Técnicas de profiling para o co-projeto de
hardware e software baseado em computação
reconfigurável aplicadas ao processador *soft core*
Nios II da Altera

Luiz Henrique Kiehn

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

**Técnicas de profiling para o co-projeto de hardware e
software baseado em computação reconfigurável aplicadas
ao processador softcore Nios II da Altera**

Luiz Henrique Kiehn

Orientador: Prof. Dr. Eduardo Marques

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

**USP – São Carlos
Novembro de 2012**

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

K47t Kiehn, Luiz Henrique
Técnicas de profiling para o co-projeto de
hardware e software baseado em computação
reconfigurável aplicadas ao processador softcore
Nios II da Altera / Luiz Henrique Kiehn; orientador
Eduardo Marques. -- São Carlos, 2012.
140 p.

Dissertação (Mestrado - Programa de Pós-Graduação em
Ciências de Computação e Matemática Computacional) --
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo, 2012.

1. CIÊNCIA DA COMPUTAÇÃO. 2. COMPUTAÇÃO
RECONFIGURÁVEL. 3. SISTEMAS EMBARCADOS. 4.
PROFILING. 5. MEMORY USAGE. I. Marques, Eduardo,
orient. II. Título.

AGRADECIMENTOS

Pelos imprevisíveis caminhos de nossas vidas, encontramos muitas pessoas. A maioria poucas lembranças deixam, pois nada fizeram para nos deixar recordações delas. Algumas participam de forma mais próxima de nosso dia a dia por mera contingência profissional ou social. Mas as pessoas que com suas atitudes positivas e desprendidas transformam de forma surpreendente e definitiva as nossas vidas, levando-nos para caminhos que não ousávamos pensar que percorreríamos de alguma forma, deixam-nos um sentimento de muito carinho e de eterna gratidão.

É esse sentimento que dedico em especial a meu mestre e orientador, que em momentos tumultuados de minha vida estendeu-me sua mão generosa e conduziu-me de forma elevada por novos caminhos.

A meus pais, em especial à minha mãe, que com sua constante dedicação e preocupação sempre foram responsáveis pela minha boa formação pessoal e educacional.

À minha esposa e às minhas filhas, que me dão força e vontade de dedicar a elas a dedicação que recebi de todos os meus entes queridos.

À minha irmã, que com seu pensamento positivo fez-me erguer minha cabeça nas dificuldades para nunca desistir.

Aos meus amigos do Mestrado, pelo apoio que me deram por todo o curso, possibilitando-me chegar aonde cheguei.

À Universidade de São Paulo, pela oportunidade de retornar à essa tão prezada casa para mais uma vez adquirir preciosos conhecimentos.

À CAPES, por propiciar-me, com a bolsa de estudos fornecida, melhores condições para adquirir os conhecimentos que adquiri.

Ao ICMC e seus profissionais, que mui gentilmente me trataram e acolheram.

Sumário

Lista de Figuras	7
Lista de Tabelas	9
Resumo	i
Abstract	iii
1 Introdução	1
1.1 <i>Contextualização</i>	1
1.1.1 Avanços tecnológicos	1
1.1.2 Desafios das novas tecnologias	3
1.1.3 As ferramentas EDA como solução para os desafios	4
1.2 <i>Objetivo Geral</i>	6
1.3 <i>Objetivos Específicos</i>	8
1.4 <i>Justificativa</i>	8
1.5 <i>Método de Pesquisa</i>	9
1.6 <i>Organização da Monografia</i>	9
2 Sistemas Embarcados	11
2.1 <i>Sistemas Embarcados</i>	11
2.1.1 Definição	11
2.1.2 Restrições	11
2.1.3 Arquiteturas	12
2.1.4 Particionamento de Hardware e Software	13
2.1.5 Sistemas Embarcados na Robótica	14
2.1.6 Otimização de Sistemas Embarcados	14
2.2 <i>Modelos de Implementação de Sistemas Embarcados</i>	15
2.2.1 SoC	15
2.2.2 EDA e Co-projeto de Sistemas	16

3	Técnicas de Profiling	23
3.1	Definição	23
3.2	EDAs e <i>Profiling</i>	25
3.3	O Estado da Arte	29
3.4	O Módulo Profiling	31
3.4.1	Aplicações	31
3.4.2	Monitoramento	32
3.4.3	Métodos de implementação	35
4	Desenvolvimento	43
4.1	<i>Considerações iniciais</i>	43
4.2	<i>Implementação</i>	46
4.3	<i>Resultados</i>	58
5	Conclusão	71
5.1	Contribuições deste Trabalho	71
5.2	Trabalhos Futuros	72
	Referências Bibliográficas	75
A	IPXAct	79
B	Níveis de Representação de Hardware	81
B.1	RTL	81
B.2	ESL	81
B.3	TLM	83
C	Programa nios2-elf-Hippie.c	85
C.1	Código fonte do programa nios2-elf-Hippie.c	85
D	Programa InstCode.c	95
D.1	Código fonte do programa InstCode.c	95
E	Programa nios2-elf-HipReport.c	99
E.1	Código fonte do programa nios2-elf-HipReport.c	99
F	Benchmarks	107
F.1	Códigos fonte dos programas usados como Benchmark	107

Lista de Figuras

1.1	Ilustração da ampla gama de áreas e objetos que usam sistemas embarcados. (Vahid e Givargis, 2002)	2
1.2	Diagrama do ciclo de vida de um projeto de sistema embarcado. (Berger, 2002)	5
2.1	Diagramas em nível de: a) Transistor; b) Gates; c) RTL; d) ESL. (adaptado de (Dia, 2010))	19
2.2	A ferramenta RoboArch. (Bonato e Marques, 2009)	22
3.1	Evolução desejada para os Sistemas Embarcados.	26
3.2	Fluxo de projeto SoC típico. (Tabbara e Kashmi, 2004)	27
3.3	Reaproveitamento de componentes armazenados em biblioteca.	28
3.4	Características dos tipos de monitoramento (Ravagnani, 2007).	34
3.5	(Hübert e Stabernack, 2009) Comparação entre diversas ferramentas de Profiling .	38
3.6	Interação Memtrace x ARMulator. (Hübert e Stabernack, 2009)	41
4.1	Algoritmo geral do processo para obtenção do perfil de uso de memória <i>heap</i> .	47
4.2	Fluxo de processamento da ferramenta Hippie.	48
4.3	Arquivo <code>InstCode.c</code> .	49
4.4	Mensagem apresentada quando há erro de alocação	51
4.5	Relatório após apresentada mensagem de erro de alocação	51
4.6	Arquivo <code>nios2-elf-HipReport.c</code> .	52
4.7	Copiar <code>nios2-elf-Hippie.exe</code> e <code>nios2-elf-HipReport</code> para <code>.../bin</code>	54
4.8	Criar uma pasta para os <code><benchmark></code>	54
4.9	Criar uma pasta para os <code><benchmark></code>	55
4.10	Copiar <code>HipInstr.c</code> para pasta principal do projeto	55
4.11	Configurar os <i>file systems</i> de entrada e de saída	55
4.12	Configurar o <i>file system</i> de entrada	56
4.13	Configurar o <i>file system</i> de saída	56
4.14	Efetuar o 'build' da pasta 'syslib' do projeto	56
4.15	Efetuar o 'build' da pasta principal do projeto	57
4.16	Programação da memória <i>flash</i>	57
4.17	Executar o projeto principal no modo <i>debug</i>	57

4.18	Execução do programa <code>HipInstr.elf</code>	58
4.19	Gerado o arquivo <code>HipReport.out</code>	58
4.20	Execução do gerador do relatório de análise	58
4.21	Arquivo de relatório de análise	59
4.22	Resultados Elemento Máximo	61
4.23	Resultados Fatorial	61
4.24	Resultados Sequential Search	62
4.25	Resultados Binary Search	63
4.26	Resultados Bubble Sort	63
4.27	Resultados Insertion Sort	64
4.28	Resultados Selection Sort	64
4.29	Resultados Merge Sort	65
4.30	Resultados Quick Sort	65
4.31	Resultados Determinante	66
4.32	Resultados Multiplicação de Matrizes	66
4.33	Resultados Transposição de Matriz	67
4.34	Resultados Fibonacci	67
4.35	Resultados Gauss Jordan	68
4.36	Resultados Sequência de Jacobi	68
4.37	Resultados Filtro de Sobel	69
4.38	Resultados Processamento de Imagem BMP	69
A.1	(SPIRITConsortium, 2010) - IPXAct	80

Lista de Tabelas

Resumo

Com o avanço dos paradigmas de desenvolvimento de sistemas eletrônicos, novos conceitos, modelos e técnicas resultaram dessa evolução, gerando ferramentas mais eficientes e objetivas. Entre estas, as de automação de projetos eletrônicos (EDA - *Electronic Design Automation*) em nível de sistema (ESL - *Electronic System Level*) trouxeram um incremento considerável de produtividade à confecção de sistemas eletrônicos, inclusive de sistemas embarcados. Já no que se refere ao desempenho do sistema elaborado, monitorar sua execução e determinar seu perfil de funcionamento são tarefas essenciais para avaliar, a partir do seu comportamento, quais os pontos que representam gargalos ou pontos críticos, afetando sua eficiência geral. Dessa forma, faz-se necessário pesquisar princípios de verificação e otimização dos sistemas elaborados que estejam mais bem adaptados aos novos paradigmas de desenvolvimento de projetos. O presente trabalho tem por objetivo implementar um módulo de coleta e processamento de dados para análise de perfil de programas escritos na linguagem C e que sejam executados em processadores *soft core*, como o NiosII, da Altera. Entretanto, diferentemente das estatísticas oferecidas pela ferramenta GProf (GNU Profiling) com relação à análise de desempenho, em que cada amostra obtida implica no incremento de um contador para a função flagrada, o presente trabalho volta seu interesse à análise do perfil de uso de memória *heap*, que encontra-se mormente no volume alocado constatado em cada amostragem. Dessa forma, para diferentes amostragens de uma mesma função interessa saber qual a maior quantidade de memória utilizada pela função entre todas as amostras coletadas. Isso significa que, ao invés de incremento por amostragem, adotar-se-á o princípio do registro do maior valor, em número de *bytes*, de uso de memória constatado em cada função. Os principais recursos do módulo proposto são: a) o armazenamento das informações de uso de memória *heap* obtidas no processo de **Profiling** em formato apropriado para uso posterior por aplicações de co-projeto de hardware e software; e b) a geração de relatórios de **Profiling** que apresentem o volume de memória dinâmica alocada durante o processamento dos programas analisados para

que se possa identificar os locais onde esse uso é mais crítico, permitindo ao projetista tomar decisões quanto à reformulação do código fonte, ou quanto ao incremento no tamanho da memória a ser instalada no sistema, ou quanto à reformulação da arquitetura de um modo geral.

Abstract

Due to the advancement of the paradigms of development of electronic systems, new concepts, models and techniques resulted from this evolution, generating more efficient and objective tools. Among them, the system-level (ESL) electronic design automation (EDA) ones has brought a considerable increase to the productivity of electronic systems manufacturing, especially including the embedded systems. In what refers to elaborated systems, monitoring its execution and determining its operating profile are the essential tasks to assess, from its behavior, which points in this system represent bottlenecks or hot spots, affecting its overall efficiency. Thus, it is necessary to study the principles of verification and optimization of the elaborated systems that are better adapted to the new paradigms of projects development. The present work has as its aim implementing a processing module for data collection and analysis of C language written programs profile, which will run in soft core processors, like Altera's NiosII. However, unlike the statistics offered by the tool GProf (GNU Profiling) tool with respect to performance analysis, in which each sample obtained implies the increment of a counter to the function caught, this paper turns his interest to the analysis of memory usage profiling, which is especially found in volume allocated in each sample. Thus, for different samples of the same function, the matter is to know the most amount of memory used by the function among all samples collected. This means that instead of increasing sample we will adopt the principle of registration of the highest number of bytes of memory usage observed in each function. So, this tool's main features are: a) storing the information of memory use in the heap memory obtained in the process of **Profiling** in an appropriate format for later use by hardware and software codesign applications; and b) the reporting of **Profiling** that shows the dynamic memory volume allocated during analyzed programs processing so one can identify where such use is more critical, allowing the designer to make decisions regarding the reformulation of source code, or as to the increase in memory size to be installed into the system, or as to the architecture redesign.

Introdução

1.1 *Contextualização*

1.1.1 Avanços tecnológicos

Entre as áreas de estudo que envolvem a compreensão das relações humanas entre si e das pessoas com seu ambiente encontra-se a Economia. Uma de suas definições diz que "Economia é o estudo do esforço do homem para satisfazer suas necessidades ilimitadas utilizando-se dos recursos limitados de que dispõe e que escolhe com liberdade"((Rocha, 1975)). Dessas necessidades, são várias as que dependem de recursos tecnológicos para serem supridas, recursos esses que evoluem diariamente.

Os avanços tecnológicos vêm para prover a sociedade com recursos e ferramentas que facilitem as atividades cotidianas do ser humano e, numa etapa subsequente, tenham a sua operacionalização agilizada. Tais recursos e ferramentas podem ser constatados nas mais diversas áreas de aplicação do cotidiano do ser humano, sejam aplicações de uso geral ou restrito, cotidiano ou eventual, sejam, como ilustrado na figura 1.1, para eletrodomésticos (televisores, geladeiras, aspiradores de pó, robótica doméstica, etc.), equipamentos audiovisuais (tocadores de música, câmeras fotográficas e filmadoras, decodificadores de TVs por assinatura, etc.) e de comunicação (de telefones a satélites, passando por geradoras de sinais de rádios e TVs), ferramentas de trabalho (computadores, handhelds, smartphones, tablets, roteadores, placas de

redes, impressoras, máquinas para produção, robótica industrial), meios de transporte e de localização (automóveis, aviões, embarcações navais, embarcações espaciais, GPS), segurança (alarmes, robótica de vigilância territorial - pública ou corporativa) e exploração (robótica de mineração, robótica de exploração submarina, robótica de exploração sideral).

