

4. Arquitetura Proposta

A arquitetura proposta para este trabalho está dividida em uma infra-estrutura de hardware e uma infra-estrutura de software. A infra-estrutura de hardware comporta todos módulos de hardware utilizados para dar suporte ao funcionamento do NCAP, bem como o simulador do TIM. E a infra-estrutura de software é responsável pelo comportamento do NCAP, pelo suporte ao padrão IEEE 1451.1 e aplicação específica. A seguir será feita a descrição das mesmas.

4.1. Infra-estrutura de Hardware

A figura 1 mostra a arquitetura deste trabalho. Através dela pode-se observar que a placa DE2 da Altera é utilizada para a prototipação do NCAP. O processador embarcado NIOS II é utilizado para dar suporte ao software que é escrito em C, onde se encontram o padrão e a aplicação específica. O Barramento Avalon é o responsável por comunicar o NIOS com o driver RS-232 que dá suporte à comunicação de alto nível, com a memória utilizada para armazenamento de dados e com o módulo de interface padronizada TII (*Transducer Independent Interface*), que é utilizado para comunicação com o simulador do TIM. Este em conjunto com a interface TII são descritos através da linguagem VHDL.

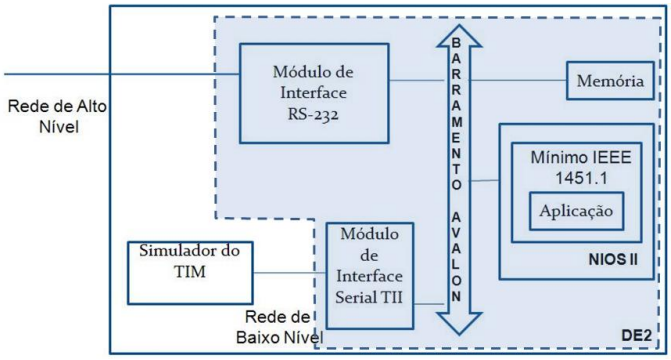


Figura 1. Arquitetura do NCAP proposto.

Na figura está claramente identificado que todos os componentes do NCAP estão situados dentro da zona tracejada em destaque. No caso o simulador do TIM também se encontra na mesma placa em que o módulo NCAP, porém não faz parte da proposta deste trabalho e foi implementado apenas para dar suporte às simulações e análises de resultados.

4.1.1. Interface TII

A interface serial padronizada TII é definida no padrão IEEE1451.2. Essa interface contém as portas de comunicação mostradas na figura 2. Neste trabalho ela é responsável por realizar a comunicação entre o módulo NCAP e o simulador do módulo TIM. De acordo com a figura a porta DIN é utilizada para envio de dados seriais do NCAP para o TIM e na porta DOUT os dados são enviados do TIM para o NCAP. A DCLK é utilizada para que o NCAP forneça seu clock para que o TIM possa trabalhar em sincronia com o mesmo. A NIOE é ativada durante o envio de mensagens do NCAP para o TIM. NTRIG é usada pelo NCAP para enviar um sinal de disparo para o TIM e este deve reconhecer

esse sinal através da porta NACK. O sinal de disparo é enviado pelo NCAP toda vez que ele deseja solicitar que o TIM realize alguma tarefa. A NINT é utilizada quando o TIM deseja enviar um sinal de interrupção para o NCAP. Neste trabalho ele é utilizado para que o TIM informe ao NCAP que alguma das grandezas medidas por ele ultrapassou um limiar pre-determinado. O NSDET é utilizado pelo NCAP para reconhecer a presença do módulo TIM, ou seja, ele deve sempre estar ativo. As demais portas são utilizadas como fontes de alimentação energética.

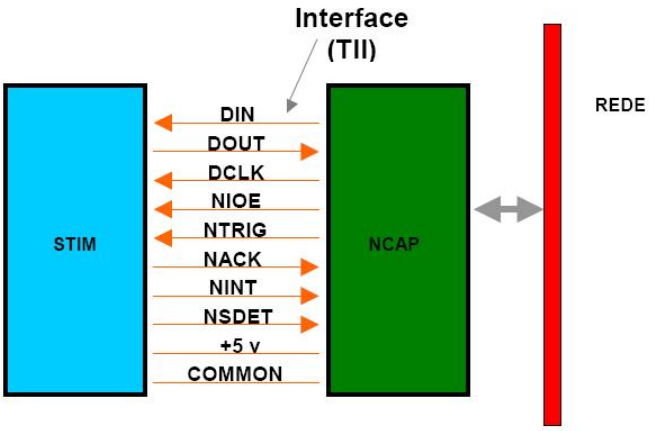


Figura 2. Interface padronizada TII.

O envio de mensagens entre o NCAP e o TIM também é feito seguindo o padrão IEEE 1451.2. De acordo com ele as mensagens devem ser enviadas ao TIM iniciando-se com o envio da função a ser realizada, em seguida do canal transdutor que deve executar a função e caso a função seja de escrita em um canal transdutor (atuador) o dado a ser escrito é enviado, senão é feita a leitura do canal (sensor). Para tanto segue-se o modelo da figura 3, onde o endereço da função é passado serialmente do bit mais significativo para o bit menos significativo, depois o mesmo ocorre com o endereço do canal transdutor. O bit mais significativo do endereçamento de função identifica se ela é uma escrita ou uma leitura, o zero representa leitura e o um representa uma função de escrita.

Endereço de função Byte mais significativo								**Endereço de canal** Byte menos significativo							
r/w	Código de função							Número de canal							
msb							lsb	msb							lsb

r =leitura, w= escrita

Figura 3. Endereçamento de funcão e de canal.

4.1.2. NIOS II

O NIOS II é um processador *soft-core* configurável, em contrapartida aos microcontroladores, que são elementos de processamento fixo. Nesse contexto, configurável quer

dizer que características podem ser adicionadas ou retiradas para que objetivos de desempenho e custo sejam alcançadas [Altera 2010].

O NIOS permite que um software desenvolvido na linguagem C/C++ possa ser prototipado e executado em uma FPGA. O ambiente de desenvolvimento do NIOS II é baseado no compilador GNU C/C++ e na IDE (*Integrated Development Environment*) do Eclipse que provê um ambiente familiar e conceituado para o desenvolvimento de software. Usando essa IDE é possível projetar e simular aplicações para esse processador.

4.1.3. SOPC Builder

A integração dos componentes do sistema é feita utilizando o SOPC Builder. O SOPC Builder é uma ferramenta que automatiza a tarefa de integração de hardware, pois ele permite gerar um completo SOPC (*System-On-a-Programmable-Chip*) em menos tempo do que utilizando métodos de integração tradicionais. Usando métodos tradicionais torna-se necessário escrever manualmente toda a linguagem de descrição do hardware que define a junção de todos os componentes do sistema. Enquanto que usando o SOPC Builder torna-se necessário apenas especificar os componentes do sistema através de uma GUI (*Graphical User Interface*) e o SOPC Builder gera todas as interconexões lógicas automaticamente.

4.2. Infra-estrutura de Software

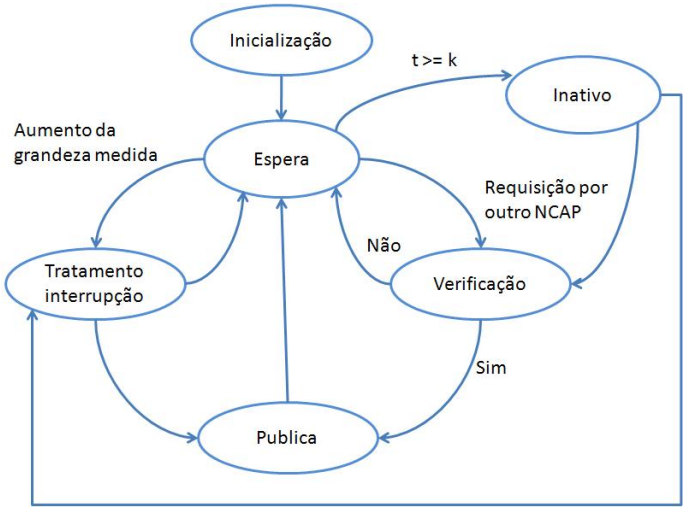


Figura 4. Máquina de estados do software do NCAP.

O software do NCAP trabalha de acordo com a máquina de estados da figura 4. Na inicialização é feita a solicitação de leitura das TEDS (*Transducer Eletronic Data Sheets*) armazenadas no simulador do TIM e todas as configurações necessárias ao funcionamento do NCAP. No estado de espera o NCAP fica à espera de solicitações de dados à partir de outro NCAP. Caso isso aconteça ele vai para o estado de verificação, onde verifica o dado solicitado e se possuir publica o mesmo para o NCAP requisitante. Também no estado de espera pode ocorrer uma interrupção por parte do simulador do TIM. Essa interrupção existe quando o TIM deseja enviar algum aviso de urgência, como no caso da

temperatura aumentar bruscamente. Nesse caso o NCAP deve publicar essa informação através da rede. Passado um determinado tempo sem receber requisição ou interrupção o NCAP entra no estado inativo, no qual ele permanece até a próxima ocorrência de uma das mesmas.

O tipo de comunicação de alto nível utilizado é o *Publish/Subscriber*, pois é o que mais se adéqua à realidade das redes de sensores. A comunicação de baixo nível é feita através de operações como *IORead* e *IOWrite* definidas no padrão. A aplicação específica irá verificar a variação de temperatura do sensor, calcular qual a variação deve ser feita no atuador do aquecedor e escrever esse valor no mesmo para manter a temperatura ideal.

5. Cenário de Uso

O NCAP proposto faz parte de um sensor inteligente, que por sua vez deve estar inserido em uma rede de sensores. Deste modo o cenário de uso para o NCAP em questão seria uma rede de sensores, onde cada sensor possui capacidade para medir uma grandeza específica. Um sensor chamado de *sink*, normalmente situado próximo à central de observação, faz requisições aos demais sensores da rede chamados *sources* e estes respondem com os dados requeridos caso seja capaz de medir a grandeza requisitada.

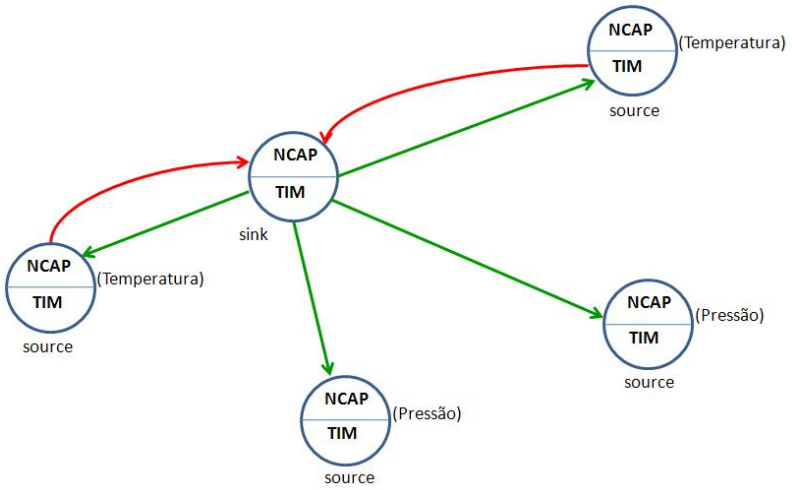


Figura 5. Cenário de uso para o NCAP.

Como pode ser observado na figura 5 a rede de sensores é formada por um nó *sink* que tem acesso direto aos demais nós da rede. Essa rede não segue uma abordagem *multihop* para entrega de dados ao *sink*. O objetivo desse cenário de uso é testar o correto funcionamento do módulo NCAP e o uso do padrão IEEE 1451.1. Na seção 7 que apresenta as conclusões e trabalhos futuros a este é deixada a implementação da comunicação através de uma interface Ethernet que possibilite simular uma rede de sensores contendo nós roteadores.

Na simulação são usadas duas placas DE2, de acordo com a figura 6, uma delas simulando um NCAP *sink* e outra simulando um NCAP *source*. O NCAP *sink* requisita a leitura de dados de pressão e temperatura do outro. Este deve responder apenas quando possuir a grandeza que é capaz de medir, no caso a temperatura. Eventualmente a temperatura pode aumentar demais, nesse caso o *source* deve enviar uma notificação ao *sink*, onde o conteúdo da notificação é o valor da temperatura.

Durante a comunicação os NCAPs utilizam as funções do padrão da seguinte maneira. Inicialmente a placa simulando o *sink* utiliza a operação *AddSubscriber()* para solicitar seus interesses através da rede. Essa operação tem como argumentos a chave de subscrição *key*, o domínio de subscrição *domain* e o qualificador de subscrição *qualifier*. A placa que simula o *source* ao receber interesses vindos da rede verifica se possui aquele tipo de interesse e caso possua realiza a operação de *publish()* para enviar os dados. Essa operação envia a chave de publicação *key*, o domínio de publicação *domain*, o tópico de publicação *topic* e o conteúdo da publicação. Ambos, o *sink* e o *source*, ficam sempre monitorando os canais transdutores do seu TIM através de solicitações de leitura usando a operação de *IORead()*, que recebe como argumento o endereço do canal transdutor a ser lido. Caso queira realizar uma escrita em algum canal transdutor é usada a operação *IOWrite()* que tem como argumentos o endereço do canal transdutor e o dado a ser escrito.

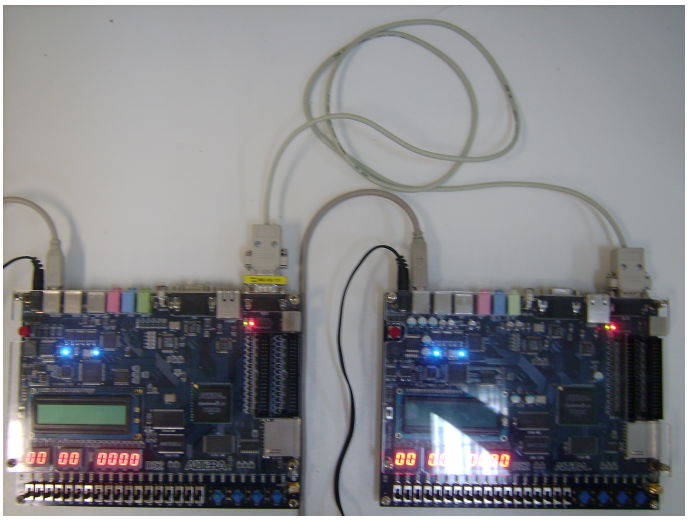


Figura 6. Placas DE2 se comunicando através de uma interface RS-232.

6. Resultados

O módulo NCAP desenvolvido para este trabalho ainda não se encontra completamente desenvolvido. Ainda falta finalizar o módulo de comunicação entre os NCAPs através da interface RS-232. Apesar de ainda não estar completo já existem alguns resultados obtidos a partir de simulações dos módulos de interface TII e do simulador do TIM. As operações de escrita e leitura solicitadas ao TIM a partir do NCAP estão funcionando, bem como a operação de interrupção vinda do TIM.

Uma operação de leitura utiliza 41 ciclos de clock do sistema para disponibilizar o dado ao software. A operação de escrita utiliza 37 ciclos. Desde a ocorrência de uma condição de interrupção o simulador do TIM leva 18 ciclos de clock para disponibilizar o valor da grandeza medida ao software. Como já foi explicado antes, uma interrupção ocorre sempre que uma grandeza medida ultrapassa um limiar pre-determinado.

Na versão que se encontra desenvolvido o NCAP, ele trabalha com uma frequência de máxima de 50MHz e seu hardware possui 4204 elementos lógicos, ocupando 13% em área da FPGA, no caso a Cyclone II EP2C35F672C6 da Altera.

7. Conclusões

Este trabalho apresenta uma arquitetura composta por uma infra-estrutura de hardware e uma infra-estrutura de software como abordagem de desenvolvimento de um módulo NCAP. Esse módulo segue o padrão IEEE 1451.1 definido pela IEEE em conjunto com o NIST. As linguagens utilizadas para o desenvolvimento foram a linguagem C para o software e a linguagem VHDL para a descrição do comportamento do hardware. Um simulador do comportamento de um módulo TIM também foi implementado para dar suporte aos testes e análises do NCAP, que é prototipado na placa DE2 da Altera. O módulo desenvolvido trás como contribuição o incentivo ao uso do padrão IEEE 1451, o qual apesar de ser considerado como um avanço na área de sensores inteligentes ainda é pouco aderido devido à falta de módulos NCAPs disponíveis seguindo este padrão.

Apesar de estar em fase de conclusão o presente trabalho deixa trabalhos futuros interessantes. Um deles seria o desenvolvimento do módulo de interface Ethernet para dar suporte a comunicação entre vários NCAPs. Outro seria a implementação de um protocolo de roteamento para o NCAP desenvolvido.

Referências

Altera, C. (2010). Altera corporation. http://www.altera.com/. Acessado em: 3 de Março de 2010.

IEEE-Std-1451.1-1999 (1999). Standard for a smart transducer interface for sensors and actuators - network capable application processor (ncap) information model. IEEE Instrumentation and Measurement Society, TC-9,The Institute of Eletrical and Eletronic Engineers, Inc.

Kochan, R., Lee, K., Kochan, V., and Sachenko, A. (2004). Development of a dynamically reprogrammable ncap [network capable application processor]. volume 2, pages 1188–1193 Vol.2. Instrumentation and Measurement Technology Conference, 2004. IMTC 04. Proceedings of the 21st IEEE.

Mayikiv, I., Stepanenko, A., Wobschall, D., Kochan, R., Sachenko, A., and Vasylkiv, N. (2007). Remote reprogrammable ncaps: Issues and approaches. pages 109–113. Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2007. IDAACS 2007. 4th IEEE Workshop on.

Oliveira, A. L. (2004). Modelo e algoritmos para organização de redes de sensores sem fio hierárquicas. Master's thesis, Departamento de pós-graduação em Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte - MG.

SAUSEN, P. S., CAMPOS, M. d., and SALVADORI, F. (2007). O estado da arte do padrão ieee 1451 aplicado às redes de sensores. *Conselho em Revista*, (36):36 – 36.

Song, E. and Lee, K. (2008). Understanding ieee 1451-networked smart transducer interface standard - what is a smart transducer? *Instrumentation & Measurement Magazine, IEEE*, 11(2):11–17.

System, S. S. (2010). Smart sensor system. http://www.altera.com/. Acessado em: 30 de Março de 2010.

Turchenko, I., Kochan, R., Kochan, V., Sachenko, A., Maykiv, I., and Markowsky, G. (2005). Network capable application processor based on a fpga. volume 2, pages 813–

817. Instrumentation and Measurement Technology Conference, 2005. IMTC 2005. Proceedings of the IEEE.

Viegas, V., Dias Pereira, J., and Silva Girao, P. (2005). Using a commercial framework to implement and enhance the ieee 1451.1 standard. In *Instrumentation and Measurement Technology Conference, 2005. IMTC 2005. Proceedings of the IEEE*, volume 3, pages 2136–2141.

Wobschall, D. (2002). An implementation of ieee 1451 ncap for internet access of serial port-based sensors. In *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*, pages 157–160.

I Workshop de Sistemas Embarcados

♦

Sessão Técnica 3

Arquiteturas II

Redes de Interconexão Multiestágios em Plataformas Reconfiguráveis para Sistemas Embarcados

Júlio Cesar Goldner Vendramini[1], Ricardo Ferreira[1]

[1]Depto de Informática – Universidade Federal de Vicosa 36570-000, Vicosa, Brazil

{julio.vendramini,ricardo}@ufv.br

Abstract. *This work presents an evaluation of multistage interconnection network (MIN) for embedded systems, mapped on FPGAs. The total area in LUTs, the latency, the data width, the number of reconfiguration bits as well as the switch complexity are analyzed. The experimental results show that the FPGA synthesis tools are enable to capture the MIN regularity, the area and the reconfiguration bits have complexity O(N Lg N), and the latency has complexity O(Lg N). Therefore, MINs are good candidates as an interconnection network for parallel embedded systems on FPGA.*

Resumo. *Este trabalho apresenta uma avaliação das redes multiestágio como mecanismo de comunicação para sistemas embarcados sintetizados em FPGA. As redes são comparadas levando em consideração área em LUTs, a latência, os bits de configuração, a capacidade dos comutadores e a largura da palavra. Como resultado é mostrado que a regularidade das redes é capturada pelas ferramentas de síntese de FPGA, apresentando complexidade O(N Lg N) em área e em bits de configuração e O(Lg n) em latência, mostrando a viabilidade do uso das redes em sistemas embarcados.*

1. Introdução

Sistemas embarcados e computação paralela são dois grandes desafios da atualidade. As redes de interconexão tem um papel fundamental na implementação de soluções paralelas. Além disso, os sistemas embarcados demandam desempenho e flexibilidade para se adaptar ao mercado muito dinâmico. Os sistemas reconfiguráveis como os FPGAs são uma alternativa interessante. Entretanto a alta flexibilidade dos FPGAs elevam a complexidade da síntese e mapeamento, aumentando o tempo de projeto dos sistemas. Este trabalho se enquadra em arquiteturas paralelas de sistemas embarcados compostas por vários elementos de processamento (EP) e/ou módulos de memória. Este trabalho apresenta as redes multiestágios como uma alternativa para interconexão dos EPs e memórias em FPGAs.

As redes multiestágios já foram usadas em sistemas de emulação e prototipação baseados em Multi-FPGAs [Barbie,1999]. Para emular um sistema eram necessários vários chips de FPGAs e redes de interconexão entre eles. Ao invés de usar chips crossbar, que tem custo $O(n^2)$, alguns trabalhos propuseram dedicar um ou mais FPGAs as interconexões, aumentando a flexibilidade. Este artigo difere dos anteriores ao propor o uso das redes multiestágios internamente no FPGA, em conjunto com elementos de processamento e memória para implementação eficiente de sistemas embarcados.

Este texto está estruturado da seguinte maneira. Primeiro são apresentados os trabalhos correlatos com multiestágio e suas implementações em VLSI e FPGA. Na seção 3, as redes são classificadas e os algoritmos de roteamento são apresentados. Na seção 4 apresentamos uma série de experimentos de implementação das redes em FPGA. A seção 5 finaliza com as principais conclusões e trabalhos futuros.

2. Trabalhos Correlatos

As redes multiestágios [Clos,1953][Benes,1965] foram inicialmente propostas nas décadas de 50/60 para o sistema de telefonia. Na década de 70, redes mais simples como as redes Omega [Lawrie,1975], foram propostas para interconexão de processadores e memória em máquinas paralelas MIMD e SIMD. O custo da rede Omega era menor assim como sua capacidade de roteamento. Porém alguns padrões de comunicação como os algoritmos de FFT e ordenação são realizados de forma eficiente. Durante os anos 70/80 até o início dos anos 90, muitos estudos dos tipos, classes de equivalência para redes bloqueantes e rearranjáveis [Wu,1980] [Yeh,1992], bem com algoritmos paralelos de roteamento foram realizados [Yeh,1992] e outras redes equivalentes sugiram como as fat-tree [Leiserson,1985]. A rede se mostrava um mecanismo paralelo de comunicação e tolerância a falhas, porém como programá-la rapidamente, era um dos grandes desafios para construção das máquinas paralelas. Muitos trabalhos teóricos sobre os limites inferiores do problema foram feitos [Çam,1999], porém estes algoritmos paralelos não foram implementados. Como redes comutadoras de pacotes, as multiestágios foram usadas em redes ATM e supercomputadores no final dos anos 90 com comutadores ópticos. Na comutação de pacotes, a rede é auto-roteável, porém pode gerar congestionamento das filas dos comutadores intermediários quando a carga de trabalho se eleva.

Durante os anos 80 e 90 foram feitos estudos e implementações físicas em VLSI para avaliar o potencial de implementações dos comutadores e conexões das redes multiestágios em silício [Dinizt,1999][Dehon,2000]. Apesar da complexidade O(N Lg N) em termos de número de comutadores, as implementações VLSI mostram um custo $O(N^2)$ em área de silício [Dinizt,1999], e exigiam também conexões com multi-camadas. Para circuitos reconfiguráveis como FPGA, as redes também foram avaliadas como um mecanismo de interconexão programável. Em [Dehon,2000], uma nova arquitetura física para FPGA com o uso de multiestágios para interligar as LUTs foi proposta, ao invés dos tradicionais elementos chaveadores em 2D que interligam segmentos de barramentos em clusters (tipo ilha). Entretanto, a construção eficiente do circuito depende de recursos multi-camadas para os fios. Ademais, nenhum FGPA comercial foi construído com redes multiestágios e continuam a ser fabricados no estilo ilha.

Com a popularização dos FPGAs nos anos 90, alguns trabalhos propunham sistemas grandes compostos de vários chips FPGA e uma arquitetura de comunicação entre eles, conhecidas como arquiteturas multi-FPGA. Ao contrário de construir um novo FPGA no nível físico, a proposta é usar os FPGAs comerciais. Além de usá-los para realizar a computação, como por exemplo a emulação de circuitos, os FPGAs eram usados também para interconexão. A ideia era uma rede de interconexão no nível lógico, sintetizando a multiestágio sobre o FPGA, como as redes Clos [Barbie,1999], Folded-

Clos [Lin,1997], redes crossbar parciais [Ejnioui,1999]. Neste caso, o FPGA inteiro é usado para implementar uma rede e interligar os outros chips de FPGA que realizam computação. São usadas redes Clos que tem um custo maior (3n Lg n) estágios com comutadores 2x2 ou apenas três níveis e comutadores NxK, MxM e KxN.

Recentemente, trabalhos propõem o uso de redes em arquiteturas reconfiguráveis de grão grosso [Tanigawa,2008][Ferreira,2008][Ferreira,2009] e em sistema com multiprocessadores MPSoCs [Neji,2008]. Em [Tanigawa,2008], uma arquitetura em linhas com redes *Benes* sintetizadas em VLSI é proposta, ou seja, propõe construir fisicamente a rede em circuito para interligar unidades funcionais. Os resultados mostraram que o layout é bem compacto, cerca de 1.000 vezes menor que um processador multicore. Mesmo com um tempo de relógio 4 vezes maior, a versão multiestágio apresenta o mesmo tempo de execução para aplicações multimídia quando comparada aos processadores Multi-core. Porém o mapeamento das aplicações é feito manualmente na arquitetura. Semelhante a abordagem anterior, porém acoplado a um processo de tradução binária, onde em tempo de execução o código binário da aplicação é mapeado na arquitetura reconfigurável, o uso de rede multiestágio foi proposto em [Ferreira,2008]. Neste trabalho, as redes *Omega* [Lawrie,1975] são usadas para interligar em linhas de um arranjo de unidades funcionais acopladas a um processador MIPS. Um algoritmo de posicionamento e roteamento é feito em hardware, mostrando que o roteamento pode ser realizado em tempo de execução. Este trabalho também demanda a construção física em VLSI de um circuito reconfigurável com as redes.

Posteriormente, em um trabalho anterior dos autores [Ferreira,2009], as redes *Omega* foram usadas em FPGAs no nível lógico. A ideia era melhorar a capacidade de roteamento de um grid de ALUs para grafos de fluxos de dados. Os grafos eram mapeados no grid que não possui capacidade de roteamento, apenas comunicação direta dos vizinhos (norte,sul,leste e oeste). Quando dois vértices do grafo de fluxo de dados não eram vizinhos na arquitetura, a comunicação deles é roteada na multiestágio. Este trabalho mostrou dois resultados interessantes. Primeiro, as multiestágios, como recurso de roteamento em nível lógico em FPGA, apresentaram um área menor que a integração de capacidade de roteamento nas unidades do grid. Segundo, mesmo usando uma rede com conflitos (redes *Omega*), como a demanda de roteamento era da ordem de 30% dos sinais, a multiestágio realiza com sucesso o roteamento. Ademais, o algoritmo de roteamento, mesmo implementado em software, executa em milisegundos que possibilita sua integração em ambientes de compilação Justi-in-Time. Foram usadas redes com largura de 32 bits para as palavras de dados. Mais recentemente, as redes em FPGA juntamente com elementos de computação [Ferreira,2010], foram usadas em Bioinformática para modelar grafos de rede reguladoras de genes dinamicamente, os resultados preliminares apresentaram aceleração de duas ordens de grandeza. No caso das redes de genes, a largura de 1 bit é suficiente para implementação dos modelos.

Recentemente, um trabalho avalia a área e o desempenho dos FPGAs com as redes multiestágios em sistemas multiprocessadores do tipo Multiprocessors System on Chip (MPSoCs) [Neji,2008]. São sintetizados até 8 processadores miniMIPS, onde cada um possui uma memória de instrução dedicada e usam a rede para comunicar com 8 módulos de memória de dados. Os resultados mostram que o tempo de síntese e a área das multiestágios é bem menor quando comparada as redes crossbar. A complexidade da

rede é O(N Lg N) após a síntese. Porém não se diz qual a largura da palavra, os miniMIPs são em geral de 32 bits. Isoladamente as redes são avaliadas com até 64 entradas. Um algoritmo de multiplicação de matrizes é usado para validar o sistema.

Motivado por trabalhos anteriores dos autores [Ferreira,2009][Ferreira,2010], e o estado atual dos FPGAs, este trabalho difere dos anteriores ao propor uma avaliação de diversos aspectos das redes sintetizadas sobre FPGAs. Diferente do trabalho [Neji,2008], buscamos uma caracterização e estimativa de custo para diferentes larguras de bits e redes bloqueantes e rearranjáveis. Em [Neij,2008] apenas redes bloqueantes são avaliadas. Ademais, redes com até 512 entradas são avaliadas. O trabalho de [Neji,2008] é baseado em ferramentas da Altera, este trabalho é baseado em ferramentas Xilinx. Os novos FPGAs possuem um grande volume de LUTs, multiplicadores embarcados e módulos de memória. Por exemplo, uma Virtex6 possui aproximadamente 150.000 Luts e 416 módulos de memória. A ideia deste trabalho é explorar qual o espaço ocupado pela rede como um recurso de comunicação reconfigurável no FPGA. O espaço restante pode ser usado para implementar sistemas embarcados paralelos com elementos de processamento e/ou memória. Serão avaliadas: a área em LUTs, os comutadores, o roteamento, a latência, os bits de configuração e a largura da palavra. Este trabalho visa uma caracterização das interconexões gerando opções de projeto para sistemas embarcados.

3. Redes Multiestágios

As interconexões podem ser diretas ou indiretas. Os elementos de computação são interligados ponto a ponto em conexões diretas como por exemplo em uma malha bidimensional. Nas conexões indiretas, os elementos se comunicam através de comutadores. As conexões diretas são estáticas, rápidas, tem custo baixo mas perdem em flexibilidade. As conexões indiretas são dinâmicas e oferecem flexibilidade. As redes crossbar são um exemplo de rede dinâmica, porém o custo em comutadores é $O(N^2)$ para comunicar N elementos. As redes multiestágios surgiram nos anos 50/60 como uma alternativa com o custo O(N lg N) em comutadores, e desde então tem sido muito estudadas.

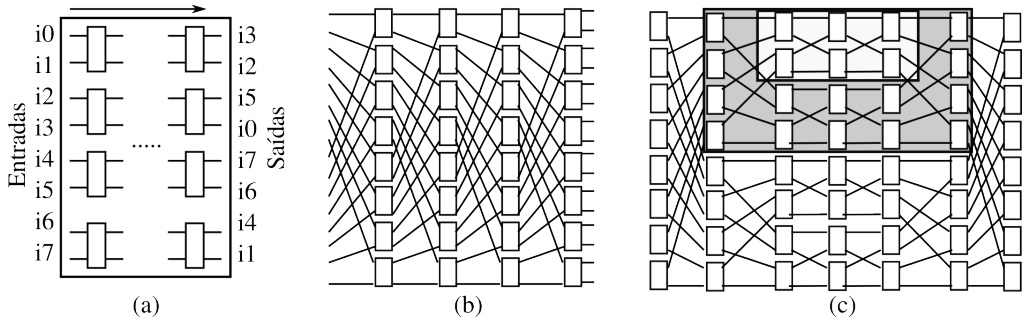


Figura 1. Multiestágios: (a) Permutação de Entrada; (b) Omega; (c) Benes.

Uma rede multiestágio possui N entradas e N saídas, é formada por uma série de colunas de comutadores (estágios), onde os comutadores do estágio j se conectam aos comutadores do estágio j+1. Formalmente a rede é definida pela tupla M(N,E,R,P,B), onde N é o número de entradas e saídas, E é o número de estágios, R é o radix do comutador, P é o padrão de permutação de bits entre os estágios e B é o número de configurações para cada comutador. Na seção 3.1 iremos classificar as redes e na seção

3.2 apresentamos os algoritmos de roteamento.

3.1. Classificação

As redes são classificadas como bloqueantes, rearranjáveis, estritamente não bloqueantes e não bloqueantes. Uma conexão completa da rede equivale a uma permutação de entrada como ilustra a Figura 1(a), onde as entradas i_0 à i_7 se conectam as saídas o_3,o_7,o_1,o_0,o_6,o_2,o_5 e o_4, respectivamente.

Uma rede é bloqueante quando não consegue rotear todas as permutações de entrada/saída. A rede *Omega* ou *Butterfly* são exemplos de redes bloqueantes. A Figura 1(b) ilustra uma rede Omega [Lawrie,1976] com 16 entradas/saídas. Se considerarmos comutadores 2x2, a rede Omega terá Lg_2 N estágios, no exemplo da Figura 1(b) tem 4 estágios. Para facilitar a nomenclatura, os termos logaritmos são da base 2. O número de comutadores é N/2*Lg N e o número de comutadores que um sinal tem que atravessar entre a entrada e a saída da rede é Lg N, ou seja, o atraso é O(Lg N) e o custo em comutadores O(N Lg N). O padrão de conexão entre os estágios que determina o tipo de rede. Estas redes são conhecidas com redes de permutação de bits. No caso da rede *Omega*, o padrão de bit é uma rotação à esquerda no endereço da linha, também conhecido como *perfect-shuffle*. Por exemplo, a linha 1 ou 000**1** em binário irá conectar a linha 2 ou 00**1**0 em binário como ilustrado na Figura 1(b), ou seja a linha $x_3x_2x_1x_0$ conecta a linha $x_2x_1x_0x_3$ para 16 entradas. Depois de realizada a rotação, o sinal passa pelo comutador do estágio, que pode trocar o último bit, por isso a rede *Omega* também é conhecida com *Shuffle-Exchange*. Apesar de bloqueante, a rede *Omega* realiza vários padrões com conectar a entrada i_j à saída o_{j+t} onde t é um inteiro e representa uma translação da entrada, outros padrões são detalhados em [Lawrie,1975]. A *Butterfly* conecta a linha $x_{n-1}...x_i...x_1x_0$ na linha $x_{n-1}...x_0...x_1x_i$ ou seja troca o bit x_i pelo x_0 em função do estágio da rede. Tanto a *Omega* quando a *Butterfly* são conhecidas com redes de *Bayan* onde uma entrada pode alcançar qualquer saída, e existe apenas um único caminho para rotear uma entrada em uma saída. Esta característica simplifica o roteamento que será detalhado na seção 3.2. Vários padrões de redes bloqueantes foram propostos nos anos 70, e no inicio da década de 80 foi demonstrado que estes padrões eram equivalentes [Wu,1980].

Uma rede é rearranjável se consegue realizar todas as permutações, desde de que as conexões já estabelecidas possam ser rearranjadas entre alguns entradas e saídas da rede para não gerar conflitos. A rede *Benes* [Benes,1965] ilustrada na Figura 1(c) é um exemplo de rede rearranjável. Entretanto o custo da rede *Benes* é o dobro da rede *Omega*, tem pelo menos 2*(Lg N)- 1 estágios, o que aumenta também o atraso. Os algoritmos de roteamento serão discutidos na seção 3.2. Assim como as redes bloqueantes, as redes rearranjáveis são organizadas em classes [Yeh,1992].

A rede é estritamente não bloqueante, quando as requisições de conexão entre entrada e saída chegam de forma assíncrona e é sempre possível roteá-las. A rede será não bloqueante se for necessário respeitar um protocolo de roteamento para não alterar as conexões pre-estabelecidas. A rede *Clos* é um exemplo de rede não bloqueante. Entretanto, o custo dos comutadores aumenta significativamente, como ilustra a Figura

2(a) que apresenta uma rede *Clos* genérica com 3 estágios proposta em [Clos,1953]. O primeiro estágio tem m comutadores n x k, o segundo estágio tem k comutadores m x m e o último estágio tem m comutadores k x n, para n entradas/saídas.

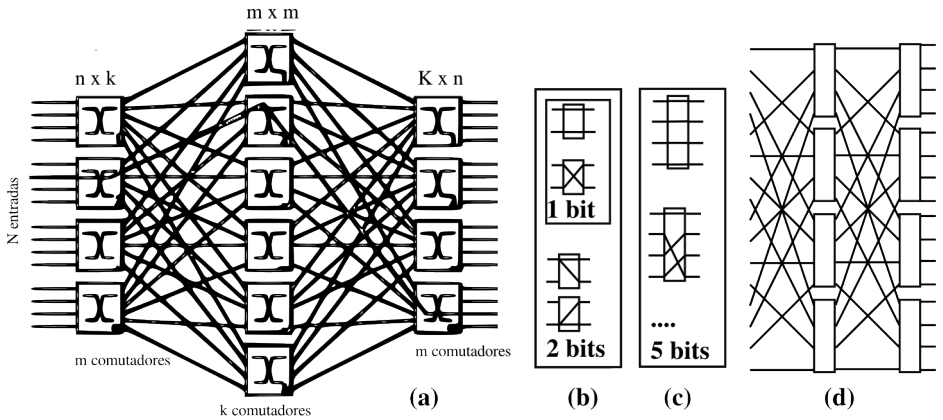


Figura 2. Multiestágios e Comutadores: (a) Clos; (b) 2x2; (c) 4x4 (d) Radix4.

O comutador da rede pode ser simples como um comutador 2x2 (veja Figura 2(b)) ou mais complexo com MxK entradas e saídas como os comutadores da rede Clos (veja Figura 2(a)). Quando maior o comutador, menor será o número de estágios. Ou seja, tem um custo benefício entre a complexidade do comutador e o número de estágios. Para comutadores 2x2 ou radix 2, uma rede bloqueante terá Lg_2N estágios como vimos para a rede Omega na Figura 1(b). Agora, se usarmos comutadores 4x4 ou radix 4, o número de estágios será Lg_4N como ilustrado na Figura 2(c) e na Figura 2(d) para uma rede Omega radix 4 com 16 entradas que terá apenas 2 estágios. Um comutador simples 2x2 tem 1 bit de configuração se opera no modo direto ou cruzado e 2 bits de configuração se faz um broadcast local (veja Figura 2(b)). A configuração broadcast local serve para broadcast ou multicast global, quando uma entrada da rede é conectada a um subconjunto de saídas da rede. Porém, o uso de multicast dobra o número de bits para radix 2. O comutador 4x4 terá 24 configurações simples 1 para 1 (5 bits) e 16 bits se considerarmos todas as possibilidade com multicast. A Tabela 1 ilustra o número de bits para diversos tamanho de redes *Omega*.

Tabela 1: Número de Bits de Configuração para os comutadores em Redes Omega com e sem capacidade de multicast. NA – não se aplica.

Tamanho N	Omega 2x2	Omega 2x2 Multicast	Omega 4x4	Omega 4x4 Multicast
8	12	24	NA	NA
16	32	64	40	128
32	80	160	NA	NA
64	192	384	240	768
128	448	896	NA	NA
256	1024	2048	1280	4096

Podemos observar que a radix4 precisa de mais bits de configuração, e fica restrita a valores de N potência de 4. Uma memória de configuração da rede irá armazenar os bits dos comutadores. Para uma rede de 256 entradas e radix2 temos 1024 bits ou 128 bytes para cada configuração da rede.

A rede pode ser usada para conectar elementos como ALUs [Ferreira,2009], elementos de processamento com recursos acoplados (registros, controle, alu) ou processadores simples [Neji,2008]. Pode ser usada para compartilhar recursos como multiplicadores ou módulos de memória, para acesso paralelo a módulos de memória. Se usada como uma rede global para interligar todas as ALU, simplifica o mapeamento, uma vez que pode-se posicionar em qualquer ALU que a rede fará o roteamento global. As redes já foram usadas para comutação por pacotes (ATM, supercomputadores, NoC). Neste trabalho avaliamos comutações por circuito mais adequadas para conectar ALUs ou elementos de processamento simples. O trabalho para MPSoCs usa comutação por pacotes [Neji,2008] entre os processadores miniMIPS. A Figura 3 ilustra diversas opções de arquiteturas onde as redes podem ser usadas. A largura da palavra é o número de bits que cada linha de dados transmite. Redes com 1 bit podem ser úteis para computação modeladas como autômatos celulares ou para sincronização dos elementos de processamento, enviando sinais de controle no lugar de sinais de dados.

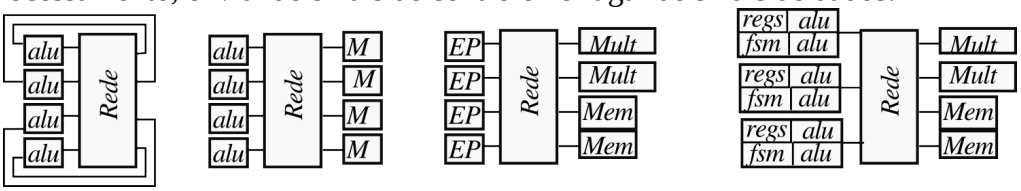


Figura 3. Diferentes Arquiteturas com redes de interconexão

3.2. Roteamento

O roteamento em multiestágio depende do tipo de padrão de bits entre os estágios e do tipo da rede. Nesta seção iremos descrever o roteamento em redes bloqueante *Omega* com e sem estágios extras e o roteamento em redes rearranjáveis.

A rede *Omega* é conhecida como auto-roteável, e pode ser usada com comutação por pacote baseado nos endereços de origem e destino. Se calcula um XOR dos endereços, e o resultado é percorrido bit a bit do mais significativo ao menos significativo, se o bit for 1, o comutador liga cruzado, se o bit for 0 o comutador ligado direto. Cada par origem destino tem um único caminho. Um conflito ocorre quando dois pares de entradas/saídas passam pela mesma linha de um comutador em um determinado estágio. Para verificar os conflitos, o roteamento pode ser feito usando janelas. Suponha uma palavra de roteamento formada pela concatenação dos endereços de origem e destino, ou seja, $O_{n-1}O_{n-2}....O_1O_0D_{n-1}D_{n-2}...D_1D_0$. Para facilitar a explicação vamos supor uma rede com 16 entradas como a ilustrada na Figura 4(a). Uma janela de 4 bits irá deslocar sobre a palavra de roteamento. A entrada 1 ou em binário 0001 é conectada a saída 6 ou em binario 0110, fazendo o caminho 0**0010**110, 00**0101**10, 000**1011**0 e 0001**0110**, onde a janela é mostrada em negrito. A janela representa a linha em cada estágio. Neste exemplo no primeiro estágio passará pela linha 2, depois pela linha 5, depois pela linha 11 e finalmente a linha 6 de destino. Quanto um outro par entrada e saída, tenta conectar pode ocorrer conflito se for passar pela mesma linha no mesmo

estágio. Por exemplo, a entrada 9 não pode se conectar a saída 4, pois gera a palavra 10010100, que terá conflito no primeiro estágio 1**0010**100 na linha 2, como ilustra a Figura 4(a).

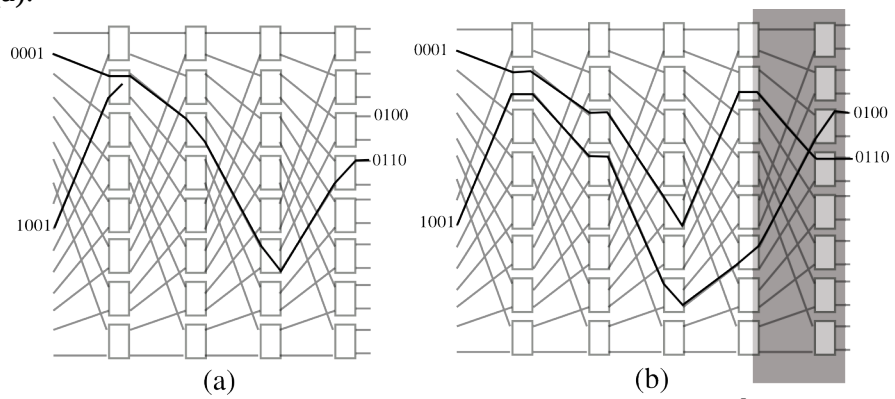


Figura 4. Roteamento em Rede Omega: (a) Caminho Único e Conflito; (b) Estágio Extra.

Uma solução para melhorar a capacidade de roteamento da Omega sem aumentar muito o custo é acrescentar estágios extras. Ao adicionar um estágio, no lugar de um caminho teremos dois caminhos. Cada estágio, dobra o número de caminhos. O roteamento é semelhante. Agora temos um bit a mais no caminho. Para conectar a entrada 0001 na saída 0110, teremos a palavra 0001X0110, onde X pode ser 0 ou 1. Suponha 0. Teremos o caminho 0**0010**0110, 00**0100**110, 000**1001**10, 0001**0011**0 e 00010**0110**, ou seja passa pelas linhas 2, 4, 9, 3 e finalmente 6. Para ligar sem conflito a entrada 9 na saída 4, pode-se usar X=1, no roteamento e nenhum conflito é gerado, como ilustra a Figura 4(b). A medida que a rede cresce aumenta a probabilidade de conflito. Apesar de vários estudos, ainda é um problema em aberto determinar a capacidade de roteamento com estágios extras de uma rede Omega [Gazit, 1989].

Uma implementação em hardware para o roteamento em redes Omega com estágios extras foi apresentada recentemente em [Ferreira,2008]. Para cada comutador, uma unidade simples de controle é gerada. As unidades são interligadas seguindo o padrão da multiestágio e um decodificador de prioridade faz a seleção de caminhos. Esta implementação pode ser realizada em FPGA em ambientes dinâmicos. Se a aplicação for compilada, um roteamento estático em software cuja a complexidade é polinomial O(N Lg N) pode ser usado.

Para redes rearranjáveis, a maioria dos algoritmos supõe conhecer a priori a permutação completa entrada/saída. Vários algoritmos sequenciais com o primeiro trabalho proposto por [Waksman,1968], e posteriormente várias versões paralelas ao longo das décadas de de 80 e 90 também foram propostas [Çam,1999]. A motivação para as versões paralelas era o uso da rede em computadores paralelos. A rede tem baixa latência O(Lg N), porém para executar o roteamento era necessário O(N Lg N). Apesar de várias propostas, os modelos de computadores que executavam os algoritmos paralelos eram teóricos, e não se encontra resultados de implementações em máquinas paralelas na literatura. O algoritmo básico pode ser executado rapidamente em um processador simples. A ideia é programar a rede, estágio por estágio, dos estágios externos para os mais internos. A cada passo se programa N comutadores e em Lg N passos se programa a rede. Um ponto em aberto, são implementações em hardware de

roteamento rearranjável para Benes.

A maioria dos algoritmos aborda apenas a conexão 1 para 1, de um entrada para uma saída sem multicast. Se considerarmos multicast além de aumentar o número de bits de configuração, aumenta-se a complexidade do roteamento. O multicast pode ser interessante em algumas aplicações.

4. Resultados Experimentais

As redes multiestágios foram descritas com um código parametrizado em VHDL em função do radix, número de estágios, bits de configuração, largura da palavra, etc. O objetivo foi avaliar o custo das redes e como as ferramentas de síntese de FPGA capturam uma descrição alto nível das redes. Como ferramenta usamos o ISE da XILINX versão 11 e como arquitetura alvo usamos uma Virtex6 que possui módulos de memória e multiplicadores embarcados. A Tabela 2 apresenta os resultados para redes Omega com vários tamanho e com a largura de bits variando de 1, 8, 16 e 32 bits.

Tabela 2: Área, Tamanho das Redes Multiestágios em Luts para N variando de 32 a 512 entradas/saídas e com largura de bits: 1, 8, 16 e 32 bits. Sintetizado em uma Virtex6 com 150730 Luts.

	32	64	128	256	512
1 bit	392	425	1108	1838	4017
8 bits	1227	2443	6039	12592	29107
16 bits	2251	4747	11501	24880	57779
32 bits	4299	9355	22765	49456	115123

Podemos observar da Tabela 2, que o espaço ocupado no FPGA para a rede *Omega* em função do tamanho da entrada cresce com a complexidade O(N Lg N), ou seja, a ferramenta de síntese captura a regularidade da rede. Em termos de espaço, vemos que uma rede de 512 entradas com largura de 32 bits ocupa 70% do FPGA. Entretanto, a complexidade de um sistema com 512 conexões de 32 bits é bem elevada. Para outros contextos, como uma rede com 128 conexões de 32 bits, apenas 15% do FPGA é ocupado, sendo que o restante pode ser usado para implementação de unidades de processamento. Outro ponto é que ao aumentar de 1 bit para 8 bits, a área em LUTs aumentou em média apenas 5,7 vezes, ao aumentar para 16 bits em relação a 1 bit, a área aumentou em 11 vezes em média, e para 32 bits o aumento médio foi de 22 vezes. Mostrando que a regularidade da rede é um aspecto importante capturado pela síntese. Além disso, as redes com largura pequena em bits apresentam um custo proporcionalmente maior.

A Tabela 3 mostra os resultados de latência para as redes da Tabela 2. Podemos observar que a latência não varia com a largura de bits, apenas o número de estágios. A cada coluna estamos acrescentando um estágio a mais na rede, ao dobrar o seu tamanho. Por exemplo, a rede de 32x32 tem 5 estágios e latência de 1,97 ns. Ao dobrar para uma rede de 64x64 teremos 6 estágios e a latência de 2,28, ou seja, 0,33 ns para atravessar um estágio a mais. Podemos observar que a latência aumenta em média 0,34 ns por

estágio, ou seja, a rede tem um comportamento linear em atraso. Se adicionarmos 1 estágio, a latência aumenta 0,34 ns. Este recurso pode ser usado para aumentar a capacidade de roteamento da rede com estágios extras. Cada estágio extra dobra o número de caminhos para um par de entrada e saída. Entretanto, uma rede bloqueante pode ter no máximo 2 Lg N estágios, mais estágios não alteram a capacidade de roteamento.

Tabela 3: Latência (ns) Redes Multiestágios para N variando de 32 a 512 entradas e saídas e com largura de bits: 1, 8, 16 e 32 bits. Sintetizado em uma Virtex6.

	32	64	128	256	512
1 bit	1,97	2,28	2,64	2,95	3,32
8 bits	1,97	2,28	2,64	2,95	3,32
16 bits	1,97	2,28	2,64	2,95	3,32
32 bits	1,97	2,28	2,64	2,95	3,32

Outro ponto importante é o numero de módulos de memória para armazenar as configurações. A Tabela 4 apresenta os resultados. Uma virtex6 tem 416 módulos, para a configuração de 512 são usados 72, ou seja, apenas 17% dos módulos. A primeira coluna com redes com 32 entradas usa LUTs para armazenar a configuração, enquanto que as outras colunas com redes de 64 à 512 entradas, a configuração é armazenada nas memórias embarcadas. A ferramenta da Xilinx faz a escolha pelas LUTs para reduzir o uso dos recursos do FPGA na rede de 32, porém para as outras possibilidades, a ferramenta verifica que os módulos são mais adequados para armazenar as configurações. Cada memória pode armazenar até 512 linhas, ou seja, sem aumentar o custo da implementação, podemos gerar 512 configurações diferentes de interconexão e programá-las no FPGA estaticamente ou dinamicamente. Ou seja, uma computação que envolva até 512 padrões diferentes de comunicação entre os elementos de processamento e memória pode ser configurada. No caso de uma rede com 128 entradas, usamos apenas 14 módulos, e 17% das LUTs do FPGA, sendo que praticamente todos os módulos de memória (mais de 400) juntamente com 80% do FPGA poderão implementar recursos de computação.

Tabela 4: Módulos de Memória sintetizado em uma Virtex6 com 416 módulos.

	32	64	128	256	512
Número de Módulos	0	6	14	32	72

Para avaliar o impacto da rede junto com unidade funcionais, a Tabela 5 apresenta resultados de síntese com ALUs de 8, 16 e 32 bits conectadas a uma rede com 32 entradas. Como cada ALU tem duas entradas e uma saída, a rede tem as 32 saídas conectadas as entradas das ALU, e as 16 saídas das ALUs juntamente com 16 entradas externas são conectadas a entrada da rede. As entradas externas servem para alimentar as ALUs. Podemos observar na Tabela 5, que o tamanho da rede em separado pode ser usado com uma base de estimativa da área ocupada. Ao realizar a síntese em conjunto a

área total foi próxima da estimativa da soma em separado da área das ALUs e da rede.

Tabela 5: Área das ALU sintetizadas em separado e em conjunto com uma rede Omega de 32 entradas.

largura	8bits	16 bits	32 bits
Área em LUTs das 16 ALUs	1824	3824	7920
Área em LUTs 16 ALUs + Rede sintetizado em separado	2848	5872	12016
Área em LUTs das ALUs + Rede sintetizado em conjunto	2910	6187	12228

Para aumentar o desempenho da rede podemos fazer uso de pipeline, ao inserir registros entre os estágios da rede. A Tabela 6 mostra os resultados de área e latência para a rede com registros entre os estágios. A vantagem é a redução do tempo de relógio para 0,5 ns. O aumento médio em área foi de 30% para uma rede com largura de 8 bits. Os módulos de memória podem armazenar a configuração da execução em pipeline, aumentando a vazão dos resultados.

Tabela 6: Rede com Pipeline (Registros entre os estágios). Largura de 8 bits

	32	64	128	256	512
Área com pipeline	1549	3370	7403	17104	37294
Área sem pipeline	1227	2443	6039	12592	29107
Acréscimo	26,00%	38,00%	23,00%	36,00%	28,00%
Atraso em ns	0,52	0,52	0,52	0,52	0,52

O padrão butterfly também foi sintetizado e apresentou resultados semelhantes a Omega. A radix4 ocupa um espaço menor. O tamanho da radix2 evolui com O(N Lg N), mostrando que a síntese em FPGA não aumenta a complexidade de custo em área da rede. Já o padrão radix4 com comutadores 4x4 reduz a área pois o número de estágios é $N\ Lg_4N$. A Tabela 7 apresenta os resultados para síntese com radix 4 para uma rede de 1 bit de largura e de 32 bits de largura. Os resultados são comparados com a rede radix2. Em alguns casos não se aplica (NA) pois o radix4 tem que ser potência de 4. Para a rede com largura de 1 bit, a radix4 ocupa uma área em LUTs duas vezes menor. Para a largura de 32 bits, a área da radix4 é em torno de 60% da área da radix2. A latência reduz de 10 a 20% com a radix4. Se for considerado multicast, o número de bits de configuração para a radix4 será o dobro da radix2. Trabalhos futuros são necessários para avaliar a número de conflitos em função do radix. Para máquinas paralelas, um trabalho recente apresentou uma versão plana dos comutadores agrupados por linha e como comunicação entre as linhas [Kim,2007]. A abordagem é baseada em redes com padrão *butterfly*. As interligações e o roteamento perde um pouco em regularidade e um estudo detalhado deve ser feito para avaliar esta alternativa.

Tabela 7: Rede Radix 4 e Radix 2. Largura de 1 e 32 bits. Área em LUTs e atraso em ns.

	16	32	64	128	256
1 bit, Radix2 – Área (LUTs)	-	392	425	1108	1838
1 bit, Radix4 – Área (LUTs)	32	NA	192	NA	1024
32 bits, Radix2 – Área (LUTs)	-	4299	9355	22765	49456
32 bits, Radix4 – Área (LUTs)	1024	NA	6144	NA	32768
Radix2 – Latência (ns)	-	1,97	2,27	2,64	3,20
Radix4 – Latência (ns)	1,46	NA	2,08	NA	2,69

Para redes rearranjáveis, como estudo de caso, a rede *Benes* foi sintetizada. Um código VHDL parametrizado também foi gerado. A Tabela 8 mostra os resultados para área em LUTs e o atraso em ns considerando uma largura de 8 bits para palavra. Podemos observar que ao dobrar o tamanho da rede, a latência aumenta em média 0,60 ns, pois em uma rede *Benes*, dobrar o tamanho significa incluir dois estágios a mais. Em relação a rede Omega bloqueante com largura de 8 bits, a área em LUTs é 70% maior. Para aplicações com compilação estática as redes Benes são interessantes pois não geram conflitos e o roteamento tem complexidade O(N Lg N) e pode ser facilmente incorporado no compilador.

Tabela 8: Rede Benes. Largura de 8 bits. Área em LUTs e atraso em ns.

	32	64	128	256	512
Área em Luts para 8 bits de largura	1792	4224	9984	23040	52224
Latência em ns	3.45	4,07	4,75	5,43	6,11

5. Conclusão

Este trabalho propõe um estudo dos custos de implementação de redes multiestágios em FPGA para implementação de sistemas embarcados paralelos. As redes podem ser descritas com código parametrizável em VHDL. As ferramentas de síntese em FPGA capturam a regularidade da rede com a complexidade O(N Lg N) em área e em bits de configuração bem como a latência com complexidade O(Lg N). Os módulos de memória embarcados nos FPGAs de última geração constituem uma boa alternativa para armazenar internamente várias configurações (até 512 em uma Virtex6). O espaço ocupado pelas redes é reduzido e é viável a síntese de várias unidades de processamento simples como ALUs. As unidades podem ser homogêneas ou heterogêneas, uma vez que a rede é um mecanismo global de roteamento, e não é necessário se aplicar algoritmos de posicionamento, uma vez que todas as posições são equivalentes. Trabalhos futuros incluem a implementação de algoritmos para sistemas embarcados com arquiteturas paralelas. Outro ponto importante é a implementação em hardware no FPGA de algoritmos de roteamento bloqueantes e rearranjáveis, bem como a implementação em softcore do roteamento internamente no FPGA em conjunto com as redes e unidades de

processamento para suporte de compilação Just-in-Time (JIT).

Referências

Barbie, J., e Reblewski, F. (1999) "Emulation System having a Scalable Multi-level Multi-Stage Hybrid Programmable Interconnection Network", US Patent 5907679.

Benes, V. (1965) “Mathematical Theory of Connecting Networks and Telephone Traffic”, Academic Press, New York.

Clos, C. (1953) “A study of non-blocking switch networks”, *Bell System Tech. J*. 32:407-425.

Çam, H. and Fortes, J. A. 1999. Work-Efficient Routing Algorithms for Rearrangeable Symmetrical Networks. IEEE Trans. Parallel Distrib. Syst. v10(7)

DeHon, A. (2000) “Compact, multilayer layout for butterfly fat-tree” In *Proceedings of the 12th ACM symposium on Parallel algorithms and architectures*, pp.206-215.

Dinitz, Y., Even, S., Kupershtok, R., and Zapolotsky, M. (1999). Some compact layouts of the butterfly. In 11Th *ACM Symposium on Parallel Algorithms and Architectures.*

Ejnioui, A., Ranganathan, N. (1999). Multi-terminal net routing for partial crossbar-based multi-FPGA systems. In Proceedings of the ACM/SIGDA Seventh international Symposium on Field Programmable Gate Arrays

Ferreira, R. S. ; Laure, M. ; Rutizig, M. ; Beck, A. C. ; Carro, L. (2008). “Reducing interconnection cost in coarse-grained dynamic computing through multistage network.” IEEE International Conference on Field Programmable Logic and Applications (FPL), p. 47-52.

Ferreira, R. S. ; Damiany, A. ; Vendramini, J. ; Teixeira, T. ; Cardoso, J. (2009). On Simplifying Placement and Routing by Extending Coarse-Grained Reconfigurable Arrays with Omega Networks. In: 5th International Workshop on Applied Reconfigurable Computing,

Ferreira, R. S. ; Vendramini, J. ; Carvalho, L. (2010). Simulação em FPGA de Redes Reguladoras com Topologia Livre de Escala. In: VI Jornadas sobre Sistemas Reconfiguráveis, Aveiro, Portugal.

Gazit, I. and Malek, M. (1989). On the Number of Permutations Performable by Extra-Stage Multistage Interconnection Networks. *IEEE Trans. Comput*. v38(2)

Kim, J., Dally, W. J., and Abts, D. (2007). Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of the 34th Annual international Symposium on Computer Architecture* (San Diego, California, USA). ISCA '07. ACM, New York, NY, 126-137.

Lawrie, D.H. (1975) “Access and Alignment of Data in an Array Processor”, IEEE Trans. on Computers, V24 (12)

Leiserson, C. E. (1985) “Fat-trees: Universal networks for hardware efficient supercomputing.” IEEE Transactions on Computers, v34 (10) p. 892–901,

Lin,S., Lin,Y. e Hwang, T.(1997)“Net Assignment for the FPGA-Based Logic

Emulation System in the Folded-Clos Network Structure", IEEE Trans. CAD, v16(3)

Neji, B., Aydi, Y., Ben-atilallah, R., Meftaly, S., Abid, M., Dykeyser, J-L. (2008) Multistage Interconnection Network for MPSoC: Performances study and prototyping on FPGA, IEEE Design and Test Workshop (IDT)

Tanigawa, K., Zuyama, T., Uchida, T. , e Hironaka, T. (2008) “Exploring compact design on high throught coarse-grained reconfigurable architectures”, IEEE *Proceedings of the International Workshop on Field-Programmable Logic (FPL).*

Yeh, Y-M., e Tse-yun Feng, T-Y (1992) “On a Class of Rearrangeable Networks”, *IEEE Trans. on Computers*, v41(11), pp. 1361—1379.

Wu, C-L., Feng, T-Y. (1980). “On a Class of Multistage Interconnection Networks.” *IEEE Trans. Comput.* V29 (8), 694-702.

Waksman, A. (1968) “A Permutation Network”, J. ACM, vol. 15, no. 1, pp. 159-163

Comparação de Modelos de Memória para Plataformas MPSoC Usando SystemC

Bruno Cruz de Oliveira[1], Ivan Saraiva Silva[2]

[1]Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte (UFRN) – Natal, RN – Brasil

[2]Departamento de Informática e Estatística
Universidade Federal do Piauí (UFPI) – Teresina, PI – Brasil

bcruz@lasic.ufrn.br, ivan@ufpi.edu.br

Abstract. *This paper presents two memory organizations exploration for MP-SoC virtual platforms. A general purpose platform is present alongside its description in SystemC. The STORM platform allows one to describe architectures comprising processors, networks-on-chip and cache memories. The focus of this paper is to evaluate the impact of shared and distributed memories organizations in the design of multiprocessor systems. To evaluate such scenarios, specific instances of STORM were submitted to a oil reservoir simulation application.*

Resumo. *Este artigo apresenta a exploração de dois modelos de memória para plataformas virtuais MPSoC. Uma plataforma de propósito geral é apresentada em conjunto com sua descrição em SystemC. A plataforma STORM permite descrever arquiteturas incluindo processadores, redes-em-chip e memórias caches. O foco desse artigo é avaliar o impacto dos modelos de memória distribuída e compartilhada no projeto de sistemas multiprocessados. Para avaliar tais cenários, diferentes instâncias da plataforma STORM foram utilizadas para executar uma aplicação de simulação de reservatório.*

1. Introdução

O constante aumento de complexidade das aplicações demanda suporte de hardware cada vez mais poderoso do ponto de vista de sua capacidade computacional. Com a aproximação do limite de freqüência dos processadores, a solução mais explorada e, provavelmente uma das mais viáveis é o paralelismo. Sendo assim, a integração de múltiplos núcleos processantes em um único chip se torna interessante, dada a alta densidade de transistores em chip permitida pelas tecnologias de fabricação atuais. As principais pesquisas nesse sentido estão focadas no conceito de Sistemas em Chip Multiprocessados (MPSoC). No entanto, sistemas arquiteturalmente complexos como os MPSoCs apresentam diversos desafios aos projetistas. Muitos desses desafios, tais como, manutenção da coerência de cache e consumo de energia vêm sendo estudados há algum tempo. Contudo, muitas soluções arquiteturais são possíveis, dado o número de componentes disponíveis, o que evidencia um enorme espaço de projeto a ser explorado.

Este artigo tem como foco o problema da organização de memória em MPSoCs como forma de atender as restrições de desempenho de aplicações distribuídas. Neste

trabalho é abordado um cenário onde múltiplas tarefas podem ser alocadas em diferentes elementos processantes de um MPSoC. Para avaliar o desempenho, quanto ao tráfego injetado pelas tarefas, no subsistema de comunicação, dois modelos de arquitetura de memória são explorados em instâncias diferentes de um MPSoC baseado em Redes-em-Chip (NoC – Network-on-Chip).

O primeiro é o modelo de memória compartilhada. Neste modelo, o espaço de endereçamento é formado por um ou mais módulos de memória, distribuídos na rede-em-chip e compartilhada por todos os elementos processantes do MPSoC. O segundo é o modelo de memória distribuída. Neste modelo cada elemento processante do MPSoC tem seu próprio módulo de memória, com espaço de endereçamento privado. Enquanto no modelo de memória compartilhada é usada a comunicação baseada em variáveis compartilhadas, no modelo de memória distribuída a comunicação é baseada em troca de mensagens. Para avaliar o desempenho em termos de comunicação e processamento entre os dois modelos de memória, são comparadas: a latência dos pacotes, a carga injetada na rede, o número médio de ciclos por instrução e o número de instruções executadas.

Como estudo de caso, foi implementada uma aplicação da área de exploração de petróleo e gás natural, a simulação de reservatório [Fachi, Harpole e Bujnowski 1982]. Esse tipo de aplicação é conhecida por demandar alta capacidade de processamento.

Este trabalho está organizado da seguinte forma: na seção 2 é apresentada a plataforma virtual STORM (MPSoC Directory-Based Platform) [Girão et al. 2007]. A seção 3 detalha a aplicação usada como estudo de caso. A seção 4 mostra os experimentos e discute os resultados obtidos. A conclusão é apresentada na Seção 5.

2. A Plataforma STORM

Nos últimos anos alguns trabalhos sobre o projeto de plataformas MPSoC foram publicados. A maioria deles com foco em plataformas baseadas em barramentos heterogêneos [Benini et al 2006], [Fummi et al 2004], [Suh, Blough e Lee 2004], enquanto que um número menor, mas crescente, tem foco em plataformas baseadas em NoC [Petrot, Greiner e Gomez 2006], [Forsell 2002]. A maioria dos trabalhos de projeto de plataformas baseadas em NoC focam o desenvolvimento e a execução de aplicações e a exploração arquitetural. No entanto, a organização e a otimização da memória devem ser consideradas durante a fase de projeto, visto que grande quantidade de aplicações requer estruturas de dados complexas. Não obstante a importância da escolha do modelo de memória, no projeto de sistemas em chip, de um modo geral e MPSoCs em particular. Outros aspectos também precisam ser levados em consideração para manter o funcionamento correto e consistente das operações de acesso à memória. Este trabalho leva em consideração a coerência e consistência de memória. Estas propriedades são necessárias para garantir a precisão das operações de leitura e escrita na memória e podem gerar impacto no desempenho de sistemas-em-chip.

STORM é uma plataforma virtual baseada em NoC, desenvolvida em SystemC. Por ser uma plataforma, STORM não possui uma arquitetura definida, mas sim um conjunto de módulos de hardware e especificações sobre como devem ser utilizados, além de bibliotecas de software, sendo possível para o projetista criar diversas instâncias de arquiteturas com diferentes características. Isto não só está de acordo com a atual tendência de projeto baseado em plataforma, como também permite a integração

de diferentes módulos e uma maior facilidade na exploração de espaço de projeto. Quanto aos módulos de hardware, STORM fornece um processador (SPARC V8), caches de instruções e dados, módulos de memória, módulo de diretório para manutenção da coerência das caches, um modelo de rede-em-chip (mesh 2D) e módulos de interface com a NoC. Estas características permitem ao projetista criar múltiplas instâncias de arquiteturas com diferentes propriedades para executar aplicações de propósito geral ou específico.

Atualmente, dois modelos de memória estão disponíveis. O primeiro é o modelo de memória compartilhada. Neste modelo um ou mais módulos de memória, fisicamente distribuídos entre os nós da NoC, fornecem um espaço de endereçamento único. Deste modo, todos os processadores no sistema podem acessar qualquer endereço de memória transparentemente, com diferentes custos de latência. Este modelo de memória oferece um modelo de programação baseado em variáveis compartilhadas. O segundo modelo de memória disponível na plataforma STORM é o modelo de memória distribuída. Neste modelo cada nó da NoC tem um processador com seu próprio módulo de memória. Um processador tem acesso apenas ao espaço de endereçamento fornecido pelo seu módulo de memória. A comunicação entre processadores é feita através de troca de mensagens.

As implementações dos nós do modelo de memória compartilhada e do modelo de memória distribuída podem ser vistas na Figura 1.a e Figura 1.b, respectivamente. Nessa figura é possível ver os módulos básicos de hardware da plataforma STORM, processador e NoC, sendo possível também ver os módulos que compõem a interface e fornecem as funcionalidades do protocolo de comunicação entre o processador, a NoC e a memória.

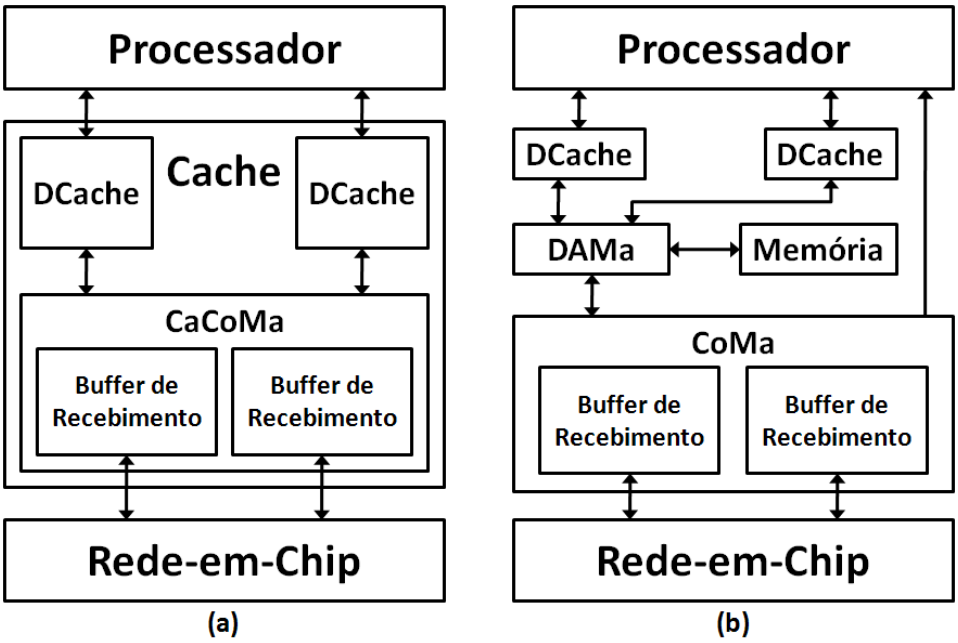


Figura 1. Nós da plataforma STORM: (a) memória compartilhada, (b) memória distribuída.

No modelo de memória compartilhada, Figura 1.a, o CaCoMa (Cache Communication Manager) é responsável por enviar e receber pacotes através da NoC. Ele também é responsável pela tradução de endereços lógicos (endereços de memória) em endereços físicos (endereço de NoC do nó de memória correspondente ao endereço

de memória requisitado). O processo de tradução utiliza uma tabela de endereços (ATA – Address Table). A ATA associa um endereço físico ao último endereço lógico armazenado por um módulo de memória. Assim, para traduzir um endereço lógico X para endereço físico, o CaCoMa deve percorrer as entradas de sua ATA e retornar o endereço físico da primeira entrada cujo valor do último endereço lógico é maior que X. Um exemplo de ATA é mostrado na Tabela 1.

Tabela 1. Exemplo de ATA.

Endereço Lógico	Endereço Físico
0x000007FF	0,1
0x00000FFF	1,0
0x000017FF	1,2

No modelo de memória distribuída (Figura 1.b) a interface entre o processador, a memória e a NoC é feita pelos módulos DAMa e CoMa. O DAMa (Data Access Manager) fornece uma interface entre as caches (ICache e DCache) e os outros módulos no nó. Para cada módulo ligado ao DAMa é atribuída uma faixa de endereços de memória. Ao receber uma requisição da cache, o DAMa verifica a qual módulo pertence o endereço acessado, realiza a comunicação com o módulo correspondente e repassa os dados para a cache. Quando necessário, o DAMa pode enviar junto com os dados um sinal para avisar a cache que os dados não devem ser memorizados, mas apenas repassados ao processador. O COMa (Communication Manager) é responsável por enviar e receber pacotes através da NoC. Como pode ser visto na figura 1.b, o CoMa possui buffers de envio e recebimento. O tamanho de cada buffer é configurável e independente um do outro. Um sinal de interrupção é gerado sempre que o buffer de recebimento do CoMa atinge um determinado nível. Esse nível é configurável e deve ser ajustado de forma a evitar que o buffer de recebimento encha e acabe sobrecarregando os buffers dos roteadores da NoC.

2.1. Programabilidade da plataforma STORM

A plataforma STORM não conta atualmente com um sistema operacional, dessa forma, uma biblioteca em C foi implementada para fornecer comunicação e sincronização em nível de usuário. No modelo de memória distribuída, a comunicação entre os processadores ocorre por troca de mensagens. Assim, um subconjunto das funções do padrão MPI (Message Passing Interface) foi implementado. Este subconjunto (Tabela 2) é suficiente para implementar uma grande quantidade de aplicações uma vez que todos os elementos básicos necessários para comunicação entre processos estão inclusos. Apesar de não ser completo, esse subconjunto pode ser facilmente expandido, uma vez que todos os tipos básicos e constantes especificadas pelo padrão MPI foram implementadas. A utilização do padrão MPI permite também que aplicações existentes sejam portadas para a plataforma STORM com pouco ou nenhum esforço.

No modelo de memória compartilhada, a comunicação é realizada através de variáveis compartilhadas (variáveis globais) sincronizadas por mutexes. Dessa forma, toda sincronização deve ser realizada de forma explícita pelo programador, tornando a tarefa de portar aplicações para este modelo difícil e passível de erros.

Tabela 2. Operações do padrão MPI implementadas e suas descrições

Operação	Descrição
MPI_Comm_size	Retorna o número de processadores em um grupo
MPI_Comm_rank	Retorna o rank de um processador em um grupo
MPI_Init	Inicializa o ambiente MPI
MPI_Finalize	Finaliza o ambiente MPI
MPI_Recv	Recebe uma mensagem de outro nó da rede
MPI_Send	Envia uma mensagem a outro nó da rede
MPI_Bcast	Envia uma mensagem de um processador raiz para todos os outros processadores em um grupo
MPI_Reduce	Combina os dados fornecidos por cada processador de um grupo em um dado único
MPI_Pack	Empacota dados em uma região contígua de memória.
MPI_Unpack	Desempacota dados contidos em uma região de memória contígua de acordo com o tipo de dado fornecido
MPI_Pack_size	Retorna a quantidade de bytes necessários para empacotar um determinado dado
MPI_Op_create	Cria uma função de combinação definida pelo usuário.

3. Estudo de caso

A aplicação escolhida para os experimentos é a simulação de reservatórios de petróleo, que consiste de um modelo matemático cujo objetivo é gerar previsões precisas do comportamento dos fluidos (água, gás e óleo) em um reservatório de petróleo (meio poroso) durante um determinado espaço de tempo e sob diferentes condições de operação [Ertekin, Abou-Kassen e King 2001]. Esse modelo tem como base a teoria de fluxo em meios porosos [Soares 2002]

Este trabalho utiliza o modelo de reservatório black-oil, um dos modelos mais comum entre simuladores comerciais de reservatórios [Fachi, Harpole e Bujnowski 1982], [CMG 1995], [Schlumberger 2009]. Este modelo leva em consideração três componentes e três fases: óleo, água e gás. As equações de fluxo para o modelo black-oil são obtidas através das equações de conservação de massa, equações de estado e a lei de Darcy, que descreve o fluxo de fluidos através de um meio poroso.

A precisão da simulação de reservatório é diretamente proporcional a quantidade de características do reservatório que foram utilizadas para obter o modelo matemático. No entanto, o uso de um grande número de características também aumenta o custo computacional da simulação. Assim é importante encontrar um equilíbrio entre precisão e o custo computacional.

As complexas equações resultantes do modelo matemático gerado não podem ser resolvidas analiticamente, sendo necessária a utilização de técnicas de discretização numérica para obter uma solução aproximada. No processo de discretização o intervalo de tempo é divido em subintervalos de valores fixos e a geometria do reservatório é, semelhantemente, divida em uma malha de blocos de tamanhos iguais. As técnicas de discretização do espaço e tempo geram, para cada bloco da malha, um sistema de equações lineares que precisa ser resolvido para cada subintervalo de tempo. De acordo com [Silva et al 2003], cerca de 80% do tempo total de execução de um simulador de reservatório é gasto na etapa de solução do sistema de equações lineares. Por isso, é de fundamental importância empregar métodos de solução eficientes, capazes de resolver o problema rapidamente, utilizando pouca memória, sem perder a precisão na solução.

O método de resolução de sistema de equações lineares utilizado em nossos experimentos é o succesive over-relaxation (SOR). Este método foi considerado por ter uma rápida convergência e ser adotado por simuladores comerciais de reservatório largamente utilizados, como o Black Oil Applied Simulation Tool [Fachi, Harpole e Bujnowski 1982].

As Equações 1 e 2 descrevem o método SOR:

$$x_i^{k+1} = \frac{1}{a_{i,i}}\left(b_i - \sum_{j<i}^{n} a_{i,j}x_j^{(k+1)} - \sum_{j>i}^{n} a_{i,j}x_j^{(k)}\right) \qquad i = 1, 2, \dots, n \qquad \textbf{(1)}$$

$$x_i^{k+1} = \omega x_i^{k+1} + (1-\omega)x_i^k \qquad \textbf{(2)}$$

Onde:

- a é um elemento da matriz de coeficientes A ($A\bar{x} = \bar{b}$);
- $\bar{x}$ e $\bar{b}$ são elementos dos vetores do sistema de equações lineares;
- x_i^{k+1} é o componente da iteração $k+1$, encontrado em função dos demais elementos da iteração anterior, k;
- ω é o fator de sobre-relaxação, o uso do fator adequado acelera a convergência do método, reduzindo o número de iterações necessárias para resolver o sistema.

A cada iteração, as Equações 1 e 2 são usadas para encontrar o novo valor de cada uma das n variáveis do sistema. Assim o algoritmo pode ser paralelizado de maneira que cada processador p seja responsável por encontrar, a cada iteração, o novo valor de n/p variáveis do sistema.

4. Resultados

Os resultados apresentados aqui correspondem a *n* processadores executando um algoritmo de simulação de reservatório em instancias da STORM configuradas com os modelos de memória distribuída ou compartilhada.

No modelo de memória compartilhada os processadores se comunicam através de mutexes: cada vez que processador quer se comunicar com outro processador, ele bloqueia até que o mutex pertencente ao destino seja liberado. Por outro lado, cada vez que um processador termina suas operações de processamento ele libera seu mutex, permitindo que os outros processadores se comuniquem com ele.