Figura 1.1: Ilustração da ampla gama de áreas e objetos que usam sistemas embarcados. (Vahid e Givargis, 2002)

Muitas dessas funcionalidades apresentam-se embutidas em dispositivos que contêm componentes integrados de hardware, ou um misto de hardware e software, os chamados SoCs,

ou *Systems-on-a-Chip* (Ghenassia, 2006). Esses sistemas são, por isso, conhecidos pelo termo Sistemas Embarcados.

Entretanto, a oferta constante de novos recursos e aplicações ao mercado implica em aumento de complexidade dos produtos que os portam e, em um mercado extremamente competitivo, de pressão por prazos de lançamento e disponibilização (*time-to-market*) exíguos e custos mínimos, ditados pela concorrência acirrada. Isso representa novos desafios a serem enfrentados para se alcançar o aumento da capacidade geral dos circuitos integrados de forma a comportar a complexidade exigida.

Esses desafios demandam soluções que permitam maior agilidade e eficiência na elaboração e no desempenho geral dos produtos fabricados. E essas soluções podem ser obtidas através do uso de ferramentas e de métodos de desenvolvimento eficazes que alcancem, da melhor forma possível, os objetivos de redução de prazos e custos e de incremento de benefícios e de eficiência dos produtos ofertados.

1.1.2 Desafios das novas tecnologias

Em 1965, Gordon E. Moore, então presidente da Intel, prognosticou um aumento de 100% no número de transistores em um chip a cada 18 meses, tendo, em 1975, revisado essa previsão para 100% a cada 24 meses (embora muitos considerem que o prazo de 18 meses ainda prevalece). Esse aumento tem-se observado de fato nessa razão e, como consequência, proporciona o lançamento de circuitos eletrônicos fabricados sobre pastilhas de silício (chips), no mercado, em curtos períodos de tempo e com capacidade crescente, em progressão geométrica, de implementação de recursos.

Os recursos e ferramentas apresentam-se na forma de produtos - tais como computadores, telefones móveis, televisores, dispositivos controladores de veículos de transporte e de eletrodomésticos, entre outros - que contêm funcionalidades diversas - tais como telecomunicação; interconexão; integração; obtenção, armazenamento e apresentação de imagens e sons; entretenimento; etc.

Na exploração comercial dos circuitos que implementam as aplicações anteriormente citadas - os chamados SoCs, ou Systems-on-Chip, que apresentam componentes tanto de hardware quanto de software -, o incremento de capacidade dos chips descrito suscita os seguintes fatos:

- O aumento na complexidade das aplicações, causado pela demanda de mais e mais funcionalidades incorporadas aos SoCs para serem oferecidas aos consumidores, complexidade essa que envolve não só as funcionalidades em si, mas também os atributos de

eficiência na sua execução, de autonomia de uso contínuo obtida com a redução no consumo de energia, de disponibilidade para uso a qualquer tempo e em qualquer lugar e de confiabilidade geral do sistema.

- Uma cada vez mais estreita janela de tempo (*time-to-market*) para concepção e elaboração de novos produtos para seu lançamento no mercado em prazo hábil que permita o maior retorno de investimentos (ROI, ou Return on Investment) possível. Estatisticamente, sabe-se que, em alguns setores, o primeiro fornecedor a chegar ao mercado com seu produto mantém, após a chegada dos concorrentes, pelo menos 60% da fatia de consumidores disputada (Ghenassia, 2006).
- Por conta da concorrência e da disposição do público consumidor, uma extenuante batalha para contenção de custos nos projetos de SoCs.

Se por um lado esses fatos representam benefícios para o mercado consumidor, por outro lado eles apresentam-se como obstáculos para o desenvolvimento desses produtos. Para superá-los, requer-se a adoção de ferramentas e de procedimentos que não só agilizem de forma segura o trabalho de elaboração dos sistemas, mas que também proporcionem produtos mais eficientes, abrangentes, confiáveis, econômicos e acessíveis.

1.1.3 As ferramentas EDA como solução para os desafios

As aplicações EDA (Electronic Design Automation) são ferramentas que auxiliam os projetistas a construir sistemas de forma mais ágil, rápida e segura. A partir de recursos de construção, utilização e reutilização de blocos funcionais, amparados por mecanismos de IDE (Integrated Development Environment) que facilitam a especificação e integração de componentes, sistemas podem ser desenvolvidos em condições favoráveis que permitam melhores resultados, seja em termos de ganho de tempo de desenvolvimento, seja em termos da qualidade geral do produto final.

Às ferramentas de auxílio ao projeto de sistemas eletrônicos devem-se alinhar métodos e técnicas de desenvolvimento que proporcionem o melhor aproveitamento dos recursos oferecidos. Na construção de um sistema eletrônico, o objetivo maior é chegar à configuração mais apropriada de um SoC de forma a se obter o melhor desempenho possível no menor prazo de entrega. Isso pode ser alcançado partindo-se de uma especificação em software, por ser presumidamente a que demandará menos tempo para ser concluída quando comparada à especificação em hardware. Executando um protótipo em software da solução pretendida sob monitoramento para análise de desempenho, os resultados obtidos das análises podem indicar

pontos do sistema que demandem otimização. Reformulações do software podem ser consideradas e efetuadas, realizando-se novas análises de desempenho. Após esses procedimentos, a especificação a que se chegou pode ser dividida, ou particionada, em diversos componentes, para que cada um tenha sua performance melhorada. Cada componente, cuja implementação poderá ser efetuada em hardware e/ou em software, poderá ser elaborado por diferentes equipes de trabalho, caracterizando um princípio de design denominado co-projeto. Depois de concluídos, os componentes então divididos serão integrados ao sistema original para nova análise de desempenho global. Caso o sistema ainda não atenda aos requisitos iniciais de eficiência, o processo de otimização é retomado, e o ciclo prossegue até que o sistema respeite as condições de funcionamento pré-estabelecidas para ele (figura 1.2). Como, nas etapas descritas, a análise de desempenho é feita sobre o perfil de funcionamento do sistema analisado, esse processo de análise é denominado *Profiling*.

Figura 1.2: Diagrama do ciclo de vida de um projeto de sistema embarcado. (Berger, 2002)

Para o caso do trabalho ora proposto, a ferramenta EDA para desenvolvimento de projetos em nível de sistema (ESL) abordada nesta monografia é a **RoboArch**, que está sendo concebida - por outros pesquisadores - para implementar as funcionalidades de modelagem, síntese, verificação, entre outras. A **RoboArch** está sendo desenvolvida no âmbito do ICMC/USP conforme

especificação definida em (Bonato e Marques, 2009). Entretanto, como esta ferramenta ainda está sob desenvolvimento, os recursos desenvolvidos neste trabalho serão implementados em uma ferramenta EDA equivalente, no caso, a que permite a implementação do processador *soft core* Nios II, a EDA Quartus II, da Altera.

Já no que concerne à implementação de uma aplicação de **Profiling** - para a obtenção de sistemas mais eficientes e econômicos - para a **RoboArch** e suas equivalentes, a implementação de uma ferramenta de **Profiling** a partir de um estudo aprofundado sobre o assunto é o tema aqui proposto para dissertação de mestrado. No capítulo sobre a proposta de desenvolvimento de projeto são elencadas algumas das opções de **Profiling** passíveis de implementação. Cada opção é mais ou é menos adequadamente aplicável a determinados componentes de um SoC.

1.2 **Objetivo Geral**

Atualmente, o uso de sistemas embarcados, além de difundir-se por um amplo domínio de aplicações, também apresenta uma alta gama de funcionalidades embutidas em um único aparelho. Basta vermos os recursos demandados, por exemplo, em um simples telefone celular, em que, mesmo sendo a comunicação de voz entre pessoas o objetivo principal desse aparelho, funções tais como acesso à Internet, câmera de fotos e de vídeos, tocador de músicas, receptor de rádio AM/FM e de televisão, envio de mensagens de texto (assim como audiovisuais), etc., são atrativos que cativam cada vez mais consumidores. Embora a capacidade dos chips atuais suporte todos esses recursos, o trabalho de elaboração dos sistemas que executarão as funções desejadas envolve, devido a sua sofisticação e complexidade, altos riscos e enormes esforços, o que é refletido nos custos finais de fabricação dos produtos.

Uma ferramenta como a proposta no Projeto **RoboArch**, juntamente com o módulo de **Profiling** ora proposto, vem a auxiliar os profissionais que concebem, projetam e elaboram sistemas embarcados, uma vez que, além de utilizar-se das mais modernas tecnologias de codificação em nível de sistema, permitirá um trabalho de forma integrada entre as equipes relacionadas aos projetos globais, em que se destaca, principalmente, o reúso de módulos e códigos prontos e testados, o que representa grandes ganhos de produtividade.

A funcionalidade de **Profiling** tem suma importância no co-projeto de SoCs por propiciar o direcionamento dos rumos do projeto em termos de particionamento dos diversos elementos que o compõem e cujas respectivas otimizações podem levar à otimização do sistema como um todo.

O presente trabalho aborda o tópico **Profiling** - ou seja, a técnica para análise de perfil de desempenho dos componentes de um sistema em cada um de seus diversos elementos e

níveis -, efetuando o estudo e a compilação de princípios que envolvam esse tema. A despeito das diversas técnicas de **Profiling** (enumeradas e descritas mais à frente nesta monografia), estabeleceu-se foco na obtenção de perfis de desempenho de módulos processadores *soft cores* a serem usados nos projetos implementados na RoboArch, mais especificamente do processador NIOS II, da Altera (Alt, 2012).

Entretanto, em vez de percorrer a rota do **Profiling** de desempenho como boa parte das ferramentas de **Profiling** existentes faz, este trabalho segue o caminho do estudo do perfil de uso de memória em um sistema embarcado. Quando se efetua uma análise de uso de memória de um sistema, o que se pretende é saber quem (quais partes do código) está gerando o maior consumo de memória, pondo o programa em risco de apresentar *memory overflow*. Com isso, o programador poderá tentar reduzir o uso de memória em seu programa ou aumentar o tamanho da memória a ser utilizada. Cabe observar que o consumo de memória deve ser considerado em termos da maior quantidade utilizada por cada função durante a execução do programa, pois é o ponto máximo que vai determinar o tamanho mínimo de memória necessário ao sistema.

Há várias abordagens de uso de memória que podem ser feitas: memória "heap"(usada em alocações dinâmicas feitas por funções como a "malloc" da linguagem C), memória "stack"(pilha de parâmetros de chamadas de funções e alocação de variáveis locais), memória total alocada (texto, dados, bss, "heap" e "stack"). Cada abordagem apresenta suas técnicas de apuração da quantidade utilizada, seja por detecção de comandos de alocação dinâmica de memória, seja por chamadas a serviços de sistema operacional que ofereçam informações sobre memória alocada, seja por qualquer outra forma.

A técnica escolhida neste trabalho envolve o uso de um pré-processador que permita a instrumentação do código fonte do programa analisado, instrumentação essa que faz com que seja gerado um arquivo onde são registradas as alocações/desalocações dinâmicas de memória durante a execução do programa. Semelhantemente ao **Profiling** de desempenho realizado pelo GCC/GProf (*GNU Compiler Collection/GNU Profiling*), a técnica aqui apresentada possui um módulo de processamento do arquivo gerado durante a execução do programa verificado, tendo como saída um relatório onde constam as funções que contêm comandos de alocação/desalocação de memória e as respectivas quantidades máximas alocadas. São as informações desse relatório que permitirão ao programador identificar os locais críticos de uso de memória e proceder à reformulação de seu código visando à diminuição da memória usada, ou a determinar a instalação de mais memória, ou a reformular a arquitetura de construção (bancos de memória) do sistema embarcado.

Futuramente, o presente trabalho poderá evoluir para o acréscimo de outras técnicas que monitorem o uso de todo tipo de memória, além da detecção de *cache misses*, *memory leaks*, dentre outras ocorrências.

1.3 *Objetivos Específicos*

Ao final do desenvolvimento do trabalho ora proposto, pretende-se ter concretizado:

- i. O estudo, compilação e descrição dos princípios e métodos inerentes a um módulo de **Profiling** de sistemas para ferramentas EDA, documentando, fundamentando e validando a implementação desse módulo.
- ii. O desenvolvimento de um módulo de **Profiling** para uso na ferramenta RoboArch que, como uma funcionalidade dessa ferramenta, permita ao projetista/programador analisar o perfil de uso de memória do(s) software(s) executado(s) no(s) processador(es) *soft core* NIOS II e efetuar os devidos ajustes, alterações e implementações no sistema que estiver sendo projetado.

1.4 *Justificativa*

No projeto e co-projeto de Sistemas Embarcados, as ferramentas de EDA podem atualmente contar com recursos que permitem avaliar o desempenho dos sistemas elaborados, porém, carecem de ferramentas que mostrem como o sistema faz uso das memórias acopladas ao SoC implementado.

Durante a elaboração de um sistema utilizando uma EDA, saber como estão sendo utilizadas as memórias *heap*, *stack* e *caches* pode levar ao aumento da qualidade do circuito gerado em termos da redução de riscos de ocorrência de travamentos que seriam provocados, em outras situações, por estouro de memória ou por algum tipo de seu mal uso.

O acesso à memória tem sido um dos grandes gargalos dos Sistemas Embarcados, principalmente naqueles que exigem uma grande movimentação de dados, como, por exemplo, os de processamento de imagens. Além de onerar o tempo de processamento, essa movimentação consome muita energia da bateria, o que compromete o sistema como um todo. Portanto, controlar, estudar e dimensionar esses acessos são de suma importância para desenvolvimento de Sistemas Embarcados.