No modelo de memória distribuída, os processadores se comunicam exclusivamente por troca de mensagens, assim, cada vez que um processador precisa de informações de outros processadores, uma comunicação é estabelecida. Para isso, uma biblioteca MPI com funções bloqueantes é utilizada.

A Figura 2.a mostra os resultados de carga injetada na NoC para cada modelo de memória. Como esperado, o modelo de memória compartilhada produz muito mais tráfego na NoC quando comparada ao modelo de memória distribuída, já que todos os dados, inclusive os não compartilhados, são armazenados em módulos de memória distribuídos pela NoC.

Na Figura 2.b, a carga injetada para os dois modelos de memória pode ser observada em diferentes escalas. A escala do lado esquerdo da figura (Megabytes) corresponde ao modelo de memória compartilhada, enquanto que a escala do lado direito da figura (Kilobytes) corresponde ao modelo de memória distribuída. Nessa figura fica claro que, apesar de ser baixa, a carga gerada pelo modelo de memória distribuída se comporta de forma semelhante à carga gerada pelo modelo de memória compartilhada.

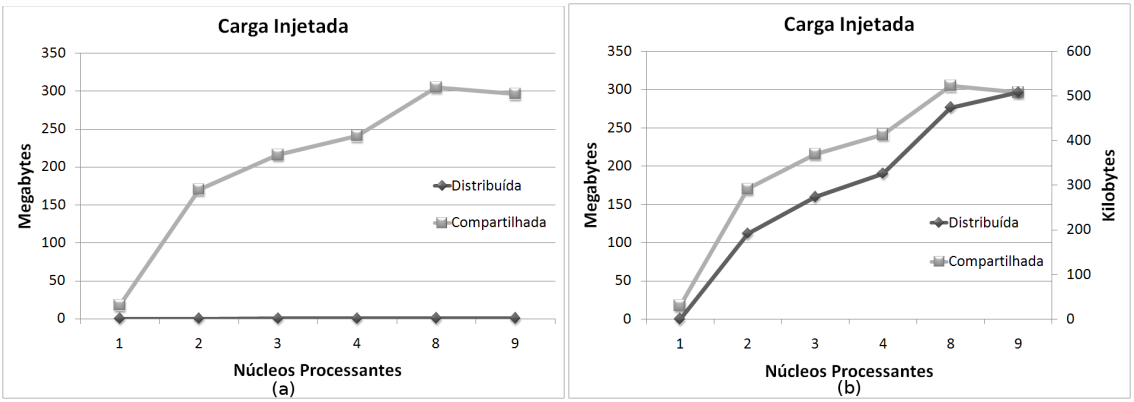


Figura 2. Carga injetada na NoC: (a) mesma escala, (b) escalas diferentes.

A Figura 3.a mostra a carga injetada apenas para o modelo de memória compartilhada. Essa carga está dividida em carga gerada por dados da aplicação e carga gerada para manter a coerência das caches do sistema. Como pode ser observado, a carga gerada por dados da aplicação supera em muito a carga gerada para manter a coerência das caches. Como mostrado em [Girão et al 2007], este resultado pode ser explicado pelo fato dos pacotes de coerência enviarem apenas pacotes de controle que são bem menores que os pacotes de dados transferido pela aplicação. Dessa forma, a linha que representa os dados injetados na NoC pela aplicação na Figura 3.b tem forma semelhante a linha da carga total (dados e coerência) injetada pelo modelo de memória compartilhada na Figura 2.a.

Na figura 3.b, o número de ciclos por instruções (CPI) é comparado para os dois modelos de memória. Como pode ser visto, o número de ciclos por instrução é consideravelmente menor no modelo de memória distribuída. Esse comportamento pode ser explicado pela concorrência no acesso aos dados que é característico do modelo de memória compartilhada.

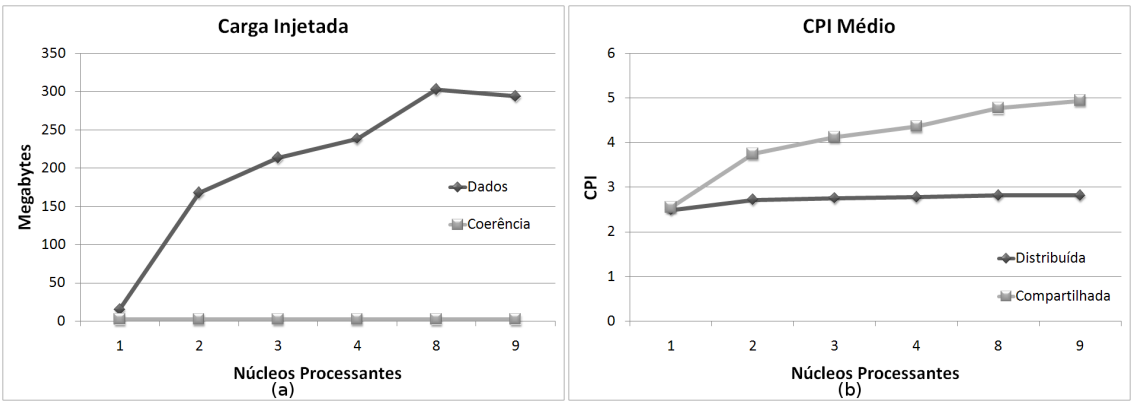


Figura 3. Carga Injetada (a) e CPI Médio (b)

A quantidade de ciclos que cada instância simulada levou para executar a aplicação é exibida na Figura 4.a. Mesmo os resultados de número de ciclos por instrução e carga injetada exibidos anteriormente sendo favoráveis ao modelo de memória distribuída, podemos notar que a quantidade de ciclos é semelhante em ambos os modelos de memória simulados. Isso pode ser creditado ao overhead causado pela execução da biblioteca de troca de mensagens (MPI), pois, como podemos observar na Figura 4.b, a quantidade média de instruções executadas por processador é maior no modelo de memória distribuída.

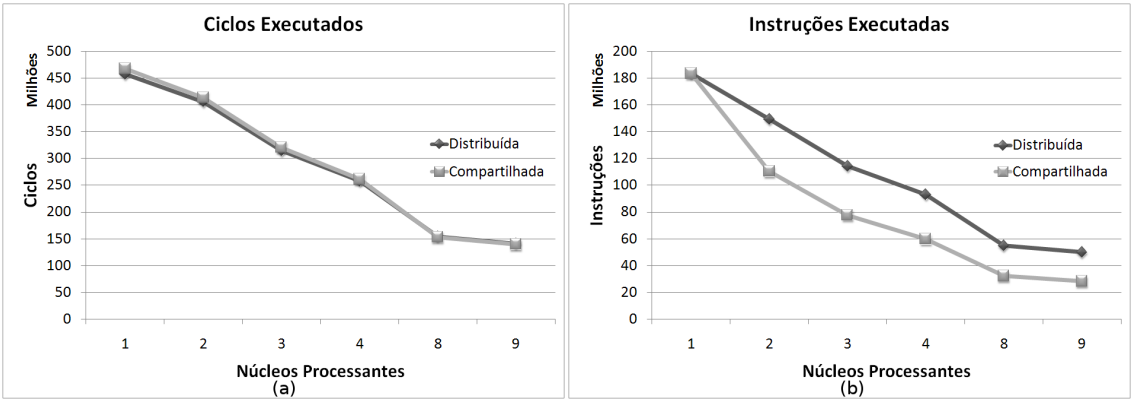


Figura 4. (a) Ciclos e (b) instruções executadas.

Os resultados de latência média dos pacotes são exibidos na Figura 5. Esses valores foram obtidos calculando a média da latência dos pacotes que chegam a cada roteador, e em seguida calculando a média desses valores. Podemos ver no gráfico que a latência média dos pacotes no modelo de memória distribuída é aproximadamente três vezes maior que a latência média do modelo de memória compartilhada. Esse fato ocorre por uma combinação de dois fatores: o chaveamento store-and-forward utilizado pela NoC, e o tamanho dos pacotes de NoC

No modelo de memória compartilhada, o tráfego é totalmente gerado pelos pedidos de dados das caches. Dessa forma, o tamanho médio do pacote de NoC é proporcional ao tamanho do bloco de cache. Já no modelo de memória distribuída, o tamanho do pacote depende do tamanho das mensagens enviadas pela aplicação. Como na aplicação simulada é necessária a troca de vetores entre os processadores do sistema, os pacotes de NoC tendem a ser maiores no modelo de memória distribuída. No

chaveamento store-and-forward os pacotes devem ser armazenados inteiramente nos roteadores intermediários. Assim, a latência tende a aumentar quando o tamanho do pacote aumenta.

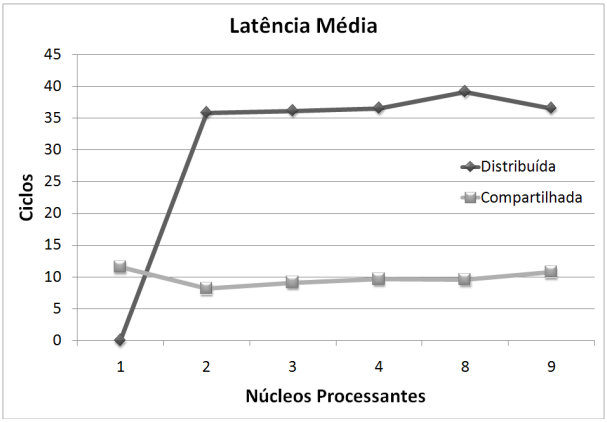


Figura 5. Latência média dos pacotes da NoC.

5. Conclusões e Trabalhos Futuros

Este artigo apresentou o projeto de uma plataforma MPSoC virtual que propõe a utilização de um conjunto de requisitos funcionais, módulos de hardware e bibliotecas de software. Tal projeto provê flexibilidade quando novas instâncias arquiteturais precisam ser desenvolvidas. Diferentemente de muitos artigos sobre MPSoC na literatura, este apresentou uma aproximação de propósito geral. A aplicação utilizada como estudo de caso foi uma simulação de reservatório computacionalmente intensiva. Esta aplicação foi simulada utilizando um sistema com 200 equações lineares

A aplicação foi executada com os modelos de memória compartilhada e distribuída disponíveis na plataforma STORM. As simulações foram realizadas considerando a carga injetada na NoC, o CPI médio, o número de ciclos e instruções executadas e a latência dos pacotes de comunicação da NoC.

Os resultados mostraram que o modelo de memória distribuída apresenta melhores resultados quando a carga injetada e o CPI médio são considerados. Por outro lado, o modelo de memória compartilhada foi melhor quanto a latência dos pacotes na NoC. Esse último resultado pode ser explicado pelo tamanho dos pacotes injetados na NoC pela aplicação. Enquanto no modelo de memória compartilhada os pacotes são do tamanho aproximado do tamanho dos blocos da cache, no modelo de memória distribuída os pacotes são proporcionais ao tamanho das mensagens enviadas pela aplicação. Esses resultados mostraram a relevância de considerar a organização da memória durante a concepção de um MPSoC, uma vez que ela impacta na comunicação (latência) e processamento (CPI, número de instruções executadas).

Como trabalhos futuros nós planejamos testar outras aplicações e observar o impacto em outras caracteristicas, como consumo de energia, quando arquiteturas de processamento e comunicação são combinadas com diferentes organizações de memórias para criar MPSoCs.

Referências

Benini, L. et al, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC". The Journal of VLSI Signal Processing, 41(2): 169 – 182, September 2005.

CMG Ltd. "IMEX Reference Manual", version 96.00, [S.l.: s.n.], 1995.

Ertekin, T., Abou-Kassem, J. H. e King, G. R. "Basic applied reservoir simulation". SPE Textbook Series Vol. 7. Richardson: SPE, 2001.

Fanchi, J. R., Harpole, K. J. e Bujnowski, S. W. "BOAST: A three-dimensional, three-phase black oil applied simulation tool". Oklahoma: [s.n.], 1982.

Forsell, M., "A scalable high-performance computing solution for network on chip". IEEE Micro, vol. 22, no. 5, pp. 46–55, 2002.

Fummi, F. et al, "Native ISS-SystemC Integration for Co-Simulation of Multi-Processor SoC". Design, Automation and Test in Europe (DATE), Proceedings, 2004

Girão, G. et al. "Cache Coherency Communication Cost in a NoC-based MPSoC Platform". In: Symposium on Integrated Circuits and Systems Design, 20, 2007, Rio de Janeiro. Proceedings... New York : Association for Computing Machinery, 2007. v. 1. p. 288-293.

Petrot, F., Greiner, A. e Gomez, P., "On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory Multiprocessor SoC Architectures". In: Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on 30-01 Aug. 2006 Page(s):53 – 60.

Schlumblerger Ltd. "ECLIPSE Reservoir Engineering Software". Disponível em: http://www.slb.com. Abril.

Silva, F. A. et al. "Parallelizing Black Oil Reservoir Simulation Systems for SMP Machines". In: Annual Symposium on Simulation, 36, 2003, Proceedings... Washington: IEEE Computer Society, 2003, p. 224 - 230.

Soares, A. A. M. "Simulação de reservatórios de petróleo em arquiteturas paralelas com memória distribuída". 2002. Dissertação (Mestrado em Ciências e Engenharia Civil) – Programa de Pós-Graduação em Engenharia Civil. UFPE, Recife, 2002.

Suh, T., Blough, D. M. e Lee H. H. S., "Supporting cache coherence in heterogeneous multiprocessor systems". Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings Volume 2, 16-20 Feb. 2004 pp.1150 - 1155 Vol.2

ARP: Um Gerenciador de Pacotes para Sistemas Embarcados com Processadores Modelados em ArchC

Rodolfo Azevedo, Bruno Albertini, Sandro Rigo

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 6172 – 13083-852 – Campinas – SP – Brazil

{rodolfo,balbertini,sandro}@ic.unicamp.br

Resumo. *Este artigo apresenta a ferramenta ARP, criada para facilitar o gerenciamento de projetos de sistemas embarcados. A ARP permite uma melhor organização dos componentes do sistema, isolando-os de forma a facilitar o reuso e a exploração de espaço de projeto dos sistemas. Na versão atual, ela foi desenvolvida para utilizar a linguagem SystemC. O artigo apresenta dois exemplos dos casos de uso principais: um com finalidade didática e outro focado em pesquisa de memórias transacionais. A ARP está disponível para download através da página da internet do ArchC.*

1. Introdução

A análise de espaço de projeto dos sistemas embarcados requer a criação de múltiplas versões do mesmo sistema, que são diferentes entre si desde pequenos parâmetros até grandes componentes. Assim, o gerenciamento das múltiplas versões desta plataforma, seus parâmetros e componentes é uma tarefa essencial no desenvolvimento dos novos sistemas embarcados.

Este trabalho apresenta as linhas gerais da ferramenta ARP, algumas de suas plataformas de exemplo e casos de uso. O foco principal da ferramenta foi a facilidade de uso, com suporte para o gerenciamento, simultâneo, de diversas configurações do sistema. Desta forma, a ARP facilita a exploração do espaço de projeto, bem como a validação de parâmetros como escalabilidade do número de componentes do sistema.

Do ponto de vista da ARP, um projeto é denominado plataforma, que possui diversos componentes. A versão atual da ARP está desenvolvida em SystemC, embora os conceitos possam ser utilizados em outras linguagens de descrição de sistemas se necessário.

Este artigo está organizado da seguinte forma: A Seção 2 apresenta os trabalhos relacionados à ARP, seguida pela Seção 3 que apresenta uma visão geral da ferramenta. A Seção 4 descreve as funcionalidades fornecidas pela ARP, seguida pelas as Seções 5 e 6 que descrevem dois casos de uso da ARP. Por fim, a Seção 7 conclui o artigo.

2. Trabalhos Relacionados

O gerenciamento de plataformas virtuais tem obtido destaque em diversas ferramentas comerciais da atualidade, sejam elas produtos de fabricantes de EDA como as aquisições da Virtio e CoWare pela Synopsys, que passou a oferecer os produtos Innovator, Platform Architect, CoMet e METeor [Synopsys 2010]. A Altera, na linha de FPGAs, oferece o

SOPC Builder integrado ao Quartus II [Altera 2010]. No outro extremo, a GreenSocs [GreenSocs 2010] oferece um conjunto de ferramentas para gerenciar plataformas baseadas no modelo de distribuição do Debian Linux.

As duas primeiras (Synopsys e Altera) oferecem ferramentas com interfaces gráficas, com suporte a diversos componentes proprietários, que podem ser acoplados aos componentes desenvolvidos pelos usuários. Uma particularidade da Synopsys é o suporte ao padrão TLM 2.0, que permite maior integração entre os componentes. No caso da Altera, a grande facilidade de integração com a ferramenta de sintese para FPGA é o destaque.

Já a GreenSocs, por ter foco na oferta de serviços de projeto, fornece uma infraestrutura mais focada na distribuição de componentes e na integração entre eles.

Um ponto fraco dos projetos acima é a flexibilidade de gerenciamento dos componentes da plataforma. As versões comerciais amarram os componentes às suas próprias plataformas enquanto que a GreenSocs foca todo o seu trabalho ao redor do GreenBus, para fornecer a interconexão.

A ARP tem por objetivo apenas gerenciar as plataformas e seus componentes, dando flexibilidade ao projetista de escolher que componentes utilizar, bem como a forma de utilizá-los.

3. Visão Geral da Ferramenta

Após instalar a configuração básica da ARP, o usuário terá uma estrutura de diretórios com os componentes indicados abaixo:

processors: Diretório que contém modelos de processadores. Embora este artigo foque nos processadores descritos em ArchC, esta não precisa ser a única fonte de processadores para a plataforma.

is: Diretório que contém os elementos de interconexão disponíveis para construção de plataformas, como barramentos, roteadores, NoCs, etc.

ip: Diretório que contém os IPs disponíveis no momento. Cada novo IP, colocado neste diretório, poderá ser utilizado por qualquer plataforma disponível.

sw: Diretório que contém os softwares que serão executados nas plataformas. Em linhas gerais, um mesmo software pode ser utilizado em mais de uma plataforma. No entanto, como é comum a mudança de parâmetros na plataforma, o software terá que ser programado adequadamente para satisfazer a esta variedade de configurações.

wrappers: Diretório que contém os conversores de protocolos que podem ser utilizados para conectar componentes distintos das plataformas.

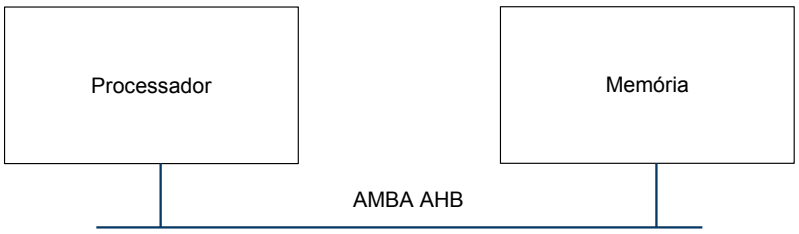
platforms: Diretório que contém as plataformas desenvolvidas. Cada versão da plataforma pode/deve ter um diretório diferente, facilitando a exploração do projeto e também o gerenciamento das configurações.

Junto com estes diretórios também há o utilitário `arp.py`[1], cujas funcionalidades serão descritas adiante e um `Makefile` que fornece os comandos de compilação, execução e gerência necessários.

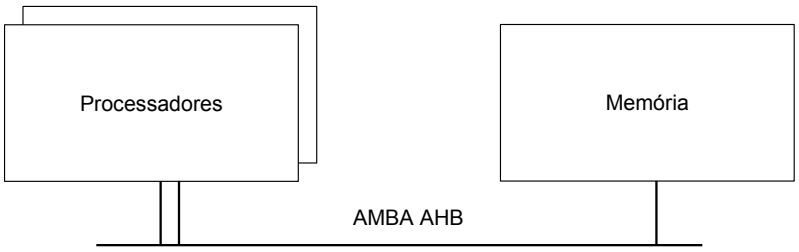
[1] A opção por manter o sufixo `.py` se deve ao conflito de nomes entre o gerenciador de plataformas e o software de rede de mesmo nome.

Embora, num primeiro momento, possa parecer uma organização não usual para uma plataforma, visto que os componentes ficam espalhados em diretórios distintos, compartilhados com outros componentes de outras plataformas, esta abordagem permite a uniformização dos componentes utilizados em mais de uma plataforma, gerando um ganho de organização ao não forçar o desenvolvedor a manter diversas cópias dos mesmos arquivos. Além disto, a ferramenta ARP possui um utilitário com funcionalidades de empacotar e desempacotar uma plataforma, permitindo a montagem de um arquivo com todos os componentes necessários para a execução da plataforma, transferência entre computadores e sua posterior restauração.

Para melhor exemplificar esta organização, suponha uma plataforma simples, como ilustrada na Figura 1(a), com um processador, um barramento e uma memória. Para armazenar esta plataforma, o diretório `processors` conteria um diretório com o código do processador (por exemplo: `sparc`), o diretório `is` conteria um diretório com o código do barramento (por exemplo: `amba_ahb`) e o diretório `ip` conteria um diretório com o código da memória. Como estes três componentes podem ser reutilizados por diversas plataformas, deverá ser criada um novo diretório, dentro de `platforms` com a instanciação dos componentes desejados, neste exemplo, este diretório poderia se chamar `simple_platform` e conteria o código em SystemC para instanciar os componentes necessários. Além dos diretórios mencionados, o software que será executado na plataforma deve ser desenvolvido e incluído dentro do diretório `sw`.



(a) Plataforma Simples



(b) Plataforma *Dual Core*

Figura 1. Duas plataformas de exemplo

Se for necessário, em algum momento posterior, criar uma versão com dois cores desta plataforma, como ilustrado na Figura 1(b), basta criar um novo diretório dentro de `platforms`, que conterá o código SystemC que instancia os dois processadores. Neste exemplo simples, como o código SystemC será muito semelhante ao da

simple_platform, basta copiar o arquivo principal do diretório anterior e realizar as modificações necessárias. Não é preciso replicar nenhum código dos componentes caso eles já possuam o suporte necessário a esta plataforma.

4. Facilidades Fornecidas pela ARP

As duas primeiras funcionalidades a destacar da ferramenta arp.py estão relacionadas com o armazenamento das plataformas. O parâmetro --pack empacota uma plataforma, gravando em um único lugar todos os arquivos utilizados durante o desenvolvimento e execução da mesma. O nome da plataforma deve ser fornecido como parâmetro. O arquivo gerado pode ser enviado para outro computador que, através das mesmas ferramentas, será capaz de reconstruir toda a árvore de diretórios e reexecutar a plataforma. Para desempacotar a plataforma, basta utilizar o parâmetro --unpack, fornecendo o arquivo criado anteriormente.

Além da facilidade de empacotar e desempacotar plataformas, que permite a transferência fácil dos componentes de um computador para outro, a organização dos componentes na ARP permite um melhor gerenciamento do processo de compilação: cada diretório de cada componente tem um arquivo Makefile, que contém instruções de compilação específicas com a finalidade de criar uma biblioteca com este componente. O diretório da plataforma contém um Makefile que compila todos os seus arquivos e realiza sua ligação com as bibliotecas dos componentes necessários. O diretório principal da ARP contém também um Makefile que executa os comandos de compilação apenas nos componentes necessários da plataforma, utilizando o mesmo conceito das bibliotecas já mencionado. Apenas o Makefile principal e o da plataforma precisam ser alterados quando da construção de uma nova plataforma. A listagem dos componentes que a plataforma requer é feita num arquivo chamado defs.arp, que fica dentro do diretorio da plataforma. Este arquivo é incluído no Makefile principal, ajudando no gerenciamento do processo de compilação. A listagem da Figura 2 mostra um arquivo defs.arp de exemplo indicando os componentes necessários para executar um programa "Hello World" numa plataforma simples como a da Figura 1(a).

```
1 IP := ac_tlm_mem
2 IS := ac_tlm_bus
3 PROCESSOR := mips1
4 SW := hello_world
5 WRAPPER :=
```

Figura 2. Exemplo do arquivo defs.arp

A primeira linha do arquivo de exemplo contém o nome do IP que implementa uma memória (ac_tlm_mem). O dispositivo de interconexão, neste caso, é um barramento genérico (ac_tlm_bus), o processador é um MIPS (mips1) descrito através da ADL ArchC e o software é o hello_world. Caso mais de um componente da mesma categoria seja necessário, basta colocar todos na mesma linha, com os nomes separados por espaços.

Uma vez que um componente possui, em seu Makefile, todos os comandos necessários para sua compilação e uso pelas plataformas, sua reutilização tende a aumentar.

Exemplos destes `Makefile` são fornecidos junto com o pacote ARP. Os comandos que devem ser suportados são:

all: Este é o padrão para todo `Makefile` e significa compilar todo o componente.
clean: Apaga os arquivos gerados pela compilação do componente. Todos estes arquivos podem ser regerados ao executar um novo `make all`.
distclean: Apaga os arquivos gerados por ferramentas auxiliares, como ArchC. Como exemplo, todos os arquivos C++ gerados pelo gerador de simuladores (`acsim`) são apagados. Eles também podem ser reconstruídos através do comando `make all`.
lib: Compila o componente atual e armazena numa biblioteca. Assim fica mais fácil reutilizá-lo. O próprio `Makefile` principal chama esta regra para os diretórios de componentes.
run: Executa o programa na plataforma atual. Para isto, se necessário, o programa sera compilado e a plataforma será executada recebendo o programa como parâmetro.

A vantagem de se criar bibliotecas com cada um dos componentes é a facilidade de passá-las como parâmetros para as plataformas através das opções `-l` e `-L` do `gcc`, como é feito por padrão pelo `Makefile` principal.

Outro aspecto importante para facilitar a interconexão dos componentes é a utilização de um padrão de comunicação como o TLM [Ghenassia 2005, Black et al. 2009] da OSCI [OSCI 2010]. Embora o suporte ou não à TLM não seja diretamente ligado às funcionalidades da ARP, todos os componentes distribuídos adotam este modelo de interface.

As próximas duas seções abordam dois casos de uso da ARP: uma plataforma multicore com finalidade educacional e outra plataforma, mais elaborada, com finalidade de pesquisa na área de memórias transacionais. Ambas as plataformas são distribuídas sob demanda.

5. Plataforma multicore básica e seu uso educacional

Esta seção ilustra como transformar a plataforma da Figura 1(a) numa plataforma multicore básica, como a versão dual-core da Figura 1(b) e como esta atividade está, atualmente, sento utilizada para fins educacionais. Será fornecida uma descrição mais detalhada da implementação do componente de gerenciamento de concorrência, por ser de grande reuso e depois serão mostradas as atividades que são fornecidas aos alunos na montagem da plataforma multicore.

5.1. Primitivas para controle de concorrência

Um dos primeiros requisitos importantes de uma plataforma multicore é o fornecimento de mecanismos para controle de concorrência. Os mais simples e efetivos são instruções atômicas como *test-and-set*, *load-and-increment* e *compare-and-swap*. Estas instruções precisam ser implementadas nos processadores mas, devido às restrições de atomicidade, precisam também de um suporte da plataforma. Uma outra alternativa, com boa efetividade, é trabalhar com um periférico (IP) que implemente um mecanismo atômico em hardware, como no caso da plataforma MPARM [Loghi et al. 2004]. O MPARM implementa um número restrito de *locks*, que podem ser utilizados na implementação de

uma biblioteca de controle de concorrência geral. Devido aos requisitos da plataforma alvo, serão utilizados como exemplo, sem perda de generalidade, os componentes básicos (processador e protocolo de interconexão) já disponíveis na ARP.

Não seria uma boa prática de projeto escolher uma das instruções atômicas mencionadas e implementá-la diretamente num processador sem antes verificar o suporte do seu conjunto de instruções. Como as três instruções mencionadas são equivalentes[2], apenas a instrução *test-and-set* será descrita a seguir.

A instrução *test-and-set* possui dois parâmetros: valor a comparar e endereço de memória. Uma versão primitiva dela pode operar, atomicamente, da seguinte forma: primeiro lê o valor armazenado no endereço. Depois compara este valor com o fornecido pela instrução, se eles forem diferentes, grava o novo valor na memória e retorna o valor antigo. Se os valores forem iguais, retorna o valor sem afetar a memória. Com base no valor de retorno é possível saber se a instrução foi bem sucedida ou não e implementar um *mutex*. O grande requisito, neste momento, é o suporte a operações atômicas no barramento e, potencialmente, os demais componentes entre o processador e a memória.

A implementação desta operação atômica pode ser feita através da implementação TLM disponível nos modelos de processador gerados por ArchC, que suportam a operação *lock* para bloquear um endereço dos componentes onde os processadores estão conectados. Assim, é necessario adquirir um *lock* através da interface TLM e, depois, efetuar as operações descritas anteriormente, liberando o *lock* a seguir.

A segunda alternativa, implementação de um periférico para gerenciar um *lock* em hardware. Uma especificação possivel para um único *lock* é a seguinte: o periférico contém um registrador de *lock*, inicialmente com valor zero. Toda leitura retorna o valor disponível no registrador de *lock*, alterando seu valor para 1. Toda escrita grava o valor fornecido no registrador. A Figura 3 ilustra os passos necessários para utilizar efetivamente o *lock* em hardware. O primeiro passo é a declaração de um ponteiro para a posição de memória onde o *lock* está localizado. A seguir, o laço aguarda pela liberação do *lock* (no nosso exemplo, o *lock* está liberado quando tiver valor 0). A seguir é executada a região crítica e o *lock* é liberado através da escrita do valor zero nele liberando, potencialmente, outras *threads* concorrentes.

```
1 volatile int *lock = (int *) ENDERECO_LOCK; // Ponteiro para o
    lock em hardware
2
3 while (*lock); // Obtem o lock
4   // trecho do programa a executar
5 *lock = 0; // Libera o lock
```

Figura 3. Exemplo de uso de um lock em hardware

Através desta implementação, é possível criar duas primitivas `AquireGlobalLock` e `ReleaseGlobalLock`, como mostrada na Figura 4, que são fragmentos do código de exemplo da Figura 3.

Estas duas funções são suficientes para gerenciar uma outra posição de memória

[2]A partir de qualquer uma delas, é possível implementar a funcionalidade das outras duas.

```
1  volatile int *lock = (int *) ENDERECO_LOCK;
2
3  void AquireGlobalLock()
4  {
5    while (*lock);
6  }
7  void ReleaseGlobalLock()
8  {
9    *lock = 0;
10   }
```

Figura 4. Implementação das funções de acesso ao *lock* global

de forma similar a um lock, como mostram as implementações da Figura 5. Tanto a função `AquireLock` quanto a `ReleaseLock` primeiro obtém o *lock* global e depois fazem a alteração no *lock* local. Assim, garantidamente somente uma *thread* do programa terá acesso, por vez, às variáveis dos *locks* locais.

```
1  void AquireLock(int *l)
2  {
3    int lockValue = 0;
4
5    while (!lockValue) {
6      AquireGlobalLock();
7        if (! *l)
8          lockValue = *l = 1;
9      ReleaseGlobalLock();
10   }
11 }
12 void ReleaseLock(int *l)
13 {
14   AquireGlobalLock();
15   *l = 0;
16   ReleaseGlobalLock();
17 }
```

Figura 5. Implementação das funções de *lock* local baseadas no *lock* globall

Tanto o componente quanto o software gerados são de grande reuso, fornecdendo um bom mecanismo de controle de concorrência no sistema.

5.2. Atividades seguintes

Geralmente a criação dos componentes de hardware (IP) e software (biblioteca) para controle de concorrência são feitas como atividades iniciais para adaptação ao uso da ARP. A seguir, as atividades de uso de plataformas e montagens de sistemas dedicados tomam forma na disciplina. A seguir são descritas, brevemente, algumas das atividades:

Definição do espaço de endereçamento dos componentes: Dado um conjunto de IPs, incluindo a memória, definir o espaço de endereçamento dos periféricos, configu-

rar o barramento para ativá-los nos momentos corretos e demonstrar o funcionamento através de um software que verifica as funcionalidades implementadas. É neste momento que o IP de gerenciamento de *locks* é inserido na plataforma.

Ampliação do número de processadores: O próximo passo é aumentar o número de processadores da plataforma, tornando-a realmente multicore. Para isto, são tratadas questões de individualização das pilhas, separação de linhas de execução (criação de *threads* sem infraestrutura de sistema operacional, divisão do software em partes paralelas, etc).

Testes de escalabilidade: O aumento no número de processadores permite testar a escalabilidade de todo o software desenvolvido e também avaliar os gargalos de concorrência por recursos globais compartilhados.

Migração de funcionalidades de software para hardware: Aqui o aluno é exposto às primeiras noções de hardware/software *co-design*. Após realizar um *profile* do código em execução, ele deve escolher funções importantes a serem migradas para hardware. Estas funções precisam ser implementadas como periféricos em alto nível de abstração. Tambem cabe ao aluno definir o protocolo de comunicação entre o software restante e o periférico, bem como encapsular as chamadas necessárias para obter a mesma funcionalidade no programa. É comum que os alunos encontrem e resolvam problemas relacionados com a diferença de endian das arquiteturas.

Uma característica importante do gerenciador de plataformas, bastante utilizada nos exercícios acima, é a facilidade de fornecer soluções parciais e também trocar componentes entre diversos computadores/usuários. Desta forma, os alunos podem trabalhar em grupos e garantir que as funcionalidades desenvolvidas por um deles será aproveitada corretamente pelo outro, além de garantir o completo funcionamento das plataformas no computador do docente que irá avaliá-la. Como exemplo, é comum a entrega de uma plataforma básica, como a versão de um único processador, através de um pacote ARP e a solicitar a resposta como um outro pacote ARP, agora com todos os componentes.

6. Plataforma multicore e seu uso na pesquisa de memórias transacionais

Dentre os projetos que contaram com suporte da ARP podemos destacar a criação de uma plataforma de prototipagem e simulação de sistemas de memória transacional com suporte específico em hardware.

A plataforma de prototipagem desenvolvida nesta pesquisa foi inicialmente baseada na proposta original de Herlihy e Moss [Herlihy and Moss 1993] e estendida para dar suporte a transações aninhadas. Neste trabalho, utilizamos modelos de processadores descritos em ArchC, um IP adicional para implementar as *caches* transacional e normal, um barramento para ligar as várias instâncias de processadores e alguns adaptadores.

Uma das tarefas necessárias ao desenvolvimento da plataforma foi a adição de novas instruções aos modelos funcionais das arquiteturas MIPS, SPARC e PowerPC. Este processo é bastante simples uma vez que os modelos são descritos em um nível de abstração bastante alto e revelam a flexibilidade obtida ao empregarmos uma ADL. Também introduzimos novos tipos de mensagem TLM para permitir a comunicação adequada entre processador e o controlador da cache, uma vez que a responsabilidade da implementação do comportamento das novas instruções é compartilhada entre processador e cache. Os modelos de processadores emitem apenas uma instrução a cada ciclo e

as instruções são executadas em sua ordem de emissão. Uma vez tendo os modelos de processadores disponíveis dentro do arcabouço, a tarefa de trocar o processador usado na plataforma se resume a uma simples alteração no arquivo de definições da mesma, como explicado anteriormente.

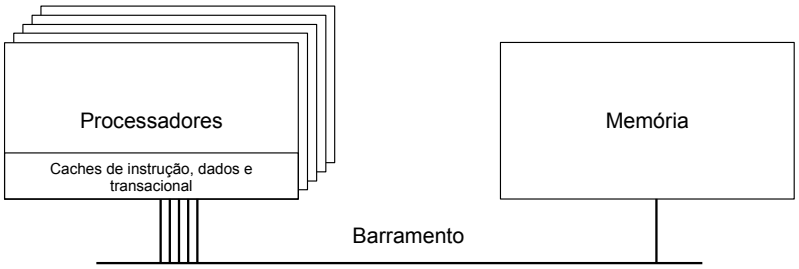


Figura 6. Plataforma de Prototipação de TM

A plataforma de prototipação está ilustrada na Figura 6. Neste ambiente, partimos de uma plataforma dual core e montamos diversas plataformas com números variados de processadores para explorar a escalabilidade do sistema de memória transacional modelado, avaliando de 2 até 256 processadores, com suas respectivas *caches* com suporte a memória transacional em hardware. Como o foco não era no mecanismo de interconexão, colocamos um barramento funcional para interligar os processadores. Essa plataforma pode ser facilmente instrumentada para colher estatísticas relacionadas ao sistema de TM, como número de abortos, confirmações, etc, e foi base do estudo publicado em [Kronbauer et al. 2007].

Partindo deste trabalho, algumas extensões podem ser feitas à plataforma como:

- Avaliação de mecanismos de interconexão como NoCs. O uso de um barramento real certamente degradaria o desempenho do sistema, como mencionado no próprio artigo [Kronbauer et al. 2007].
- Avaliação de protocolos de coerência de *caches*. Variações nos protocolos podem ser de extrema importância em situações de grandes conflitos das transações.
- Avaliação de *tradeoffs* entre memórias transacionais em software e hardware.

7. Conclusões

Gerenciar as diversas versões de um sistema embarcado pode levar a erros de inconsistências entre os componentes, seja de modificações necessárias ou desnecessárias. A ferramenta ARP surge neste contexto para fornecer um repositório organizado em diretórios e arquivos auxiliares para compilação, empacotamento e desempacotamento. Este artigo mostrou os dois principais ramos de uso da ARP atualmente: educacional, dando suporte a atividades de disciplinas de laboratórios e científico, dando suporte a pesquisas avançadas, como na área de memória transacional.

Alguns exemplos iniciais de plataformas, juntos com o pacote de gerenciamento, estão disponíveis no site da linguagem ArchC (http://www.archc.org). Um repositório com mais componentes está em fase de criação e a participação de uma comunidade pode ajudar no seu crescimento.

Como trabalhos futuros nesta direção estão: a criação de uma interface gráfica para gerenciamento dos pacotes, o suporte a padrões de empacotamento como SPIRIT e maior disponibilidade de componentes básicos para iniciar novas plataformas no repositório da ARP.

8. Agradecimentos

Este trabalho foi financiado com recursos da FAPESP, CNPq e CAPEs.

Referências

Altera (2010). Altera design software. online. `http://www.altera.com/products/software/sfw-index.jsp`.

Black, D., Donovan, J., Bunton, B., and Keist, A. (2009). *SystemC from the Ground Up*. Springer.

Ghenassia, F., editor (2005). *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer.

GreenSocs (2010). Green projects overview. online. `http://www.greensocs.com/node/1564`.

Herlihy, M. and Moss, J. E. B. (1993). Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300.

Kronbauer, F., Baldassin, A., Albertini, B., Centoducatte, P., Rigo, S., Araujo, G., and Azevedo, R. (2007). A flexible platform framework for rapid transactional memory systems prototyping and evaluation. In *RSP '07: Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping*, pages 123–129, Washington, DC, USA. IEEE Computer Society.

Loghi, M., Poncino, M., and Benini, L. (2004). Cycle-accurate power analysis for multiprocessor systems-on-a-chip. In *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 410–406, New York, NY, USA. ACM.

OSCI (2010). Open systemc initiative. online. `http://www.systemc.org/home/`.

Synopsys (2010). Synopsys system-level design tools. online. `http://www.synopsys.com/Tools/SLD/Pages/default.aspx`.

Otimizando o Desempenho de Rádios Definidos por Software Através do Desacoplamento de Canais

Roberto de Matos[1,2], Antônio Augusto Fröhlich[2], Leandro Buss Becker[1]

[1]Departamento de Automação e Sistemas – DAS
Universidade Federal de Santa Catarina – UFSC

[2]Laboratório de Integração de Software e Hardware – LISHA
Universidade Federal de Santa Catarina – UFSC

{rmatos,lbecker}@das.ufsc.br, guto@lisha.ufsc.br

Abstract. *The demand for embedded systems with wireless communication has created specific standards to carry out different requirements of range, security and power consumption. This increases diversity and is an obstacle to create a single standard. Software Defined Radios emerged as an alternative to this scenario as it possesses a physical layer which is completely adaptable, however with an overhead much greater than desired. This paper presents an architecture of channels for SDRs based on GNU Radio and USRP2, which demonstrated a significant decrease in the processing cost of the Host, without limiting the flexibility of radio implementations.*

Resumo. *A demanda de sistemas embarcados com comunicação sem fio tem criado padrões específicos para o cumprimento de diferentes requisitos de alcance, segurança e consumo de energia, aumentando a diversidade e impossibilitando a convergência para um único padrão. Os Rádios Definidos por Software surgem como alternativa a esse cenário por possuírem a camada física completamente adaptável, entretanto com o sobrecusto muito maior que o desejado. Este trabalho apresenta uma arquitetura de canais para SDRs baseados no GNU Radio e USRP2, que demonstrou uma diminuição significativa no custo de processamento no Host, sem limitar a flexibilidade das implementações dos rádios, possibilitando assim, uma transição mais suave para o mundo dos sistemas embarcados.*

1. Introdução

Um dos aspectos mais importantes e estudados na área de sistemas embarcados é a forma de comunicação e interação dos vários equipamentos existentes. Atualmente, essa interação tem caminhado cada vez mais para o mundo sem fio, onde é necessário levar em conta o domínio da aplicação, o qual delimita os requisitos com relação a capacidade de dados a serem transmitidos, imunidade a fatores externos, alcance, segurança e consumo de energia. Consequentemente, implicando na criação de diversos protocolos e estruturas de comunicação coexistentes e muitas vezes incompatíveis entre si.

Considerando impossível a convergência de todas as redes sem fio para um único protocolo, a solução tem sido equipamentos capazes de se comunicarem com vários padrões de redes, o que tradicionalmente tem sido alcançado integrando diversos componentes, cada um com compatibilidade para um tipo de rede. Exemplos desses equipamentos

são os novos televisores que se conectam em redes locais sem fio (802.11x) para download de conteúdos especiais da internet e possibilitam a integração com redes pessoais (ex.: Bluetooth) para transmissão de som para fones de ouvidos ou transmissão de dados de câmeras fotográficas e celulares. Ou ainda os gateways residenciais que se comunicam com a rede 3G de telefonia celular e distribuem internet para uma rede local sem fio (ex.: 801.11x).

As desvantagens do modelo tradicional são o aumento do espaço físico e do consumo de energia na adição de cada novo componente, além da inflexibilidade para adição ou adaptação de novos padrões de comunicação sem fio. Assim uma alternativa a esse cenário tem sido o uso de Rádios Definidos por Software (SDR - *Software Defined Radio*) [Mitola 1995], os quais possuem sua camada física completamente reconfigurável, permitindo flexibilidade em vários parâmetros de comunicação como faixa de freqüência, tipo de modulação, protocolo e potência de transmissão [Reed 2002]. Além disso, como os componentes são desenvolvidos em software, eles podem ser reaproveitados em outras plataformas, testados e modificados facilmente [Blossom 2009].

Atualmente, existem várias propostas [Blossom 2009, Ettus 2009, Balister et al. 2007, WARP 2009, KUAR 2009, Ackland et al. 2005] para implementação de SDRs, as quais tipicamente utilizam modelos de alto nível para definição do funcionamento do rádio e o conceito de distribuição de processamento por várias unidades heterogêneas, como FPGAs, DSPs e GPPs (*General Purpose Processors*) nas plataformas de uso específico ou no *host*. As diferenças são percebidas nas regras de criação dos blocos de processamento digital de sinais, conexão desses blocos e a maneira como o sinal é processado pela estrutura que forma o rádio. Um dos projetos mais populares da área é o GNU Radio [Blossom 2009], que juntamente com uma plataforma de baixo custo, chamada de USRP [Ettus 2009] (*Universal Software Radio Peripheral*), possibilita a criação de rádios funcionais a partir de modelos de alto nível utilizando computadores pessoais.

Entretanto, apesar da flexibilidade total da camada física e do provimento de modelos de alto nível para descrição dos rádios, o sobrecusto para implementação de SDRs ainda é muito maior que o desejado para sistemas embarcados. Mesmo computadores pessoais modernos não conseguem processar mais de quatro canais do protocolo IEEE 802.15.4, utilizado em redes de sensores sem fio [Choong 2009]. Assim um dos desafios atualmente é diminuir o custo da execução dos blocos em software utilizando arquiteturas que distribuam de maneira eficiente as tarefas entre as unidades de processamento heterogêneas (FPGA, DSP e GPP) sem a perda de flexibilidade dos blocos de software. Isto possibilitaria, por exemplo, a sua utilização em sistemas embarcados de alto desempenho [Mccarthy et al. 2008] ou de recursos mais limitados [Balister et al. 2007].

Este trabalho apresenta a concepção de uma arquitetura para SDRs que propõem o desacoplamento do conceito de canal da camada física. Isto possibilita a implementação do oneroso estágio de separação de canais (translação de freqüência e decimação/interpolação) de forma mais eficiente em hardware reconfigurável (FPGA), sem perder a flexibilidade de funcionamento do rádio definido por software. O uso da arquitetura proposta mostrou uma melhora significativa no desempenho global do sistema e uma simplificação na interface com a camada física, uma vez que não há necessidade de configurar as variáveis relacionadas com os ajustes do próprio hardware.

O restante do artigo está organizado da seguinte forma: Na próxima seção é apresentado uma visão geral sobre SDR, GNU Radio e a USRP. A seção 3 apresenta a arquitetura de canais proposta. A implementação e os testes comparativos de desempenho são apresentados na Seção 4. Finalmente as conclusões e os trabalhos futuros são apresentados na Seção 5.

2. Software Defined Radio

Rádio Definido por Software (*Software Defined Radio* – SDR) é uma tecnologia emergente que tem sido pesquisada por mais de uma década e vem modificando a forma como os novos rádios de comunicação estão sendo projetados. O termo SDR foi adotado pela primeira fez por Joseph Mitola [Mitola 1995] para designar rádios com a capacidade de funcionamento em várias faixas de freqüência, tipo de modulação, protocolo e potência de transmissão, onde pelo menos 80% das funcionalidades são providas por software, podendo ser reconfiguráveis ou adaptáveis.

O objetivo principal dessa tecnologia é conceber um rádio que virtualmente possa se comunicar com qualquer nova tecnologia apenas atualizando o software, ou seja, o esforço é colocar o software mais próximo da antena, e usá-lo para filtrar, modular, demodular e executar outros estágios da transmissão e recepção. O SDR ideal elimina quase todo hardware, utilizando somente um ADC (*Analog to Digital Converter*) para amostrar o sinal vindo da antena e enviar ao software. No entanto, nem os ADCs mais rápidos são o suficiente para amostrar toda a fatia do espectro de freqüência desejada, sendo necessário mais hardware para converter a faixa interesse para banda base.

O uso de software para implementar rádios garante um ciclo de desenvolvimento muito mais rápido e atualizações em campo com modificações completas das funções, entretanto a extrema flexibilidade e a reconfiguração *on-the-fly* ocasionam um maior consumo de energia se comparados com os ASICs (*Application-Specific Integrated Circuit*), assim, se a flexibilidade da camada física não é importante, este custo seria desnecessário.

Atualmente várias arquiteturas e *frameworks* estão disponíveis para implementar SDRs. Um dos mais utilizados é o GNU Radio, sendo escolhido para desenvolver este trabalho por fazer parte de uma comunidade altamente ativa de software livre, o que garantiu fácil acesso ao *core* para execução das modificações. Além disso, o GNU Radio possui suporte a várias plataformas de hardware, incluindo a USRP (*Universal Software Radio Peripheral*), que é uma placa de baixo custo e serve como interface com o mundo RF. As próximas duas subseções darão uma visão geral sobre o GNU Radio e a USRP, respectivamente.

2.1. GNU Radio

GNU Radio é uma ferramenta de software livre para o desenvolvimento de Software Defined Radios [Blossom 2009]. Foi iniciado com o financiamento de John Gilmore e tem sido desenvolvido desde 2001, com uma reformulação completa em 2004. A facilidade de uso e o alto desempenho para processamento digital de sinais são alcançados pela sua natureza híbrida, que usa C++ para o núcleo dos componentes de processamento de sinais (*Processing Blocks*) e Python para as conexões entre esses componentes na forma de grafos de fluxo (*flowgraphs*) possibilitando a criação de filtros, moduladores, demoduladores e outras estruturas que compõem os rádios.

Sendo executados em plataformas de uso geral, o GNU Radio faz parte de uma vertente iniciada pelo projeto SpectrumWare [Welborn et al. 2009], que desenvolveu uma abordagem que separa temporalmente os fluxos de amostras dos módulos de software, relaxando as restrições temporais sobre os algoritmos de processamento e a sua execução [Tennenhouse and Bose 1996]. O uso em computadores pessoais ordinários permite desenvolvimento facilitado, o que criou uma grande comunidade de desenvolvedores, propiciando um fortíssimo suporte para novas aplicações. Atualmente, o GNU Radio pode ser executado nos sistemas operacionais Linux, Windows e MAC OSX.

O GNU Radio é formado por três principais entidades, são elas: os blocos de processamento (*Processing Blocks*), *Flowgraph* e escalonador. Os blocos de processamento são os componentes que realmente atuam no fluxo de dados e, normalmente, são desenvolvidos em C++ para garantir o desempenho exigido. O *Flowgraph* é responsável por abstrair o fluxo de dados, ou seja, a seqüência dos blocos de processamento de sinais e as conexões entre eles. Finalmente, o Escalonador movimenta os dados (amostras) pelo *flowgraph*. Passando repetidas vezes por cada bloco, ele verifica a relação entre os buffers de entrada e saída para determinar se o bloco deve executar ou não.

2.2. USRP

A USRP (*Universal Software Radio Peripheral*) é uma plataforma aberta, de baixo custo e extremamente flexível desenvolvida sob medida por Matt Ettus [Ettus 2009] para o projeto GNU Radio, o que a fez ganhar rapidamente uma grande comunidade de desenvolvedores em todo o mundo. Sendo basicamente composta por ADCs, DACs, uma FPGA, slots para placas filhas e uma interface de comunicação, a qual conecta o GNU Radio ao mundo RF. O principal objetivo da USRP é permitir aos desenvolvedores a criação de SDRs de baixo orçamento e mínimo esforço inicial. Existem diversas placas filhas, que tem a função de *front-end RF*, as quais são responsáveis por converter a faixa de interesse do espectro de freqüência para uma freqüência intermediária na recepção do sinal e ao contrário para transmissão.

Atualmente, existem duas versões da placa USRP que possuem basicamente as diferenças apresentadas na Tabela 1. O principal problema da USRP1 é a interface de conexão com o PC, o padrão USB2 tem uma taxa máxima teórica de 60MB/s, limitando na prática transferências a 32MB/s. Como cada amostra complexa é composta por quatro bytes, 2 bytes do canal I e 2 bytes do canal Q, a USB2 limita a transferência de oito milhões de amostras complexas por segundo (8MS/s), o que permite uma janela de amostragem de 8MHz. Enquanto o ADC pode amostrar uma janela de aproximadamente 32MHz e o DAC pode gerar sinais dentro de uma janela de 60MHz. Isso caracteriza uma grande limitação na flexibilidade dos SDRs implementados com a USRP.

As melhorias com relação a interface na USRP2 foram alcançadas substituindo a interface USB por uma interface Gigabit Ethernet, com uma capacidade de transferência de 125MB/s, permitindo uma janela de amostragem de 25MHz. Outras melhorias foram o aumento das taxas de amostragens do ADC e do DAC e uma FPGA mais rápida e maior, permitindo a implementação de mais funções no hardware, o que combina a flexibilidade da reconfiguração e a performance requerida por algumas aplicações. Por esse motivo, essa foi a versão escolhida para desenvolvimento do trabalho.

	USRP1	**USRP2**
Interface	USB 2.0	Gigabit Ethernet
FPGA	Altera EP1C12	Xilinx Spartan3 2K
RF Bandwidth to/from host	8MHz @ 16bits	25MHz @ 16bits
Cost	700	1400
ADC Samples	12-bit,64 MS/s	14-bit,100 MS/s
DAC Samples Daughterboard	14-bit, 128 MS/s	16-bit, 400 MS/s
capacity	2 TX, 2 RX	1 TX, 1 RX
SRAM	None	1 Megabyte
Power	6V, 3A	6V, 3A

Tabela 1. Diferenças entre as Versões da placa USRP [Blossom 2009]

3. Arquitetura Proposta

A relação de um para um da interface física (componentes analógicos, ADCs/DACs) para camada física também é comum nas implementações atuais para SDRs, mais por uma questão da herança de implementação dos rádios tradicionais do que por necessidade (Figure 1). Por exemplo, a USRP exporta somente um canal por interface, o qual deve ser configurado para possuir as características esperadas pela implementação da camada física, o que inutiliza o resto do espectro capturado pela interface para outras camada física (Figura 2).

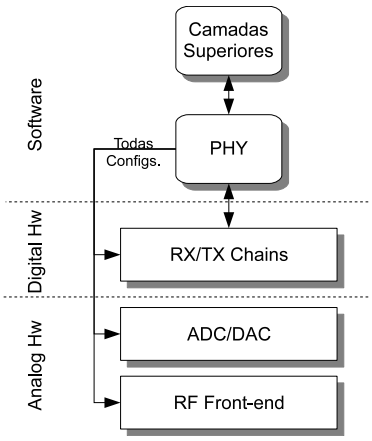


Figura 1. Implementação tradicional das camadas físicas.

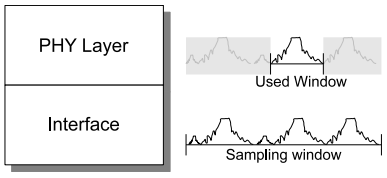


Figura 2. Desperdício do espectro capturado pela interface física.

Outra possibilidade é configurar a USRP para envio de todo o espectro de interesse e fazer a separação de cada canal em software (Figura 4), o que é um processo altamente oneroso para o processamento do *host* e deve ser feito sob-medida para cada conjunto de camadas físicas. Como os parâmetros de configuração do hardware são inferidos a partir das características dos canais, cada mudança exige um esforço grande de reconfiguração que pode variar com o tipo de hardware ou com o novo conjunto de canais alocados.

Com uma análise mais cuidadosa percebe-se que todas as camadas físicas utilizam o conceito de canal, o qual possui apenas dois parâmetros: freqüência central e largura de banda. Padrões para comunicação digital (ex.: 802.11, 802.15.4) constantemente fixam a largura do canal e utilizam identificadores numéricos para referenciá-los, os quais podem ser facilmente traduzidos utilizando a tabela de padronização. Ou seja, o desacoplamento

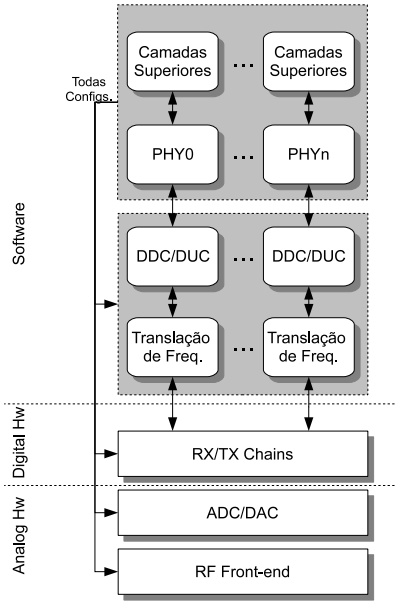


Figura 3. Abordagem em software altamente onerosa para separação de canais.

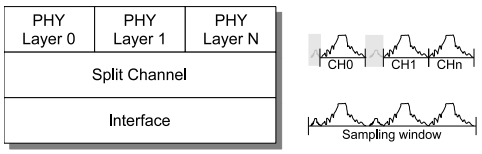


Figura 4. Múltiplas camadas utilizando uma única interface física.

do conceito de canal da implementação da camada física é simples e adiciona flexibilidade no desenvolvimento de aplicações com múltiplas camadas físicas.

O conceito de canal simplifica a interface com a camada física, uma vez que não há necessidade de configurar as variáveis relacionadas com os ajustes do próprio hardware, como é exigido na interface que lida diretamente com o espectro de freqüência. Essas variáveis podem ser inferidas a partir dos canais solicitados pelo usuário, bem como as configurações dos blocos relacionados com a translação de freqüência e decimação/interpolação. Outra vantagem é que a dependência da camada física passa a ser um canal ou um grupo de canais e não mais a interface física completa. Isso permite uma estrutura que extraia vários canais de uma mesma interface e divida-os entre as camadas físicas que estão funcionando em paralelo.

Assim, uma visão geral da arquitetura de canais proposta nesse trabalho é apresentada na Figura 5, onde o principal conceito é uma interface simples e única para alocação de canais independentes, os quais podem ser reconfigurados a qualquer momento no processo de comunicação, respeitando os, aqui classificados, fatores dinâmicos e estáticos.

3.1. Controle

Fatores dinâmicos e estáticos influenciam no projeto e funcionamento da camada física de uma rede sem fio. O fatores dinâmicos estão relacionados com diferentes requisitos das aplicações, ambiente RF e a taxa de consumo de recursos finitos do sistema. Enquanto que os fatores estáticos têm relação com os limites do hardware e regulamentação. No caso da arquitetura proposta, os parâmetros de controle de cada canal são: Freqüência central e largura de banda. A limitação dos recursos do sistema aparece como fator estático e a alocação de canais concorrentes que consomem esses recursos é o fator dinâmico.

O fator estático, recursos do sistema, que limita a configuração da freqüência cen-

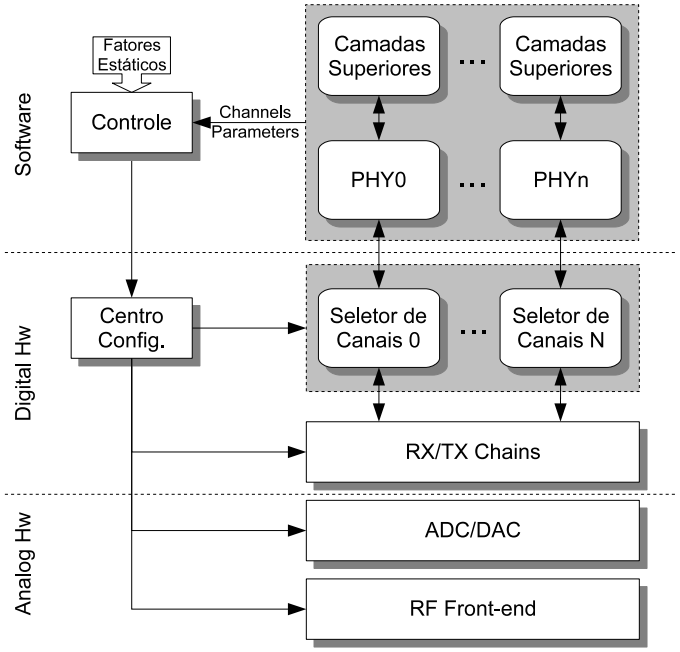


Figura 5. Arquitetura de múltiplos canais.

tral e largura de banda dos canais são:

Capacidade do *RF Front-End*: A função do RF Front-End é deslocar uma janela de interesse para uma freqüência intermediária (IF), onde o sinal é amostrado por ADCs e o inverso para a transmissão. Cada RF Front-End possui uma faixa de freqüência de trabalho com limites máximo e mínimo.

Janela de amostragem: Após o deslocamento para IF uma faixa do espectro é amostrado pelo ADC, esse por sua vez tem limitações ditadas pelo teorema de *Nyquist-Shannon*, onde a freqüência de amostragem tem que ser o dobro da freqüência de interesse. Ou seja, para uma janela de 100MHz o ADC deve trabalhar amostrando a 200MHz.

Interface de comunicação: O hardware transmite os canais selecionados para uma unidade de processamento onde será executado o software que define o comportamento da camada física. Se o processador estiver ligado via interfaces de alta velocidade, esse limite será maior que a freqüência de trabalho do ADC, tendo pouca influência no sistema. Entretanto, não é raro a utilização de interfaces USB ou Giga Ethernet, limitando a transferência de espectros com janelas de 8MHz ou 25MHz, respectivamente.

As equações (1), (2) e (3), apresentam a relação entre a freqüência central (f_{cx}) e a largura de banda (L_{cx}) de um canal qualquer com os limites impostos pela capacidade do *RF Front-End* (f_{FEmin} e f_{FEmax}), pela largura da janela de amostragem do ADC (L_{max}) e pela capacidade da interface de comunicação (LI_{max}). A Tabela 2 resume os parâmetros utilizados nas equações.

$$f_{FEmin} \leq f_{cx} \leq f_{FEmax} \quad (1)$$

$$L_{cx} \leq L_{max} \tag{2}$$

$$L_{cx} \leq LI_{max} \tag{3}$$

Parâmetro	Variável
Freqüência central de um canal *x*	f_{cx}
Largura de banda de um canal *x*	L_{cx}
Limites mínimo e máximo das freqüências de trabalho do *RF Front-End*	f_{FEmin} e f_{FEmax}
Largura máxima da janela de amostragem do ADC	L_{max}
Largura máxima da janela transmitida pela interface de comunicação	LI_{max}

Tabela 2. Características do canal e limitantes do recurso do sistema.

Além dos fatores estáticos existem a concorrência dos recursos pelos canais, ou seja, dinamicamente os canais concorrem por fatias do espectro e a soma dos recursos alocados não podem ultrapassar os limites estabelecidos do sistema. Assim a alocação de todos os canais gera uma janela que possui uma freqüência inferior (B_i) e um freqüência superior (B_s) que tem relação com os canais de menor (f_{cmenor}, L_{cmenor}) e maior freqüência (f_{cmaior}, L_{cmaior}), respectivamente. As equações (4) e (5) apresentam essa relação.

$$B_i = f_{cmenor} - \frac{L_{cmenor}}{2} \tag{4}$$

$$B_s = f_{cmaior} + \frac{L_{cmaior}}{2} \tag{5}$$

Após calcular as freqüências inferior e superior da janela formada pela alocação dos canais, os limites impostos pela capacidade do *RF Front-End* (f_{FEmin} e f_{FEmax}) e pela largura da janela de amostragem do ADC (L_{max}) devem ser verificados. As equações (6), (7) e (8) apresentam essa relação.

$$B_i \geq f_{FEmin} \tag{6}$$

$$B_s \leq f_{FEmax} \tag{7}$$

$$B_s - B_i \leq L_{max} \tag{8}$$

Finalmente, após amostrados, cada canal gera uma quantidade de dados proporcional a sua largura de banda L_c. Assim, a somatória da largura de banda de todos os canais alocados deve ser menor ou igual a capacidade de transmissão da interface de comunicação (equação (9)).

$$\sum_{i=0}^{n} L_{ci} \leq LI_{max} \tag{9}$$

O Algoritmo 1 apresenta o pseudocódigo para uma das possíveis implementações da verificação dos recursos do sistema e alocação de uma lista de canais seguindo as equações descritas acima.

Algorithm 1 Verificação dos recursos do sistema para alocação de uma lista de canais.

```
l_ca = 0
for i := 1 to n_channels do
  if f_FEmin ≤ f_c[i] ≤ f_FEmax then
    l_ca = l_ca + l_c[i]
    v[ ].insere_ordenado(f_c − L_c/2)
    v[ ].insere_ordenado(f_c + L_c/2)
    if (v[ ].max() − v[ ].min() > L_max) or (l_ca > LI_max) then
      error("Impossivel alocar todos os canais")
    end if
  end if
end for
```

4. Implementação e Cenários de Aplicação

A arquitetura proposta foi implementada utilizando a placa USRP2 e o GNU Radio. A USRP2 sofreu modificações no hardware reconfigurável, para adição dos blocos responsáveis pela separação dos canais e implementação das estruturas de suporte. Além disso, a lógica de controle foi modificada para fazer a verificação e roteamento apropriado do fluxo de amostras para cada canal adicionado. O número de canais é limitado pelos recursos da FPGA da plataforma. Já no GNU Radio foi criado um novo par de blocos *Source* e *Sink*, os quais implementam as interfaces de configuração e recepção/transmissão, onde os canais podem ser alocados dinamicamente seguindo as restrições apresentadas na Seção 3.

Para os testes e análise do desempenho foram utilizadas uma implementação do demodulador multi-canal IEEE 802.15.4 da UCLA [Choong 2009] seguindo os componentes tradicionais do GNU Radio e uma implementação equivalente utilizando a arquitetura de canais proposta. As duas implementações foram executadas no mesmo PC com processador Intel QuadCore de 2.83 GHz, com 4GB de RAM e rodando Ubuntu 9.10 com o kernel 2.6.31-19. A placa de rede gigabit utilizada para interface com a USRP2 foi a Broadcom BCM5755 integrada.

O ambiente de experimentações montado é composto por 4 motes MicaZ, uma USRP2 e um *host*, como mostrado na Figura 6. Os motes MicaZ utilizam o rádio CC2420 com a implementação do padrão 802.15.4, cada um transmitindo em um canal, que por limitações da implementação padrão, devem ser vizinhos. Cada mote envia mensagens contendo apenas seu identificador, que servem apenas para *debug*, onde o intervalo e o sincronismo não fazem diferença, pois a ocupação do canal não influência no desempenho

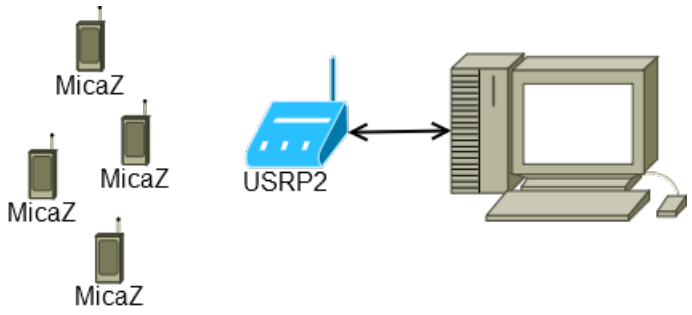


Figura 6. Ambiente de testes montados.

dos testes. Cada implementação foi executada com prioridade "tempo real" e testada durante 300 segundos com dados de performance sendo lidos em intervalos de 2 segundos.

O primeiro teste foi executado utilizando a implementação tradicional, que possui a estrutura mostrada na Figura 7. A predominância dos blocos em software e do paralelismo das funções como "Translação e DDC" em altas taxas de amostras por segundo são a causa principal do baixo desempenho da implementação. Tais resultados são mostrados na Figura 8, que apresenta três medidas principais: *IDLE*, *USR* e *SYS*.

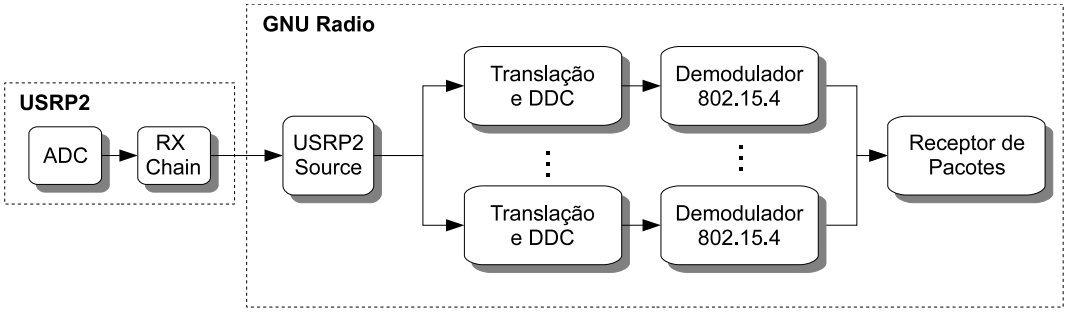


Figura 7. Diagrama da implementação tradicional.

A curva *USR* mostra a porcentagem de utilização das aplicações executados em espaço de usuário. Considerando o mesmo ambiente de teste e a prioridade "tempo real" da tarefa que executa a aplicação do rádio, consegue-se comparar o impacto no sistema das diferentes implementações. A curva *SYS* representa a porcentagem de utilização da *CPU* por aplicações/serviços em espaço de sistema. Finalmente, a curva *IDLE* apresenta a porcentagem da capacidade de processamento livre da *CPU* durante os testes.

A estrutura definida para o segundo teste, desenvolvida sobre a arquitetura de canais proposta, pode ser vista na Figura 9. O deslocamento do processamento dos canais para o hardware reconfigurável permite que a janela do espectro de freqüência capturada pelo *RX chain* seja "fatiada" em canais com freqüências centrais (FC) e larguras distintas. Cada um desses parâmetros pode ser configurado pelo *host*, não diminuindo em nada a flexibilidade alcançada pelos blocos de software utilizados no primeiro teste, conseguindo assim, o mesmo nível de configuração em tempo de execução. Utilizando essa estrutura, cada camada física pode livremente escolher a sua freqüência de trabalho, com a possibilidade do uso de um ou mais canais. As medições de desempenho podem ser vistas na Figura 10.

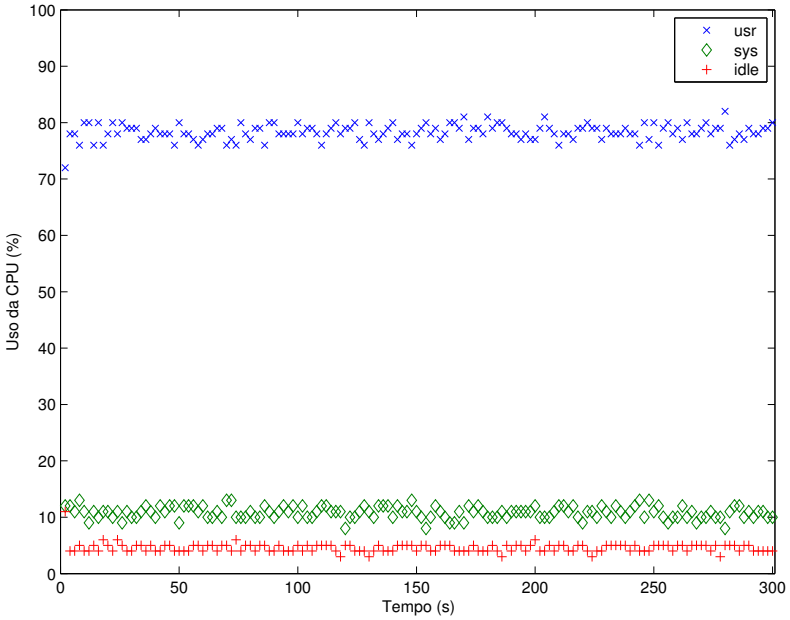


Figura 8. Resultados dos testes de desempenho da implementação tradicional.

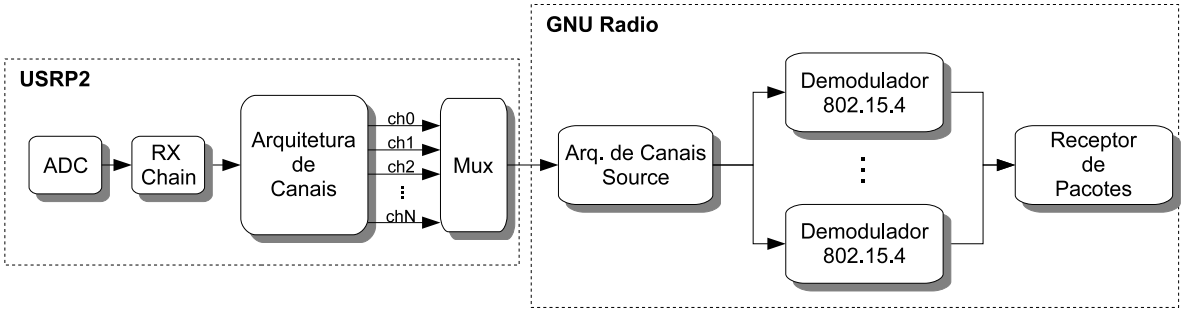


Figura 9. Diagrama da implementação com a arquitetura de canais.

A Figura 11 apresenta as médias de ocupação da CPU. A diminuição de 64,3% de ocupação da CPU quando a arquitetura de canais é utilizada mostra um ganho de performance significativo para execução da mesma tarefa. Além disso, a diminuição do fluxo de dados entre a USRP2 e o *host* traz diversos benefícios, como por exemplo, a diminuição do número de interrupções e consequentemente o *overhead* do sistema para o tratamento das mesmas, o que pode ser visto com a diminuição de 49,3% na ocupação da CPU.

Uma das limitações mais drásticas no modelo original é largura máxima da janela transmitida pela interface de comunicação que interliga a USRP2 e o *host* ($LI_{max} = 25MHz$), sendo um dos fatores estáticos discutidos na Seção 3. Assim, outra vantagem da capacidade do gerenciamento de canais diretamente no hardware é o melhor aproveitamento dos recursos do sistema. Com a possibilidade de separar somente as fatias do espectro que contém informações relevantes nos estágios iniciais, a propagação de dados sem informação útil gera uma economia na largura de banda das interfaces de comunicação.

Por exemplo, o padrão 802.15.4 prevê o uso de 16 canais em uma janela de

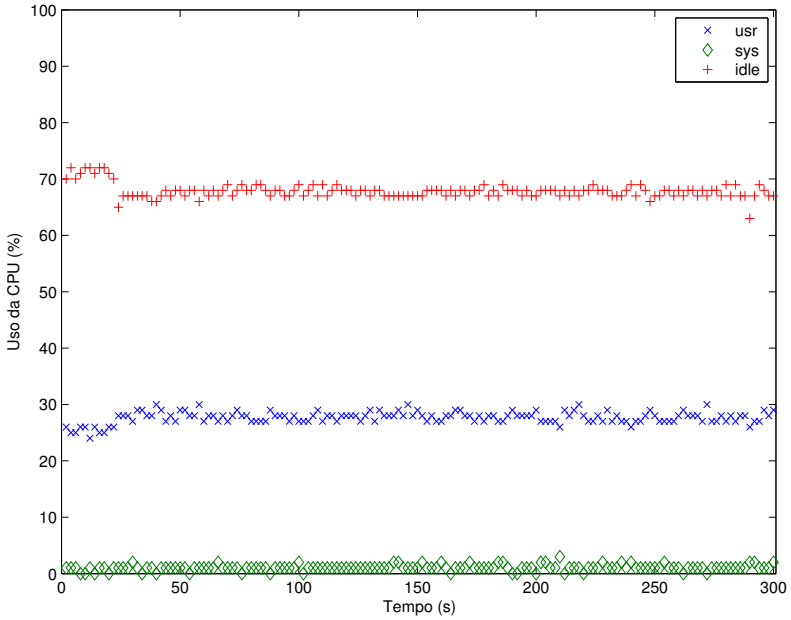


Figura 10. Resultados dos testes de desempenho da implementação com a arquitetura de canais.

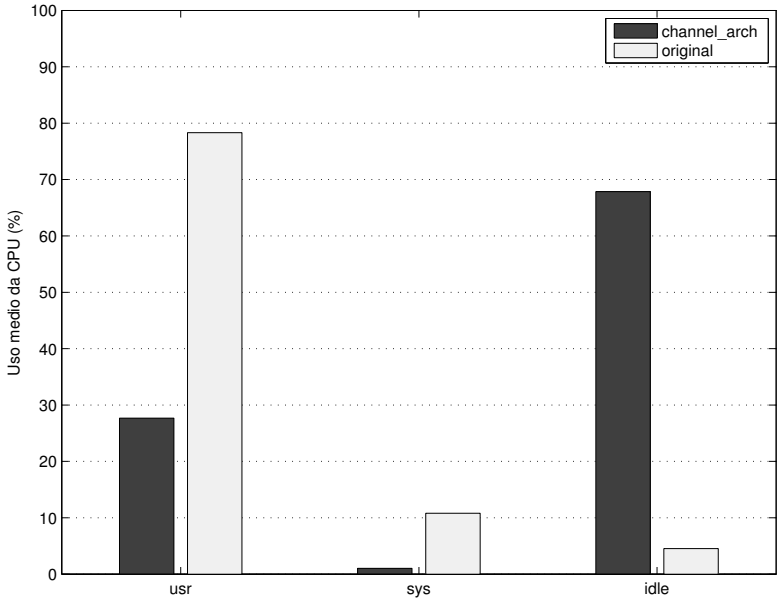


Figura 11. Ocupação média da CPU.

freqüência de 2400MHz até 2483.5MHz. Cada canal tem uma largura de 2MHz e um espaço entre canais de 3MHz. Com a implementação original, uma janela de 25MHz é enviada ao *host* sem processamento de canais prévio, ou seja, as amostras referentes aos espaços do espectro sem informação útil ocupam banda do canal de comunicação. Assim, podemos considerar que cada canal na verdade ocupa 5MHz e pela Equação (9) chegamos a um número máximo de apenas 5 canais e ainda com a limitação de serem consecutivos, como abordado anteriormente. O aproveitamento da janela de 25MHz é de apenas 10MHz de informações úteis (40%). Com a arquitetura proposta, podemos pro-

gramar cada canal separadamente enviando apenas 2MHz de informação por canal para o *host*. Permitindo assim, até 12 canais sem a necessidade de serem consecutivos, com um aumento do aproveitamento da janela de 25MHz de 96%.

5. Conclusões

Esse artigo apresentou uma nova arquitetura para SDRs, que propôs o conceito de desacoplamento de canais da camada física, possibilitando o deslocamento das fases com alto consumo de processamento, devido as altas taxas de amostras por segundo, para o hardware reconfigurável. Isso permitiu uma ganho de performance significante sem qualquer perda de flexibilidade. A arquitetura foi implementada utilizando o GNU Radio e a USRP2, e os testes comparativos entre a implementação original e a arquitetura proposta demonstraram ganhos significativos no aproveitamento dos recursos do sistema, relacionados mais especificamente ao desempenho e as interfaces de comunicação. Com a exigência de um desempenho menor para processar os blocos de software, a possibilidade do uso dessa tecnologia em sistemas embarcados torna-se factível.

Como trabalhos futuros, o porte da arquitetura de canais para plataformas embarcadas será finalizado, permitindo análises dos benefícios da utilização da proposta em um ambiente embarcado real. Além disso, problemas relacionados com a latência de comunicação entre o momento em que o sinal é capitado pela antena e as amostras chegam r nos blocos de software e a melhor integração entre os projetos da camada física e o MAC despontam como grandes desafios para a área.

Referências

Ackland, B., Raychaudhuri, D., Bushnell, M., Rose, C., and Seskar, I. (2005). High performance cognitive radio platform with integrated physical and network layer capabilities. Technical report, http://www.winlab.rutgers.edu/pub/docs/NeTS-ProWiN1.pdf.

Balister, P., Tsou, T., and Reed, J. H. (2007). Software defined radio on small form factor systems.

Blossom, E. (2009). Gnu radio. http://www.gnu.org/software/gnuradio.

Choong, L. (2009). Multi-channel ieee 802.15.4 packet capture using software defined radio. Technical report.

Ettus, M. (2009). Universal software radio peripheral. http://www.ettus.com/.

KUAR (2009). Kansas university agile radio. https://agileradio.ittc.ku.edu/.

Mccarthy, N., Blossom, E., Goergen, N., Oshea, T., and Clancy, C. (2008). High-performance sdr: Gnu radio and the ibm cell broadband engine. In *Virginia Tech Wireless Personal Communications Symposium*.

Mitola, J. (1995). The software radio architecture. *Communications Magazine, IEEE*, 33(5):26–38.

Reed, J. (2002). *Software radio: a modern approach to radio engineering*. Prentice Hall Press, Upper Saddle River, NJ, USA.

Tennenhouse, D. L. and Bose, V. G. (1996). The spectrumware approach to wireless signal processing. *Wireless Network Journal*, 2.

WARP (2009). Rice university wireless open-access research platform (warp). http://warp.rice.edu.