Portanto, um trabalho que inicie a criação de um produto voltado à detecção de problemas com a alocação de memórias, começando pela análise da memória *heap*, onde são feitas alocações dinâmicas por funções do tipo `malloc` (da linguagem C) e assemelhadas, e, futuramente, avançando para verificações do uso da *stack* e do acesso às *caches*, é de suma importância para que o projetista aproxime-se do ponto ótimo entre quantidade de memória necessária e a minimização do seu tamanho. Tal minimização não só implica na redução da área total do

circuito projetado, implicando em uma maior compactação do produto final, como também na diminuição dos custos envolvidos.

Este trabalho visa a apresentar sua colaboração ao âmbito das pesquisas em torno da resolução dos problemas de uso de memória acima relatados.

1.5 Método de Pesquisa

Como método de desenvolvimento do produto almejado adotou-se o princípio de se construir um pré-processador *source-to-source* de programas fonte na linguagem C. Um programa fonte será consumido como entrada do pré-processador, o qual o instrumentará com os comandos necessários para que o programa fonte gerado na saída e compilado por um compilador padrão possa gerar as informações que propiciarão as devidas análises.

Esse pré-processador, aliado ao eventual uso de recurso do compilador padrão de geração de informações para análise de desempenho do sistema (por exemplo, o GProf, do compilador GCC), permitirão ao projetista buscar o equilíbrio entre o ganho de eficiência desse sistema e as melhorias no uso de suas memórias.

1.6 Organização da Monografia

Visando a propiciar a apresentação e o encadeamento lógico dos conceitos que envolvem o tema principal deste trabalho, qual seja, **Profiling**, a presente monografia estrutura-se em capítulos que definem os tópicos inerentes.

No Capítulo 3, Teoria sobre Sistemas Embarcados, apresentam-se os principais elementos para se entender o que são e para que servem os sistemas embarcados, estes desenvolvidos por projetistas/programadores de hardware e software para atender determinadas necessidades nas mais diversas áreas e utilizados nos mais diversos aparelhos e dispositivos.

Como o presente trabalho trata de aplicar as técnicas estudadas à Computação Reconfigurável, no que se refere aos modelos correspondentes, são abordadas questões que envolvem a arquitetura (SoC) e os níveis de abstração (RTL, ESL) no desenvolvimento de sistemas embarcados para serem implementados em FPGAs. E, em modelos de Alto Nível, são descritos os conceitos relativos ao ambiente de desenvolvimento de sistemas embarcados, tais como a forma de integração entre as equipes (Co-projeto), as ferramentas utilizadas (EDAs) e as formas de abstrair e modelar projetos (*platform-based*).

Também descreve-se a ferramenta de automação de projetos eletrônicos (EDA) desenvolvida no âmbito do ICMC/USP, a **RoboArch**, à qual o módulo **Profiling** proposto nesta monografia

será agregado, tornando-se uma de suas funcionalidades. Logo em seguida, é apresentado o tópico principal desta monografia, **Profiling**. Nele abordam-se os objetivos de se obter um perfil de desempenho de sistemas - principalmente sistemas embarcados, instados a funcionar sob rígidos padrões de eficiência -, bem como as principais técnicas adotadas para obtê-lo e, obviamente, a ferramenta de **Profiling** que se pretende desenvolver neste projeto de mestrado.

No Capítulo 4, é apresentada a ferramenta implementada, sua forma de trabalho e os resultados apresentados.

Finalizando, o Capítulo 5 tece considerações sobre as conclusões obtidas e as expectativas futuras para a ferramenta de **Profiling** implementada.

Sistemas Embarcados

2.1 *Sistemas Embarcados*

2.1.1 Definição

Em uma definição mais ampla, um sistema embarcado (ou embutido) é qualquer dispositivo que incorpore em si um computador programável, porém, sem o propósito de ser um computador de uso geral (Wolf, 2005). Ou seja, o uso de um dispositivo programável tem o objetivo de implementar uma aplicação configurável, ou até mesmo reconfigurável, com funcionalidades específicas a um determinado propósito (de telecomunicações, multimídia, controles de veículos, etc.), realizando um conjunto reduzido de tarefas. Por exemplo, um veículo pode apresentar dispositivos encapsulados contendo sistemas (compostos por circuitos, microprocessadores, softwares, etc.) que efetuem controles específicos sobre cada uma de suas partes. Cada dispositivo é responsável por controlar partes específicas do todo conforme as tarefas que lhe forem atribuídas, podendo haver ou não um controle geral dos dispositivos.

2.1.2 Restrições

Sistemas embarcados geralmente apresentam requisitos de qualidade e confiabilidade mais rígidos que aqueles exigidos a outros sistemas computacionais (Noergaard, 2005). É comum

encontrar rigorosas restrições no que se refere ao desempenho do sistema em termos de tempo de resposta de seus processos, à interação entre um sistema embarcado e o ambiente no qual está inserido, à precisão dos resultados computados pelo sistema, ao consumo energético do dispositivo embarcado, entre outros fatores.

Por sua natureza comum de relacionar-se a controles e aplicações de tempo real, que demandam respostas imediatas, a velocidade de processamento é um requisito importante a ser considerado em um projeto de sistema embarcado. E a interação de um sistema embarcado com o ambiente que o envolve dá-se de forma controlada, uma vez que certos requisitos devem ser respeitados e a capacidade de resposta do sistema aos demais dispositivos com os quais interage deve ser (Sangiovanni-Vincentelli, 2007) oportuna - ou seja, feita no momento exato em que é necessária, nem antes nem depois - e precisa - ou seja, os valores fornecidos devem estar corretos. Um exemplo a ser destacado, neste caso, é o dos circuitos acoplados a sistemas de freios ABS, que precisam receber os dados dos sensores, processá-los e devolvê-los de forma correta e em tempo hábil para que os acionadores funcionem no momento certo, evitando-se eventuais acidentes.

Há casos em que se exige que um sistema embarcado apresente múltiplas funcionalidades e seja flexível em suas configurações. As funcionalidades devem ser operacionalizadas sem que uma afete o desempenho da outra e todas caibam em um mesmo aparelho portátil. E os recursos devem poder ser reconfigurados, dentro de certos limites, quando necessário. É o caso de um dispositivo muito conhecido: os aparelhos de telefonia móvel, que possuem circuitos programáveis e que executam funcionalidades de comunicação com outros aparelhos - seja por voz, seja por outros tipos de sinal -, além de outras funcionalidades, como, por exemplo, acesso à Internet, câmera digital, Personal Digital Assistants (PDA), etc.

Devido ao seu uso comumente ligado a baterias, outro requisito de um sistema embarcado é apresentar baixo consumo de energia. Além disso, tais sistemas precisam ser instalados geralmente em invólucros pequenos e, às vezes, portáteis, para que possam caber em locais com pouco espaço ou na palma da mão de uma pessoa. Assim, o tamanho reduzido de um circuito de um sistema embarcado é outra restrição a ser respeitada na sua fabricação.

E, apesar de todas essas exigências, para que possa ser facilmente comercializado, ele deve, também, apresentar custo baixo (Vahid e Givargis, 2002).

2.1.3 Arquiteturas

Além do mais, para realizar as tarefas a que se propõem, os sistemas embarcados podem ser implementados sob diversos tipos de arquiteturas: a) com um único microprocessador; b) com múltiplos processadores (MPSoC, ou Multiple Processors Systems on-Chip) interconectados

(homogêneos - SMP/Symmetric Multiple Processor - ou heterogêneos - AMP/Assymmetric Multiple Processor) e funcionando em paralelo; c) como um dispositivo híbrido que se utilize de hardware e de software conjuntamente, com coprocessadores acoplados para acelerar o sistema, etc. Também nesses casos faz-se necessário o uso de sistemas operacionais (embarcados), entre outros, para controle da alocação de recursos aos processos e para controle e agendamento de processos prioritários nas aplicações de tempo real (RTOS - Real Time Operating System).

Um sistema embarcado pode ser implementado utilizando ASICs (Application Specific Integrated Circuit), circuitos reconfiguráveis (FPGAs - Field-Programmable Gate Array), processadores de sinais digitais (DSPs - Digital Signal Processing) e até mesmo uma combinação entre essas e outras opções. Os elementos usados dependem da aplicação e de uma análise dos componentes mais adequados a ela, seja em termos de obtenção de maior eficiência, flexibilidade, eficácia, ou qualquer outra característica desejada. Enquanto a eficiência pode ser alcançada com o uso de ASICs, por outro lado, caso a flexibilidade seja mais importante, esta pode ser obtida com as FPGAs, que, embora sejam um pouco mais lentas que os ASICs, apresentam, muitas vezes, desempenho compatível com os requisitos de velocidade necessários, além de permitirem um desenvolvimento mais rápido de aplicações.

2.1.4 Particionamento de Hardware e Software

O projeto de um sistema embarcado normalmente envolve componentes de hardware (microprocessadores, microcontroladores, aceleradores de hardware, etc.) e de software (os processos executados nos microprocessadores), o que demanda uma análise do sistema como um todo para que se tomem decisões sobre que partes serão implementadas em hardware e que partes serão implementadas em software. A isso chama-se particionamento em componentes de hardware e software (Berger, 2002) ou, simplesmente, particionamento de hardware e software. O fluxo de projeto e o ciclo de desenvolvimento de um sistema embarcado deve levar em conta a divisão desse projeto nesses componentes, efetuando a integração e a co-verificação das partes à medida que a conclusão das etapas de elaboração de cada uma assim o permitir.

A decisão sobre quais partes serão especificadas em hardware e quais em software estará pautada em critérios objetivos de desempenho geral exigido para o sistema e de tempo de desenvolvimento para lançamento do produto no mercado dentro da janela de tempo (*time-to-market*) apropriada.

2.1.5 Sistemas Embarcados na Robótica

Em robótica, mais especificamente na robótica móvel, em que se faz necessário um controle eficiente das ações do robô, o uso de sistemas embarcados é extremamente apropriado. Um robô, cujo projetista pretenda que ele se locomova de forma autônoma, precisa constantemente receber informações sobre o ambiente em que se encontra, processá-las e tomar decisões rapidamente sobre mover-se ou parar, acelerar ou reduzir sua velocidade, ir em frente, para trás ou virar (à esquerda ou à direita), entre outras ações. Percebe-se, portanto, a importância de se instalar sistemas que forneçam ao robô aptidões que lhe permitam controlar suas ações sem a necessidade de controle direto on-line feito por seres humanos. Esse papel é realizado por sistemas embarcados que controlam os sensores e atuadores (motores, acionadores de engrenagens, etc.) do robô, além, é claro, de lhe proporcionar capacidade de processamento das informações recebidas por seus sensores para que possa tomar em tempo hábil as decisões cabíveis, conforme sua programação prévia.

2.1.6 Otimização de Sistemas Embarcados

Percebe-se, pelo exemplo de aplicação em robótica acima exposto, a importância da otimização de sistemas para que os dispositivos embarcados funcionem dentro de restritos padrões de eficiência, precisão, tamanho, economia de energia, entre outros fatores. A exigência de eficiência se dá pela necessidade de respostas rápidas do sistema em boa parte dos casos. A precisão é desejada quando houver dependência do sistema por valores bem específicos para a correta disponibilização de recursos no tempo, no espaço e em qualquer ambiente em que isso se faça necessário. A preocupação com o tamanho ocupado também existe normalmente, uma vez que muitos aparelhos onde se devem embutir sistemas possuem dimensões pequenas. E a obtenção de economia de energia é desejada por permitir maior autonomia de funcionamento ao sistema. Conclui-se, portanto, que uma ferramenta que automatize o projeto de sistemas embarcados e, adicionalmente, ofereça uma funcionalidade que facilite a obtenção de melhores rendimentos dos sistemas nela projetados é muito bem-vinda aos meios de pesquisa e desenvolvimento.

2.2 Modelos de Implementação de Sistemas Embarcados

2.2.1 SoC

Misto de componentes de hardware e de software (Ghenassia, 2006), Systems-on-a-Chip (SoC) é um termo que designa a composição de vários desses elementos (microprocessadores para execução de determinados softwares, coprocessadores, barramentos de comunicação, etc.) em uma única pastilha (Vahid e Givargis, 2002). Essa concentração de diversos módulos funcionais em um único chip, permitida pelo avanço na tecnologia de fabricação de circuitos integrados cada vez mais compactos, refletiu sensivelmente na eficiência dos sistemas embarcados, pois tornou a comunicação entre os componentes mais rápida e o consumo de energia menor.

Um SoC pode ser composto por elementos programáveis e/ou customizados parcial ou totalmente.

Os elementos programáveis podem tanto ser o software que é executado nos microprocessadores do SoC quanto um hardware reconfigurável (por exemplo, FPGAs). Ambos serão especificados via descrição (hardware) e programação (software) por parte do(s) engenheiro(s) responsável(eis) por essa tarefa conforme os requisitos definidos para o sistema. O software pode tanto ser executado em um processador fabricado diretamente sobre silício quanto em um processador dito *soft core*, ou seja, um processador desenhado utilizando-se alguma linguagem de descrição de hardware (Verilog ou VHDL), o qual será implementado em hardware reconfigurável. Entre os processadores *soft core* existentes no mercado, podemos citar os mais conhecidos:

- NIOS II (Altera)
- Micro Blaze (Xilinx)
- Leon (Gaisler)
- Open Risc (OpenCores)
- Plasma (Steve Rhoads)
- Ultra Sparc V9 (Sun/Oracle)

Um elemento parcialmente customizado pode ser um ASIP (Application Specific Instruction-set Processor), que apresenta uma arquitetura otimizada para o processamento de uma aplicação específica, mas cuja funcionalidade (software) a ser nele executada pode ser também programada.

Já um elemento completamente customizado pode ser um circuito ASIC (Application Specific Integrated Circuit), que corresponde a um hardware definitivo, não-programável, gerado para uma determinada aplicação.