Welborn, M., Rao, S., Nathuji, R., Tuchinda, R., Ankcorn, J., Garland, S., and Guttag, J. (2009). Spectrumware project. http://nms.csail.mit.edu/projects/spectrumware/.

I Workshop de Sistemas Embarcados

♦

Sessão Técnica 4

Desenvolvimento de Software e Sistemas & Redes de Sensores

GERSE: Guia para Elicitação de Requisitos de Sistemas Embarcados

Jaime Cazuhiro Ossada[1], Luiz Eduardo G. Martins[2]

[1]FATEC-ID – Faculdade de Tecnologia de Indaiatuba
Rua Dom Pedro I, 65 – Cidade Nova – Indaiatuba – SP – Brazil

[2]UNIMEP – Universidade Metodista de Piracicaba
Rodovia do Açúcar, km 156 – Piracicaba – SP – Brazil

jossada@fatecindaiatuba.edu.br, lgmartin@unimep.br

Abstract. *The projects with embedded systems are used for many different purposes, being a major challenge for the community of developers of such systems. As we benefit from technological advances the complexity of designing an embedded system increases significantly. This paper presents GERSE, a guideline to requirements elicitation of embedded systems. Despite of advances in the area of embedded systems, there is a shortage of requirements elicitation techniques that meet the particularities of this area. The contribution of GERSE is to improve the capture process and organization of the requirements of embedded systems projects.*

Resumo. *Os projetos de sistemas embarcados são desenvolvidos para as mais diversas finalidades, apresentando-se como um grande desafio para a comunidade de desenvolvedores de software. Na medida em que nos beneficiamos dos avanços tecnológicos, a complexidade dos sistemas embarcados tem aumentado de forma significativa. Neste artigo apresentamos o GERSE, um guia para elicitação de requisitos de sistemas embarcados. Apesar dos avanços na área de sistemas embarcados, nota-se uma escassez de técnicas de elicitação de requisitos que atendam às particularidades desta área. A contribuição do GERSE vem no sentido de melhorar o processo de captura e organização dos requisitos em projetos de sistemas embarcados.*

1. Introdução

Os projetos de sistemas embarcados são criados para as mais diversas finalidades, sendo um nicho com muitos aspectos a serem explorados. A presença de sistemas embarcados (SE) tem crescido muito nos últimos anos, tanto na indústria e comércio como nas residências em geral [CANCIAN *et al.,* 2007]. Os SE estão presentes em nosso cotidiano, e a previsão é que aumentarão em grande escala nos próximos anos [WALLS, 2006], atualmente são fabricados bilhões de processadores (por ano) voltados para o mercado de SE [VAHID e GIVARGIS, 2002]. As aplicações de *software* desenvolvidas para SE estão se tornando mais sofisticadas e complexas, demandando das equipes de desenvolvimento grande esforço para gerenciar a complexidade e garantir a qualidade dos sistemas implementados. Essa sofisticação tem um impacto direto sobre o tratamento dos requisitos do sistema. Segundo Broy (1997), em sistemas computacionais embarcados mais de 50% dos problemas ocorrem após a entrega do artefato ao usuário. Contudo, os problemas relatados não são erros de implementação,

mas em grande parte equívocos cometidos na captura dos requisitos. Neste artigo apresentamos o GERSE, um guia de elicitação de requisitos para SE. O restante deste artigo está organizado da seguinte forma: na seção 2 é apresentada uma rápida caracterização do problema que levou ao desenvolvimento do GERSE; na seção 3 são discutidos os trabalhos correlatos; na seção 4 é apresentada a metodologia adotada para o desenvolvimento do guia; na seção 5 é apresentada a organização e as atividades que compõem o GERSE; na seção 6 é apresentada a conclusão do trabalho; e na seção 7 são indicados alguns trabalhos futuros.

2. Caracterização do problema

Na medida em que o *software* para SE está se tornando cada vez mais complexo, os engenheiros de sistemas embarcados estão procurando na Engenharia de *Software* técnicas, métodos e ferramentas que possam auxiliá-los na melhoria da qualidade do *software*. Por sua vez, a comunidade de Engenharia de *Software* está percebendo a necessidade de adaptar o ferramental já existente, e também propor novas abordagens, que possam atender de forma efetiva as particularidades da área de SE. Uma percepção que emerge das pesquisas na literatura da área de SE, e também a partir da interação com profissionais que desenvolvem sistemas nesta área, é que há uma escassez, senão uma ausência completa, de métodos, técnicas e ferramentas de Engenharia de Requisitos desenvolvidas especialmente para a área de SE. Para Wagner e Carro (2009), *software* embarcado possui características particulares, por exemplo, consumo de energia, quantidade de memória utilizada, requisitos de tempo real, e outras restrições que não são usualmente consideradas no desenvolvimento de *software* tradicionais.

Em junho de 2009, realizamos uma pesquisa de campo com o objetivo de mapear o estado da prática da elicitação de requisitos entre profissionais que desenvolvem SE, abrangendo principalmente os profissionais do Estado de São Paulo (participaram desta pesquisa 53 profissionais da área de SE). Entre outros fatores, essa pesquisa revelou que 54,9% dos profissionais de SE não utilizavam nenhuma metodologia para a elicitação de requisitos em seus projetos, e que 67% tinham formação voltada para o desenvolvimento de aspectos do hardware do sistema (como engenharia elétrica, eletrônica, tecnologia em automação industrial e telecomunicações), com pouca (ou nenhuma) educação formal em disciplinas da Engenharia de Software [OSSADA, 2010].

3. Trabalhos correlatos

Para a elaboração do guia de elicitação de requisitos dirigido aos SE, inicialmente foi realizado um estudo sobre oito trabalhos, publicados entre 1997 e 2009, nos quais são apresentadas propostas para as atividades da Engenharia de Requisitos. Os trabalhos analisados foram:

(i) Broy (1997), o qual estabelece que, os processos da Engenharia de Requisitos para os SE pode ser divididos em duas fases, a saber: a pré-fase e a fase principal. Durante a pré-fase, são elaboradas as estratégias gerais do produto e as suas restrições gerais, enquanto na fase principal são determinados os requisitos técnicos. Seu trabalho é centrado principalmente na fase principal, indicando que para descrever o comportamento de um SE é necessária a seleção de um bom modelo de domínio para expressar os requisitos, o qual deve ser simples, abstrato e o mais sugestivo possível.

(ii) Nasr et al. (2002) estudam os casos de uso para elicitar e especificar os requisitos para um SE, mas com foco no que foram considerados requisitos técnicos, não indicando, em nenhum momento, a utilização de uma técnica ou de um modelo dirigido à elicitação de requisitos para a captura de requisitos informais.

(iii) Graaf et al. (2003) realizaram um trabalho que consistia em determinar o estado da prática, envolvendo sete empresas e um instituto de pesquisa, em três países da Europa. As empresas variavam desde fabricantes de eletrônicos de consumo a indústrias especializadas em máquinas industriais. Os resultados mostram que, para a captura e a especificação de requisitos, normalmente é empregada a linguagem natural, em um processador de textos, e se utiliza um *template* que descreve quais aspectos devem ser especificados. Para organizar o gerenciamento de requisitos, nenhuma ferramenta apresentou resultados satisfatórios, por serem incompletos ou o tempo de aprendizado da ferramenta ser demasiadamente longo.

(iv) Botaschanjan et al. (2005) apresentam uma metodologia que aponta para quatro fases durante o desenvolvimento de SE: fase de especificação, fase de modelagem, fase de verificação e fase de integração. A primeira é a de especificação de requisitos, por meio da utilização da linguagem natural.

(v) Chae (2006) propõe a utilização de uma metodologia ágil para o desenvolvimento de sistemas embarcados, através do particionamento temporário de hardware e software. Porém, em seu trabalho não há menção à fase inicial da Engenharia de Requisitos, focando-se especificamente no *codesign* de hardware e software.

(vi) Pretschener et al. (2007) apontam em direção à existência de duas fases para as atividades de elicitação de requisitos e ressaltam a importância da Engenharia de Requisitos para os SE como uma disciplina chave para o sucesso no domínio dos SE automotivos. Além disso, realçam a relevância de um modelo sistemático para estruturar os requisitos após a sua captura.

(vii) Boulanger e Dao (2008) propõem a utilização do System Modeling Language (SysML), desenvolvido pela OMG, que consiste no reuso do conjunto da UML 2, adicionando-se novos diagramas, e modificando-se alguns para diminuir a lacuna entre software e outras disciplinas envolvidas no processo de um SE automotivo. Para as atividades de elicitação de requisitos, eles citam a utilização do diagrama de requisitos para integrar os modelos do sistema com requisitos baseados em texto, capturados por uma ferramenta de gerenciamento de requisitos.

(vii) Liggesmeyer e Trapp (2009) apresentam como sugestão a utilização de um modelo dirigido para especificar, gerenciar mudanças, rastrear e testar requisitos obtendo assim os requisitos funcionais inicialmente, sem considerar nenhuma execução técnica, e depois, durante o desenvolvimento da arquitetura do sistema, baseados nos requisitos funcionais. Contudo, em seu trabalho é proposta a utilização do MDA (*Model-Driven Architecture*).

Todos os trabalhos citados anteriormente apontam para fases durante o estágio inicial de desenvolvimento dos sistemas embarcados. A primeira delas (pré-fase) consiste em capturar as necessidades dos *stakeholders*, baseando-se em requisitos informais, e a segunda focaliza a transformação dos requisitos informais em requisitos técnicos. Porém, não foi observada a sugestão da utilização de uma metodologia

específica ou genérica para a captura e a análise dos requisitos, ou ainda de um guia para refinar e transformar os requisitos informais em técnicos.

4. Metodologia de Desenvolvimento

Para a elaboração do guia de elicitação de requisitos para SE, foi realizada inicialmente uma pesquisa de campo, utilizando-se um questionário com 25 questões, que foi aplicado aos profissionais que atuam no mercado brasileiro de SE, com o propósito de verificar o estado da prática em relação à elicitação de requisitos. Para participar da pesquisa, foram convidados profissionais que atuavam em várias áreas de SE, a maioria alocada em empresas do Estado de São Paulo, envolvendo vários segmentos de mercado: sistemas automotivos, automação industrial, eletrônicos de consumo, domótica, equipamentos médicos, telecomunicação e entretenimento. Após a tabulação e análise dos resultados da pesquisa de campo, realizou-se um estudo aprofundado do documento IEEE 830-1998 [IEEE, 1998], o qual fornece recomendações para a especificação de requisitos de *software*, e do *template* Volere [ROBERTSON E ROBERTSON, 2009], que fornece uma estrutura para documentar e organizar os requisitos de *software*. Esses três elementos de estudo constituíram a base para a elaboração das atividades que comporiam o guia de elicitação de requisitos (Figura 1).

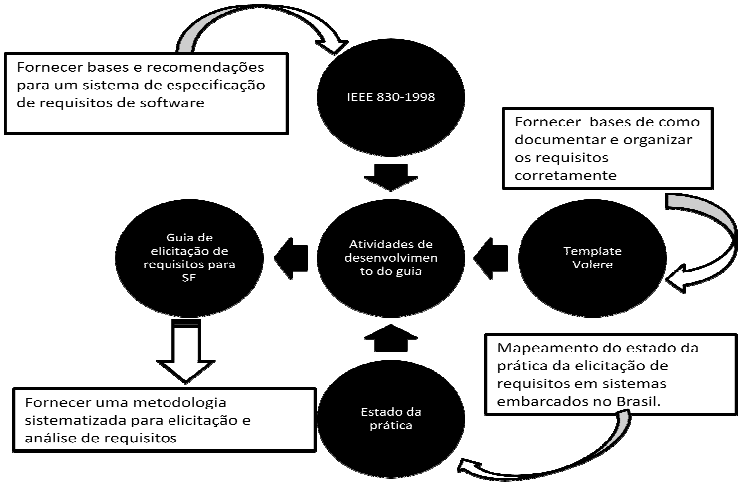


Figura 1. Elementos que Fundamentaram a Elaboração do GERSE

4.1. Condução da Pesquisa de Campo

A pesquisa foi realizada no período de junho e julho/2009. O questionário abrangeu 25 questões divididas em três grupos:

I) Perfil do profissional: neste grupo, foram elaboradas questões objetivas para se obter o tipo de formação, experiência e tempo de atuação em projetos de SE.

II) Tecnologia utilizada: neste grupo de questões o objetivo foi identificar os diversos ambientes de desenvolvimento adotados pelos profissionais, abrangendo famílias de processadores, compiladores e simuladores de hardware e software.

III) Processo de elicitação de requisitos: o foco neste grupo de questões foi identificar os principais procedimentos adotados, bem como as ferramentas utilizadas e os principais problemas encontrados durante a coleta e definição de requisitos. Também houve a preocupação de identificar quais eram os critérios adotados para separação de

funcionalidades e restrições a serem implementadas por hardware e software. A seguir são apresentados os resultados mais relevantes:

- Perfil do entrevistado: a maioria possui formação com ênfase em hardware, como engenharia elétrica, engenharia eletrônica, tecnologia de automação industrial, tecnologia em sistemas eletrônicos e tecnologia em telecomunicações, totalizando 67% dos profissionais participantes. Apenas 33% possuem formação (superior) exclusivamente em computação. A Figura 2 mostra o perfil da formação técnica dos profissionais que participaram da pesquisa.

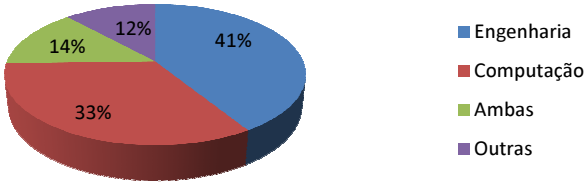


Figura 2. Formação técnica dos participantes

- Tecnologias utilizadas: os resultados foram abrangentes e diversificados. A família de microcontroladores *PIC* revelou-se a mais utilizada, abrangendo 23% das respostas, seguida pela família *ARM* com 21%. A família *Motorola MCS* 51 ficou com 14% de utilização, 10% para a família *Freescale*, contra 9% da família *ATMEL*. Outros MCs foram citados durante a pesquisa, como *MIPS* (3%), *Rabbit* (3%), *Zilog* (2%) e citações aos MCs *Nios, Infineo, Silabs, Cypress, Hitachi e MSP340*. Neste item alguns desenvolvedores trabalhavam com mais de uma família de MC.

- Processos de elicitação de requisitos: o foco central foi identificar se eram adotados processos consistentes de elicitação e documentação de requisitos, quais seriam as maiores dificuldades durante a elicitação de requisitos e o grau de importância atribuído ao tratamento dos requisitos em projetos de sistemas embarcados. Quando perguntados se utilizavam alguma metodologia para elicitação de requisitos, mais da metade (54,9%) responderam que não utilizavam nenhuma metodologia. Os que responderam sim (45,1%), afirmaram que os procedimentos adotados são baseados em metodologias oriundas da literatura em Engenharia de Software (47%), seguidos por aqueles criados pela própria empresa (24%), criados e adotados por iniciativa própria do desenvolvedor (19%) e outras formas (10%). Questionados ainda se os procedimentos adotados eram estáveis, 41,2% responderam negativamente. Outro item analisado foi sobre as técnicas de coleta de requisitos adotadas, as respostas foram diversas, distribuídas da seguinte forma: análise de documentação existente (26%), entrevistas (19%), troca de mensagens por fax e email (18%), análise de mercado (17%), questionário (15%) e JAD (2%). Chamou atenção o fato de vários profissionais considerarem a troca de mensagens via fax e e-mail como técnicas de coleta de requisitos, dado o grau de informalidade que normalmente acompanha a utilização desses recursos.

Questionados sobre as principais dificuldades encontradas durante a coleta e análise de requisitos, as respostas ficaram assim distribuídas: falta de clareza por parte do cliente (22%), requisitos incompletos (20%), confusos (18%) e ambíguos (17%), pouco tempo para coleta e análise (10%). Outras citações menos freqüentes foram: não importância aos requisitos por parte do stakeholder, ou da própria equipe de desenvolvimento por ser multidisciplinar, além de requisitos muitas vezes impossíveis

de serem implementados. Na Tabela 1 são apresentadas as principais dificuldades encontradas por desenvolvedores de SE durante a elicitação e análise de requisitos.

Tabela 1. Principais dificuldades encontradas na elicitação de requisitos

Principais dificuldades	Respostas
Falta de clareza por parte do cliente	22%
Requisitos incompletos	20%
Requisitos confusos	18%
Requisitos ambíguos	17%
Pouco tempo para coleta e análise dos requisitos	10%
Dificuldade em organizar os requisitos coletados e ausência e ineficácia de um guia de elicitação	5%
Alterações de requisitos, pouca importância aos requisitos por parte do cliente, equipe não dá importância aos requisitos e requisitos não realizáveis.	1%

5. Organização e Atividades do GERSE

O guia desenvolvido foi dividido duas fases (pré-fase e fase principal), organizado em sete categorias, que totalizam quarenta e seis atividades específicas, responsáveis pela geração de sete artefatos que devem compor os requisitos do sistema embarcado (Figura 3). A cada passo, os artefatos serão gerados com o propósito de auxiliar na transição dos requisitos informais para os requisitos técnicos.

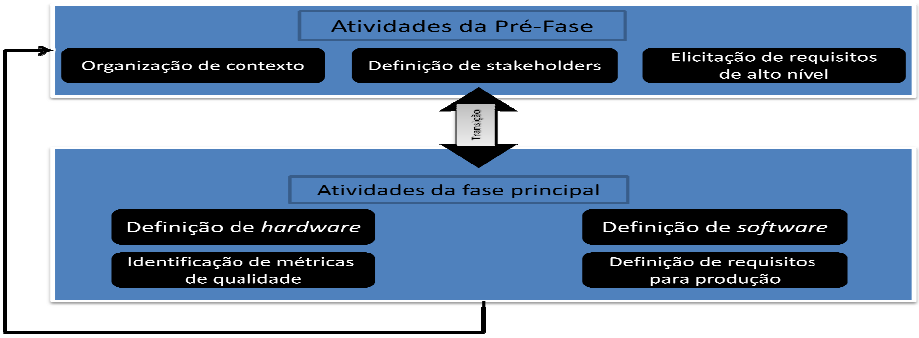


Figura 3. Os setes artefatos gerados pelo GERSE

5.1. Resumo Geral das Atividades do GERSE

Durante a pré-fase, as atividades foram organizadas em três categorias (organização de contexto, definição de *stakeholders* e requisitos de alto nível). Na fase principal foram criadas quatro categorias (requisitos de *hardware*, requisitos de *software*, requisitos de qualidade e de produção) cada qual com um conjunto de atividades específicas. A Tabela 3 apresenta o resumo geral das atividades a serem realizadas adotando-se o GERSE. Ao finalizar as atividades da pré-fase, é possível ao engenheiro de requisitos analisar os requisitos de alto nível e também fazer a conversão para os requisitos técnicos. Cada categoria possui objetivos diferentes, com atividades específicas, cujo propósito é gerar artefatos (saída) para auxiliar na especificação dos requisitos.

Tabela 3. Resumo Geral das Atividades do GERSE

Atividades da Pré Fase	
1 Organização de Contexto	
1.1 Obter o propósito e as metas organizacionais do produto frente ao mercado.	1.2 Definir as características gerais do produto.
1.3 Definir os impactos organizacionais com o desenvolvimento do produto.	1.4 Definir os impactos negativos com o não desenvolvimento do produto.

1.5 Definir as expectativas de tempo total de desenvolvimento do produto.	1.6 Definir o público a ser atingido.
1.7 Recuperar projetos de sistemas legados.	
2 Atividades de definição de *stakeholders*	
2.1 Definir principais stakeholders.	2.2 Definir stakeholder especialistas de domínio.
2.3 Definir stakeholders contrário ao projeto.	2.4 Definir perfil do usuário.
3 Atividades de elicitação de requisitos de alto nível	
3.1 Definir as funções do produto (Requisitos Funcionais).	3.2 Definir as restrições do produto (Requisitos não funcionais).
3.3 Definir as restrições físicas do ambiente.	3.4 Definir as características de consumo de energia.
3.5 Definir as características físicas e mecânicas.	3.6 Definir a interface.
3.7 Definir as situações críticas.	3.8 Definir grau de confiabilidade.
3.9 Definir solução encontrada.	3.10 Definir estimativa de custos.
Atividades da Fase Principal	
4 Atividades de identificação de Hardware	
4.1 Definir sensores.	4.2 Definir atuadores.
4.3 Definir interação com o usuário	4.4 Definir interrupções de HW.
4.5 Definir botões.	4.6 Definir memórias.
4.7 Definir portas de comunicação externa.	4.8 Definir requisitos de componentes.
4.9 Definir requisitos de layout da placa controladora.	4.10 Definir parâmetros de HW legados.
4.11 Definir parâmetros de COTS especiais.	4.12 Definir microcontroladores.
5 Atividades de identificação de Software	
5.1 Definir variáveis de ambiente.	5.2 Definir funções de SW.
5.3 Definir exceções.	5.4 Definir funções de interrupções
5.5 Definir requisitos de idioma.	5.6 Definir interface de comunicação (software).
5.7 Definir funções de monitoramento.	5.8 Definir funções de armazenamento de dados.
6 Atividades de definição de métricas de qualidade.	
6.1 Definir grau de segurança.	6.2 Definir desempenho.
6.3 Definir métricas de manutenção.	
7 Atividades de definição de métricas de linha de produção.	
7.1 Definir aspectos de produção.	7.2 Definir embalagem.

5.2. Relação Requisitos Informais *versus* Requisitos Técnicos

Os requisitos informais possuem um alto grau de influência nos requisitos técnicos. Cada atividade específica identificada (conjunto de requisitos de alto nível) na pré-fase será utilizada intensamente para definir os requisitos técnicos. A Tabela 4 mostra as influências que puderam ser notadas de forma mais direta entre as atividades da pré-fase (requisitos de alto nível) e as atividades da fase principal (requisitos técnicos). Para a construção do quadro de dependência, foram observadas quais as atividades de definição dos requisitos de alto nível da pré-fase (nove atividades específicas) influenciariam no conjunto de vinte e seis atividades para a definição dos requisitos técnicos

O GERSE foi desenvolvido inicialmente para a elicitação de requisitos de SE de pequeno e médio porte, possibilitando aos engenheiros de requisitos realizarem as atividades de forma sistemática, em duas fases distintas, o que implica em uma transição natural a qual ordena os requisitos em grupos específicos. O *template Volere* [ROBERTSON E ROBERTSON, 2009] e a recomendação *IEEE 830-1998* [IEEE, 1998], no entanto, são modelos gerais e não contemplam em sua estrutura a oportunidade de serem utilizadas diretamente em projetos de SE. As atividades previstas no GERSE apresentam um maior grau de detalhamento, obtido por meio de um conjunto de atividades específicas para projetos em SE, permitindo aos engenheiros de requisitos serem contemplados com uma ferramenta mais adequada às atividades de elicitação de requisitos, produzindo uma documentação completa, facilitando e organizando as demais fases do ciclo de vida de desenvolvimento do produto.

Tabela 4. Relação Requisitos Informais *versus* Requisitos Técnicos

Atividades da Pré -Fase (requisitos de alto nível)	Atividades da Fase Principal (requisitos técnicos)																								
	4.1 Definir Sensores	4.2 Definir atuadores	4.3 Definir interação com o usuário	4.4 Definir interrupções de *HW*	4.5 Definir botões	4.6 Definir memórias	4.7 Definir portas de comunicação externa	4.8 Definir requisitos de componentes	4.9 Definir requisitos de *layout* da placa controladora	4.10 Definir parâmetros de *HW* legados	4.11 Definir parâmetros de COTS especiais	4.12 Definir microcontrolador	5.1 Definir variáveis de ambiente	5.2 Definir funções de *SW*	5.3 Definir exceções	5.4 Definir funções de interrupções	5.5 Definir requisitos de idioma	5.6 Definir interface de comunicação (*software*)	5.7 Definir funções de monitoramento	5.8 Definir funções de armazenamento de dados	6.1 Definir grau de segurança	6.2 Definir desempenho	6.3 Definir métricas de manutenção	7.1 Definir aspectos de produção	7.2 Definir embalagem
3.1 Definir as funções do produto (Requisitos Funcionais)	x	x	x	x	x			x		x	x	x	x	x	x	x		x	x	x	x	x	x		
3.2 Definir as restrições do produto (Requisitos não funcionais)			x						x	x	x	x		x	x		x			x	x	x	x	x	
3.3 Definir as restrições físicas do ambiente	x	x									x	x	x												
3.4 Definir as características de consumo e energia	x	x	x		x	x		x		x	x	x						x		x	x				
3.5 Definir as características físicas e mecânicas	x	x	x		x			x	x	x	x	x				x		x		x	x		x	x	x
3.6 Definir interface de Comunicação	x	x	x	x	x	x	x	x	x	x	x	x		x		x	x	x		x	x	x			
3.7 Definir Situações Críticas		x	x	x				x		x	x			x	x	x					x	x			
3.8 Definir grau de confiabilidade	x	x	x	x	x	x		x		x	x	x		x				x	x	x	x		x		
3.9 Definir solução encontrada	x	x	x	x	x	x	x	x	x	x	x	x	x	x						x	x	x	x	x	x
3.10 Definir estimativas de custos	x	x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x

5.3. Estudo de Caso

Após a elaboração do guia, o mesmo foi instanciado em um estudo de caso, que foi a elicitação de requisitos de um relógio digital de xadrez. O objetivo dessa experiência foi avaliar na prática o guia proposto para a elicitação de requisitos de SE, demonstrando a aplicação do guia e os artefatos gerados. A seguir serão apresentados fragmentos da documentação gerada no projeto instanciado.

Quadro 1. Organização de contexto

1.1 Obter o propósito e as metas organizacionais do produto frente ao mercado.

O produto a ser desenvolvido tem o objetivo de automatizar a marcação de tempo de uma partida de xadrez. O produto tem como meta ser adotado em competições nacionais de xadrez e para uso doméstico por enxadristas de qualquer nível.

1.2 Definir as características gerais do produto.

O produto deverá marcar tempo com exatidão, ser resistente, leve, fácil de transportar e ser de baixo custo. Deverá utilizar *displays* de cristal liquido que permita fácil visualização do tempo decorrido por parte dos jogadores e árbitro.

1.3 Definir os impactos organizacionais com o desenvolvimento do produto .

Com o lançamento do produto, através de campanhas publicitárias, é desejo da organização tornar-se líder nacional no segmento de fornecimento de relógios digitais para partidas de xadrez.

1.4 Definir os impactos negativos com o não desenvolvimento do produto.

Com o não desenvolvimento do produto, os esforços empregados no projeto serão perdido. Trata-se de um produto inédito no mercado nacional, na qual similares são todos importados e com interface de comunicação com o usuário geralmente na língua estrangeira (inglês), oferecendo uma grande oportunidade para o lançamento do produto devido às características adequadas para o mercado brasileiro. A não realização do projeto fará com que a organização perca grande oportunidade de mercado que se encontra inexplorado por produtos nacionais similares.

1.5 Definir as expectativas de tempo total de desenvolvimento do produto.

É desejo da organização o lançamento do produto no mercado nos próximos 12 meses.

1.6 Obter o público a ser atingido.

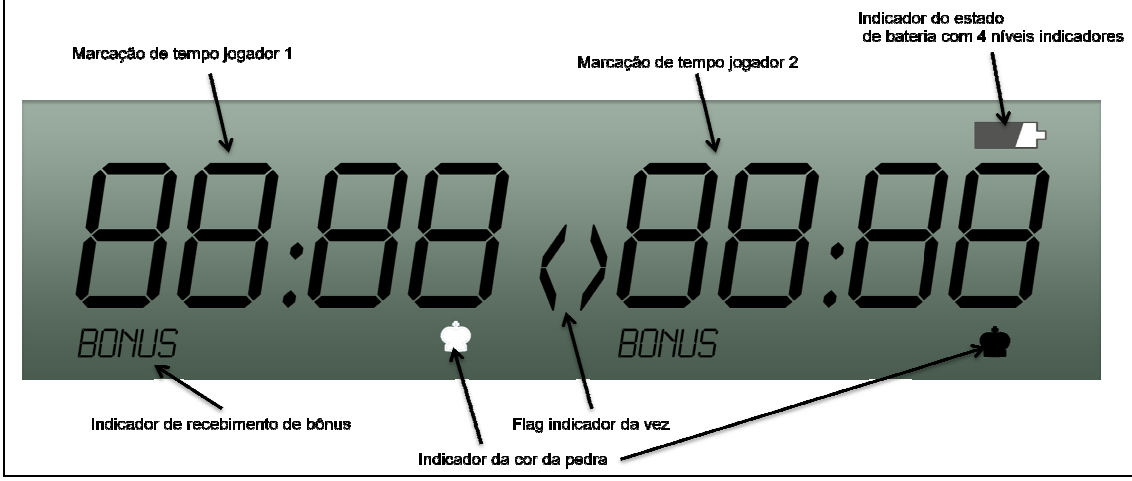
Usuários do produto serão enxadristas brasileiros, amadores e profissionais, Confederação Brasileira de Xadrez, e outras entidades ligadas a esse esporte.

1.7 Recuperar projetos de sistemas legados.

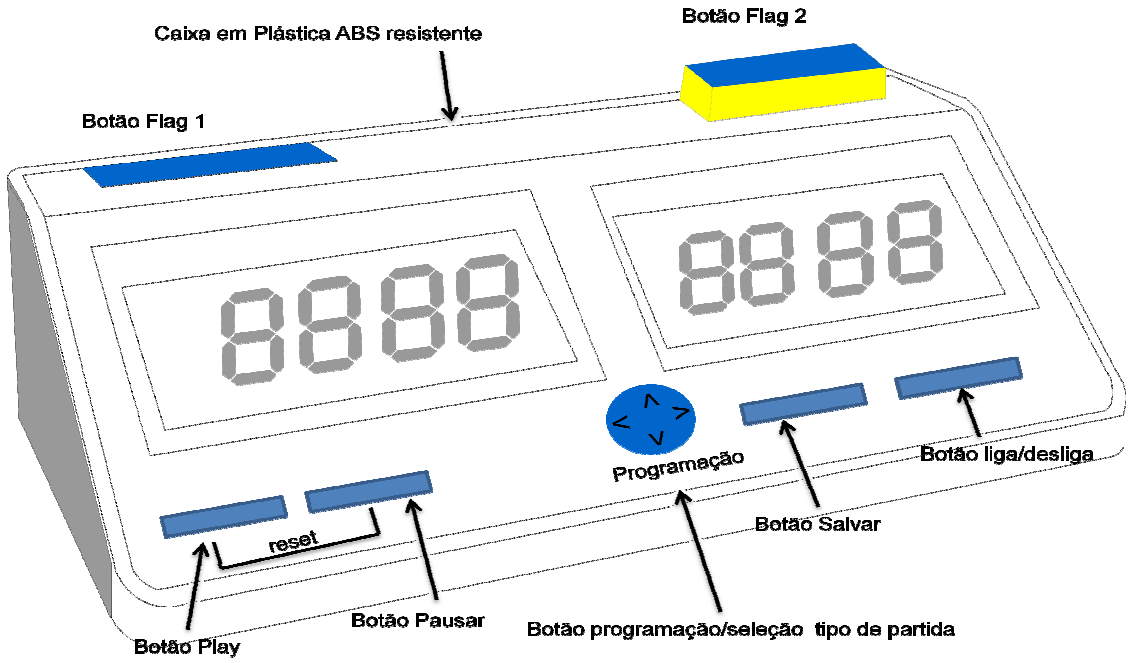
Trata-se de um projeto que será realizado pela primeira vez, não sendo utilizado nenhum sistema legado.

Quadro 2. Sugestão de mensagens de display (Atividade específica: definir as características físicas e mecânicas)

Sugestão das mensagens a serem apresentadas ao usuário com base no *display*.



Sugestão para o gabinete, a disposição dos botões de ação e suas respectivas funções.



Quadro 3. Definição das variáveis de ambiente a serem empregadas durante a programação do *firmware*

Variável	Tipo	Faixa de valores
Contador de Tempo	Inteiro	0 a 99999
Sensor de temperatura	Inteiro	0 a 100
Sensor de umidade	Inteiro	0 a 100
***Flags* 1 e 2**	Inteiro	0 e 1
Botão Play	Inteiro	0 e 1
Botão Pausar	Inteiro	0 e 1
Botão Parar	Inteiro	0 e 1
Botão Salvar	Inteiro	0 e 1
Botão Ligar-Desligar	Inteiro	0 e 1
Salvar *log*	Char	A|Z e 0|10
Display 1 e 2	Char	A|Z e 0|10
Botão Programação	Inteiro e char	A|Z e 0|10 e 0 e 1

5.4. Avaliação do GERSE

A documentação do GERSE foi submetida a quatro projetistas de SE (profissionais que atuam na área), que realizaram a avaliação formal do guia, por meio de um questionário. Salientou-se aos avaliadores que o GERSE foi elaborado para atender às diversas disciplinas presentes em um projeto de SE, e não apenas ao desenvolvimento de HW e

de SW, mas sim às questões globais do projeto. As respostas do questionário permitiram uma avaliação prática da viabilidade da utilização do guia proposto em projetos de SE, e na identificação de eventuais correções, não detectadas anteriormente. De modo geral os resultados da avaliação demonstraram que o GERSE satisfaz a proposta de um guia de elicitação de requisitos para SE. A utilização do GERSE, dentro de um projeto de sistema embarcado, com as atividades dispostas de forma organizada, produz como resultado final um documento bem estruturado e compreensível. Os resultados mostraram-se promissores e o GERSE apresentou-se como uma ferramenta que pode facilitar e auxiliar na especificação de requisitos de SE. A tabela 5 apresenta o resultado final da avaliação qualitativa do GERSE, após a tabulação das respostas enviadas pelos participantes da avaliação.

Tabela 5 – Resultado final da avaliação qualitativa do GERSE

Questões	Concordo Totalmente	Concordo parcialmente	Discordo Parcialmente	Discordo Totalmente
O guia apresentado é claro suficiente para ser utilizado em um projeto de sistemas embarcados de pequeno e médio porte.	50%	50%	0%	0%
O guia apresentado é completo e atende as necessidades para projetos de sistemas embarcados de pequeno e médio porte.	50%	25%	25%	0%
Adotaria o guia apresentado para elicitação de requisitos de projetos futuros.	50%	25%	25%	0%
O guia apresentado é de fácil utilização.	50%	50%	0%	0%
O guia apresentado contribui na melhoria da qualidade de desenvolvimento de sistemas embarcados.	50%	50%	0%	0%
O guia apresentado atende às necessidades de definições de requisitos em projetos de sistemas embarcados.	50%	50%	0%	0%

Os comentários acerca da avaliação geral do GERSE são promissores. Alguns deles são apresentados a seguir:

> *"Acho um bom guia, um ponto de partida de grande utilidade para as equipes de projetos de sistemas embarcados que, em geral, fazem muito pouco nesse sentido. É tudo muito ad-hoc. Creio que esse trabalho pode ajudar bastante.".*

> *"Acredito ser perfeitamente possível utilizá-lo em projetos de sistemas embarcados de pequeno e médio porte, como sugere o seu guia, principalmente na fase inicial, em que a grande maioria dos projetistas o faz de maneira desorganizada. Um guia de elicitação veio a calhar".*

> *"Dentre todos os projetos de sistemas, acredito que, o de sistemas embarcados seja o mais difícil pela diversidade de requisitos de engenharia, tecnologias, processos produtivos e, principalmente, por requerer um grande conhecimento de hardware (até mesmo para codificar o software). Vale ressaltar que, o programador vai encontrar uma máquina desprovida de sistema operacional. Só a tarefa de inicializar o hardware, já é um grande desafio. Acho que não existe uma caracterização, a qual englobe todas as suas características. Podemos, contudo, enxergar um conjunto mínimo, como faz essa proposta. Acredito que o guia em questão englobe uns 80% das atividades necessárias."*

6. Conclusão

A elicitação de requisitos em sistemas embarcados é uma atividade fundamental para contribuir com a melhoria da qualidade desses sistemas. Sistemas embarcados, após sua implementação, dificilmente poderão sofrer alterações, devido ao alto acoplamento que possuem em relação ao artefato físico em que estão embutidos. Portanto, falhas na

captura dos requisitos, percebidas depois da implementação do sistema, são desastrosas, podendo levar o projeto ao insucesso completo. Com os resultados do questionário de avaliação do GERSE, por intermédio de profissionais que atuam no mercado de trabalho de SE, pode-se observar que alguns ajustes no guia ainda se fazem necessários, por exemplo o aperfeiçoamento e a minimização de *hardware* e de *software*, oferecendo ao projetista a oportunidade de uma escolha adequada para executar uma função por *hardware* ou *software* para minimizar os recursos. Ou ainda a inserção de mais atividades de requisitos de confiabilidade e qualidade. Entretanto, de maneira geral, o GERSE foi considerado satisfatório, contribuindo para preencher a lacuna existente na fase inicial de um projeto de SE. O GERSE contribui para atenuar de forma significativa os riscos de falhas na captura dos requisitos de SE, alcançado por intermédio da captura sistematizada dos requisitos e através da transição dos requisitos informais para requisitos técnicos, auxiliando também no particionamento dos requisitos do produto. Os respondentes do questionário de avaliação do GERSE fizeram uma avaliação positiva do guia, destacando que ele contribui para a melhoria da qualidade do desenvolvimento de SE, e que é de fácil utilização, com boa cobertura dos principais aspectos relativos à concepção de um SE. A realização deste trabalho permitiu ampliar a visão de profissionais e estudantes de Ciência da Computação em relação a SE. No meio acadêmico ainda é pouco estudado ou relatado sobre os processos em um projeto de SE. Alguns dados são cercados de sigilo profissional por parte das equipes de desenvolvimento que atuam nas organizações, por se tratarem de documentos internos às empresas em uma área com inúmeras particularidades, mas constantemente em evolução, e que carece de ferramentas que auxiliem a diminuir o tempo de desenvolvimento sem deixar de lado aspectos de qualidade, segurança e confiabilidade. Sistemas embarcados estão presentes no cotidiano das pessoas e a sua presença está aumentado cada vez mais, necessitando de mais pesquisas neste campo.

7. Trabalhos Futuros

O trabalho apresentado neste artigo pode ser ampliado com uma série de novas propostas, as quais podem ser realizadas em trabalhos de conclusão de cursos de graduação, dissertações de mestrado e teses de doutorado. Entre essas propostas, destacam-se:

- Ajustes, adequações e inserção de atividades no guia, identificadas através da aplicação do questionário utilizado para avaliação, realizada junto aos profissionais da área;
- Adequação do GERSE às atividades de elicitação de requisitos para SE de grande porte (baseados em microprocessadores);
- A experimentação do GERSE em outros sistemas microcontrolados, como por exemplo, um sistema de codificação e decodificação MP4, um sistema de cadeira de rodas automatizado ou ainda a aplicação em um projeto de SE com uso de redes de computadores;
- O desenvolvimento de uma ferramenta automatizada que apóie e facilite a utilização do guia, bem como permita a criação de um repositório para armazenamento e recuperação dos artefatos gerados ao longo da utilização do GERSE;

- A complementação do GERSE para conseguir a cobertura das demais fases da Engenharia de Requisitos (modelagem, especificação e validação);
- A integração com outros *templates*, destinados à utilização na Engenharia de Requisitos para os sistemas computacionais de uso geral;
- A integração do guia com ferramentas automatizadas comerciais (*Doors*, *Enterprise Architect*) e das ferramentas livres (*Eclipse*, *Net Beans* etc.);
- A aplicação do GERSE em outros projetos de SE, como por exemplo, que utilizem *DSP*, *PLD* e outros tipos de processadores além de microcontroladores.

Referências

Boulanger J.; Van Quang D. (2008). "Experiences from a model-based methodology for embedded electronic software in automobile", at Information and Communication Technologies: From Theory to Applications, In: ICTTA 3rd International Conference on.