Todos os elementos que comporão um SoC deverão ser, pois, devidamente conectados entre si para permitir que se obtenha a funcionalidade desejada de forma adequada, segura, eficiente e com o menor consumo possível, de acordo com os requisitos definidos. Percebe-se, nesse caso, que a montagem de um SoC envolve a elaboração de componentes isolados que, em um dado momento, comunicar-se-ão uns com os outros. Em função disso, o design de cada um desses elementos pode ser efetuado de forma autônoma, embora interdependente. O design de um projeto de SoC demanda, pois, um trabalho inicial de particionamento das funções necessárias ao circuito para, então, distribuí-las entre as equipes responsáveis pelo desenvolvimento de cada bloco funcional, seja de hardware, seja de software. Esse trabalho de particionamento de um projeto é denominado co-projeto de hardware e software. Nele devem ser levadas em conta questões como comunicação entre os componentes no chip, sincronização de tarefas paralelas, adequação do software, além da otimização global do sistema.

2.2.2 EDA e Co-projeto de Sistemas

As ferramentas que têm por finalidade auxiliar na construção de SoCs de forma ágil, permitindo desde a elaboração até a verificação (do cumprimento das especificações) e validação (da correteza do produto em relação às expectativas do usuário), por simulação e testes, dos módulos especificados, são denominadas EDA (Electronic Design Automation). Essas ferramentas permitem que haja ganhos de desempenho e produtividade em geral, ganhos esses obtidos: com a maior interação entre as diversas equipes envolvidas no desenvolvimento dos respectivos projetos, as quais compartilham recursos comuns; com o reúso de módulos previamente elaborados; e com o aumento do nível de abstração na especificação e modelagem de projetos de SoC.

No que se refere aos aspectos da ferramenta que permitem que haja uma maior integração entre as equipes envolvidas no projeto, esse objetivo é alcançado a partir da disponibilização dos componentes desenvolvidos por cada grupo em bibliotecas compartilhadas de IPs, bem como pelo acesso de todas as equipes a todos os componentes disponíveis. Dessa forma, as equipes trabalham de forma concorrente e colaborativa sobre componentes de hardware e de software isoláveis do sistema, os quais serão, posteriormente, integrados entre si. Porém, para que isso possa ser realizado, faz-se necessário estabelecer padrões de trabalho que devem ser rigorosamente obedecidos, em que é preciso respeitar critérios de padronização de identificações e de procedimentos, critérios esses normalmente definidos por comitês ou consórcios mundiais de

grande aceitação, como, por exemplo, o padrão IPXACT (**Apêndice A**) definido pelo Consórcio SPIRIT (SPIRITConsortium, 2010). Só assim aos resultados obtidos poder-se-á atribuir um nível satisfatório de confiabilidade, o qual refletirá na qualidade global do produto final. Dessa forma, faz-se extremamente necessário garantir que métodos, técnicas e princípios sejam adotados no uso de uma ferramenta EDA.

Os métodos e ferramentas utilizados nesse caso proporcionam uma manipulação de componentes virtuais de hardware e de software previamente elaborados seguindo uma abordagem *platform-based*¹. Nessa abordagem orientada à integração de elementos, utilizam-se modelos que representam funcionalidades pré-definidas, disponíveis para uso e reúso, e que estão armazenados em bibliotecas, onde estão especificadas, para cada elemento, as composições válidas em termos de suas interligações e regras. Esses modelos, na prática, correspondem a componentes que, em uma implementação de um projeto, podem ser inseridos, removidos ou substituídos por outros componentes para criar a arquitetura desejada. Isso permite que a construção de arquiteturas se faça através de um processo de conexão de componentes pré-existent, armazenados em uma biblioteca própria, efetuando-se um refinamento contínuo em que cada elemento da biblioteca pode ser utilizado, reutilizado e/ou substituído no circuito alvo, até o projetista chegar ao resultado pretendido.

Além disso, as IDEs utilizadas nas ferramentas EDA devem adotar princípios intuitivos de elaboração de interfaces homem-computador, permitindo conexões entre componentes, a partir de canais de comunicação, em nível de transação, de onde a sigla, em inglês, **TLM (Transaction Level Modeling)** - originalmente, Transaction-based Modeling. Também convém, para que o desenvolvimento seja mais rápido e seguro, que a codificação utilize linguagem de alto nível - dita, no jargão da área, "em nível de sistema" ou, da sigla **ESL** em inglês, **Electronic System Level**.

Ao se permitir que os elementos de um sistema projetado sejam decompostos em módulos isoláveis e distribuídos para diversas equipes para que cada uma desenvolva a sua parte e, posteriormente, entregue-a para ser integrada às demais, tem-se como consequência o estabelecimento de um fluxo de projeto que contempla o desenvolvimento paralelo das atividades distribuídas.

Um caso específico de divisão de tarefas destaca-se então: as tarefas efetuadas sobre componentes de software e as tarefas efetuadas sobre componentes de hardware. Na elaboração de sistemas eletrônicos, contrapõem-se as questões que definem a escolha entre uma implementação mais voltada à síntese do circuito em hardware e outra mais relacionada à prototipação em software. Enquanto no hardware obtém-se maior ganho de desempenho na execução do

¹ Conforme (Carlioni *et al.*, 2005), "uma plataforma é uma camada de abstração que encapsula os detalhes de diversos possíveis refinamentos de implementação das camadas subjacentes".

sistema, no software o ganho dá-se na redução do tempo e, conseqüentemente, dos custos de desenvolvimento total (Gupta, 1995), uma vez que há redução das quantidades (e valores inerentes) de horas-homem envolvidas nos projetos, além de outros custos relativos ao tempo de elaboração.

Na prática, uma abordagem mista na elaboração de um sistema embarcado traz maiores vantagens mercadológicas, pois resulta em um produto com performance aceitável - dentro das requisições de sistema apuradas - lançado em um *time-to-market* apropriado. Dessa maneira, procura-se estabelecer um fluxo de projeto que prestigie a divisão, ou particionamento, do sistema desenvolvido em elementos de hardware e de software conforme a melhor conveniência para se obter o maior ganho na elaboração das partes e, conseqüentemente, do todo. Cada componente é, portanto, desenvolvido por sua respectiva equipe paralelamente ao desenvolvimento dos demais componentes, propiciando considerável ganho de tempo na finalização do projeto como um todo. E, depois de concluído cada componente, executam-se de forma global os testes necessários integrando-se as partes que compõem o sistema.

A esse processo de particionamento de sistemas em componentes de hardware e de software que serão desenvolvidos isoladamente para posterior integração ao todo, resultando em um trabalho coletivo e colaborativo do qual se obtém ganho de tempo de elaboração e de desempenho geral do produto final, chamamos de co-projeto de hardware e software. Uma ferramenta EDA deve permitir o desenvolvimento de projetos de sistemas conforme o fluxo de design de co-projetos.

As ferramentas EDA atuais consistem em uma evolução das formas de elaboração de sistemas, quais sejam, desde os, inicialmente, **diagramas esquemáticos de transistores**, seguidos pelo design em nível de **portas lógicas (gates)** e, posteriormente, pelo design em nível de **registradores - RTL (Register Transfer Language)** -, chegando à atualmente cada vez mais utilizada elaboração de projetos em nível de **sistema - ESL (Electronic System-Level)**, como apresentado na figura 2.1. Percebe-se, nesse processo evolutivo, que o desenvolvimento dos circuitos integrados está-se distanciando da especificação detalhada do hardware em prol do uso de ferramentas que apresentem alto nível de abstração. Detalhamentos sobre esses diversos níveis de representação de hardware podem ser vistos no **(Apêndice B)**.

O desenvolvimento em nível de sistema apresenta as vantagens de: facilitar o co-projeto de hardware e software, uma vez que os módulos particionados podem ser desenvolvidos de forma independente pelas equipes para, depois, integrarem-se entre si através do acoplamento de suas interfaces de I/O via canais de comunicação; proporcionar maior velocidade na simulação; permitir o teste e o ajuste da arquitetura do sistema em alto nível, o que garantirá o aumento de produtividade global a partir da possibilidade de se permitir que os testes sejam efetuados por cada equipe em seus respectivos módulos, o que proporciona redução de tempo nos testes finais

do sistema como um todo, uma vez que a maior parte dos problemas existentes em cada módulo já foi depurada; permitir o reuso do mesmo testbench, tanto na fase de validação do modelo como na de teste da arquitetura do sistema, utilizando, entre outros, os métodos definidos pela OVM¹ (Open Verification Methodology).

Figura 2.1: Diagramas em nível de: a) Transistor; b) Gates; c) RTL; d) ESL. (adaptado de (Dia, 2010))

As ferramentas EDA voltadas a FPGAs, largamente utilizadas na atualidade, oferecem uma abordagem RTL para a elaboração de projetos de sistemas. Entre as mais conhecidas temos:

- **Quartus II**, da Altera, bem como os produtos agregados, como o QSys/SoPC Builder e DSP Builder, para desenvolvimento de hardware, e o NIOS II, para desenvolvimento de software, todos da própria Altera, além dos produtos de outros fabricantes suportados

¹ A OVM é uma iniciativa conjunta, estabelecida entre a Graphics® e a Cadence® Design Systems, fundamentada em múltiplas plataformas para verificação de sistemas descritos em System-Verilog e que fornece uma biblioteca de classes com componentes modulares e reutilizáveis que se comunicam via interfaces padrões de TLM.

pelo Quartus II, tais como o ModelSim, da Mentor Graphics, para simulação de hardware, e o HandelC, da Celoxica, para desenvolvimento de hardware (Bonato e Marques, 2009).

- **ISE**, da Xilinx, também com seus produtos agregados (Platform Studio, System Generator e EDK).

Entretanto, já há, no mercado, ferramentas comercializadas (a altos custos) que oferecem a possibilidade de elaboração de projetos de sistemas em nível ESL. Cada qual apresenta vantagens e desvantagens, porém, todas são ferramentas proprietárias. A seguir são apresentados alguns exemplos de ferramentas dessa natureza.

- **Magillem 4.0**. http://www.magillem.com/index.php?page_id=Overview&ANIM=ESL
- **NAUET Design Assembler** (Matai Tech). <http://www.mataitech.com/>
- **Platform Express** (Mentor Graphics). http://www.mentor.com/products/esl/news/platform_express_supports_spirit
- **coreAssembler** (Synopsys). https://www.synopsys.com/dw/doc.php/ds/o/coretools_ds.pdf
- **Vergil - Ptolemy II** (Berkeley University). <http://ptolemy.berkeley.edu/ptolemyII/>
- **Metropolis Project**. <http://portal.acm.org/citation.cfm?id=766189>
- **PeaCE**. <http://portal.acm.org/citation.cfm?id=1255461&dl=GUIDE&coll=GUIDE&CFID=73273195&CFTOKEN=13372329>
- **Aldec**. <http://www.aldec.com/>

Além de permitir o desenvolvimento ágil de projetos de sistemas eletrônicos, uma ferramenta EDA deve propiciar a elaboração de sistemas complexos, de alto desempenho, confiáveis, a um custo reduzido. Para isso, ela deve oferecer um modelo de desenvolvimento de projetos que, além de otimizar cada etapa¹ do trabalho, permita que o próprio produto em si possa ser otimizado em termos de velocidade de processamento de suas funcionalidades, de redução no consumo de energia, de uso de recursos de memórias (cache, RAM, etc.), entre outros elementos que caracterizam o desempenho global de um sistema.

¹No PU, ou Processo Unificado, são consideradas as etapas de Concepção, Elaboração, Construção e Transição (LARMAN, 2008).

Otimizar as etapas de desenvolvimento significa garantir que: a Concepção e a Elaboração do produto sejam efetuadas de forma rápida, permitindo a sua modelagem em alto nível, sem que haja, inicialmente, preocupação com detalhes técnicos de implementação; a Construção do produto - que poderá ser dividida em módulos que serão desenvolvidos por equipes distintas - possa ser efetuada utilizando-se recursos que otimizem os trabalhos; na fase de Transição do produto, haja a utilização de mecanismos de verificação e testes - com o uso de simuladores, etc. - que permitam agilmente corrigir eventuais erros no projeto.

Já otimizar o produto em si significa melhorar a qualidade global do sistema desenvolvido. Nesse caso, o desafio é balancear a excelência de suas propriedades, respeitando os requisitos mínimos exigidos para o sistema, com o fator *time-to-market*. A utilização de uma ferramenta que propicie buscar tal excelência adjacente a uma ferramenta EDA vem ao encontro do objetivo de se alcançar a otimização do processo de desenvolvimento de um sistema eletrônico concomitantemente à otimização do próprio projeto desenvolvido. Tal ferramenta, cujos princípios de trabalho serão abordados nos próximos capítulos, denomina-se **Profiling**.

Para o caso do trabalho ora desenvolvido, a ferramenta EDA para desenvolvimento de projetos em nível de sistema (ESL) abordada nesta monografia é a **RoboArch**, que está sendo concebida - por outros pesquisadores - para implementar as funcionalidades de modelagem, síntese, verificação, entre outras. A **RoboArch** (figura 2.2) está sendo desenvolvida no âmbito do ICMC/USP conforme especificação definida em (Bonato e Marques, 2009). Entretanto, enquanto a **RoboArch** não atingir um estágio de desenvolvimento que permita testes da ferramenta de **Profiling** aqui apresentada, as verificações e validações que se fizerem necessárias serão realizadas no processador *soft core* Nios II, da Altera.

Figura 2.2: A ferramenta RoboArch. (Bonato e Marques, 2009)

Técnicas de Profiling

3.1 Definição

Como dito no capítulo anterior, a elaboração de um sistema em software dá-se em prazo menor que a elaboração do mesmo sistema em hardware. Em contrapartida, a implementação em hardware é mais eficiente que a implementação em software. O ponto de equilíbrio pode ser atingido iniciando-se da implementação mais rápida (em software), ou seja, o sistema começa consistindo em um processador. A partir de então, efetuam-se testes no sistema, monitorando sua execução para coleta de dados sobre seu perfil de funcionamento. Após coletados esses dados, analisam-se os pontos críticos e gargalos do código observado, substituindo-os, conforme o caso, por implementações em hardware, até que se atinjam os requisitos de desempenho desejados. A esse processo geralmente recorrente de coleta e análise de dados dá-se o nome de **Profiling**. O princípio ora descrito representa a concepção geral da técnica utilizada para obtenção do **Profiling** de um sistema. Adicionalmente, em termos mais específicos, há diversas técnicas de monitoramento e análise de dados que implementam o princípio geral de um **Profiling**.