Botaschanjan, J.; Kof, L.; Kuhnel C.; Spichkova, M. (2005). "Towards verified automotive software", In: Proceedings of the Second international Workshop on Software Engineering For Automotive Systems.

Broy M. (1997). "Requirements engineering for embedded systems". In: Proc. ofFemSys.

Cancian R.; Stemmer M.; Frohlich A. (2007). "New Developments in EPOS Tools for Configuring and Generating Embedded Systems", In: Proceedings of the 12th IEEE International Conference on Emerging Technologies and Factory Automation, Patras, p. 776-779.

Chae H. (2006). "The Partitioning Methodology in Hardware/Software Co-Design Using Extreme Programming: Evaluation Through the Lego Robot Project", In: Proceedings of the sixth IEEE International Conference on Computer and information Technology.

Graaf B.; Lormans M; Toetenel H. (2003). " Embedded software engineering: the state of the practice", In: IEEE Software archive, v. 20, n. 6, p. 61-69.

IEEE Computer Society Software Engineering Standards Committee (1998), "IEEE Recommended Practice for Software Requirements Specifications". IEEE Std 830-1998.

Liggesmeyer P.;Trapp M. (2009). "Trends in Embedded Software Engeneering", IEEE, In: Software, IEEE, v. 26, n. 3, 2009, p. 19-25.

Nasr E.; Mcdermid J.; Bernat G.(2002). "Eliciting And Specifying Requirements With Use Cases For Embedded Systems", In: Proceedings at 7th International Workshop on Object-Oriented Real-Time dependable systems (WORDS 2002).

Ossada J. C. (2010). "GERSE: Guia de Elicitação de Requisitos para Sistemas Embarcados de Pequeno e Médio Porte". Dissertação de Mestrado do Programa de Pós-Graduação em Ciência da Computação – Universidade Metodista de Piracicaba - Piracicaba.

Pretschner A.; Broy M.;Kruger, Ingolf H.; Stauner T.(2007). “Software Engineering for Automotive Systems: A Roadmap Future of Software Engineering”, In: International Conference on Software Engineering - Future of Software Engineering (FOSE’07).

Robertson S.; Robertson J. (2006). “Mastering the Requirements Process”. Addison-Wesley Pub Co; 2st edition.London.

Vahid F.; Givargis T. (2002). “Embedded System Design: A Unified Hardware/Software Design”. John Willey & Sons.

Wagner, F.; Carro, L. (2009). “Metodologias e Técnicas de Engenharia de Software para Sistemas Embarcados”, In: JAI’09 – XXIII Jornadas de Atualização de Informática, Capítulo 4. SBC/PUC, Bento Gonçalves.

Walls C. (2006). “Embedded Software – The Works”. Elsevier. London.

Esqueletotipação: Um Método para Desenvolvimento de Software Embarcado

Diogo Branquinho Ramos[1], **Adilson Marques da Cunha**[1]

[1]Divisão de Ciência da Computação
Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos – SP – Brasil

{diogobr,cunha}@ita.br

Abstract. *This paper describes the Skeletotyping Method designed for embedded software development comprised of five steps: 1) requirements modeling, 2) use cases modeling, 3) design pattern application, 4) source-code generation application techniques, and 5) updates and changes application. The major contributions of this paper are the Skeletotyping Method and its 3rd and 4th steps. The 3rd step involving the design pattern application has improved modeling by allowing proper boudaries between business and infrastructure layers needed to systems' operations. Finally, the 4th step involving source-code generation application techniques was based upon models and helped to ensure the embedded software development according to product specification and design.*

Resumo. *Este artigo descreve o Método de Esqueletotipação concebido em cinco passos para o desenvolvimento de software embarcado: 1) diagramação de requisitos; 2) diagramação de casos de uso; 3) aplicação de um padrão de projeto (design pattern); 4) aplicação de técnicas para geração de códigos-fonte; e 5) aplicação de alterações e mudanças. As principais contribuições deste artigo são o Método de Esqueletotipação e os seus 3º e 4º passos. O 3º passo melhorou a modelagem, permitindo delimitações apropriadas entre as camadas de negócio e de infraestrutura necessárias à operação de sistemas. Finalmente, o 4º passo ajudou a garantir o desenvolvimento de um software embarcado, de acordo com a especificação e o projeto de um produto.*

1. Introdução

Nas duas últimas décadas, constatou-se uma crescente utilização de sistemas e software embarcados anteriormente empregados apenas em ambientes complexos industriais, aeronáuticos e aeroespaciais, atualmente utilizados em veículos, televisores, celulares, entre outros dispositivos [Taurion 2005].

Dentro desse contexto, projetos de sistemas embarcados costumam envolver rigorosas especificações de requisitos, muitas vezes para cumprirem restrições de tempo. Além disto, estes projetos possuem vários fatores limitantes tais como baixa quantidade de memória disponível, pouco poder de processamento, limitação de consumo de energia sem perda de desempenho, curtos espaços de tempo para desenvolvimento, entre outros [Barr 1999].

O aumento do número de funções incorporadas a um único sistema embarcado vem ocasionando a necessidade de utilização de metodologias, técnicas e ferramentas para o gerenciamento de complexidades.

Entre outros fatores, vêm se constatando que as multidisciplinaridades das equipes de projetos devem se utilizar, cada vez mais, de comunicações claras e uniformes, de modo a minimizar os erros causados por interpretações incorretas dos requisitos e das funcionalidades dos sistemas [Douglass 2004].

Logo nas fases iniciais do desenvolvimento dos projetos, a utilização de protótipos vem representando outro fator relevante para o gerenciamento de complexidades e possibilitando oportunidades de produções de versões iniciais (à priori) de sistemas disponibilizáveis.

As construções de protótipos de hardware vêm facilitando os testes de projeto (*design*) e as elaborações de protótipos de software, auxiliando sobremaneira nas elucidações, verificações e validações de seus requisitos funcionais.

A modelagem visual, ao contemplar a possibilidade de abstrações de problemas do mundo real em modelos, vem contribuindo para reduzir a complexidade na compreensão dos requisitos dos sistemas.

A Linguagem de Modelagem Unificada (*Unified Modeling Language* - UML) e a Linguagem de Modelagem de Sistemas (*System Modeling Language* - SysML) representam os padrões notacionais mais utilizados para se abstrair software [Ramos et al. 2008]. Além disso, a modelagem visual possibilita a aplicação do paradigma de Desenvolvimento Dirigido a Modelos (*Model-Driven Development* - MDD), permitindo tanto a execução de modelos quanto a geração automática de códigos-fonte [Loubach et al. 2008].

A partir da crescente complexidade das metodologias da Engenharia de Software, surgiram as ferramentas de Ambientes Integrados de Engenharia de Software Ajudada por Computador (*Integrated - Computer Aided Software Engineering - Environment* - I-CASE-E), para o apoio as comunidades de engenharia, envolvendo gerenciamento, análise, codificação e testes, por meio de métodos e técnicas.

Entretanto, o uso desses ambientes de ferramentas, ainda se apresenta como incipiente, principalmente, por resistência dos desenvolvedores, que consideram suas configurações e usos complexos, não percebendo com isso os ganhos de qualidade, confiabilidade e segurança (*safety*) nos produtos gerados [Cunha 2006]. Cabe ressaltar também que o uso destes ambientes de ferramentas deve sempre levar em conta as necessidades dos projetos.

A principal motivação do trabalho de pesquisa que deu origem a este artigo surgiu da necessidade de se obter métodos e técnicas mais apropriadas para apoio ao desenvolvimento de software embarcado que reduzissem, principalmente, as deficiências existentes quanto aos desenvolvimentos baseados em modelos e as utilizações de ferramentas I-CASE-E.

2. Sistemas Embarcados

Os sistemas embarcados são sistemas computadorizados contendo integração de hardware e software fortemente acoplados, podendo ainda combinar possíveis partes adicionais mecânicas, elétricas ou de software para executar funções dedicadas e específicas nos contextos dos sistemas [Li and Yao 2003].

Os projetos de sistemas embarcados distinguem-se razoavelmente dos projetos de computadores pessoais, mesmo envolvendo componentes de software, de hardware e de mecânica. Os computadores pessoais não foram concebidos para desempenhar funções específicas e possuem capacidade de servir a muitos propósitos, de forma geral e não aprofundada.

2.1. Software Embarcado

O Software, como citado anteriormente, pode fazer parte de um Sistema Embarcado, assim como o hardware e as demais partes mecânicas. Neste segmento de software embarcado, empregam-se técnicas de desenvolvimento diferenciadas das normalmente utilizadas no desenvolvimento do software de propósito geral. Enquanto o software de propósito geral busca portabilidade de plataformas, o software embarcado deve ser especificado para plataformas previamente designadas, acarretando o aumento de complexidades para o seu desenvolvimento [Barr and Massa 2006].

Ultimamente, algumas técnicas de desenvolvimento de software de propósito geral vêm sendo adaptadas para o desenvolvimento de software embarcado. Entre elas cabe destacar a aplicação de: padrões de projetos (*design pattern*); técnicas de modelagem; geração automática de códigos-fontes; técnicas de testes de software; e testes de modelos.

Um padrão de projeto representa uma solução generalizada para problemas recorrentes. Neste caso, considera-se que a solução do padrão de projeto seja suficientemente geral para abranger um maior conjunto de aplicações, em diferentes domínios de conhecimento [Douglass 2003].

3. Desenvolvimento Dirigido a Modelos

As técnicas de modelagem visual vêm auxiliando o desenvolvimento de software e a sua utilização vem contribuindo sobremaneira nos seguintes aspectos:

- facilidades nos entendimentos dos problemas;
- enriquecimento de comunicações entre desenvolvedores e usuários;
- preparação de documentações dos sistemas;
- auxílio nos projetos (*design*) de software; e
- aproveitamento de automatizações de códigos-fonte e de testes.

Além disso, considerando-se a possibilidade de geração automática de códigos-fonte baseada em modelos, a modelagem visual auxíliou no advento do paradigma de MDD, focando muito mais no modelo ou abstração do sistema de software do que no aplicativo final [Azevedo 2008]. Desta forma, a aplicação de MDD evidência as vantagens da modelagem dos sistemas nos diversos níveis de abstração da solução e nas suas integrações e fluxos de informação.

Os modelos executáveis caracterizam-se como um componente-chave do MDD. As aplicações dos seguintes conceitos possibilitam a interoperabilidade entre as diversas ferramentas baseadas em MDD: transformações automáticas de modelos; validações de modelos; e suas padronizações [Kossiakoff and Sweet 2003].

No paradigma de MDD, modelos não são utilizados somente como rascunhos de um novo Sistema. Eles fazem parte da solução, compondo um conjunto de artefatos primários que, por meio de transformações automáticas ou manuais, podem gerar implementações mais eficientes.

A UML, como uma das linguagens de notação de modelos do MDD, oferece um alto grau de flexibilidade necessária para os engenheiros de software entenderem e expressarem as complexas relações e interações de sistemas embarcados. Ela, entretanto, não é uma linguagem formal, oferecendo uma certa liberdade na modelagem. Em contra partida, possui algumas desvantagens, devido às poucas regras envolvidas [UML 2003].

A SysML, assim como a UML, tem a intenção de unificar o modo pelo qual o desenvolvimento de sistemas é realizado. Ela representa uma linguagem de desenvolvimento de aplicações de domínio específico para Sistemas de Engenharia.

Derivada da UML, a SysML suporta as fases de especificação, análise, projeto verificação e validação de uma gama de Sistemas e de Sistemas de Sistemas - SdS (*Systems-of-Systems* - SoS), que podem incluir hardware, software, informações, processos, pessoas, e instalações [Azevedo 2008].

Segundo [Haywood 2004], um dos pontos fracos do paradigma de MDD, que representa ainda uma questão em aberto (*open issue*), refere-se a "como desenvolver software utilizando modelos tanto abstratos quanto suficientemente completos que permitam execuções diretas e transformações automáticas em um programa executável para uma plataforma especifica".

4. O Método de Esqueletotipação

Considerando as principais vantagens da utilização do paradigma de MDD e suas limitações, esta seção apresenta o Método de Esqueletotipação como um possível auxílio para minimizar as principais dificuldades existentes e inerentes ao desenvolvimento de software embarcado dirigido a modelos.

A Engenharia de Sistemas prevê, para o desenvolvimento de sistemas embarcados, as fases de elicitação de requisitos, concepção do produto, análise e projeto, bem como a sua divisão em partes sistêmicas, envolvendo software, hardware, partes mecânicas, entre outras. O método proposto de Esqueletotipação concentra-se apenas no software embarcado e, mais especificamente, no seu desenvolvimento e integração com as demais partes dos sistemas embarcados. Além disso, a respectiva aplicação do método deve ser inciada logo após a conclusão das fases de definições da Engenharia de Sistemas.

A aplicação do método de Esqueletotipação não exige a adoção de um processo de desenvolvimento específico, esperando-se apenas que o processo utilizado permita o uso de técnicas de modelagem de software e de ferramentas de I-CASE-E, envolvendo UML, SysML e MDD.

Este método de Esqueletotipação é composto por cinco passos estruturados: 1) diagramação de requisitos; 2) diagramação de casos de uso; 3) aplicação de um padrão de projeto; 4) aplicação de técnicas para geração de códigos-fonte; e 5) aplicação de alterações e mudanças. A Figura 1 ilustra a sua estruturação em passos.

A escolha da forma de implementação do software embarcado encontra-se fortemente interligada com as características do Produto a ser desenvolvido. Segundo [Barr and Massa 2006], os sete seguintes requisitos afetam as escolhas de projeto (*design*) no desenvolvimento de sistemas embarcados: capacidade de processamento; quantidade de memória; número de unidades a serem fabricadas; consumo de energia; custo de desenvolvimento; custo do produto; tempo de vida do produto; e confiabilidade do produto.

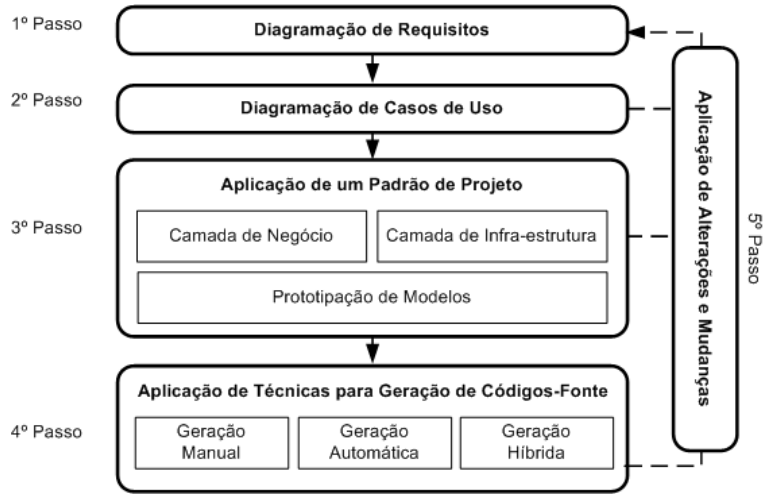


Figura 1. Estrutura do Método de Esqueletotipação

Os autores deste trabalho sugerem que, durante a aplicação deste método em projetos de sistemas embarcados, sejam sempre levadas em consideração: a capacidade de processamento; a quantidade de memória; o custo de desenvolvimento; e o custo do produto. Estas características são fundamentais para a realização de cada passo do método de Esqueletotipação e servem para auxiliar na abordagem do desenvolvimento de software embarcado.

4.1. 1º Passo - Diagramação dos Requisitos

No primeiro passo, a diagramação de requisitos deste método foca na organização e classificação dos requisitos que envolvem o projeto de software embarcado.

Neste passo, sugere-se a utilização do Diagrama de Requisitos da SysML, para representar graficamente os requisitos baseados em textos, além de possibilitar o relacionamento entre eles, definindo suas hierarquias, derivações, satisfações, verificações e refinamentos [OMG 2007].

Este diagrama pode ainda estar contido em outros diagramas, relacionando-se com outros elementos da modelagem. Por exemplo, com o diagrama de Casos de Uso. Segundo [Jacobson et al. 1999], esta funcionalidade, chamada de rastreabilidade, tem como metas ajudar a:

- Compreender a origem dos requisitos;
- Gerenciar as mudanças nos requisitos;
- Avaliar o impacto na mudança de um requisito; e
- Verificar se a implementação satisfaz a todos os requisitos do sistema.

4.2. 2º Passo - Diagramação de Casos de Uso

O segundo passo do método de Esqueletotipação traduz os requisitos para um modelo de casos de uso, representando as funcionalidades externa e internamente observáveis do sistema e suas interações. O modelo de casos de uso é parte integrante da especificação de requisitos, moldando os requisitos do sistema por meio do diagrama de casos de uso.

O diagrama de casos de uso possui em sua composição atores, casos de uso e relações. O caso de uso representa a especificação de uma sequência de interações entre os agentes externos do sistema, definindo o uso de suas funcionalidades. Os atores representam elementos que não fazem parte do sistema, interagindo com ele no envio e na recepção de informações. Completando o diagrama, os elementos podem se relacionar de acordo com os seguintes tipos: comunicação, inclusão, extensão e generalização [Bezerra 2002].

4.3. 3º Passo - Aplicação de um Padrão de Projeto (*Design Pattern*)

O padrão de projeto concebido, considerado uma das principais contribuições desta pesquisa, possui seu foco na organização da modelagem dos componentes de software embarcado em duas camadas bem definidas: a Camada de Negócio e a Camada de Infraestrutura. Diferentemente de outros padrões que visam a organização arquitetural do software, este padrão de projeto visa, além da organização arquitetural, a estruturação dos modelos e componentes de software para transformações em códigos-fonte.

A camada de negócio contém entre outros elementos: a lógica do funcionamento do software, reunindo as regras relativas a solução do problema sistêmico; a definição da ordem de execução das tarefas; e a transformação de dados.

A camada de infraestrutura reúne os procedimentos de controle dos dispositivos de hardware e possui como principal função manter um baixo acoplamento com a camada de negócio, fornecendo e recebendo dados de sensores, atuadores, comunicadores, entre outros dispositivos.

Desta forma, a substituição de um equipamento de infraestrutura ou até mesmo de uma tecnologia, minimizam o impacto de desenvolvimento na camada de negócio do software embarcado, tornando-a independente de plataformas de infraestrutura e conservando, como parte mais valiosa do sistema, as suas regras de funcionamento.

A aplicação do padrão de projeto na modelagem de software embarcado não exclui outros padrões já existentes, que podem ser utilizados em conjunto, atentando-se apenas para que o padrão de projeto deste método de Esqueletotipação seja o último a ser aplicado.

Esta pesquisa considera cada modelo/diagrama como protótipos executáveis [Asur and Hufnagel 1993]. Desta forma, o seu desenvolvimento pode ser testado, incrementalmente, verificando-se o seu funcionamento, a medida que novas funcionalidades são associadas ao projeto. Consequentemente, a execução dos modelos auxilia nos testes visuais de verificação da solução do problema modelado, incluindo testes de caixa-preta, de caixa-branca, entre outros.

Os protótipos executáveis podem ser implementados, segundo dois métodos: de prototipagem descartável (*throwaway*) e de prototipagem evolutiva.

Segundo [Kotonya and Sommerville 1998], a prototipagem do tipo descartável auxilia na validação da implementação dos requisitos e não é utilizada no produto final. Já a prototipagem do tipo evolutiva minimiza o tempo de desenvolvimento do software, podendo ser utilizada no desenvolvimento do produto. No caso deste método de Esqueletotipação, um modelo evolutivo será transformado em códigos-fonte.

A Figura 2 ilustra a aplicação do padrão de projeto. Nela, pode-se observar a divisão entre camadas e a classificação dos protótipos executáveis, por meio de esteriótipos.

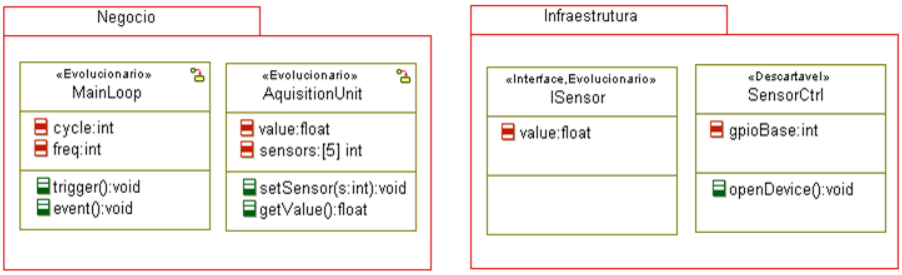


Figura 2. Exemplo de aplicação do padrão de projeto

4.4. 4º Passo - Aplicação de técnicas para geração de códigos-fonte

Concluído o terceiro passo de aplicação do padrão de projeto, o quarto passo inclui a geração de códigos-fonte baseada em modelos que pode ser realizada a partir de uma das três seguintes técnicas de: geração manual, geração automática ou geração híbrida.

A escolha de uma das técnicas encontra-se fortemente ligada com a natureza do projeto de sistema embarcado. Como citado anteriormente, os requisitos de capacidade de processamento, quantidade de memória, custo de desenvolvimento e custo do produto devem ser considerados para a escolha da técnica de geração.

Por exemplo, a geração automática requer uma ferramenta de I-CASE-E para realização desta funcionalidade, podendo impactar no custo de desenvolvimento, considerando também a sua aquisição. Além disto, a ferramenta de I-CASE-E exige um espaço maior de memória para o arquivo binário, impactando no custo do produto. Outra consideração importante é o ganho de produtividade.

Entre outros fatores, essas características devem ser analisadas e julgadas antes da escolha das técnicas de geração de códigos-fonte.

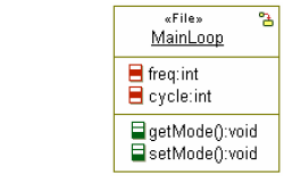
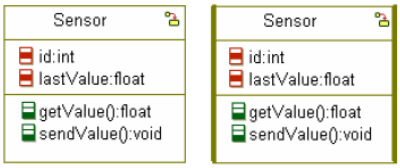
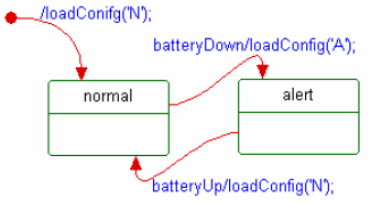
4.4.1. Técnica de Geração Manual

A técnica de geração de códigos-fonte manual consiste na transformação visual da modelagem de software, utilizando-se um roteiro ou guia transformacional para cada elemento do modelo. Esta técnica é indicada para o desenvolvimento de sistemas embarcados com acoplamentos altos ou complexos entre software e hardware. Estes sistemas embarcados apresentam pouco poder de processamento e de memória disponível, porque necessitam de implementações tecnológicas especificas nas interfaces de dispositivos de hardware e de um melhor aproveitamento dos recursos do Sistema Operacional Embarcado - SOE.

Considerando a modelagem do software em camadas e os testes de execução do modelo, a partir deste ponto, basta se realizar a transposição dos modelos para os códigos-fonte. As transformações, neste caso, não devem ser consideradas como uma relação 1 para 1, mas sim como um roteiro ou guia para facilitar a concepção do algoritmo a ser desenvolvido.

A Tabela 1 apresenta as relações entre os elementos do modelo e o guia transformacional para códigos-fonte. Neste trabalho, foi utilizada a ferramenta de I-CASE-E *IBM Rational Rhapsody 7.4*. Note-se que, apesar da utilização da UML, algumas representações gráficas dos modelos poderão ocorrer de forma diferente em outras ferramentas:

Tabela 1. Guia transformacional para os elementos do modelo

Elementos do Modelo	Guia de transformação
 Arquivo - C	O elemento *File* do modelo representa, na linguagem C, um arquivo .c e .h. No primeiro contêiner, encontra-se o nome do arquivo; já no segundo, as variáveis e seus tipos; e no terceiro contêiner, as funções e seus retornos. Mais informações podem ser acessadas, ao se navegar no modelo.
 A B Classe – C++	O elemento a esquerda representa uma classe na linguagem C++. O elemento classe possui, no primeiro contêiner, o seu nome; no segundo, a relação de seus atributos; e no terceiro, os métodos da classe. A representação gráfica distinta significa que a classe A é executada na mesma *thread* do sistema e a B é executada em uma *thread* própria.
 Máquina de Estados Finitos	O elemento Máquina de Estados Finito pode ser transformado em um simples comando *Switch* (em C ou C++), onde cada *Case* representa um estado da máquina. A mudança de transição (*batteryDown/Up*) é representada na variável de escolha do *Swtich,* armazenada em uma Enumeração. As ações da transição (*loadConfig('A/N')*) representam os comandos, dentro de cada *Case*.

4.4.2. Técnica de Geração Automática

Esta técnica consiste na geração automática de códigos-fonte, utilizando-se as regras transformacionais já existentes nas ferramentas de I-CASE-E. Ela é indicada, principalmente, para o desenvolvimento de software embarcado em plataformas de hardware de propósitos mais gerais ou com recursos de processamento e de memória menos escassos.

A geração automática de códigos-fontes pode ser considerada também para a comunicação entre dispositivos de hardware e software, contendo acoplamentos baixos ou simples, como por exemplo: RS-232, RS-485, *Ethernet*, entre outros.

4.4.3. Técnica de Geração Híbrida

A aplicação desta técnica no quarto passo do Método de Esqueletotipação, considerada também como uma das principais contribuições desta pesquisa, reúne os pontos fortes das gerações de códigos-fonte manual e automática, de forma sincronizada, no desenvolvimento de software embarcado.

Nesta técnica, a geração manual é fortemente indicada para a camada de infraestrutura, visando um melhor controle de dispositivos de hardware com acoplamentos altos ou complexos com o software. Já a geração automática ajusta-se melhor para transformação da camada de negócio em códigos-fonte. No entanto, não existem restrições para que um elemento da camada de negócio seja gerado pela técnica de geração manual e vice-versa.

4.5. 5º Passo - Aplicação de Alterações e Mudanças

O quinto e último passo do Método de Esqueletotipação pode ser considerado como um passo alternativo utilizado no decorrer da aplicação deste método no desenvolvimento de software embarcado. A função principal deste passo é a de garantir o sincronismo do projeto em relação as alterações e mudanças necessárias para o sistema de software.

5. Estudo de Caso

A verificação da aplicação do Método de Esqueletotipação foi realizada por meio de um estudo de caso que contempla uma parte de um projeto real denominado Projeto de Integração e Cooperação Amazônica para a Modernização do Monitoramento Hidrológico - ICA-MMH. Este projeto, financiado pela Financiadora de Estudos e Projetos - FINEP, encontra-se em fase final de desenvolvimento no Instituto Tecnológico de Aeronáutica - ITA, em parceria com a Agência Nacional de Águas - ANA e com o Instituto de Pesquisas Hidráulicas - IPH.

O objetivo geral do Projeto ICA-MMH é desenvolver, para uma região hidrográfica de referência, um sistema-piloto que contemple a modernização e a integração de pontos de coletas telemétricas de dados hidrometereológicos.

O estudo de caso envolveu a aplicação do Método de Esqueletotipação em uma parte significativa do desenvolvimento do software embarcado da Plataforma de Coleta de Dados - PCD do Projeto ICA-MMH, responsável pela aquisição e transmissão de dados hidrometereológicos via satélite, celular ou rádio-frequência.

5.1. O Projeto ICA-MMH

Na concepção sistêmica do Projeto ICA-MMH, realizada a partir de uma análise de requisitos, concebeu-se um cenário com 5 grandes grupos de requisitos com propósitos comuns e coesos, possibilitando a formação de 5 sistemas que se constituíram em um sistema maior, o Sistema de Sistemas ICA-MMH (SdS ICA-MMH).

Os seguintes Sistemas compõem o SdS ICA-MMH: Sistema de Aquisição de Dados - SAD; Sistema de Tratamento de Dados - STD; Sistema de Banco de Dados - SBD; Sistema de Monitoramento, Controle e Apoio à Decisão - SMCAD; e Sistema de Difusão de Dados - SDD.

O Sistema de Aquisição de Dados é composto pelo Sistema Embarcado da PCD e por um Sistema de Software para recepção de dados hidrometereológicos e envio de configurações remotas.

5.2. A Plataforma de Coleta de Dados - PCD

A PCD disponibiliza informações hidrometereológicas, visando um melhor acompanhamento das condições hidrológicas e uma melhor previsão de eventos extremos tais como secas, inundações, entre outros.

Cada PCD deve ser capaz de propiciar a transmissão dos dados coletados via um desses meios de comunicação previstos: celular, rádio-frequência ou satelital. A redundância de comunicação deve ser controlada por um algorítimo de comutação automática dos meios de comunicação.

De acordo com as especificações sistêmicas do Projeto ICA-MMH, o software embarcado da PCD deve ser desenvolvido em um *Kit de Desenvolvimento ARM9 - Congent CSB 637*. Esta plataforma possui um microprocessador AT91RM9200 de 184Mhz, com 64 *MByte* SDRAM e 8 *MByte* de memória *flash*, com a utilização de um Sistema Operacional Embarcado - SOE Linux.

5.3. Aplicação do Método de Esqueletotipação no Projeto da PCD

A partir das definições sistêmicas do Projeto ICA-MMH, das divisões dos sistemas, das definições da plataforma embarcada, entre outras definições, tornou-se possível aplicar o Método de Esqueletotipação no sistema embarcado do Projeto.

Para a aplicação neste Estudo de Caso, considerou-se as seguintes funcionalidades da PCD: modos de operação (normal, alerta e crítico); atualização dos parâmetros de configuração; e aquisição de dados dos sensores (pluviômetro, sonda multiparamétrica e pressão barométrica).

Inicialmente, despendeu-se algum tempo considerado irrelevante para a configuração e integração do ambiente de ferramentas de I-CASE-E. As principais integrações ocorreram entre a ferramenta de gerência de requisitos *IBM Rational RequisitePro 7.1* e a ferramenta de modelagem de software embarcado *IBM Rational Rhapsody 7.4*.

5.3.1. Aplicação do 1º Passo no Projeto da PCD

Antes de se iniciar a diagramação de requisitos, realizou-se a importação para a ferramenta de modelagem, dos requisitos textuais contidos na ferramenta de gerência de requisitos. A partir da relação dos requisitos do SAD, decorrentes da importação anterior, envolvendo o Projeto da PCD, realizou-se a diagramação dos requisitos e as definições das relações entre eles. A Figura 3 ilustra a representação gráfica dos requisitos diagramados.

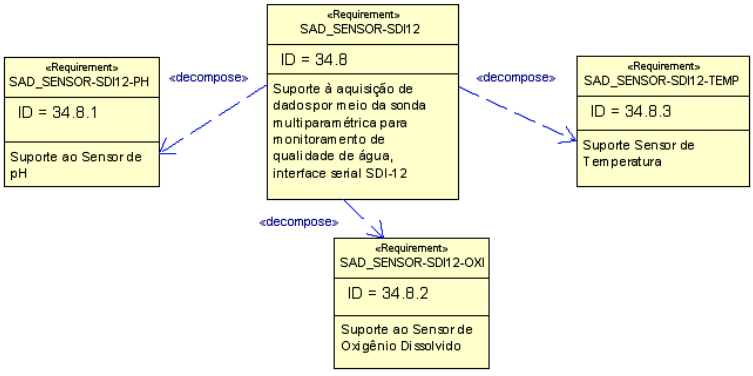


Figura 3. Diagramação dos Requisitos do software embarcado da PCD

5.3.2. Aplicação do 2º Passo no Projeto da PCD

Após a realização da organização dos requisitos, a diagramação dos modelos de casos de uso foi elaborada, de forma relativamente simples, identificando-se os atores externos do sistema, as suas conexões e as interações entre o sistema.

A partir deste ponto, os requisitos diagramados puderam ser relacionados com os casos de uso, obtendo-se a rastreabilidade do requisito, tornando possível a verificação dos requisitos contemplados pelo sistema.

A Figura 4 ilustra um exemplo do modelo de casos de uso relacionado com os requisitos, por meio da ligação *trace*.

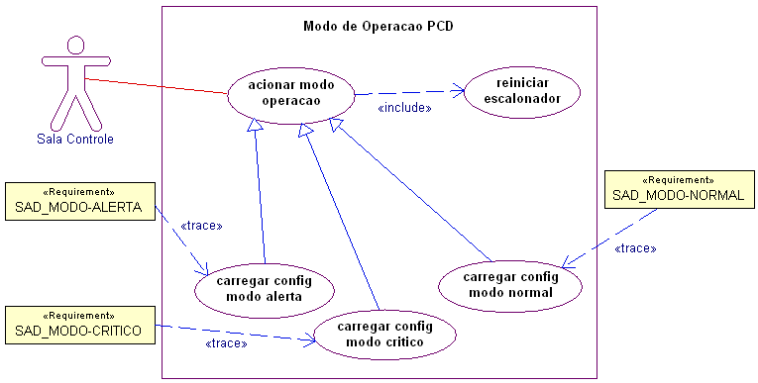


Figura 4. Diagramação dos Casos de Uso do software embarcado da PCD

5.3.3. Aplicação do 3º Passo no Projeto da PCD

Considerando o escopo dos requisitos e seus detalhamentos nos modelos de casos de uso, pôde-se iniciar a aplicação do terceiro passo.

Este passo, considerado uma das principais contribuições da pesquisa, consistiu na aplicação do padrão de projeto concebido, que dividiu o software embarcado em camadas de negócio e infraestrutura. Durante a concepção da modelagem, observou-se a divisão das camadas e a classificação dos modelos/protótipos em evolucionário ou descartável, ambas previstas no terceiro passo do Método de Esqueletotipação. A classificação auxiliou na identificação de quais elementos deveriam participar da geração dos códigos-fonte.

Nesta oportunidade, constatou-se que os modelos puderam ser executados no ambiente *host*, para fins de simulação e verificação da solução em desenvolvimento. Constatou-se também que algumas das ferramentas de I-CASE-E possuíam a capacidade de integração com outras ferramentas de modelagem. Por exemplo, a ferramenta de I-CASE-E *Rhapsody 7.4* pôde ser integrada com a ferramenta de modelagem *MathWorks Matlab Simulink*, viabilizando possíveis futuras simulações e verificações de integrações de produtos.

No Projeto da PCD, existem alguns sensores hidrometereológicos que possuem interfaces de acoplamentos altos ou complexos com o hardware. Neste caso, constatou-se, durante a aquisição de dados, via *General Purpose Input/Output* - GPIO ou via con-

versores Analógicos/Digitais - A/D, que estas interfaces não conseguem ser bem representadas pela UML e as tentativas de representações são todas baseadas em adaptações de baixa qualidade dos elementos do modelo. Além disso, constatou-se também que a implementação desta interface torna-se mais vantajosa, ao se considerar a utilização de técnicas de SOE. Por exemplo, quando se considera a implementação de módulos do *kernel* (*devices drivers*).

A Figura 5 ilustra os elementos de modelagem da PCD e suas classificações, quanto aos modelos evolucionários ou descartáveis. Neste caso, os sensores que possuem interfaces de baixo nível foram considerados descartáveis.

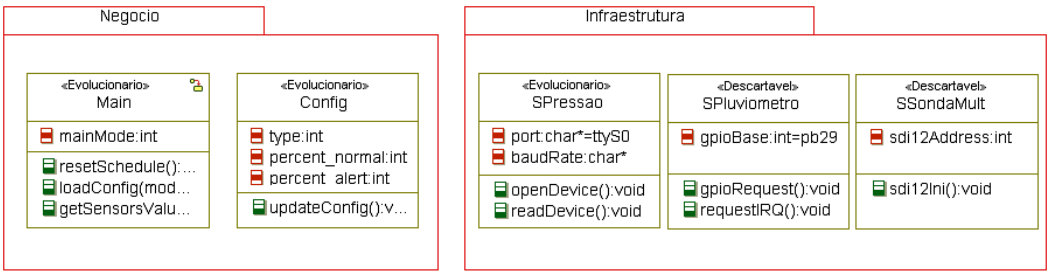


Figura 5. Aplicação do padrão de projeto na modelagem do software embarcado da PCD

5.3.4. Aplicação do 4º Passo no Projeto da PCD

Após a modelagem do software embarcado, a aplicação do quarto passo consistiu na escolha de uma das técnicas de geração de códigos-fonte.

Este passo, também considerado com uma das principais contribuições desta pesquisa, consistiu na aplicação da técnica de geração automática. Esta técnica foi considerada, inicialmente, como a mais adequada para o Projeto da PCD pelas suas características e especificações da plataforma de hardware. Porém, ao se constatar a existência de alguns sensores com necessidades de implementações especiais, a técnica de geração híbrida mostrou-se mais eficiente para a geração de códigos-fonte baseados em modelos.

Desta forma, toda a modelagem da camada de negócio foi gerada automaticamente pela ferramenta *Rhapsody 7.4*, incluindo os elementos de interface e os sensores com acoplamentos baixos ou simples entre o software e o hardware da camada de infraestrutura. Quanto aos demais sensores, foram construídos baseados nas técnicas de geração manual de códigos-fonte.

A Figura 6 apresenta, no item A, as configurações necessárias para a geração automática de códigos-fonte na ferramenta de desenvolvimento. Esta figura, no item B, apresenta a implementação do módulo do *kernel* do SOE utilizado no Projeto da PCD.

6. Conclusão

A aplicação do Método de Esqueletotipação no desenvolvimento da Plataforma de Coleta de Dados - PCD do Projeto de Integração e Cooperação Amazônica para a Modernização do Monitoramento Hidrológico - ICA-MMH mostrou-se satisfatória, principalmente

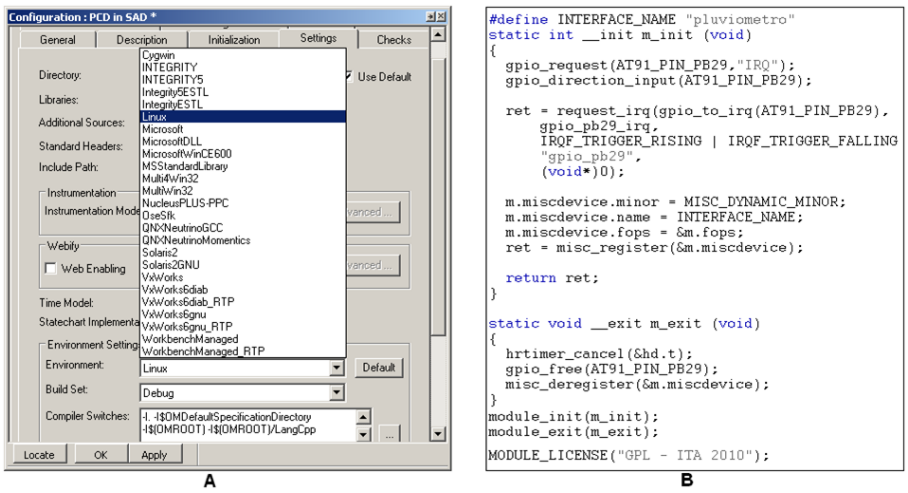


Figura 6. A - Configuração da ferramenta de I-CASE-E e B - Implementação do módulo do *kernel*

quanto a possibilidade de documentação, via modelos, de todos os elementos de software e de interfaceamento de hardware.

Este artigo apresentou como principais contribuições: a concepção de um método denominado de Esqueletotipação para o desenvolvimento de Software Embarcado; o desenvolvimento e a aplicação de um padrão de projeto (*design pattern*), no terceiro passo deste método; e a elaboração e aplicação de uma técnica de geração híbrida de códigos-fonte, no seu quarto passo.

A aplicação de um padrão de projeto permitiu uma melhor divisão e delimitação das fronteiras entre as camadas de negócio e de infraestrutura necessárias ao funcionamento do sistema. Desta forma, qualquer dispositivo de infraestrutura (sensores, atuadores, comunicadores, entre outros) passou a poder ser substituídos por dispositivos equivalentes, sem causar mudanças e/ou adequações à camada de negócio do sistema.