Elaborado um projeto de sistemas, saber se sua eficiência global encontra-se no nível máximo (de benefícios) que pode ser explorado ainda é uma questão a ser avaliada posteriormente à confecção do projeto, por mais que o projetista tenha se preocupado em atingir os melhores resultados, durante o processo de desenvolvimento do sistema, através: de uma análise e es-

colha de algoritmos apropriados para a resolução do problema em questão; de uma codificação enxuta das instruções necessárias para implementar o algoritmo escolhido; da utilização de aceleradores de hardware onde, à primeira vista, isso pareceu ser conveniente; de remanejamentos de módulos na arquitetura do sistema; e de quaisquer outras medidas que lógica ou supostamente apontassem para a melhor solução.

A despeito da ampla disseminação de Sistemas Embarcados nos dias de hoje, seu desenvolvimento ainda é fortemente afetado, como observado mais à frente, por gargalos que representam obstáculos não só às etapas para sua concepção geral, mas também à obtenção de bons resultados finais. Os fatores complexidade (qualidade funcional), time-to-market, área total ocupada, segurança, custo (da área fabricada) e desempenho geral (eficiência, uso de memória, consumo energético, etc.) apresentam um alto grau de inter-relacionamento, uma vez que alcançar bons resultados em um desses fatores pode implicar em perdas (*trade-offs*) sensíveis em um ou mais dos demais fatores. Portanto, encontrar o ponto de equilíbrio otimizado entre todos os fatores envolvidos é um desafio a que todos os profissionais da área lançam-se diariamente, desafio esse em que cada pequeno avanço é considerado uma grande contribuição na busca por *tecnologias mais eficientes, abrangentes, confiáveis, econômicas e acessíveis*. Portanto, aprofundando um pouco as questões levantadas no final do capítulo Sistemas Embarcados desta monografia, para a obtenção de resultados satisfatórios, devem-se superar as questões relacionadas:

- i. À **complexidade** das aplicações e respectivas funcionalidades. Quanto mais funcionalidades uma tecnologia apresentar mais atrativa ela se torna, ao mesmo tempo em que maiores recursos (linhas de código, circuitos, etc.) são demandados, implicando em mais tempo de desenvolvimento exigido, além de aumentar os riscos relacionados à qualidade final dos produtos.
- ii. Ao prazo concedido pelo mercado para elaboração do produto e sua disponibilização ao consumidor (*time-to-market*). Esse prazo está cada vez mais reduzido, uma vez que há muitos concorrentes investindo na evolução de seus produtos, o que exige tempos cada vez mais curtos na obtenção de resultados.
- iii. Aos **custos de** concepção inicial de produto (**NRE** - *Non-Recurrent Engineering*) e custos contínuos de produção. O primeiro envolve a produtividade no design dos circuitos, aí consideradas todas as etapas previstas para o ciclo de desenvolvimento de um sistema, quais sejam, levantamento de requisitos (Concepção), projeto (Elaboração), implementação & testes (Construção) e, eventualmente, integração (Transição; consiste na delegação para utilização da ferramenta a candidatos a serem seus futuros usuários, para realização de testes com visões diversificadas sobre seu uso). Esses custos, embora incidam apenas uma

vez no orçamento geral do projeto, podem determinar a viabilidade econômica deste. Já os custos contínuos de produção envolvem valores de fabricação, tais como mão-de-obra, matérias-primas (proporcionais à área total do dispositivo fabricado) e outras despesas inerentes. O custo final do produto deve ser tal que permita o retorno sobre os investimentos (ROI) efetuados sem que o preço praticado torne-se inacessível ao público alvo.

- iv. Ao **desempenho**: Para que haja aceitação de uma tecnologia pelo mercado, além do fator custo, uma questão primordial é a eficiência no desempenho do sistema oferecido. Assim sendo, sua concepção deve buscar uma implementação otimizada que permita - respeitando os fatores anteriores - oferecer aos usuários um produto que apresente desempenho no mínimo satisfatório, embora, por fatores mercadológicos, espera-se que o desempenho obtido seja sempre o melhor possível. Além da eficiência, considera-se também como quesito de desempenho a ser avaliado em um SoC a questão da economia de consumo energético do sistema, uma vez que, por normalmente dependerem de baterias para o seu funcionamento, a autonomia de uso é um fator altamente considerado pelos consumidores quando da sua aquisição.

3.2 EDAs e Profiling

As pesquisas realizadas em todo mundo na área de Sistemas Embarcados buscam incessantemente por técnicas e ferramentas que possibilitem a elaboração de sistemas eletrônicos de forma a permitir que se atinja o nível de complexidade e desempenho almejados em prazo e custo mínimos (**Figura 3.1**), conforme as exigências do mercado. E nessa busca são utilizados diversos métodos e ferramentas que possam trazer ganhos de produtividade na concepção e implementação de projetos e de performance dos produtos gerados. Entre as ferramentas utilizadas com esse objetivo temos as EDAs (*Electronic Design Automation*), adotadas na concepção, elaboração e otimização de projetos de sistemas eletrônicos.

Dessa forma, uma ferramenta EDA que se proponha a superar os obstáculos acima descritos e a alcançar o patamar de tecnologia que ofereça as propriedades anteriormente relacionadas deve apresentar as seguintes características:

- a) **Permita aos desenvolvedores trabalharem de forma colaborativa seguindo princípios de fluxo de projeto decomponível (particionado)**: Um fluxo de projeto (**Figura 3.2**) especifica a sequência de passos a ser seguida na elaboração do projeto como um todo (especificação de requisitos funcionais, definição arquitetônica, modelagem, implementação,

Figura 3.1: Evolução desejada para os Sistemas Embarcados.

testes, síntese). No início de um projeto, é difícil saber qual a melhor combinação de recursos para implementá-lo, ou seja, quais funcionalidades convêm serem implementadas em software e quais convêm serem implementadas em hardware; em outras palavras, qual o melhor particionamento de tarefas a ser definido. Como possíveis soluções, pode-se partir de especificações totalmente no domínio de hardware ou totalmente no domínio de software (Tong e Khalid, 2007). Na abordagem inicial em hardware, o projetista poderá analisar e alterar partes de seu projeto para software conforme os requisitos de desempenho do sistema. Na abordagem inicial em software, o projetista poderá verificar o desempenho do sistema utilizando um **Profiler** para detectar os pontos críticos da aplicação, implementando em hardware as funções que mais afetam o desempenho global. A segunda abordagem (software) consiste na forma mais ágil de se iniciar a definição de um projeto, o qual, portanto, é especificado, primeiramente, como um algoritmo que descreve seus módulos funcionais e, logo em seguida, codificado como um protótipo em uma linguagem adequada (normalmente C ou C++). Tal codificação é submetida, então, a uma análise de desempenho com o objetivo de se perseguir a otimização do sistema. A otimização pode ser obtida por medidas que en-

volvem desde a adoção de algoritmos mais eficientes e/ou a recodificação do software (ou de partes dele) até a substituição das partes críticas do código em software por implementações em hardware, as quais, supõe-se, apresentarão desempenho sensivelmente maior. Cabe observar aqui que as questões relacionadas a essa otimização compõem o elemento principal desta dissertação, vindo a ser abordadas mais profundamente logo a seguir, quando do desenvolvimento do tópico 3.4. Quando ocorre um particionamento de projeto, os subprojetos gerados podem ser adotados por equipes distintas, embora interdependentes, e os respectivos desenvolvimentos podem-se dar de forma paralela, obtendo-se, com isso, ganho de tempo no desenvolvimento do projeto como um todo. Assim sendo, uma ferramenta EDA deve propiciar o desenvolvimento de co-projetos dentro dos princípios ora descritos, permitindo a integração, no momento em que isso se fizer necessário, dos componentes gerados, bem como a verificação, tanto isolada quanto unificada (coverificação), desses componentes.

Figura 3.2: Fluxo de projeto SoC típico. (Tabbara e Kashmi, 2004)

- b) **Reúso de componentes:** Uma ferramenta de modelagem de sistemas - mono ou multi *cores* - que seja *platform-based*, ou seja, que explore o reaproveitamento de componentes previamente elaborados (Figura 3.3), testados e armazenados em uma biblioteca para esse fim, vem ao encontro das medidas necessárias para a agilização do desenvolvimento de proje-

tos e as consequentes reduções de tempo de conclusão das tarefas e de riscos e custos envolvidos. Sempre que necessário, o programador poderá buscar, nessa biblioteca, módulos prontos que desempenhem as funções desejadas ou que mais se ajustem a elas (Carloni *et al.*, 2005), atendendo às suas expectativas e economizando-lhe o tempo que despenderia na sua elaboração. E, havendo mais de uma opção de módulo compatível com a funcionalidade requerida, um processo de coleta de dados para a elaboração de um **Profiling** sobre cada alternativa adotada pode levar à escolha do componente que ofereça as melhores condições de execução do projeto como um todo.

Figura 3.3: Reaproveitamento de componentes armazenados em biblioteca.

- c) **Especificação/Implementação em nível de sistema (ESL):** Segundo (McDonald, 2005), aproximadamente 70% dos projetos de sistemas embarcados alcançam não mais que 90% do desempenho esperado, sendo que cerca de um terço não chega a mais que 50% das metas estabelecidas. E cerca de 30% dos projetos atendem às especificações funcionais somente em até 50% do que fora estipulado inicialmente. Vale destacar ainda que, conforme essa mesma pesquisa (realizada pela Embedded Market Forecasters, segundo citado por McDonald), por volta de um terço dos projetos apresenta atraso médio de quatro meses em sua conclusão.

Nos *designs* de SoCs, a modelagem funcional que utiliza abstração em nível algorítmico oferece melhores condições para se obter maiores ganhos de produtividade, bem como maior

confiabilidade sobre os circuitos gerados, diferentemente da especificação em nível lógico (no caso, RTL), que, por sua complexidade, tende a tornar a elaboração do projeto mais morosa e mais sujeita a riscos. Também nos processos de verificação, e mesmo de coverificação, esse fato é constatado, haja vista que as simulações realizadas sobre modelos algorítmicos são mais rápidas do que as efetuadas sobre o nível lógico (Ghenassia, 2006). Portanto, na elaboração de sistemas eletrônicos, níveis mais elevados de abstração elevam a produtividade e levam à obtenção de maior eficiência e confiabilidade na conclusão dos projetos.

E para evitar perda de tempo com a identificação e a compreensão de um módulo de biblioteca, quando do reaproveitamento de componentes, visando-se efetuar a devida escolha e integração do componente funcional mais apropriado ao projeto, faz-se necessário que tanto a documentação básica dos componentes quanto a especificação dos módulos seja feita de forma clara, precisa e concisa. É nesse sentido que se pode afirmar que quanto mais alto o nível de abstração da especificação de um módulo, mais adequado ele se encontra para ser utilizado no projeto de outros sistemas de forma a alcançar os objetivos de redução de tempo de desenvolvimento, custos e riscos.

Atualmente, os projetos desenvolvidos em nível de sistema eletrônico (ESL) são os mais compatíveis com o nível de abstração ora descrito. Também nesse nível cabe falar da adoção de técnicas de análise de desempenho e utilização de recursos com vistas à otimização global do sistema, seja essa otimização feita em termos de reformulação algorítmica, seja em termos de substituição ou inserção de componentes de hardware, seja em termos de modificações arquitetônicas em geral, sendo que a estratégia a ser adotada dependerá da avaliação das partes dos componentes ou do componente como um todo dentro da arquitetura especificada. Tal avaliação é efetuada a partir de levantamentos do perfil **Profiling** de funcionamento dos componentes envolvidos, assunto a ser abordado em seção posterior deste capítulo e que, como já dito, corresponde ao tema principal desta monografia.

3.3 O Estado da Arte

O Estado da Arte em termos de ferramentas de Profiling pode ser considerado o produto Valgrind ((Val, 2011)). Essa ferramenta consiste em um framework de instrumentação para a construção de ferramentas de análise dinâmica. Ela executa sobre Linux (X86, AMD, ARM, PPC64, S390X), Android (ARM) e Darwin (X86 e AMD), sendo *Open Source/Free Software* sob a licença GNU GPL versão 2.

A Valgrind é composta por diversos módulos funcionais, entre os quais destacam-se:

- A) MEMCHECK: Detecta acesso a memória em áreas ainda não alocadas, áreas já liberadas, áreas além do final de blocos de heap, áreas inacessíveis da stack; uso perigoso de variáveis não inicializadas; vazamento de memória (memória não liberada após seu uso); má liberação de blocos de heap (liberação dupla ou liberação incompatível); passagem de código fonte sobreposto e blocos de memória de destino a `memcpy()` e outras funções correlatas (`strcpy()`, etc.);
- B) CACHEGRIND: É um profiler de cache, que identifica cache misses, referências de memória e instruções executadas em cada linha de código (por função, por módulo, programa inteiro).
- C) CALLGRIND: É uma extensão do Cachegrind. Provê as mesmas informações que o Cachegrind e mais as informações de callgraph. Apresenta interface visual, que permite ver também as saídas do Cachegrind.
- D) MASSIF: É um profiler de heap. Apresenta a utilização da heap no decorrer do tempo e quais partes do programa são responsáveis pela maioria das alocações.
- E) HELGRIND: É um debugger de threads. Detecta concorrência de dados em programas multi-threading e procura por áreas de memória acessadas por mais de uma (POSIX p-)thread sem haver um bloqueio adequado (pthread-mutex).
- F) DRD: Detecta erros em programas C/C++ com multi-threading.
- G) LACKEY e NULGRIND: Usados para propósitos de testes e demonstrações.
- H) Além de outras ferramentas criadas para a VALGRIND (ver <http://valgrind.org/download/variants.html>).

Entretanto, cabe observar que a Valgrind não é portada para o processador Nios II e nem para os sistemas operacionais que são executados nesse processador *soft core*. Portanto, o presente trabalho tem a intenção de iniciar a inserção de recursos existentes na Valgrind em um produto voltado para Sistemas Embarcados e que envolva computação reconfigurável.

Uma outra ferramenta de **Profiling** que vale a pena destacar é a **PIN**, desenvolvida pela Intel em parceria com a Universidade do Colorado, EUA. Ela está publicamente disponível para Linux nas arquiteturas IA32 (32 bits x86), EM64T (64 bits x86), Itanium[®] e ARM (Luk *et al.*, 2005).

Projetada para ser, na medida do possível, independente de arquitetura, ao mesmo tempo em que permite acesso a informações específicas a uma determinada arquitetura, a PIN utiliza-se de compilação dinâmica (JIT - Just In Time *compiler*) para instrumentar códigos executáveis durante seu processamento. Multifuncional, entre suas principais funcionalidades estão atuar

como *profiler*, efetuar simulações de *cache*, realizar análises de rastreamento e verificar erros de memória.

Pretendendo ser um ferramenta de fácil utilização, portátil, transparente, eficiente e robusta, a PIN implementa técnicas de realocação de registradores (*register reallocation*), incorporação de sub-rotinas (*inlining*), análise de execução (*liveness analysis*) e agendamento de instruções (*instruction scheduling*).

Entretanto, assim como a Valgrind, a PIN não é, ainda, voltada à Computação Reconfigurável.

3.4 O Módulo Profiling

Fazer o **Profiling** de um sistema significa efetuar levantamentos estatísticos envolvendo as diversas etapas e processos inerentes a esse sistema com o objetivo de obter subsídios para que se possa efetuar sua otimização (Conte *et al.*, 1996). Os dados para os levantamentos estatísticos são obtidos, normalmente, através do monitoramento de execuções do sistema que se procura otimizar. A partir de indicadores que mostram o volume de participação de trechos do sistema no processamento geral da aplicação analisada pode-se, manual ou automaticamente, identificar as partes mais influentes na execução geral do sistema e estudar possíveis soluções para obter ganhos de desempenho e utilização de recursos, seja com relação ao incremento na velocidade de processamento, na alocação de memória, na redução do consumo de energia, ou em qualquer outro ponto passível de ser aperfeiçoado.

Cabe observar que, quando se fala em ganho de desempenho, um amplo espectro de fatores deve ser considerado. Pode-se obter ganho de desempenho através de procedimentos que: a) levem à execução mais rápida dos processos envolvidos (pela inserção de hardware acelerador, pela melhoria da configuração de hierarquias de memória, pela inclusão de controladores de DMA, etc.); b) que proporcionem a redução do consumo de energia do circuito; c) que conduzam à redução da área moldada do circuito; d) até mesmo que propiciem a redução da ociosidade de componentes inseridos no sistema para serem executados em paralelo, sejam esses componentes processadores homogêneos ou heterogêneos, sejam eles coprocessadores e outros circuitos de hardware inseridos especificamente para aceleração de partes do sistema.

3.4.1 Aplicações

Através da adoção de técnicas de **Profiling** sobre sistemas em geral podem-se identificar os mais variados gargalos de desempenho em sistemas, permitindo-se, com isso, analisar cada caso

e buscar soluções para melhorar sua eficiência ou mesmo obter economia de recursos. Duas das principais aplicação de análise de **Profiling** são:

- a) Otimização de sistemas eletrônicos: Ao detectar pontos críticos em um sistema eletrônico, o projetista programador pode, a partir das informações fornecidas, procurar por soluções para melhorar o desempenho geral do sistema, tais como substituir partes executadas em software por partes processadas por hardware que acelere os cálculos.
- b) Otimização de códigos em compiladores: Em geral, os compiladores, ao efetuarem conversão de um código fonte para um código objeto, permitem, de forma parametrizada, o uso de procedimentos para analisar, a partir de execuções subsequentes do código objeto criado, o perfil das rotinas e instruções geradas e, utilizando-se de determinadas técnicas previamente concebidas, automaticamente otimizar o código objeto em compilações posteriores, o que consiste em uma forma de **Profiling** dentro dos princípios preconizados para essa função (coleta de informações visando à melhoria geral do sistema) (James e Larus, 1996).

3.4.2 Monitoramento

Como dito anteriormente, os dados para **Profiling** são obtidos, em geral, através do monitoramento de execuções do sistema desenvolvido. Isso ocorre porque a simples especificação dos códigos dos programas e dos componentes do sistema projetado não indica o comportamento dinâmico do sistema, uma vez que esse comportamento normalmente depende das entradas fornecidas (valores fornecidos pelos usuários ou por sensores, tabelas, etc.). O número de vezes que ocorrem iterações de trechos de códigos ou que determinadas funções do código são executadas em muito depende das entradas aplicadas.

Há diversas formas de se efetuar o monitoramento de um sistema. A seguir são citadas classificações de formas de monitoramento sugeridas em (Ravagnani, 2007).

Uma das formas de classificar um monitoramento é em termos da localização do monitor em relação ao sistema, podendo ser síncronas ou assíncronas. No monitoramento síncrono, o módulo de coleta e armazenamento de dados, bem como os sensores que detectam eventos provocados por instruções, funções ou circuitos, encontra-se dentro do sistema. Nota-se que essa intrusão propicia que ocorram *overheads*, ou seja, o sistema em análise pode ter seu comportamento afetado pelo módulo acoplado para monitoramento de sua execução, influenciando os resultados obtidos. Essa influência pode ser relevante ou não, conforme os resultados alcançados devam ser considerados em termos absolutos ou relativos. A retirada dos monitores após se ter obtido o **Profiling** implica em uma redução de tempo proporcional, entre as funções registradas, dos tempos medidos. Se o que se deseja é saber quais os pontos que consomem mais

tempo entre todos (influência relativa), a retirada dos monitores não fará diferença nos objetivos pretendidos, podendo até ser deixados, caso se queira que os resultados obtidos durante a fase de desenvolvimento do sistema se mantenham durante o seu uso em produção. Porém, se o que se deseja é saber se o tempo consumido por cada função apresenta *throughput* suficiente para a aplicação desenvolvida (influência absoluta) - por exemplo, em uma aplicação de tempo real -, a execução com e sem monitores poderá apresentar diferenças sensíveis que invalidam a análise e outra forma de se obter o **Profiling** deve ser considerada. Já o monitoramento assíncrono, em que o módulo monitor encontra-se fora do sistema, não gera o *overhead* acima referido, porém, apresenta custo superior em relação ao síncrono, uma vez que um circuito adicional e, normalmente, específico para o sistema deve ser desenvolvido e acoplado.

Um monitoramento, seja ele síncrono ou assíncrono, pode ser realizado em diversos níveis de abstração, conforme o nível de detalhamento que se deseje obter. Um monitoramento em nível de sistema apresenta uma visão lógica geral do sistema como um todo, sendo mais útil para se saber sobre o tempo total de execução do sistema, sendo este nível de pouca ou nenhuma serventia (exceto para o caso em que se comparam arquiteturas distintas, cogitando-se a substituição e/ou eliminação de blocos componentes) em um **Profiling** que vise obter informações suficientes para otimização do sistema. Em nível de processo, observam-se os processos de um sistema, a comunicação e sincronização entre eles, os estados de cada processo e os eventos que podem alterá-los. Entretanto, assim como no nível de sistema, o nível de processo também não oferece detalhamento apropriado para a otimização do sistema. Esse objetivo passa a ser atingido a partir do nível de função, em que, sabendo-se quais procedimentos e funções demandam mais tempo para serem executados, pode-se tomar decisões sobre novas implementações desses componentes. E, no nível de maior detalhamento, o nível de instrução permite obter informações sobre diversos eventos registrados no hardware - seja do processador, seja de um eventual hardware adicionado -, tais como acessos à memória, acessos à *cache*, ocorrências de interrupções, *system calls*, paralisações de ciclos (*stall cycles*) de cada instrução, etc.

Monitoramentos de sistemas podem ser feitos por software, por hardware ou por ambos (híbrido). O monitoramento por software, embora mais intrusivo - pois se faz necessária a inclusão de código dentro da aplicação, o que afeta os resultados obtidos -, é mais flexível, já que os monitores são colocados pelo projetista em pontos de sua escolha conforme conveniência e necessidade. O monitoramento por hardware utiliza-se de um hardware passivo que observa o barramento e os sinais do sistema ao qual está conectado. Obtém-se, com isso, um monitoramento não intrusivo, que não demanda alterações no sistema monitorado e, com isso, não afeta os resultados de sua execução. Porém, as informações desse tipo de monitoramento restringem-se àquelas obtidas no(s) ponto(s) em que o hardware de monitoramento é inserido. Além do mais, ao contrário do monitoramento por software, que pode utilizar-se de progra-

mas genéricos, o monitoramento por hardware implica na criação de módulos específicos (que, muito provavelmente, não poderão ser utilizados em outros projetos) para se observar cada sistema, possivelmente até servindo apenas para determinadas partes do sistema, o que significa custo, pelo menos na fase de desenvolvimento do produto (se o módulo não puder ser retirado na fase de produção, esse custo estender-se-á para essa fase também). O uso pontual desse monitoramento também significa baixo número de informações para análise. Um método que concilia monitoramento híbrido (por software e por hardware) procura diminuir as desvantagens de cada um desses métodos adotado isoladamente. Nesse princípio, insere-se algum código em software como o menor número possível de instruções para gerar sinais específicos que serão coletados por um hardware externo. Com isso, o "ruído" provocado no sistema pelo software é minimizado, enquanto se obtém algum nível de flexibilidade que o hardware sozinho não ofereceria. A Figura 3.4 apresenta uma classificação dos tipos de monitoramento conforme suas propriedades, vantagens e desvantagens.

Tipo de Monitoramento	Nível de Abstração	Tamanho do Log de Eventos	Overhead da Instrumentação	Flexibilidade
Hardware	Baixo	Grande	Baixo	Baixa
Software	Alto	Pequeno	Alto	Alta
Híbrido	Alto	Pequeno	Baixo	Alta

Figura 3.4: Características dos tipos de monitoramento (Ravagnani, 2007).

Podem-se coletar dados de um sistema monitorado de diversas formas, as quais podemos classificar em orientadas por relógio e orientadas por evento. Nas orientadas por relógio, faz-se uma observação dos estados dos processos monitorados em períodos pré-determinados e constantes de tempo. Por exemplo, o PC (*program counter*) de um processador ficar constantemente sob observação. A cada período de tempo, obtém-se o endereço da instrução que está sendo executada, encontra-se a qual função ela pertence e incrementa-se, em uma tabela, um contador para o número de vezes que essa função foi flagrada em execução. Isso é feito para cada instrução/função detectada no PC. Ao final, gera-se um arquivo contendo as estatísticas relacionadas a essa coleta. Uma ferramenta que utiliza esse princípio é o **gprof** (*GNU profiler*). Uma vez que há um tratamento para contagem dos dados coletados, algum *overhead* é causado na execução do sistema. E nas orientadas por evento, o monitoramento somente é ativado após a ocorrência de um evento previamente definido (comunicação sistema x usuário, mudança de

estado de um processo, chamada de função, interrupção, etc.). Os dados coletados podem ser registrados indicando: a) o tempo demandado por uma determinada função ou por um determinado processo (de acesso a memória, etc.); b) a contagem do número de constatações de uma certa propriedade feita à medida que ocorrem os eventos; c) o registro (ou *tracing*) de um (ou mais) determinado(s) tipo(s) de evento(s) em um certo intervalo de tempo, fornecendo análises mais completas; e d) o registro de solicitação de algum recurso (por exemplo, alocação dinâmica de memória) em momentos aleatórios da execução de um processo para análise do volume de recursos requisitados em cada ponto desse processo. As técnicas de monitoramento acima descritas podem ser aplicadas, conforme sua adequação a cada caso, para a observação de determinados pontos de um sistema como, por exemplo: a) em funções ou linhas de códigos de software executados nos processadores do sistema (para registro de tempo ou número de vezes ou de ciclos de *clock* de execução de cada elemento, bem como para a obtenção de *branch mispredictions* e, como já dito acima, de volume de recursos utilizados); b) nos devidos barramentos, para detecção de acesso a memórias RAM (para registro dos acessos) ou *cache* (para registro de *cache misses*); c) nos respectivos registradores de instruções de cada processador (para registro por instrução executada, podendo-se, com o uso de uma tabela de estimativa de consumo de energia de cada instrução para o processador em questão, registrar também uma estimativa de consumo total de energia pelo sistema). Geralmente, a técnica adotada apresentará o conflito de duas questões primordiais: o *overhead* inerente à técnica versus a precisão e qualidade das informações que se deseja obter (Ball *et al.*, 1998). Quanto melhor for um, maior o comprometimento do outro.

3.4.3 Métodos de implementação

Geralmente, para se obter o **Profiling** de uma aplicação efetua-se a instrumentação de seu código, ou seja, implementa-se um software e/ou um hardware para monitoramento de desempenho de um sistema coletando informações a partir de amostragens periódicas (Dean *et al.*, 1997).

(Anderson *et al.*, 1997) estabelece de forma clara e concisa que um módulo que forneça o recurso de **Profiling** normalmente é composto por dois processos básicos: (1) um para coleta de dados que extrai amostragens dos *Program Counters* durante a execução das aplicações estudadas, armazenando os dados coletados em uma base de dados própria; (2) e um conjunto de ferramentas para compilação das informações de perfil armazenadas na base de dados e apresentação dos resultados obtidos, seja em termos de tempo consumido por cada função do programa, seja por quantidade de paralisações de ciclos (*stall cycles*) de cada instrução, seja por

faltas em acessos a *caches*, seja pelo volume de recursos alocados, entre outros diversos fatores passíveis de análise.