Finalmente, a geração híbrida de códigos-fonte permitiu um melhor aproveitamento dos recursos do Sistema Operacional Embarcado - SOE, considerando a abordagem de geração manual para a camada de infraestrutura. Com isto, constatou-se um controle mais eficiente dos dispositivos de hardware do tipo *device drivers*, durante a implementação dos módulos do *kernel* do Sistema Operacional Embarcado.

7. Agradecimentos

Os autores deste artigo agradecem as seguintes Instituições que apoiaram, direta ou indiretamente, a realização deste trabalho: Comando-Geral de Tecnologia Aeroespacial - CTA; Instituto Tecnológico de Aeronáutica - ITA; Agência Nacional de Águas - ANA; Financiadora de Estudos e Projetos - FINEP; e Fundação Casimiro Montenegro Filho - FCMF.

Referências

Asur, S. and Hufnagel, S. (1993). Taxonomy of rapid-prototyping methods and tools. *Rapid System Prototyping, 1993. Shortening the Path from Specification to Prototype. Proceedings., Fourth International Workshop on*, pages 42–56.

Azevedo, R. d. C. (2008). Colibri: Um framework colaborativo para engenharia de sistemas embarcados baseada em modelos híbridos. Master's thesis, Instituto Tecnológico de Aeronáutica.

Barr, M. (1999). *Programming embedded systems in C and C++*. O'Reilly.

Barr, M. and Massa, A. (2006). *Programming embedded systems: with C and GNU development tools*. O'Reilly.

Bezerra, E. (2002). *Princípios de Análise e Projeto de Sistemas com UML*. Campus.

Cunha, A. M. d. (2006). Notas de aula da disciplina ce-235 sistemas embarcados de tempo real, do programa de pós-graduação em engenharia eletrônica e computação, na Área de informática - pg/eec-i. Instituto Tecnológico de Aeronáutica - ITA.

Douglass, B. P. (2003). *Real-Time Design Patterns: robust scalable architecture for Real-time systems*. Addison Wesley.

Douglass, B. P. (2004). *Real Time UML Third Edtion - Advances in The UML for Real-Time Systems*. Person Education, Inc.

Haywood, D. (2004). Mda: Nice idea, shame about the... Disponível em: http://www.theserverside.com/news/1365166/MDA-Nice-idea-shame-about-the.

Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley.

Kossiakoff, A. and Sweet, W. N. (2003). *Systems Engineering Principles and Practice*. John Wiley & Sons.

Kotonya, G. and Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques*. Wiley.

Li, Q. and Yao, C. (2003). *Real-Time Concepts for Embedded Systems*. CMP Books, Lawrence, KS, USA.

Loubach, D., Ramos, D., Saotome, O., and da Cunha, A. (2008). Comparing source codes generated by case tools with hand coded. *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 1292–1292.

OMG (2007). Omg systems modeling language (omg sysml tm). http://www.sysmlforum.com/.

Ramos, D. B., Loubach, D. S., and da Cunha, A. M. (2008). Developing a distributed real-time monitoring system to track uavs. *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 4.C.6–1–4.C.6–9.

Taurion, C. (2005). *Software Embarcado - A nova onda da Informática*. Editora Brasport.

UML (2003). Profile for schedulability, performance and time, ptc2003-03-02. http://www.omg.org/.

Evaluation of the impact of code refactoring on embedded software efficiency

Wellisson G. P. da Silva[1], Lisane Brisolara[1], Ulisses B. Corrêa[2], Luigi Carro[2]

[1]Universidade Federal de Pelotas (UFPEL) - Pelotas – RS – Brazil

[2]Universidade Federal do Rio Grande do Sul (UFRGS) - Porto Alegre – RS – Brazil

wguilhermino@gmail.com, lisane.brisolara@ufpel.edu.br,
ubcorrea@inf.ufrgs.br, carro@inf.ufrgs.br

Abstract. *The increasing complexity of embedded software and the hard time-to-market requirements, motivate to the use of object-oriented languages. However, this usage can negatively impact on energy consumption as well as on performance. Code refactoring are techniques that change the code in order to improve the software quality. This paper analyzes how the inline method refactoring, a software optimization technique, can impact on the performance and energy of embedded software written in Java. Three different applications are evaluated in order to discuss this impact.*

1. Introduction

Complex embedded systems have hard constraints regarding performance, memory, and energy and power consumption [1]. Although these physical properties are more closed to hardware, the way the software interacts with system resources has impact on power and energy consumption as well as on performance [2]. Moreover, the embedded system domain is driven by cost and time-to-market factors [3], which also influence the design decisions taken by embedded software developers.

To handle the increasing complexity of embedded software and the hard time-to-market requirements, the use of object-oriented languages became more important mainly due its modular and reusable code. However, object-oriented (OO) languages can introduce penalties to system power and energy consumption and performance [3]. In this scenario, embedded software designers should handle the software complexity and produce efficient software at the same time.

Refactoring is a software engineering technique that modifies the code to improve its readability and maintainability without changing its computation [4]. Method inline is a common refactoring method as well as a well-known code optimization technique. This work discusses the impact of the method inline on system performance and energy consumption when it is applied to Java codes. The objective is help designers to understand how OO practices affect these physical properties and evaluate if refactoring is a valid strategy to explore the embedded software design space.

The remaining of this paper is organized as follows. Section 2 gives the background. Section 3 presents the used methodology and target platform, while results are discussed in Section 4. Conclusions and future work are presented in Section 5.

2. Background

There are three main sources of power consumption in embedded systems [3], which are: processor power consumption, due to the processor activity, memory power consumption, for accessing data and instructions in memory, and the power consumption to connect the processor and memories. All these operations should be taken in account, when developing embedded software. Moreover, some software engineering practices can directly impact system power consumption and performance, like code refactoring.

Refactoring is a process of modifying the code to make it easier to understand and modify without changing its computation as defined in [4]. These code changes can also affect the system performance and energy consumption, since it modifies the instructions that will be used. Examples of changes that can be done are Extract Method – which creates a method to represent duplicate code – or Inline Method – which exchanges a method call for its body. Inline is not recommended by the software engineering best practices, because it can make the code hard to understand and so decrease code maintainability. However, the Inline Method is expected to increase performance since it removes the overhead of a method call – each method call in the Java Virtual Machine has a cost associated [5]. This work has as objective to analyze how method inline impacts performance and energy consumption for Java codes.

3. Methodology and Target Platform

Through dynamic profiling of the application code, information about the number of method calls are obtained then we apply the method inlines incrementally. In this way, the first most frequently called method is inlined in the first iteration (modifying original code and generating *Iteration1* version) and the second one is inlined generating the *Iteration2* version, and so on. Thus, the gains achieved by the reduction of method invocations can be increased. After that, several versions of the same application are generated.

In order to obtain performance and energy consumption for these several Java code versions, an estimation tool called DESEJOS was used [6]. As embedded platform, we adopted the FemtoJava [7] processor. This processor is a stack based Java Virtual Machine implementation, executing Java bytecodes natively. We chose FemtoJava and DESEJOS due to their use of Java, a language that is gaining attention on the embedded community.

The FemtoJava processor implements a stack machine compatible with the Java Virtual Machine (JVM) specification and that is able to execute Java code in hardware. Two different versions of the FemtoJava processor are available: multicycle and pipeline. In this experiment, we adopted the multicycle version that is targeted to low power embedded applications.

4. Experimental Results

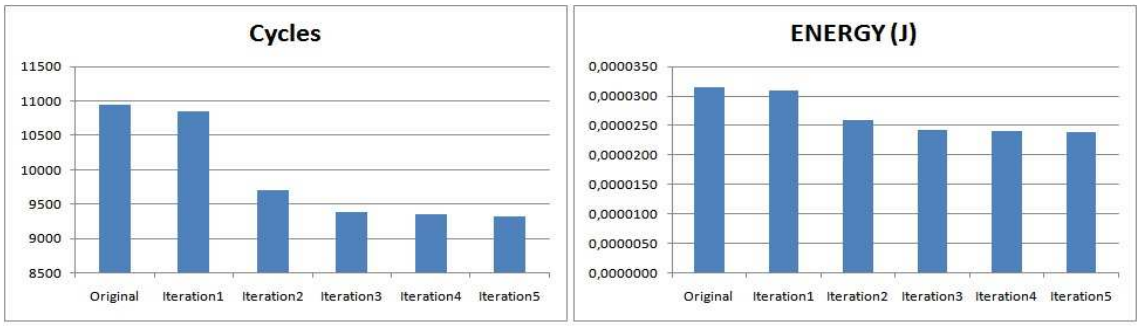
Three applications are used as case studies, an address book implemented with a hash table, a game called Sokoban, and also an Mpeg layer-3 audio decoder from the Spec jvm2008 benchmark set [8]. Table 1 presents values of metrics obtained by the Eclipse IDE Plug-in called Metrics [9] for each application. These metrics gives an idea about the complexity of the studied applications. According to these metrics, the MPEG

decoder is the used application with higher complexity, since it presents the higher number of classes, objects, and packages, besides the higher McCabe ciclomatic complexity, well-know measure for the complexity of a program.

Table 1. Complexity of the applications

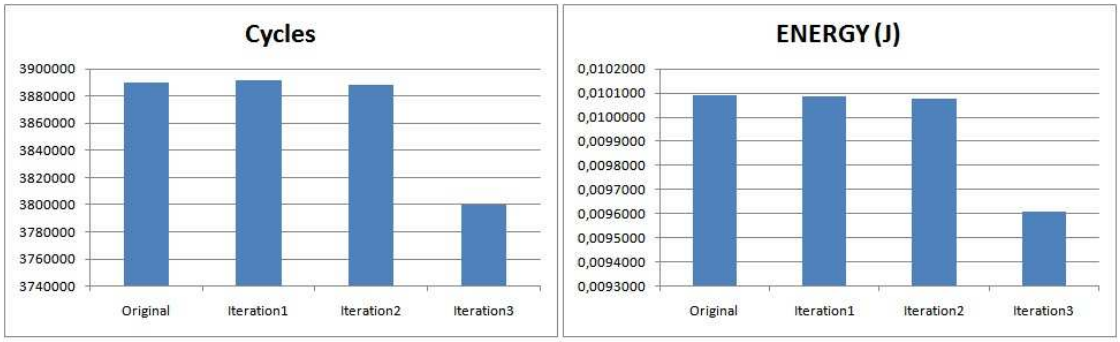
	Address Book	Sokoban	MPEG Decoder
Number of classes	2	7	51
Number of methods	31	49	184
Number of packages	1	2	80
McCabe Ciclomatic Complexity	37	109	120

Figure 1 (a) and (b) and Figure 2 (a) and (b) present performance (in cycles) and energy consumption (in Joule) for the different code versions, generated by inline refactoring, of the Address book and of the Sokoban, respectively. These results show that this refactoring method achieved improvements in performance and energy consumption for both applications.



(a) Performance results of the **Address book** (b) Energy consumption results of the **Address book**

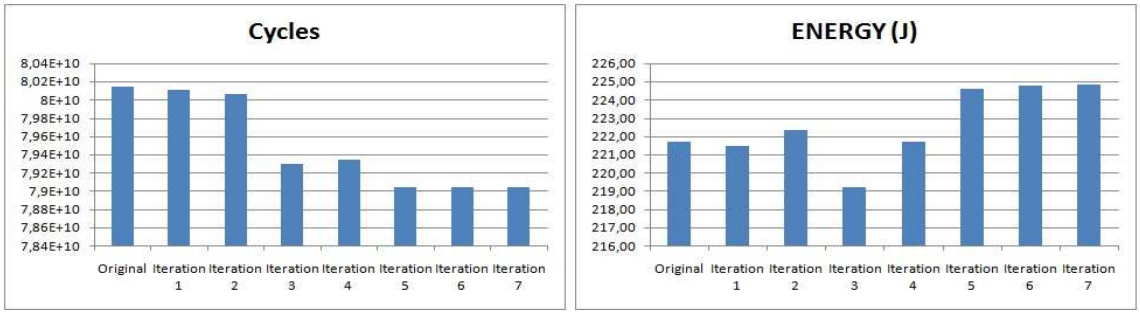
Figure 1: Performance and Energy consumption results for the Addrees Book



(a) Performance results of the **Sokoban** (b) Energy consumption results of the **Sokoban**

Figure 2: Performance and Energy consumption results for the Sokoban

Figure 3 (a) and (b) illustrates the results for performance (in cycles) and energy consumption (in Joule) achieved for the different code versions of the MPEG decoder. The performance results show that the cycles required by the application are reduced when the inline method is applied, as expected, since cycles used to create frames for each method call are eliminated. However, the energy results of the MPEG-Audio decoder were not as expected (see Fig. 3b). After the third refactoring iteration, which has a greater gain from its antecessors, all remaining versions have higher consumption and the last ones have a consumption worse than the original code (without any inline applied).



(a) Performance results of the **MPEG decoder** (b) Energy consumption of the **MPEG decoder**

Figure 3: Performance and Energy consumption results for the MPEG decoder

To better analyze the achieved results for energy consumption, we have analyzed the Java bytecodes and have obtained the instruction histogram for the execution of the different code versions in which the results are not the expected. Summarized results of these histograms can be observed in Table 2 and Table 3.

Table 2 shows the instructions that presented different number of calls between the *Iteration2* and *Iteration3*, whose versions present the lowest energy consumption (see Fig. 3b). A reduction in the number of *iload_1* instructions can be observed in the *Iteration3* results in Table 2, as well as an equivalent increase in the number of *iload* instructions. These instructions require one more access to memory than the *iload_<n>* instructions (like *iload_0*, …, *iload_3*). This extra memory access is used to load the index of which variable in the pool will be popped on the JVM operand stack, unlike the simpler *iload_<n>* instructions that has this address implicitly defined. Moreover, these results show reductions in the number of *invokevirtual* and *ireturn* instructions that represents an invocation of a class instance method in Java and the return instruction, respectively. These reductions were expected, since the inline removed method calls. Besides, a great reduction in the number of the *aload_0* instructions can be observed in Table 2.

Table 2. Main instructions used for the Iteration 2 and 3

Instruction	Iteration 2	Iteration 3	Difference
iload	2256402932	2284795276	28392344
iload_1	117059899	88667555	-28392344
aload_0	1058245488	916283768	-141961720
istore	372594809	400987153	28392344
istore_1	28458885	66541	-28392344
ireturn	41305600	12913256	-28392344
invokevirtual	37981918	9589574	-28392344

To observe the increasing on energy provoked by inline, the difference between the third and fourth iterations is analyzed. Table 3 shows the number of used instructions for each code version (Iteration 3 and Iteration 4) and the difference between these numbers. The results show a great increase in the number of *iload* instructions and also a decrease in *iload_2* instructions, and yet an increase in the number of *getfield* instructions. Moreover, Table 3 shows the effect of the inline method, where the number of method calls is decreased, provoking a reduction in the number of *invokevirtual* and *return* instructions. In addition to that, the inline affected

the number of method´s local variables causing a changing of position in the variable pool. This changing can be noted in the variation of *istore* and *istore_1* instructions. With the method scope increasing the variable that occupied the second position in the variable pool went to a new position after the fourth position, generating the necessity of an *istore* instruction. The increase of *iload* is a consequence of the inline method, which inserted more local variables to the method and the instructions *iload_0, iload_1, iload_2* and *iload_3* cannot be used.

Table 3. Main instructions used for the Iteration 3 and 4

Instruction	Iteration 3	Iteration 4	Difference
iload	2284795276	2383364176	98568900
iload_2	289277504	190708604	-98568900
aload_0	916283768	939206768	22923000
aload_1	203437622	178986422	-24451200
istore	400987153	401751253	764100
istore_2	16634671	15870571	-764100
return	6336716	5572616	-764100
getfield	1169270767	1241096167	71825400
invokevirtual	9589574	8825474	-764100

The experiments show that for more complex application with more complex methods, as the MPEG decoder (the most complex application from Table 1), the inline method cannot achieve the expected improvements for energy or performance. Loss in energetic efficiency occurs when after inlining, the new method body/scope increase too much then simpler instructions have to be changed by more complex instructions. Moreover, indiscriminate inlining generates an increasing in the number of instructions related with Object Oriented Programming (like *getfield*).

5. Conclusions and future work

This paper studies the impact on the embedded system performance and energy consumption when the method inline refactoring is applied to Java codes. The Inline Method was expected to increase performance and decrease energy consumption since it would reduce the number of cycles to create a frame in the Java Virtual Machine for every method call. The expected results were achieved for the Address Book and Sokoban applications, but not for the MPEG decoder, which is the most complex analyzed application. Moreover, the results have shown that when applying inline in a method, the complexity of the method and whole application should be taken in account. It is because the reduction of a method call does not result in a gain when the method has many variables to be addressed in the variable pool.

As future work, we plan to explore other refactoring methods and case studies.

References

[1] Graaf, B.; Lormans, M.; Toetenel, H. “Embedded Software Engineering: the State of the Practice”. IEEE Software, v. 20, n. 6, p. 61- 69, Nov. – Dec. 2003.

[2] Saxe, E. “Power Efficient Software”. Communication of the ACM. v. 53, n. 02, Feb. 2010.

[3] Chatzigeorgiou, A. and Stephanides, G. “Evaluating Performance and Power of Object-Oriented vs. Procedural Programming in Embedded Processors”. Proc. of International Conference on Reliable Software Technologies, Ada-Europe 2002, Vienna, Austria, June 17-21, 2002.

[4] Fowler, M. “Refactoring: Improving the Design of Existing Code”, Addison Wesley, 14th printing, 2004.

[5] Sun Microsystems, Inc. “The JavaTM Virtual Machine Specification”, http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html, March, 1999.

[6] Mattos, J.C.B., Carro, L. “Object and Method Exploration for Embedded Systems Applications”. Proc. of Symposium on Integrated Circuits and Systems Design, SBCCI 2007, Rio de Janeiro, Brazil, 2007.

[7] Ito, S. A., Carro, L., & Jacobi, R. P. “Making java work for microcontroller applications”. IEEE Design & Test of Computers, v.18, n. 5, p.100-110, 2001.

[8] SPECjvm2008 (Java Virtual Machine Benchmark), http://www.spec.org/jvm2008.

[9] Metrics Eclipse Plug-in, http://metrics.sourceforge.net/.

Development Process for Critical Embedded Systems

L.B. Becker[1], J.-M. Farines[1], J.-P. Bodeveix[2], M. Filali[2], F. Vernadat[3]

[1]Dept of Automation and Systems – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88040–900 – Florianopolis – SC – Brazil

[2]IRIT-CNRS, Université de Toulouse
Toulouse, France

[3]LAAS-CNRS, Université de Toulouse
Toulouse, France

{lbecker,farines}@das.ufsc.br, {bodeveix,filali}@irit.fr, francois@laas.fr

Abstract: Designing safety critical systems is a complex task due to the need of guaranteeing that the resulting model can cope with all the functional and non-functional requirements of the system. Obtaining such guarantees is only possible with the use of model verification techniques. This paper presents an approach aimed to fulfill the needs of critical system design. The proposed approach is based on the Architecture Analysis and Design Language (AADL), which is suitable to describe the system's architecture. It contains a sequence of model transformations that easies the verification of the designed AADL model and so assures its correctness. It must be highlighted that this is not performed in a single step, as it is possible to verify AADL models with different abstraction levels, which allows successive refinements in a top-down approach. We use a case study from an Autonomous Parking System to illustrate the proposed development process.

Keywords: safety-critical systems, design approach, model-verification

1. Introduction

Modern safety-critical systems are getting more and more complex and, at the same time, have become indispensable nowadays. Almost every system that in the past was simply mechanic (e.g. cars, trains, airplanes) is now equipped with an embedded computing systems. Also, most of the times, such systems are safety-critical.

In order to handle such increasing complexity it is necessary to use a development process based on System Engineering. These techniques should both facilitate the modeling discipline and provide model-verification facilities. Model-verification is crucial for safety-critical systems design because it allows guaranteeing that the designed model respect the application requirements and constraints.

In this context, the Architecture Analysis & Design Language (AADL) [Feiler et al. 2006] seems to be a suitable choice. AADL is a modeling language that allows early analysis of a system's architecture. It supports the modeling of both software and hardware components in a hierarchical manner using a set of components connected through ports. AADL defines properties that can be attached to modeling elements in order to give an abstract specification of the dynamic architecture of the system. Real-time constraints are attached to threads, ports, buses, and processors (e.g. dispatch protocol, period, deadline, processing power, hardware-software mapping, etc). The AADL lan-

guage can also be extended by defining new properties or by attaching specific languages to some elements.

Although AADL precise semantics makes it suitable for model verification, how to perform such a task is still an open question. For this reason, we present in this paper a solution that overcomes this problem. Our approach consists in supporting model verification taking into account irregular behaviors and data. Another important feature from our proposal is that it follows the Model Driven Engineering (MDE) principles, as design is intended to remain in high-level abstraction levels and does not need to worry about the low-level details from the performed model transformations.

We can say that the proposed process supports the safe design of the system's architecture, once the resulting system architecture goes through several verification steps in order to assure its correctness. To reach this goal it is performed a sequence of model transformations, maintaining the principles of MDE. It starts with an AADL model and finishes with an automaton model that can be verified.

The rest of the paper is structured as follows: Section 2 discusses some related works. Section 3 gives a brief introduction to the AADL language. Section 4 presents the proposed development process and our autonomous parking case study. Section 5 presents the techniques and toolset used to verify temporal properties of AADL models. Finally, section 6 draws the conclusions of this work.

2. Related Methodologies and Tool Support

Designing new generations of embedded real-time systems is so complex that became mandatory to work with higher abstractions (namely computational models) previous to implementation. The Model Driven Engineering (MDE) [Schmidt 2006] is, for instance, an initiative to help developers to manage software development complexity using models at the very beginning, and with different abstraction levels. The key aspect from this technology is the design of models that are decoupled from their target platform. Among the main benefits of the emerging MDE approach it should be highlighted its enhanced possibilities for early model verification.

In fact, many recent tools have been proposed to support different kinds of verification. With respect to our concerns, timing verification tools have been an active area of research over these last years. It is interesting to remark that although most of these tools are based on existing theoretical models, e.g., timed automata, Petri nets, the limitations (especially with respect to combinatorial explosion and scalability) of which are well known, the effort has been undertaken to achieve them. In fact, it is hoped that first, the abstraction and the structure brought by the model driven approach and second, the adoption of a specific execution model will help to struggle against these limitations. Along these lines, we can cite the Cheddar [Dissaux and Singhoff 2008] scheduling tool which proposes dedicated analysis for the AADL execution model. Currently, it considers mainly analytical models. Future versions should take into account more detailed behavior descriptions [Franca et al. 2007]. The tools Uppaal Port [Håkansson et al. 2008] and Pola[Berthomieu et al. 2007] are based on the traditional model checking approach. Uppaal Port is based on timed automata and supports component based development. In order to reduce the combinatorial explosion Uppaal Port adopts a synchronous like execution model which restricts interleaving of the asynchronous approach. Moreover, it

proposes partial order techniques for reducing space explorations. The tool Pola is based on timed Petri nets, and it proposes specific support for the AADL execution model.

3. A Brief Overview of AADL

AADL is an architecture design language standardized by the SAE. This language has been created to be used in the development of real time and embedded systems. As a successor of MetaH, AADL capitalizes more than 10 years of experiments. MetaH is a language developed by Honeywell Labs and used in numerous experiments in avionics, flight control, and robotic applications. AADL also benefits from the knowledge on ADLs acquired at CMU during the development of several ADLs, like ACME and Wright.

AADL contains all the standard concepts of any ADL: components, connectors used to describe the interface of components, and connections used to link components. The set of AADL's components can be divided in three partitions: the software components (process, thread, thread group, subprogram, and data), the hardware components (processor, bus, memory, device), and a system component. Components can communicate through ports, synchronous calls, and shared data. A process represents a virtual address space, or a partition, this address space includes the program defined by its subcomponents. A process must contain at least one thread or thread group. A thread group is a logical organization of threads in a process. A thread represents a sequential flow of execution, it is the single AADL component that can be scheduled. A subprogram represents a piece of code that can be called by a thread or another program. A data models a static variable used in the code, they can be shared by threads or processes.

A processor is an abstraction of the hardware and the software in charge of the scheduling and the execution of threads. The memory represents any platform component that stores data or binary code. The buses are communication channels used to connect different hardware components. The devices represent interfaces between the system described and its environment.

Systems allow composing software components with hardware components. The interactions can be defined at a logical and a physical level. At a physical level, software components are associated to hardware components, a thread to a processor, or a data to a memory for example. The logical level is used to describe the communication between hardware and software. At a logical level we can define communication connections between processors or devices and software components.

AADL uses the notion of mode to determine a set of active components. This mechanism allows describing dynamic architectures through a static set of components. We consider here the behavior annex [Franca et al. 2007] attached to threads or devices, which is used to specify an abstract behavior for these components, allowing to make data dependent analysis.

4. The Proposed Development Process

This section presents our proposed development process for critical embedded systems. It is possible to say that this process supports the safe design of the system's architecture using MDE's principles. By safe design we mean that the resulting system architecture goes through several verification steps in order to assure its correctness. To reach this

goal it is performed a sequence of model transformations, which starts with an AADL model and finishes with an automaton model that can be verified. This section skips the details of the verification chain (which is covered in the next section) and concentrates in the high-level steps of the proposed process, which are shown in Figure 1.

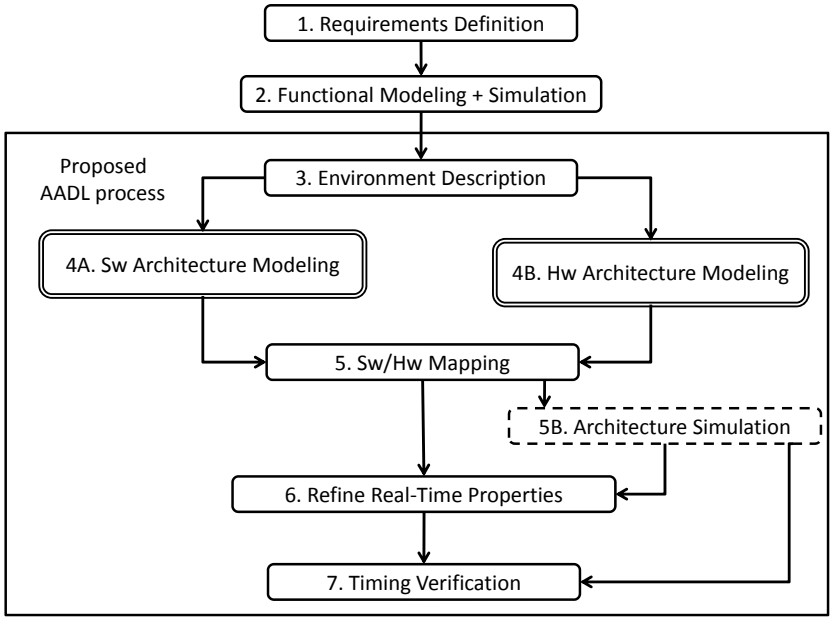


Figure 1. Proposed Design Flow

We understand that, as in any system development, the initial step is the definition of the functional and non-functional requirements of the system, resulting in a set of requirements. Then it is followed by the design of a functional model for the system (e.g. Lustre or Simulink model). The proposed process itself starts in step-3 with the design of the AADL model, providing the specification of the external devices (environment) that interact with the system. step-4 is split in two parts: (4A) software architecture modeling/verification and (4B) hardware architecture modeling. The overall result here should be an AADL model with basic properties already verified and a hardware architecture potentially capable to run the designed software model. In step-5 a mapping from the modeled software components to the hardware model is performed. The result is a complete AADL model. In step-6 it is suggested that the real-time properties of the AADL model should be updated with the precise timing information coming from the simulation of the software in the target platform, which is conducted in step-5B. The proposed development process is concluded in step-7 with the final model verification, which uses as input the AADL model updated with the precise timing information. After that, it should be possible to make automatic code-generation of the application.

It is important to highlight that the design flow among the steps is not unidirectional. Every time that a verification step fails the designer should either backtrack to higher abstraction levels of the AADL model and its properties or change assumptions made in earlier levels. For example, if there is an error in the timing verification (step-7), then the designer should be able to judge if the problem is due to the result of step-4A

(proposed software architecture) or to the result of step-4B (target hardware architecture).

The reminder parts of the current section details the steps depicted in Figure 1. We use an Autonomous Parking (AP) System case study to elucidate the work performed in each step. Moreover, we concentrate the discussions on the software architecture modeling (step-4A). The target hardware architecture definition (step-4B), although very important in the context of the proposed process, should be subject of additional investigation and therefore is left out of this work.

4.1. Requirements Definition

The initial step in any development methodology is to define the requirements of the system to be developed. This includes both functional requirements (FR) and non-functional requirements (NFR). While the former depicts the main functionalities to be performed by the system, the latter imposes restrictions to those functionalities.

Table 4.1 presents the list of requirements from the AP system, which has three main functionalities: (FR1) start/stop the system using a GUI; (FR2) search for a parking slot; and (FR3) park the car. NFRs are like properties that must be satisfied by the related FR.

4.2. Functional Modeling and Simulation

In many applications, especially those related with control systems, it is required to first design a functional model of the system and to simulate it before any design decision on the system architecture is carried on. This is used either to provide a deeper understanding of the system functionalities or to test/simulate control solutions in early development stage. Tools like Scade/Lustre and Matlab/Simulink are often used for this propose.

4.3. Environment Description

The third step of the process consists of using AADL to describe the environment that interacts with the system under development. So, the set of interactions of the system with the external devices, such as sensors, actuators, user interface, etc.

For this reason we use here a high-level AADL diagram. Figure 2 presents the diagram designed for the AP system, where it is possible to observe the main system in the center (named `ParkingCtrl`) surrounded by the devices. An advantage of using AADL for such purpose is that it allows detailing each message exchanged between the system and the devices, including information like data type, arrival pattern, and time constraints.

In this phase two different kinds of external devices can exist: reused devices and new devices. While devices like sensors and actuators are normally reused from previous applications, devices like User Interfaces (UI) are normally designed on demand for each application. New devices can be subject of formal verification prior to its use in the model. Therefore it is necessary to specify the device's behavior. In the scope of this work it is suggested to describe behavior using finite automatons.

To exemplify the verification of devices behavior in the AP system we selected the UI device (`UIController`). A possible behavior of this device is depicted in Figure 3. This state-transition diagram states that, independently of the status of the application, the

FR1 - Start/stop the system using a GUI	
Description: The system must be explicitly activated by the driver to start operation	
NFR1.1 - Maximum speed	To start the system the speed must be kept at $\leq$ 20Km/h
NFR1.2 - On operation	The system must inform the user while it is working
NFR1.3 - Finished	The system must inform the user as it is turned off
FR2 - Search for a parking slot (*real-time operation*)	
Description: When activated, the system must start searching a new park slot as the vehicle moves forward	
NFR2.1 - Driver alert	The system must inform the user when a new parking slot is found
NFR2.2 - Safety	If the speed is too high (over 20km/h) than it is not possible to search a parking slot
FR3 - Parking (*real-time operation*)	
Description: The driver must trigger the beginning of the parking after a parking slot is found. The system controls the speed and direction of the vehicle.	
NFR3.1 - Safety	The system is allowed to start parking only if the current speed is zero
NFR3.2 - Emergency Stop	The system must be halted immediately if the driver moves the wheel
NFR3.3 - Finish allert	The system must alert the driver when the parking maneuver is finished

Table 1. Requirements set of the Autonomous Parking (AP) System

driver can always turn off the system (NFR1.3). This can be proof by the existence of the user event `Off!` in every possible execution state of the system. Although very simple, this is an example to show that it is possible to use verification already at this level.

4.4. Software Architecture Modeling

The software architecture modeling (step-4A) is probably the most important phase of the proposed design process. This phase may have several steps of iterations, as designer may create several AADL submodels, from more abstract to more detailed ones, and that all these models should have its properties verified.

In the first iteration the designer must detail the AADL system process (e.g. `ParkingCtrl` at Figure 2) into a set of subcomponents (either processes or threads). As this detailing is completed, model verification is performed, as explained in the next section. If the verification fails (many times due to the lack of information in the model at the moment), a new refinement in each component should take action, starting new iterations.

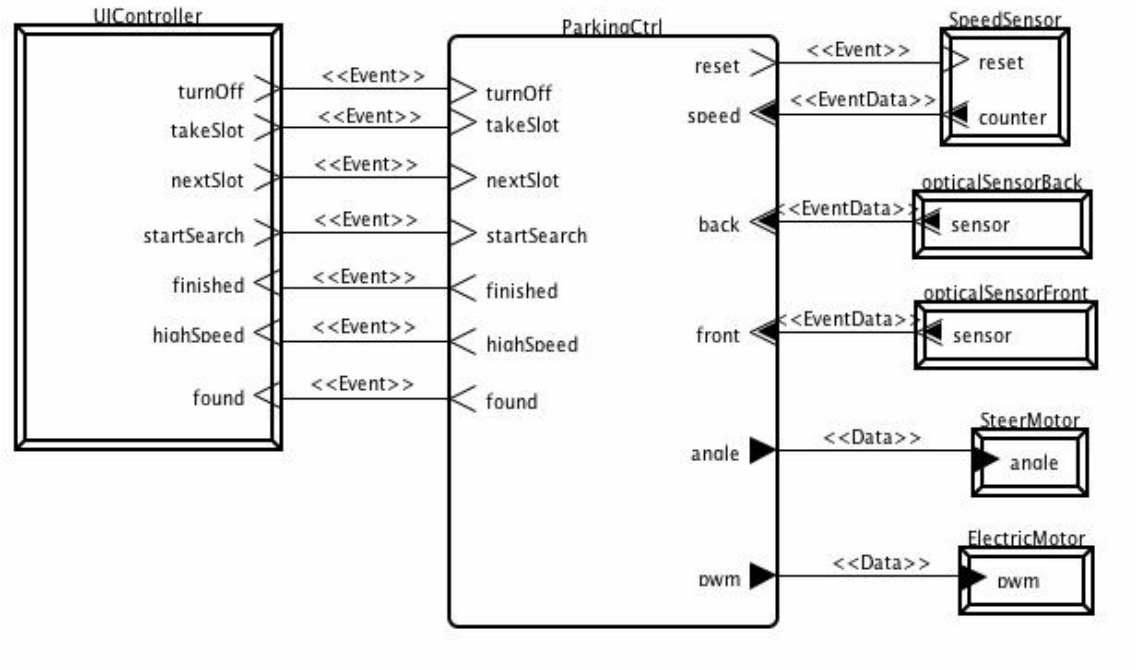


Figure 2. AP System Environment Description.

Following this approach, each component of the AADL model can derive into several subcomponents. By definition, the successive refinements will only finish as the model contains enough details to be proof correct or incorrect by the model verification. Each detailed model (i.e. iteration) should, however, cope with the abstract behavior defined for the higher level component.

4.4.1. Architecture Refinement

The architecture refinement process consists of successive model refinements and verification, as suggested in the design flow from Figure 4. It starts with identifying the operation modes (1) and threads (2) of the system, being followed by the mapping of functions to threads (3). Afterwards the designer can make the connections among the threads (4) and associate an execution mode to each thread (5).

We suggest organizing the functionalities of the system using different operation modes. This can be seen as a kind of temporal decomposition from the set of available functions. Therefore it is necessary to identify how many different operation modes the system should have. These modes can be used to guide the modeling of the distinct AADL processes that will be used to decompose the system in sub-parts. In our case study, the sub-functions of the first decomposition are more or less analogous to the operation modes. Figure 5 shows the automaton in charge of representing the AP system behavior.

After the identifications of the system (sub)functions it is possible to decompose the AADL model into different threads. This can be either the first level of decomposition of the AADL-system or a refinement of an existing thread. Defining connections means to establish the information exchange among the system subparts (threads). This also

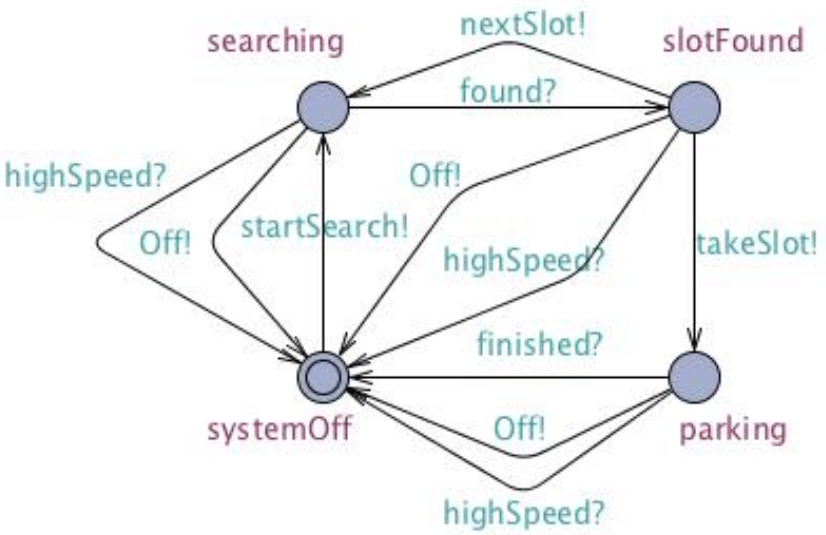


Figure 3. User interface behavior

A2.2. Architecture Refinement

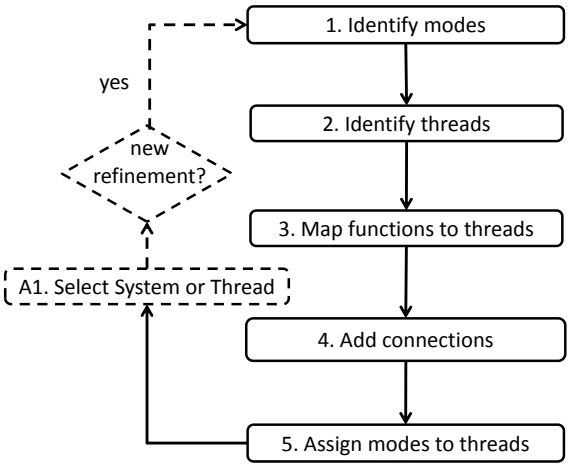


Figure 4. Refined steps from Architecture Refinement

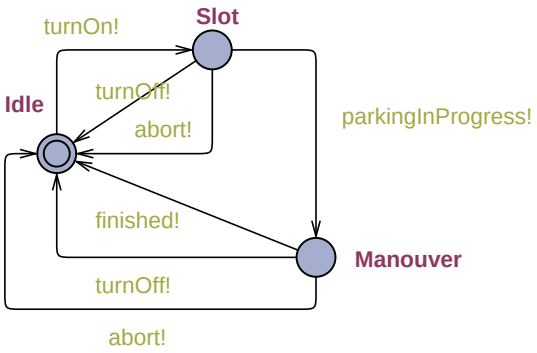


Figure 5. Basic operation modes of the AP System.

requires the definition of the data types associates with each port that transfer data.

For the AP system case study, the first level of decomposition consists basically in three threads, as shown in Figure 6. `SystemManagement` is used to start or halt the AP system by means of the graphical interface (FR1), `SlotSelection` is responsible to search for a parking slot (FR2), and finally `ParkingManeuver` is responsible to perform the parking (FR3). Every thread corresponds a FR of the system.