A partir da análise dos resultados do **Profiling** realizado, o próximo passo é efetuar o particionamento do projeto em elementos de software e de hardware, conforme os requisitos de desempenho em termos de produtividade na elaboração do projeto (caso em que se dá preferência à implementação em software) ou de eficiência no processamento (caso em que a preferência é dada à implementação em hardware). Em projetos *platform-based* (seção 3.2, item b), prestigia-se o reúso de componentes pré-definidos quando da montagem de novos sistemas ou mesmo na otimização de sistemas já anteriormente elaborados ou em elaboração (fase de ajustes/melhorias). É a partir de avaliações de desempenho de cada componente que se podem definir as partes passíveis de serem melhoradas ou até mesmo substituídas por outros componentes mais eficientes, sejam esses componentes de software, sejam (na maioria dos casos) de hardware.

Após o sistema ter sido especificado, compilado e verificado, procede-se, então, à otimização do código gerado. Discutiremos, aqui, principalmente a otimização de tempo de processamento. Porém, também otimizações relacionadas à redução de consumo de energia, de área utilizada, de faltas em acessos a *caches* (*cache misses*), paralisações de ciclos (*stall cycles*), contenção de unidade funcional (*functional unit contention*) e falha na previsão de ramificações (*branch mispredictions*), entre outras, podem ser perscrutadas.

Através de diversos tipos de análise podem-se detectar pontos onde o tempo de processamento é relativamente elevado, sendo, por isso, considerados pontos críticos ou gargalos para um desempenho eficiente do sistema como um todo. Assim, analisar esses pontos e encontrar alternativas para torná-los mais rápidos normalmente conduz o sistema ao encontro do desempenho estipulado como ideal nos requisitos de tempo real para a aplicação. Desnecessário dizer que querer também otimizar as partes do sistema que são pouco utilizadas seria perda de tempo.

Ao se efetuar uma análise de **Profiling** de um sistema, subentende-se que se pretende adotar medidas visando a melhorar a configuração geral da aplicação de forma a alcançar a solução mais adequada para aquele projeto em um escopo pré-definido. Portanto, para cada ponto crítico encontrado, adaptações serão provavelmente feitas. Muitas vezes, menos em termos de redefinição de algoritmos e de recodificação, mas mais em termos de substituição de instruções, ou sequências de instruções, por aceleradores, sejam estes coprocessadores, ASIPs (*Application Specific Instruction-set Processors*), instruções customizadas, formas de acesso a e tipos de memórias (CPU/DMA, DRAMs/SRAMs, *on-chip/off-chip caches* customizados, etc.), entre outras possibilidades.

Para cada tríade arquitetura-de-hardware/algoritmo/codificação-de-software possível para uma aplicação - cujos elementos consistem em fatores que influenciam a eficiência de cada solução

-, uma análise pode ser realizada em diversos níveis de profundidade do sistema, desde a instância mais alta - a de sua arquitetura global -, passando pelo nível de suas funções e até o de precisão de ciclos de *clock* de cada instrução do processador, bem como de formas de acesso a memórias. Obviamente, efetuar uma análise de **Profiling** de uma arquitetura ou da aplicação em si não traz grandes benefícios em termos de otimização, uma vez que se perde o potencial de identificação dos pontos críticos a serem alterados. A única vantagem, nesses casos, é poder ter uma medição de eficiência para aquela arquitetura naquela aplicação para uma comparação entre possíveis soluções a serem adotadas, numa análise em um nível mais alto. Por outro lado, detectar gargalos em nível de funções permite que o programador possa conceber blocos mais eficientes que venham a substituir tais funções na solução escolhida. E detectar quais instruções são as que mais demandam ciclos de *clock* pode permitir que decisões sejam tomadas sobre as partes do programa que usam essas instruções de forma mais intensa, decisões tais como implementar instruções customizadas, chamar coprocessadores ou até mesmo alterar a arquitetura de hardware onde for necessário, adotando-se, por exemplo, SIMD (*Single Instruction, Multiple Data*) ou VLIW (*Very Long Instruction Word*), conforme o caso.

No entanto, uma análise de **Profiling** não se dá somente sobre o código objeto gerado (de forma estática), uma vez que o desempenho geral da aplicação depende da maneira como esse código é executado, ou seja, dos dados que são fornecidos na entrada do sistema e, portanto, do número de vezes que cada parte do sistema é executada em função desses dados. Portanto, uma análise de **Profiling** é sempre acompanhada de um procedimento de simulação ou, normalmente, de *debug* da aplicação.

Há diversas maneiras de se coletar as informações necessárias para a análise de **Profiling** de uma solução. Algumas são mais invasivas que outras, o que significa que o fato de se estar gerando informações para o **Profiling** afetará a execução da aplicação, influenciando mais ou menos, conforme o método utilizado, o tempo total de processamento. Dependendo da precisão desejada, o processamento da rotina para obtenção do arquivo para análise de **Profiling** pode ser mais ou menos lento. A Figura 3.5 ilustra algumas ferramentas de **Profiling**.

São coligidas, a seguir, as principais técnicas de **Profiling** utilizadas para análise de desempenho de sistemas com o intuito de servirem de base de estudo para o desenvolvimento e implementação de uma técnica que atenda os objetivos deste trabalho.

- a) **Chamada a procedimento de acúmulo de número de chamadas por função da aplicação executada:** Técnica utilizada, entre outros, pelo GNU *Profiler* (**gprof**), a qual permite obter

PROFILING TOOL COMPARISON									
	Cycles	Memory Accesses	Power	Per Function	Per Line	Callgraph	Instrumentation	Accuracy	CPUs
Gprof	+	—	—	+	+	+	+	10 ms ⁴	Many
ARM Profiler	+	+	—	+	+	+	—	$\mu\text{s}/\text{ns}$ ⁴ & estimation	ARM
ATOMIUM	+	+	—	+	—	+	+		Generic
VTune	+	+	—	+	+	+ ¹	—	⁴	Xscale
HDL Profiling ²	+	+	+	—	—	—	—	ns	Any HDL
Valgrind	+	+	—	+	+	+	—	Cycle ⁵	x86/PPC
SimpleScalar	+	+	+	—	+ ³	—	—	Cycle ⁵	Synthetic
MEMTRACE	+	+	+	+	+	—	—	Cycle	Any ISS

¹ not for embedded processors, ² very slow, ³ per assembly address, ⁴ sampling based, ⁵ simulated CPU.

Figura 3.5: (Hübert e Stabernack, 2009) Comparação entre diversas ferramentas de **Profiling**.

informações sobre chamadas a rotinas feitas durante o processamento de uma determinada aplicação. É um procedimento pouco intrusivo, ou seja, requer poucas alterações no código da aplicação para que sejam geradas as informações necessárias. Ao se habilitar o compilador (por exemplo, o gcc, especificando os parâmetros `-pg`) para que se gerem informações para **Profiling**, a execução do arquivo gerado faz com que seja criado, ao final, o arquivo `gmon.out`, o qual contém os registros às chamadas de cada rotina (função) definida e usada na aplicação. Esses registros são feitos a partir da chamada de uma função `mcount` a cada início de execução de cada função. Essa chamada foi inserida durante a compilação, quando a opção de geração do arquivo para **Profiling** foi solicitada. A função `mcount`, por sua vez, atualiza uma tabela, em memória, de contagem do número de acessos a cada função original da aplicação. Isso é feito a partir da simples consulta, pela `mcount`, da pilha de armazenagem dos endereços de funções chamadas, em que são pegos os dois endereços anteriores, que são, respectivamente, o da função chamada (nó filho) e o da função chamadora (nó pai). Ao término da execução da aplicação, os dados da tabela são gravados em um arquivo persistente chamado `gmon.out`, o qual poderá ser utilizado pelo **gprof** para geração dos relatórios de análise, com informações tanto de *flat profile* (relação das funções chamadas por ordem descendente de utilização e seus respectivos tempos de execução) quanto de *call graph* (relação em que consta o tempo de cada função chamadora e suas respectivas funções chamadas; a importância maior do relatório *call graph* é mostrar as funções que, mesmo tendo registrado pouco tempo de processamento, chamaram outras funções que apresentaram processamento mais intenso). A IDE do NIOS II, da Altera, que suporta a compilação (utilizando o compilador gcc) e execução de programas em C/C++ no processador *soft core*, admite também a utilização do utilitário **gprof** para obtenção de relatórios para análise de **Profiling** ((AlteraCorporation, 2008)).

- b) **Observação durante a execução (levantamento estatístico sobre amostras coletadas durante processamento):** Consiste em observar, em intervalos de tempo pré-definidos, o local

onde o *program counter* esteve nos instantes verificados, registrando-se um histograma para cada local detectado. O período de observação é de, geralmente, 100 vezes por segundo, mas pode ser alterado se isso se fizer necessário (Sourceware, 2010). Essa observação pode ser feita:

- A partir de uma chamada de sistema `profil()`, prevista na maioria dos sistemas operacionais *Unix-like*, que aloca uma matriz de memória no *kernel*, juntamente com a definição de um fator de escala que determina como a área de endereços do programa mapeia essa matriz. Valores típicos de escalonamento fazem com que todo local de endereçamento de 2 a 8 bytes seja mapeado em uma posição de uma matriz. A cada mudança no clock do sistema (assumindo-se que o programa de **profiling** esteja em execução), o valor do program counter é examinado e a correspondente posição da matriz de memória é incrementada. Já que isso é feito no *kernel*, que de qualquer forma tem de manipular interrupções de *clock*, pouquíssima sobrecarga adicional ao sistema ocorre.
- Por outro lado, alguns sistemas operacionais, mais notadamente o Linux 2.0 (e anteriores) não preveem a chamada de sistema `profil()`. Nesses sistemas, são feitos ajustes para que o kernel envie, periodicamente, um sinal para o processo (geralmente via `setitimer()`), o qual efetua a mesma operação de verificação do *program counter* e incremento da devida entrada na matriz. Devido a esse método requisitar que um sinal seja enviado ao local do usuário sempre que uma amostra é pega, ele apresenta uma sobrecarga consideravelmente maior que no caso do **Profiling** de *kernel*. Além do mais, por conta de um atraso de preempção necessário para o envio do sinal, este método é ainda menos preciso.

Uma rotina especial de inicialização aloca memória para o histograma e também chama a rotina `profil()` ou define um manipulador de sinal de *clock*. Essa rotina (`monstartup`) pode ser evocada de diversas formas. Nos sistemas Linux, um arquivo especial `gcr0.o` de inicialização de **Profiling**, que evoca `monstartup` antes de evocar o `main`, é usado no lugar do default `crt0.o`. A chamada desse arquivo especial de inicialização é uma das consequências do uso de `"gcc ... -pg"` no *link*. Já nos sistemas SPARC, nenhum arquivo especial de inicialização é utilizado. Ao invés disso, a rotina `mcount`, quando evocada pela primeira vez (geralmente quando a `main` é chamada), chama a `monstartup`.

- c) **Associação de blocos de código a contadores:** Nesta técnica, utilizada no NIOS II ao se definirem *Performance Counter Peripherals*, blocos (ou seções) de código são marcados (início e fim), sendo cada um associado a registradores de contagem, até sete pares no total,

um par para cada bloco. Cada par de contadores contém um registrador de 64 bits - onde é acumulado o tempo, em números de ciclos de *clock*, gasto pela seção - e um registrador de 32 bits - onde é registrado o número de vezes que a seção foi executada. O **SoPC Builder**, da Altera, permite que controles via *performance counter* sejam definidos (AlteraCorporation, 2008).

- d) **Chamadas diretas a leituras de relógio:** Neste caso, o que se faz é efetuar chamadas explícitas para leitura de um "relógio" nas seções de código fonte onde se deseja realizar as medições. Por isso, o uso dessa forma de registro de tempo é mais apropriado na avaliação de problemas de desempenho em um único programa. O procedimento para preparar um programa para medição de tempo de partes de seu código é instrumentá-lo manualmente. Como essa instrumentação é, em geral, menos abrangente, ela se torna também menos intrusiva. Entretanto, a desvantagem deste método em relação ao utilizado pelo *Performance Counter* é o fato de se ter de efetuar duas chamadas de função para efetuar o registro do tempo despendido, uma para ler e marcar o tempo de início da seção e outra para fazê-lo no final, demandando mais ciclos de *clock* (AlteraCorporation, 2008). No NIOS II, esta técnica é implementada em um processo denominado *High Resolution Timer* (Temporizador de Alta Resolução).
- e) **Rastreamento de módulos componentes de uma arquitetura de processamento via *callback functions* implementadas em um *Instruction Set Simulator* (ISS):** Nesta modalidade de **Profiling**, um simulador de conjunto de instruções (ISS) deve estar disponível para o processador que se pretende utilizar e analisar. Assim sendo, no ISS são inseridas chamadas a funções *callback* conforme a estratégia de medição de tempo definida pelo programador (por exemplo, para cada chamada de função da aplicação e/ou para eventos de *cache misses* e/ou ocorrências de acesso ao barramentos de memória RAM e/ou memória interna/cache, etc.). Essas funções correspondem a métodos especificados em um módulo de tratamento dos resultados medidos. Esses resultados vão sendo acumulados em uma tabela em memória durante a execução do programa para, ao final, serem gravados em uma base de dados definitiva, a qual será disponibilizada, logo em seguida, para a geração dos relatórios de análise de **Profiling** desejados. Um exemplo desta modalidade de obtenção de **Profiling** é a ferramenta MEMTRACE descrita em (Hübert e Stabernack, 2009) e cujo princípio de comunicação entre o ISS (no caso, o ARMulator, simulador de conjunto de instruções do processador ARM) e a ferramenta é ilustrado na Figura 3.6.