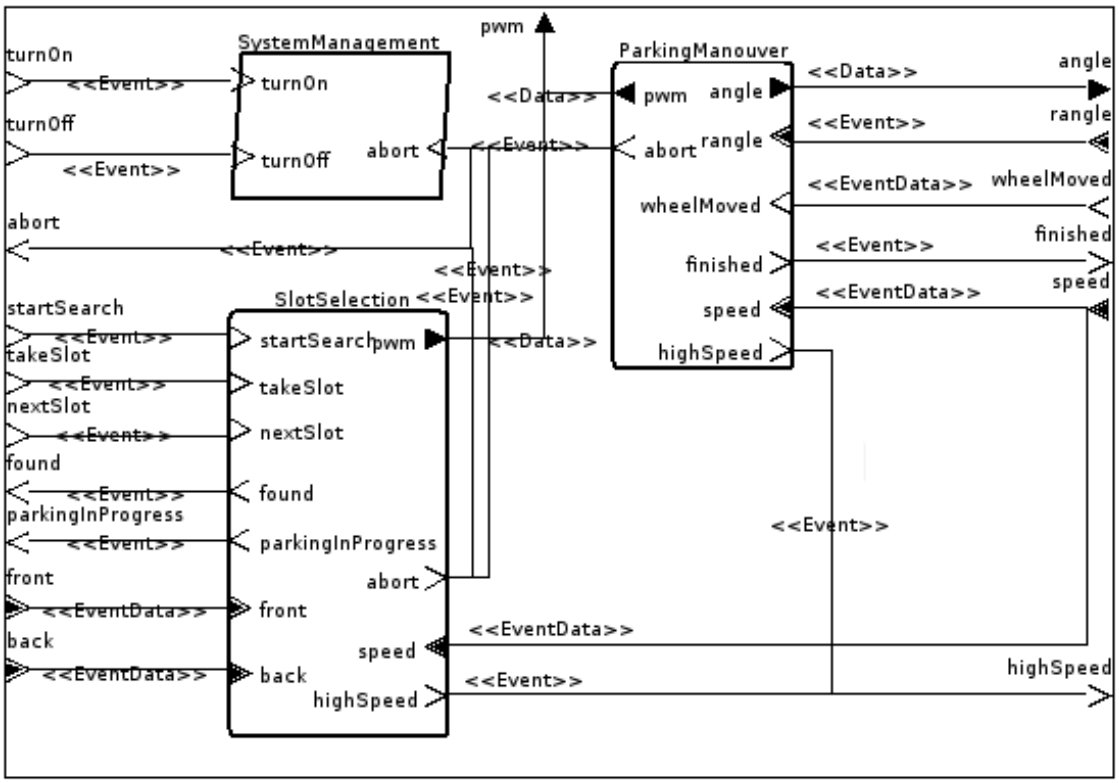


Figure 6. AADL model of parking control system (in the first decomposition)

Finally it is required to define in which operation modes each thread will be active. This represents a common modeling procedure to make the timing decomposition of the system functionalities. In AADL this is performed directly in the code, i.e. there is no graphical representation for this association. It must be highlighted, however, that it is

possible to associate a thread with several operation modes.

4.4.2. Model Verification

It is a modeler decision whether he wants to perform further refinements or to verify the behavior of the current model. In order to make the model verification it is necessary to provide the abstract behavior of each thread that belongs to the AADL model. Afterwards designer should define the set of properties of interest to be verified and perform the verification process. Such process is detailed in the section 5.

4.5. Time-Related Levels

To verify the real-time properties of the model it is necessary to make the Software/Hardware Mapping (step-5). After this step, every thread must be associated with a specific processor. The hardware architecture must have at least one processor. Thereby, in the Real-Time Properties Refinement (step-6), the designer can add additional timing information in the AADL model to be further verified. Such information must be obtained using, for example, model simulation on top of the target architecture. Thereby it is possible to obtain the worst case execution time (WCET) for each function of the system prior to its implementation. The last step of the proposed process is in charge of making the verification of the timing properties. Schedulability and response-time analysis are exemples of possible properties to be verified.

5. Verification Process

It is possible to argue that our proposed verification process supports the safe design of the system's architecture using MDE's principles. By safe design we mean that the resulting system architecture goes through several verification steps in order to assure its correctness. To reach this goal it is performed a sequence of model transformations, which starts with an AADL-like model and finishes with an equivalent automaton model that is suitable for verification.

The verification process we have been working on uses AADL models as input and performs the model checking of LTL properties. Moreover, schedulability and buffer overflow can also be analyzed, as well as user defined properties. This process is split in the following phases (Figure 7):

- Use of the OSATE-TOPCASED [Team 2004, Topcased] environment for AADL model edition and XMI generation. We consider AADL together with its behavioral annex.
- Translation of AADL XMI models to Fiacre [Berthomieu et al. 2008].
- Translation of Fiacre to the timed transition system (TTS) input format of Tina toolbox.
- Translation to an untimed automaton via an LTL-preserving time abstraction.
- Verification of LTL properties using the Selt tool from the Tina toolbox.

5.1. Verification Tools

TINA is a software environment to edit and analyze Petri nets, Time Petri nets, Time Transition Systems, and also extension of these nets handling data, priorities and temporal

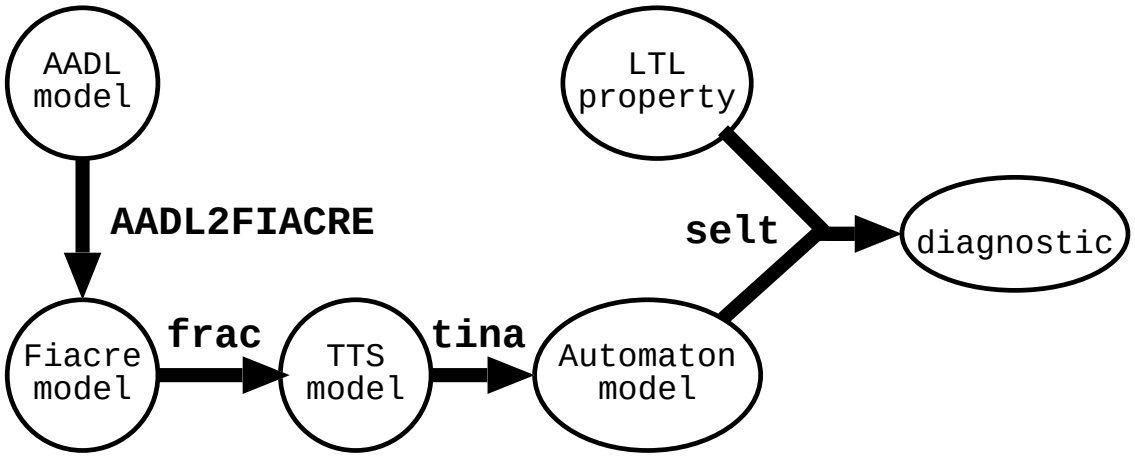


Figure 7. The verification process.

preemption. Beside the usual editing and analysis facilities of similar environments, the essential components of the toolbox are a state space abstraction tool (also called Tina) and a model checking tool (selt). Detailed information about the toolbox capabilities can be found in [Berthomieu et al. 2004].

TINA offers abstract state space constructions that preserve specific classes of properties of the state spaces of nets, like absence of deadlocks, linear time temporal properties, or bisimilarity. For untimed systems, abstract state spaces help to prevent combinatorial explosion. For timed systems, TINA provides various abstractions based on state classes, preserving reachability properties, linear properties or branching properties.

State space abstractions are provided in various formats suitable for existing model checkers. The TINA toolbox also provides a native model checker, selt. Selt allows one to check more specific properties than the general ones (boundedness, deadlocks, liveness) already checked by the state space generation tool. Selt implements an extension of linear time temporal logic known as State/Event LTL [Edmund et al. 2004], a logic supporting both state and transition properties. The modeling framework consists of Kripke transition systems (labeled Kripke structures, the state class graph in our case), which are directed graphs in which states are labeled with atomic propositions and transitions are labeled with actions.

State/Event-LTL formulas are interpreted over the computation paths of the model. They may express a wide range of state and/or transition properties. A formula p, q evaluates to true if it does so on all computation paths, constituted from the statements X (in the next step), G (globally), and F (eventually). Follows some typical formulas:

- **p** p holds at the start
- **X p** p holds at the next step (next)
- **G p** p holds all along the path (globally)
- **F p** p holds in a future step (eventually)
- **p U q** p holds until q holds (until) and q holds eventually.

Real-time properties, like those expressed in so called “timed temporal logics”, are checked using the standard technique of observers, encoding such properties into reachability properties. The technique is applicable to a large class of real-time properties and

can be used to analyze most of the "timeliness" requirements found in practice.

5.2. Properties Verification

Currently, we support the verification of three kinds of properties: (i) implicit properties taken into account by the translator and leading to deadlock when not satisfied; (ii) user properties specified through AADL real-time observers; and (iii) properties specified directly in linear temporal logic.

5.2.1. Implicit properties

For the moment, two implicit properties are taken into account by the translator:

- **Schedulability**: threads are scheduled using a fixed priority protocol with user-specified preemption points. Deadline events are generated by the translator. If a deadline occurs while a thread is still active, a specific deadlock is generated.
- **Buffer overflows**: AADL defines the property *Overflow_Handling_Protocol* which specifies what to do in case of overflow. Either the oldest or the newest data is lost, or the component is erroneous. The latest case is handled by the translator to generate a specific deadlock if the capacity of the input buffer is exceeded.

5.2.2. Real-time observers

Some properties such as bounded response time can be expressed using AADL threads acting as real-time observers. The component to be checked is linked to an observer which plays the role of its environment and checks its responses.

For example, properties of the `maneuver` component of the parking can be verified by specifying an environment as the following. It checks that the `highSpeed` signal is emitted one period (fixed here at 10ms) after the speed becomes non zero. Otherwise, the `err` state would be reached. It also checks that the `abort` signal is sent if the wheels are moved. The `selt` model checker was used to show that the `err` state is unreachable.

5.2.3. Linear time Temporal Logic

Temporal properties can be checked on the closed system. They can be expressed in linear temporal logic (LTL) and passed to the `selt` tool. Atomic properties are either event properties or state properties. For example:

- If the speed is too high, the interface cannot get the `found` message while the search has not been restarted.
- It is possible to park the car, i.e. there exists an execution path leading to a state where the car is parked. It is expressed as a negated property: it is not true that in any execution, `finished` is never sent.
- The car can be parked infinitely often.

5.2.4. Modal mu-calculus

There exists some useful properties that cannot be expressed neither in LTL, nor in CTL. For example, the fact that the user interface can be reinitializable by the user whatever the system does. To solve this problem, it can be expressed in modal mu-calculus. Such a property can be verified on atemporal models by the `muse` tool of the Tina toolbox. It must be associated with a *stability* property expressing that non-user events do not leave the initial state.

6. Conclusions

In this paper we presented a verification approach and the related toolset to design safety critical systems using the AADL language. This work is part of a more general project, which also covers the hardware architecture definition in more details, going towards producing safe models for critical applications. It must be highlighted that in the end of the process it is possible to make automatic code generation from the AADL model for a given platform.

It should be noticed, however, that given the complexity of the situation, the guarantee of the existence of a correct solution cannot be asserted. This also applies to the implementation derived from the generated model. To overcome this problem, designer feedbacks are necessary and, more generally, it should be wise to superpose to the software engineering process risk management.

Future work should cover automatic derivation of the properties to be verified from the system requirements. By using such approach, we should also assess the property languages in more details.

Acknowlegements

This work was developed with the grant CAPES STIC-AmSud 003/07 **TAPIOCA** : *Timing Analysis and Program Implementation On Complex Architectures* and supported by the French AESE project **Topcased**.

References

Berthomieu, B., Bodeveix, J.-P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., Lang, F., and Vernadat, F. (2008). Fiacre: an intermediate language for model verification in the TOPCASED environment. *Proceedings of the 4th European Congress on Embedded Real-Time Software ERTS'08(Toulouse, France)*.

Berthomieu, B., Peres, F., and Vernadat, F. (2007). Model checking bounded prioritized time petri nets. In Namjoshi, K. S., Yoneda, T., Higashino, T., and Okamura, Y., editors, *ATVA*, volume 4762 of *Lecture Notes in Computer Science*, pages 523–532. Springer.

Berthomieu, B., Ribet, P., and Vernadat, F. (2004). The tool TINA – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14).

Dissaux, P. and Singhoff, F. (2008). Stood and cheddar: Aadl as a pivot language for analysing performances of real time architectures. In *4th European Congress ERTS EMBEDDED REAL TIME SOFTWARE*.

Edmund, S. C., Clarke, E. M., Sharygina, N., and Sinha, N. (2004). State/event-based software model checking. In *In Integrated Formal Methods*, pages 128–147. Springer-Verlag.

Feiler, P., Gluch, D., and Hudak, J. (2006). The architecture analysis & design language (AADL): An introduction. Technical report, Software Engineering Institute, Carnegie Mellon University.

Franca, R. B., Bodeveix, J.-P., Filali, M., Rolland, J.-F., Chemouil, D., and Thomas, D. (2007). The AADL behaviour annex – experiments and roadmap. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 377–382, Washington, DC, USA. IEEE Computer Society.

Håkansson, J., Carlson, J., Monot, A., Pettersson, P., and Slutej, D. (2008). Component-based design and analysis of embedded systems with uppaal port. In *ATVA '08: Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, pages 252–257, Berlin, Heidelberg. Springer-Verlag.

Schmidt, D. (2006). Model-driven engineering. *IEEE Computer*, 39(2).

Team, S. A. (2004). OSATE: An extensible source aadl tool environment. Technical report, Software Engineering Institute, Carnegie Mellon University.

Topcased. (toolkit in open-source for critical apllications and systems development). `http://www.topcased.org`.

Um Algoritmo Emergente para Coleta de Dados em Redes de Sensores sem Fio

Otávio Alcântara de Lima Júnior[1], Helano S. Castro[2] e Paulo Cesar Cortez[2]

[1]Departamento de Telemática - Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE)
Campus Maracanaú – Av. Contorno Norte, 10 – CEP: 61925-315 – Maracanaú – CE – Brasil

[2]Departmento de Engenharia de Teleinformática – Universidade Federal do Ceará (UFC)
Campus do Pici – CEP:60455-970 – Fortaleza – CE – Brasil

otavio@ifce.edu.br, helano@lesc.ufc.br, cortez@lesc.ufc.br

Abstract. *Many natural and man-made systems present emergent properties, where simple and repeated interactions among internals components lead to complex global behavior patterns. This emergent behavior can be explored by designers to reach application goals. This paper presents a wireless sensor network data gathering algorithm, which defines a set of simple rules to the nodes, achieving systemic behavior. That reduces messages exchanges and increasing network lifetime by reducing power consumption. Tests experiments demonstrated that the algoritm has a higher performance than similars techniques, in scenarios which the nodes has a failure probability.*

Resumo. *Vários sistemas naturais e criados pelo homem apresentam características emergentes, onde as interações simples e repetidas dos componentes internos geram padrões complexos de comportamento em nível global. Esse comportamento emergente pode ser explorado em nível de projeto para alcançar os objetivos da aplicação. Este trabalho apresenta uma proposta de algoritmo para coleta de dados em redes de sensores sem fio que define uma série de regras simples aos nós, obtendo um comportamento sistêmico, que reduz o número de mensagens trocadas pelos nós. Os experimentos realizados foram capazes de demonstrar que este algoritmo possui um desempenho aceitável, em relação a técnicas similares, em cenários onde os nós possuem probabilidade de falhar.*

1. Introdução

Redes de Sensores sem Fio (RSSFs) consistem de pequenos nós sensores inseridos no ambiente, e que tem a missão de extrair dados de monitoramento e enviá-los para análise. Os nós de uma RSSF são geralmente limitados pela potência de sua fonte de alimentação, poder computacional e alcance de seus enlaces de rádio. As RSSFs são utilizadas em diversos segmentos como monitoramento de estruturas, ambientes militares, dentre outros [Culler et al. 2004].

A arquitetura básica de uma RSSF está ilustrada na Figura 1 . Os nós podem ser organizados em grupos ou atuarem individualmente, e os dados coletados são enviados para o nó sorvedouro que os disponibiliza para outras redes de comunicação. O roteamento das mensagens é centrado nos dados e não nos endereços. Nessas aplicações, é mais importante saber os valores ou eventos de uma determinada região do que o valor

lido por um sensor específico, sendo que a alta densidade de nós permite que o mesmo evento seja monitorado por diversos nós, criando uma redundância nas informações e maior tolerância a falhas.

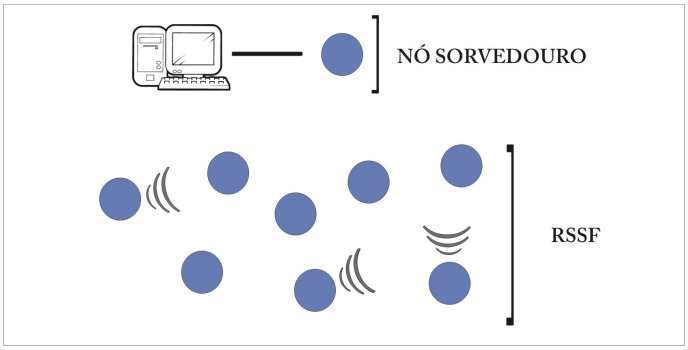


Figura 1. Modelo de Organização de uma RSSF típica.

O paradigma de comunicação das RSSFs diverge do modelo empregado em outras redes sem fio. Em uma RSSF, os nós da rede são instalados geralmente, em regiões de difícil acesso que inviabilizam a manutenção dos mesmos, bem como a troca de baterias. Por isso, questões de otimização do consumo de energia norteiam as decisões de projeto, que devem priorizar o prolongamento do tempo de vida da rede. Além disso, o fluxo de comunicação na rede é de muitos nós para um, e os nós sensores tendem a enviar os resultados de suas amostragens para o nó sorvedouro, numa espécie de multicast inverso. Por causa dessas e de outras especificidades, diversos protocolos de comunicação foram propostos para as RSSFs [Akkaya and Younis 2005].

Sabe-se que o consumo de energia é uma questão central no projeto de RSSFs, cuja principal técnica utilizada para reduzir o consumo é a fusão de dados. Esta técnica processa dados provenientes de diversas fontes e gera uma saída que é, de alguma forma, melhor do que as saídas das fontes individuais, através da cooperação entre os nós sensores. Dessa maneira, é possível reduzir o número de pacotes que trafegam pela rede aumentando seu tempo de vida [Nakamura et al. 2007].

As RSSFs são exemplos de sistema embarcado e distribuído. Nestas redes, há um fenômeno natural recorrente a auto-organização, indicando que a interação entre os componentes no nível microscópico acarretam comportamentos globais que emergem no nível macroscópico [Mills 2007]. Esses comportamentos emergentes podem ser indesejáveis e prejudiciais ao sistema, porém o projetista pode, intencionalmente, adicionar regras locais nos componentes que gerem comportamentos emergentes benéficos para a aplicação. Mills [Mills 2007] apresenta diversas abordagens que estimulam a auto-organização em RSSFs com o intuito de alocar banda de comunicação, formar grupos de sensores, disseminar informações, e organizar tarefas.

Este trabalho apresenta uma proposta de algoritmo para coleta de dados em RSSFs que utiliza os conceitos de sistemas auto-organizáveis, como forma de reduzir o número de mensagens enviadas e prolongar o tempo de vida da rede através da diminuição do consumo de energia.

O restante do trabalho está dividido da seguinte forma: a secção dois apresenta outros trabalhos relacionados ao tema; na secção seguinte são discutidas as principais

características de sistemas auto-organizáveis; a secção quatro apresenta o algoritmo proposto; em seguida apresentamos os resultados obtidos com os experimentos e a última secção trata as considerações finais.

2. Trabalhos Relacionados

Diversos protocolos para RSSFs se apóiam na interação local dos nós para atingirem objetivos globais, dentre estes podemos destacar a difusão direcionada [Intanagonwiwat et al. 2000]. Usando esse paradigma de comunicação, os dados gerados pelos sensores são nomeados através de pares de atributo-valor. Um nó solicita dados enviando uma mensagem de interesse especificando o tipo de evento desejado e a região de monitoramento. Quando um sensor da região de monitoramento recebe o interesse, este sensor passa a monitorar o ambiente buscando identificar o evento descrito. No momento que o evento é identificado, a informação retorna pelo caminho inverso da propagação do interesse. Os nós intermediários podem agregar novos dados, melhorando a precisão das medições. Essas interações são realizadas utilizando apenas troca de mensagens entre nós vizinhos. Contudo, esse algoritmo não se adequa bem para aplicações de monitoramento contínuo de ambientes.

Outro protocolo de RSSFs que busca inspiração no comportamento de sistemas biológicos foi apresentado por Cunha(2005). O objetivo desse protocolo é fornecer uma estimativa de processos variáveis espaço-temporialmente em uma dada região. Cada nó da rede identifica padrões no processo físico e reporta ao sorvedouro apenas os comportamentos que desviem do esperado [Cunha and Duarte 2005]. Por isso, esse protocolo só pode ser utilizado para monitorar processos físicos que apresentem um comportamento ao longo do tempo com alguma característica que possa ser considerada comum, ou esperada.

Madden(2002) apresenta uma RSSF, cujos nós são organizados em uma árvore de agregação. O nó raiz da árvore é o nó que envia as requisições e recebe seus resultados agregados. A requisião/resposta é propagada dos nós pais para os nós filhos. Os nós folhas enviam suas próprias medições para o pai. Nós intermediários esperam pelos valores de seus nós filhos, realizam a fusão de dados locais desses valores com suas próprias medições e envia o resultado para o seu nó pai. O nó raiz computa da mesma maneira que os nós intermediários e apresenta o resultado da agregação dos dados para o usuário [Madden et al. 2002]. Esse algoritmo requer a construção e manutenção de uma árvore de extensão. Em um ambiente real, falhas nos nós intermediários exigiriam a rotineira reconstrução da árvore de extensão. A Figura 2 ilustra a organização da rede nesse algoritmo.

Outra técnica de agregação de dados em RSSF é o protocolo *Push-Sum* [Kempe et al. 2003]. Cada nó mantém duas quantidades: um peso e um agregado. Em cada rodada, cada nó escolhe aleatoriamente um nó na rede e envia metade de seu peso e de seu agregado e o nó receptor adiciona essas quantidades as suas próprias. Se cada nó da rede for capaz de, aleatoriamente, contactar qualquer outro nó, o algoritmo calcula o valor agregado das medições da rede em um tempo que é proporcional ao logaritmo do número de nós. O cálculo do valor agregado é realizado de forma completamente descentralizada, mas para redes maiores o tempo de convergência pode se tornar um empecilho.

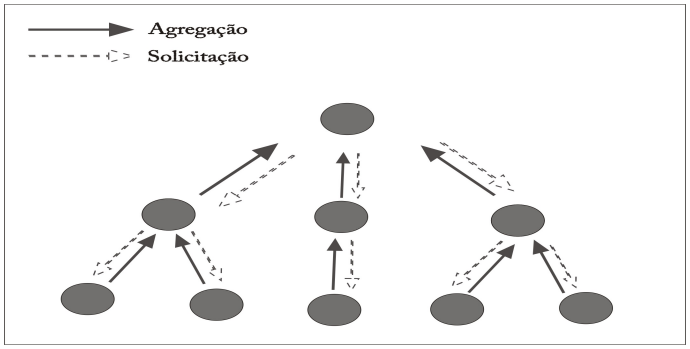


Figura 2. Estrutura da Árvore de Agregação.

3. Princípios de Auto-Organização

Um sistema com diversos componentes pode exibir comportamentos globais que aparentam ser mais complexos do que o comportamento de suas partes. Estes comportamentos emergentes aparecem naturalmente no processo de auto-organização do sistema, que é comum em vários sistemas naturais e criados pelo homem [Mills 2007]. Um exemplo são as propriedades de temperatura e pressão dos gases, que é uma medida da força das interações entre as moléculas. O projetista de protocolos para RSSFs pode tirar proveito dessas propriedades emergentes para atacar diferentes questões, como: alocação de banda, formação de grupos, fusão de dados, otimização do consumo de energia, dentre outras. No entanto, para tal é necessário ter o entendimento de quais princípios guiam e definem o comportamento emergente desses sistemas.

Sumpter (2006) enumera os princípios básicos inerentes a diferentes sistemas auto-organizáveis que usamos para entender os algoritmos que exploram o comportamento emergente. Os princípios são: integridade e variabilidade, reforço positivo, reforço negativo, limites de resposta, liderança, redundância, sincronização e egoísmo. Muitas das grandes questões do futuro dessa área se preocuparam em como trabalhar com esses princípios para gerar padrões de comportamento coletivo complexos. A seguir, discutiremos brevemente cada princípio.

Integridade e variabilidade estão relacionadas à individualidade de cada componente do sistema. É importante que cada componente possua um grau de diferenciação dos outros, é uma forma de prover novas soluções para os problemas que o grupo deseja resolver. O reforço positivo é uma forma de um indivíduo compartilhar informações com o grupo, influenciando o comportamento dos outros. Por outro lado, o reforço negativo dissuade um indivíduo a participar de uma ação, permitindo criar estabilidade no padrão do grupo. Limites de resposta estão atrelados a quantidade de estímulos necessária para um indivíduo gerar uma resposta ao sistema.

Embora liderança possa parecer pouco associável com sistemas auto-organizáveis, existem diversos exemplos de sistemas nos quais alguns indivíduos são chave para organizar as ações do grupo. Redundância é outra característica presente em sistemas auto-organizáveis, pois, permite que o sistema continue funcionando mesmo que haja uma redução no número de componentes. Sincronização é um exemplo do reforço positivo, que contribui para a melhor coordenação das ações dos componentes.

Por fim, o egoísmo que define que os atos altruístas de um indivíduo devem possuir

relação custo/benefício menor do que o relacionamento com o indivíduo que lucra com esse ato [Sumpter 2006]. Nesse trabalho, focamos na utilização de três dos princípios listados acima: integridade e variabilidade, reforço positivo e reforço negativo. Esses princípios nortearam a definição das regras impostas aos nós sensores.

4. Modelo do Algoritmo Proposto

Neste trabalho, utilizar-se agregação de dados em árvore para obter uma estimativa de um processo físico variável espaço-temporalmente em uma dada região. O algoritmo é baseado na interação entre nós vizinhos, indicando os nós que possuem medições divergentes em relação a uma média das medições. Apenas os nós localizados nas regiões críticas, cujas medições divergem significativamente da média da rede, reportam suas medições, que são agregadas ao longo do caminho de transmissão ao nó sorvedouro, que poderá, por sua vez, alterar os parâmetros que definem a classificação dos nós. Desta forma, o nó sorvedouro possui uma visão geral da variação do processo físico monitorado.

Antes de apresentarmos a estrutura do algoritmo proposto, devemos especificar as características que uma RSSF deve possuir para sua implantação. Os nós da rede devem ser fixos, cientes de sua localização geográfica e possuir *hardware* semelhante. A densidade dos nós deve ser adequada para gerar correlação nas medições e cobrir toda a área de monitoramento. A demanda de dados pela aplicação deve ser relativamente baixa, como também os requisitos de latência. De um modo geral, o usuário deve estar interessado em ter uma estimativa da variação espaço-temporal de um processo físico em determinada região.

O algoritmo define que os nós da rede devem compartilhar o mesmo ciclo de vida, que se resume a: enviar uma mensagem de *broadcast* com informações sobre o próprio nó e sua vizinhança; esperar um tempo para receber as mensagens de seus vizinhos e depois desligar o rádio e o processador até o próximo período. A partir dessa mensagem periódica, os nós são capazes de realizar algumas atividades complexas, como: sincronização; agrupamento de nós; agregação de dados; difusão das solicitações e respostas a solicitações.

A primeira etapa do algoritmo é a sincronização, que é baseada na técnica de sincronização *firefly* [Breza and McCan 2008]. Uma mensagem periódica deve conter um campo que indique qual é o tempo para a próxima ativação do nó. Os nós devem ajustar o período de ativação pela média dos períodos dos nós da vizinhança. Assim, em algumas iterações todos os nós da rede devem estar sincronizados. Essa técnica permite que os nós deixem os seus rádios desligados a maior parte do tempo, prolongando o tempo de vida da rede.

O próximo passo do algoritmo é o agrupamento dos nós em regiões, sendo que outro campo existente na mensagem identifica a qual região o nó pertence. O nó sorvedouro sempre pertence à região zero. Os nós que estão ao alcance de seu rádio, ao receberem a mensagem, configura sua região para nível um. De forma similar, os outros nós da rede configuram sua região como sendo uma região de número acima do nó vizinho de menor região. Dessa forma, em poucas iterações, todos os nós da rede conseguem determinar suas regiões. A Figura 3 ilustra a topologia da rede, após os nós identificarem suas regiões, que refletem sua distância em relação ao nó sorvedouro.

O foco do algoritmo é gerar uma representação média das medições da rede, que

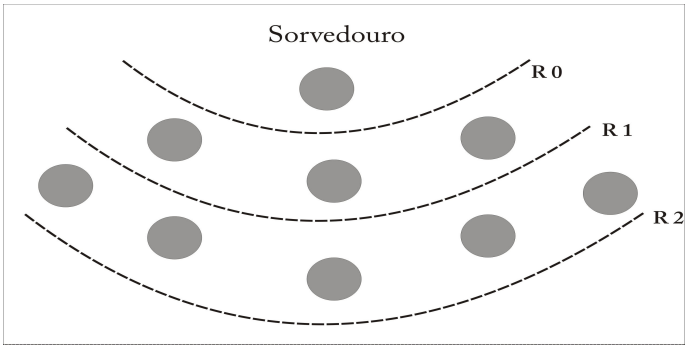


Figura 3. Regiões da RSSF.

é alcançada através da geração de uma estrutura de agregação em árvore. A cada nova iteração do algoritmo, cada nó da rede seleciona aleatoriamente um nó vizinho, que esteja na região imediatamente inferior. Este então envia para esse nó as suas medições, e o nó receptor agrega suas medições com as medições recebidas e as envia na próxima iteração. O valor agregado é representado por um par de números reais, o primeiro valor representa a soma das medições recebidas durante a iteração e o segundo representa o número de nós que enviaram suas medições. De forma que, em algumas iterações, as medições das sub-árvores chegam ao nó sorvedouro, que é capaz de calcular o valor médio de toda a rede. A Figura 4 ilustra esse algoritmo e é importante notar que essas informações são trocadas na mensagem periódica de sincronização dos nós.

Algoritmo

0 - Seja i um nó da rede alocado na região R;

1 -Seja $\{w_k\}$ todos os valores enviados para o nó i pelos k nós de sua sub-árvore na iteração t;

2 - Seja $W_{t,i} = \{\Sigma_j w_j, k\}$, o valor agregado calculado pelo nó i na iteração t;

3 - Escolha aleatoriamente um nó da região R-1, e envie $W_{z,i}$ na iteração t + 1.

Figura 4. Algoritmo de Agregação.

Após calcular a média da rede, o nó sorvedouro adiciona esse valor na mensagem da próxima iteração, com o intuito de que todos os nós da rede tenham ciência do valor calculado. Assim, os nós podem utilizar esse valor para realizarem suas classificações, utilizando os parâmetros que são difundidos na rede pelo nó sorvedouro.

Neste ponto do algoritmo, os nós estão sincronizados, agrupados em regiões e cientes do valor médio das medições da rede. Para que o usuário obtenha uma visão mais precisa dos eventos da rede, é necessário que haja um mecanismo de difusão de solicitações de dados, que são os parâmetros utilizados pelo nó para identificar se está em

uma região de interesse de monitoração. Neste algoritmo, nós propomos utilizar a própria mensagem de sincronização para difundir as solicitações, que neste caso são bem simples. Um exemplo de solicitação de dados que foi implementada é que o nó verifique se sua amostragem local está dentro de um limite percentual em relação à média da rede.

As respostas das solicitações são enviadas pelo mesmo mecanismo de árvore de roteamento utilizado para agregar os valores das medições dos sensores, utilizando agregação de dados sempre que for necessário ao longo da rota. Contudo, apenas os nós que satisfazem os critérios da solicitação encaminham seus dados.

5. Simulação e Resultados

A simulação para a obtenção dos resultados apresentados neste trabalho se baseia no modelo escrito para o simulador SHOX [Lessman et al. 2008], onde os nós foram posicionados de forma aleatória em uma região quadrada de 50 m por 50 m. O raio de alcance dos rádios foi configurado como 20 m.

Os resultados apresentados se referem a um número de sensores variando entre 20 e 50. Os nós trocam mensagens a cada um segundo, em que 80% desse tempo é passado com o rádio desligado.

São realizados três tipos de experimentos para avaliar o desempenho do algoritmo proposto:

- verificação do tempo para que os nós entrem em sincronismo;
- verificação do tempo para completar o cálculo da agregação;
- cálculo do erro percentual entre o valor real da média e o calculado pelo algoritmo, para cenários onde os nós tem uma probabilidade p de serem desativados em cada iteração. As medições foram realizadas para três casos:
  - o nó agregador é escolhido aleatoriamente entre os nós da região imediatamente inferior, que foi a proposta de nosso algoritmo;
  - o nó agregador é o nó mais próximo que esteja na região imediatamente inferior, e não há cache dos valores agregados em iterações anteriores;
  - o nó agregador é o nó mais próximo que esteja na região imediatamente inferior, e os nós armazenam os valores agregados anteriores por quatro iterações.

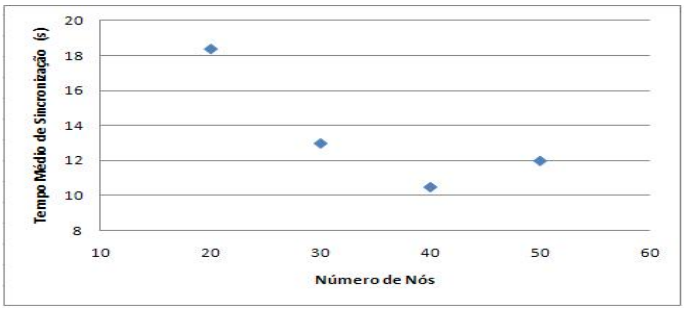


Figura 5. Tempo para Sincronização

Para cada caso de teste proposto nos experimentos foram relizadas um número arbitrário de cem iterações do algoritmo. As Figuras 5 e 6 mostram o tempo necessário

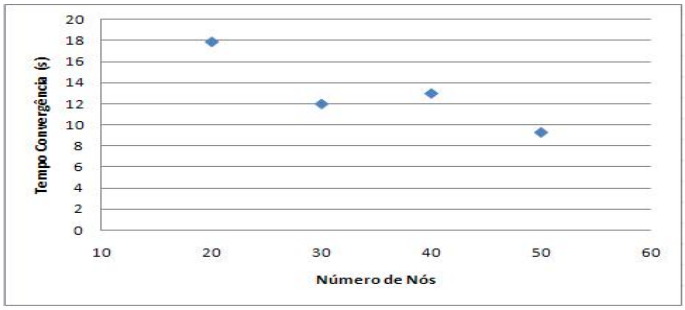


Figura 6. Tempo de Convergência da Agregação

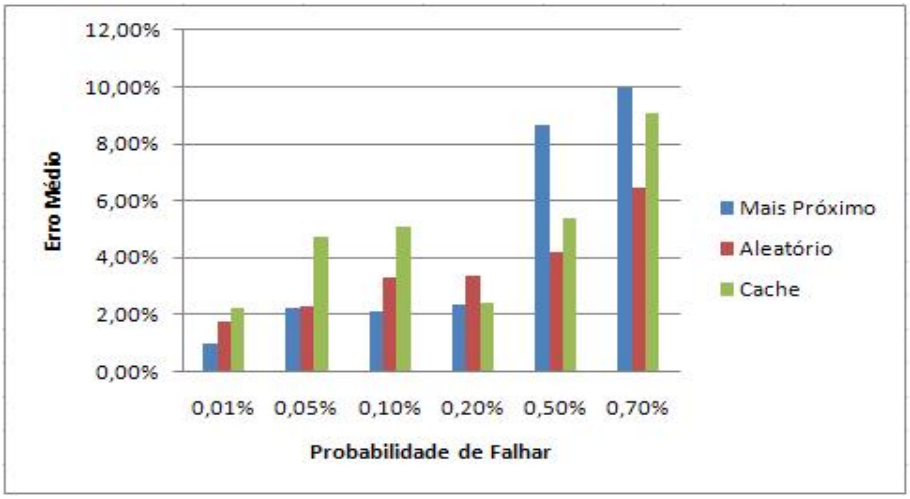


Figura 7. Erro Médio Verificado

para a convergência do sincronismo e do valor de agregado calculado pela rede. Como é de se esperar para redes menos densas, o tempo de convergência tende a ser maior.

O erro médio percentual entre o valor da agregação e o valor real da média da rede está mostrado na Figura 7. Nesse experimento, os nós possuem probabilidade p de falhar a cada iteração. Para cenários onde os nós tem alta probabilidade de falhar, escolher aleatoriamente o nó que fará parte da árvore de agregação produz um desempenho melhor do que escolher o nó mais próximo ou fazer cache de dados antigos. Para cenários com baixa probabilidade de falhas, o desempenho dos algoritmos é similar. É interessante notar que, para o cenário modelado, fazer cache de dados de agregação antigos não traz um bom desempenho devido ao fato do algoritmo de agregação usar dados de nós sensores que já estariam desativados.

6. Considerações Finais

A utilização dos conceitos de sistemas auto-organizáveis em projetos de sistemas distribuídos, em especial de RSSFs é importante e pode dar mais contribuições para a construção de protocolos que tirem maior proveito das capacidades desses sistemas. Neste artigo, utilizamos os princípios de auto-organização em RSSF para criar um algoritmo emergente que fornece uma descrição resumida da variação de um processo físico em uma determinada região. Os nós da rede trocam uma única mensagem por iteração do algoritmo que permite: sincronizar os nós; calcular a média das medições da rede e agregar

as respostas das solicitações do usuário.

Os experimentos realizados foram capazes de demonstrar que o algoritmo descrito possui um desempenho aceitável, em relação a técnicas similares, em cenários onde os nós possuem probabilidade de falhar. Ademais, o tempo de convergência do cálculo do valor médio das medições da rede e de sincronização dos nós é relativamente baixo para redes com até cinquenta nós.

Referências

Akkaya, K. and Younis, M. (2005). A survey on routing protocols for wireless sensor networks. *Elsevier Journal of Ad Hoc Networks*, 3:325–349.

Breza, M. and McCan, J. A. (2008). Lessons in implementing bio-inspired algorithms on wireless sensor networks. ESA Conference on Adaptive Hardware and Systems.

Culler, D., Estrin, D., and Srivastava, M. (2004). Overview of sensors network. *IEEE Computer Magazine*, 37:41–49.

Cunha, D. and Duarte, O. C. M. B. (2005). Um esquema bio-inspirado para estimação de campo com redes de sensores sem fio. XXII Simpósio Brasileiro de Redes de Computadores.

Intanagonwiwat, C., Govindan, R., and Estrin, D. (2000). Directed diffusion: A scalable and robust communication paradigm for sensor networks. 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking.

Kempe, D., Dobra, A., and Gehrke, J. (2003). Gossip-based computation of aggregate information. 44th Annual IEEE Symposium on Foundations of Computer Science.

Lessman, J., Heimfarth, T., and Janacik, P. (2008). Shox: An easy to use simulation plataform for wireless networks. 10th International Conference on Modelling & Simulation.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2002). Tag: A tiny aggregation service for ad-hoc sensor networks. pages 131–146. SIGOPS Oper. Syst.

Mills, K. (2007). A brief survey of self-organization in wireless sensor networks. *Wireless Communications and Mobile Computing*, 7:823–834.

Nakamura, E. F., Loureiro, A. A. F., and Frery, F. G. (2007). Information fusion for wireless sensor networks: Methods, models, and classifications. *ACM Computing Surveys*, 39:1–55.

Sumpter, D. J. T. (2006). The principles of collective animal behaviour. *Philosophical transactions of the Royal Society of London*, 361:5–22.

Índice por Autor