Figura 3.6: Interação Memtrace x ARMulator. (Hübert e Stabernack, 2009)

O trabalho de dissertação de mestrado a ser desenvolvido visa a explorar os conceitos relacionados ao processo de **Profiling** de sistemas de forma a permitir obter o perfil de uso de memória, mais especificamente, numa primeira instância, do uso de memória *heap* de programas codificados na linguagem C que rodem no processador *soft core* Nios II, da Altera.

Baseado nas técnicas analisadas até aqui, optou-se por desenvolver o trabalho nas seguintes etapas:

- 1) Partindo de um arquivo semelhante ao `gmon.out` criado pelo software executado em um processador *soft core* NIOS II, geram-se informações de **Profiling** em um formato apropriado para ser lido posteriormente por um programa analisador. Esse arquivo, denominado `HipReport.out`, é o resultado do registro de todas as chamadas a qualquer rotina de alocação/desalocação dinâmica de memória (`malloc`, `calloc`, `realloc` e `free`) e as respectivas quantidades de *bytes* alocadas/desalocadas de forma cumulativa.
- 2) Processa-se o arquivo `HipReport.out` gerado, apresentando as medições registradas, tanto em termos de valores absolutos (número de bytes alocados) quanto em termos de valores relativos (percentual de participação de cada função medida sobre o total).

Embora o trabalho ora desenvolvido volte-se, inicialmente, ao processador *soft core* NiosII, da Altera, utilizando-se sua IDE EDS - Embedded Design Suite -, posteriormente, quando a **RoboArch** estiver concluída, os conceitos e recursos aqui desenvolvidos e descritos poderão ser migrados para essa ferramenta.

Desenvolvimento

4.1 *Considerações iniciais*

Nos capítulos anteriores, foram ressaltados por diversas vezes os elementos críticos na elaboração de sistemas embarcados cuja otimização é perscrutada com o objetivo de viabilizar técnica, funcional e economicamente os produtos dependentes de sistemas computacionais. Há já diversas técnicas para apurar a eficiência de um sistema, porém, nem todas aplicam-se a sistemas embarcados, menos à computação reconfigurável e poucas à otimização do uso da memória.

As vantagens de otimizar o uso de memória referem-se tanto à consequente redução do tamanho dos dispositivos de armazenamento volátil de programas e dados (e dos custos de produção), com o respectivo encolhimento da área global do dispositivo fabricado, quanto à diminuição da probabilidade de travamento do sistema devido a estouro de memória. O tamanho da memória utilizada pode variar, durante a execução de um programa, nos casos de chamada de funções (recursivas ou não) com declaração de variáveis locais e automáticas, com o inerente incremento da memória *stack*, e de alocação dinâmica de memória, com o aumento da memória *heap*.

Portanto, técnicas que permitam a obtenção do perfil (***Profiling***) de uso de memória na elaboração de projetos (ou co-projetos) de sistemas embarcados propiciam aos projetistas identificar

os pontos críticos em seus módulos e reavaliar suas soluções, otimizando os recursos que se utilizam da alocação dinâmica e extensível de espaços de memória.

Uma ferramenta muito utilizada no desenvolvimento de sistemas embarcados para dispositivos reconfiguráveis é a Quartus II, da Altera, que permite projetar circuitos e até mesmo processadores para suas FPGAs. Uma de suas funcionalidades, a QSys (sucessora da SOPC Builder), oferece o recurso de montagem e configuração de um processador *soft core*, o Nios II. Constrói-se, com ela, um dispositivo com CPU, memória *on-chip*, barramentos de comunicação e outros recursos inerentes que propiciarão a execução de um programa que exercerá as funções desejadas em um determinado sistema embarcado. Neste trabalho foi utilizado o Quartus II versão 11.1 e seus componentes QSys e Nios II de mesma versão em um ambiente operacional Windows 7 de 32 bits.

Geralmente, os programas que são executados nos processadores *soft core* são desenvolvidos na linguagem C devido à sua maior adequação a esse tipo de aplicação. A compilação dos códigos fonte em C é feita, no caso do Nios II, utilizando-se o GCC (GNU Compiler Collection) (GCC, 2012) instalado para o *target* "nios2-elf-". O GCC apresenta o recurso de obtenção do **Profiling** de desempenho de sistemas em termos de tempo de execução a partir da compilação do código fonte utilizando-se o parâmetro `-pg`. Isso faz com que o arquivo executável criado esteja preparado para, quando de sua execução, gerar o arquivo `gmon.out`, que contém as informações para obtenção do perfil de funcionamento do sistema. O programa GProf (GPr, 2012) completa a tarefa lendo o arquivo `gmon.out` e gerando os relatórios para análise do perfil levantado.

Entretanto, os programas processados pelo conjunto GCC/GProf não fornecem informações a respeito do uso de memória. Mesmo o GDB (que permite efetuar o *debug* de programas) não apresenta as informações de uso de memória de forma apropriada para uma análise estatística considerando o funcionamento completo do programa. Baseado nesse fato é que se optou por realizar este trabalho de implementação de um recurso que, de forma paralela e complementar ao GProf do GCC, forneça informações em formato apropriado para que se façam análises de uso de memória (neste caso, alocada dinamicamente) em sistemas executados em processadores embarcados. O ambiente usado para testes é o processador *soft core* Nios II.

Se, assim como há uma análise de desempenho das funções de um programa através do GProf, houver também como analisar o uso de memória por cada função desse mesmo programa, o desenvolvedor poderá facilmente detectar as funções críticas sujeitas a problemas tais como desperdício de memória usada e/ou travamentos de sistema devido a processos que alocam memória dinamicamente de forma frequente e intensa. Funções que sobrecarregam sobremaneira a *heap* devido à intensa alocação dinâmica de memória ou que estão constantemente adicionando dados à pilha da *stack* por causa, por exemplo, de chamadas recursivas são sérias

candidatas a sofrerem um estudo mais profundo visando à sua otimização ou, até mesmo, à determinação do acréscimo de memória para que as necessidades funcionais do sistema sejam atendidas sob o menor risco possível.

Entretanto, diferentemente das estatísticas oferecidas pelo GProf com relação à análise de desempenho, em que cada amostra obtida implica no incremento de um contador para a função flagrada, o interesse na análise do uso de memória encontra-se mormente no volume alocado constatado em cada amostragem. Dessa forma, para diferentes amostragens de uma mesma função interessa saber qual a maior quantidade de memória utilizada pela função entre todas as amostras coletadas. Isso significa que, ao invés de incremento por amostragem, adotar-se-á o princípio do registro do maior valor, em número de *bytes*, de uso de memória constatado em cada função.

Além do mais, também é necessário dizer que, ao analisar as estatísticas referentes ao uso de memória por cada função e adotar medidas para sua otimização, o desenvolvedor deverá levar em consideração, entre outros fatores, a questão do desempenho do sistema. Em outras palavras, deve-se tomar cuidado para que as soluções adotadas para a melhoria do uso de memória não comprometam o desempenho do sistema em termos de seus requisitos de tempo real. A análise de uso de memória deve ser, portanto, realizada concomitante e/ou alternadamente à análise de desempenho.

Vale ressaltar que os resultados obtidos para o processador Nios II podem ser estendidos para outros produtos semelhantes - como o também processador *soft core* Micro Blaze ((Xilinx, 2010b)), da Xilinx, montado e configurado na ferramenta EDK (*Embedded Development Kit*) -, desde que efetuados os devidos estudos de viabilidade e ajustes necessários para cada caso.

Embora o escopo do trabalho atual só se limite a analisar o uso de memória de forma restrita à alocação dinâmica da memória *heap*, futuramente poder-se-á estudar formas de se obter com maior precisão informações sobre o uso global da memória, bem como sobre outros usos específicos, como o da memória *stack* e de memórias *cache* (quanto à detecção de *cache-misses*. Também a detecção de *memory leaks* e outros problemas associados ao uso de memória em sistemas embarcados poderão derivar do presente trabalho.

4.2 Implementação

Para implementar os procedimentos para obtenção do perfil de uso de memória por uma dada aplicação, definiram-se os seguintes elementos a serem utilizados:

a) **Programa pré-processador *source-to-source* com instrumentação de código** `nios2-elf-Hippie.c`):

Este programa tem a função de ler um programa fonte escrito na linguagem C e instrumentá-lo de forma a fazer com que, durante a execução do programa objeto correspondente, seja gerado um arquivo de dados contendo as informações sobre a quantidade de memória utilizada por cada função do programa processado. Depreende-se, portanto, que esse pré-processador terá como entrada um programa fonte na linguagem C e como saída também um programa fonte na linguagem C, porém, este alterado conforme os critérios de instrumentação adotados.

A referida instrumentação consiste:

- da geração de um arquivo intermediário instrumentado a partir da intercalação entre o arquivo fonte alvo (**target**) e o arquivo `InstCode.c`, que contém as definições necessárias para a chamada da função de tratamento das funções `malloc`, `calloc`, `realloc` e `free`, qual seja, `dynAlloc_HProf`; essa função de tratamento efetua a apuração e gravação da quantidade de memória usada nas suas chamadas; a gravação é feita em um arquivo cujo nome é `HipReport.out`;
- da substituição das funções `malloc`, `calloc`, `realloc` e `free` encontradas no programa fonte C pré-processado pela função `dynAlloc_HProf` de tratamento definida no arquivo `InstCode.c`.

b) **Programa analisador do perfil de uso de memória** `nios2-elf-HipReport.c`: Já este programa tem a função de ler o conteúdo gravado no arquivo `HipReport.out` gerado durante a execução do programa instrumentado, processar as informações de forma a organizá-las adequadamente para uma apresentação analítica e apresentar o relatório com as informações processadas.

O quadro da **Figura 4.1** e também a **Figura 4.2** apresentam uma visão geral do fluxo de processos inerentes.

Ao analisarmos os arquivos fonte `nios2-elf-Hippie.c` (Apêndice C) e `InstCode.c` (Apêndice D), encontraremos os códigos referentes, respectivamente, ao pré-processamento do

Algoritmo geral do processo para obtenção do perfil de uso de memória *heap*:

- Usuário executa o pré-processador *source-to-source* `nios2-elf-Hippie.exe`;
 - Lê programa fonte C;
 - Instrumenta programa fonte C;
 - * Acrescenta arquivo `InstCode.c`;
 - * Substitui chamadas a `malloc`, `calloc`, `realloc` e `free` por chamadas a `dynAlloc_HProf`;
 - * Obs.: Parâmetros das novas funções são os mesmos das funções originais acrescidos do nome da função onde são chamadas e um indicador do tipo de função que está sendo substituída, mais um eventual complemento, quando necessário;
 - Grava programa fonte C instrumentado (`HipInst.c`);
 - Chama compilador para compilar arquivo instrumentado;
- Usuário executa programa objeto (`HipInst.exe`) gerado;
 - Programa executado gera arquivo `HipReport.out`;
- Usuário executa o programa analisador `nios2-elf-HipReport.exe` que lê arquivo `HipReport.out` e gera relatório de uso da memória *heap*.

Figura 4.1: Algoritmo geral do processo para obtenção do perfil de uso de memória *heap*.

arquivo alvo e à instrumentação com o tratamento das funções `malloc`, `calloc`, `realloc` e `free`. A **Figura 4.3** apresenta o algoritmo referente ao código de instrumentação constante no arquivo `InstCode.c`.

Fluxo de processamento da ferramenta Hippie **Figura 4.2:** Fluxo de processamento da ferramenta Hippie.

Especificação das funções constantes no arquivo `InstCode.c`:

- Arquivo `InstCode.c` - função `dynAlloc_HProf`:
 - Abre arquivo `HipReport.out`;
 - Chama `malloc`, `calloc`, `realloc` e `free` normalmente;
 - Apura tamanho da memória alocada;
 - Grava em `HipReport.out` o nome da função chamadora passada nos parâmetros e a quantidade de memória alocada;
 - Fecha o arquivo `HipReport.out`;
 - Retorna, exceto para a função `free`, o ponteiro da área de memória alocada à variável da chamada original da função substituída pela `dynAlloc_HProf(malloc, calloc, realloc)`.

Figura 4.3: Arquivo `InstCode.c`.

O cômputo da área alocada em cada chamada de uma rotina de alocação (`malloc`, `calloc`, `realloc`) considera não só a quantidade requisitada como parâmetro da chamada, mas também o *overhead* advindo da forma como a memória *heap* é organizada para melhor atender às solicitações dinâmicas de espaço. O espaço total alocado, que consiste no espaço requisitado acrescido do *overhead*, é calculado da seguinte forma:

$$EspacoTotalAlocado = (int)((floor(\frac{EspacoRequisitado}{8.0} + 0.49) + 1.0) * 8.0);$$

$$Se(EspacoTotalAlocado < 16)entao(EspacoTotalAlocado = 16);$$

Para entendermos essas questões, partamos do princípio de que o espaço de memória requisitado - e o *overhead* inerente - é sempre alocado em endereços contíguos. Visando a otimizar a alocação de memória, a qual pode ser requisitada aleatoriamente em tamanhos variados, a área livre para alocação é organizada pelo compilador em blocos denominados *chunks* (os quais podem estar vinculados a uma estrutura denominada *bin*, que agrupa os *chunks* de um mesmo tamanho), que compõem uma lista encadeada de endereços livres (Aho *et al.*, 2007) e (Lea, 2012). Utilizam-se técnicas diversas (*best-fit*, *next-fit*, *worst-fit*, etc.), as quais não serão aqui detalhadas, para encontrar o *chunk* mais adequado a uma requisição efetuada. Ao encontrar a área certa para alocação, caso o tamanho solicitado não seja exatamente igual ao tamanho da área selecionada, o espaço restante (que, obviamente, será menor que o tamanho original do *chunk* ao qual pertencia) é dissociado da área reservada e devolvido à lista de endereços livres. É nesse momento que ocorre o que é chamado de fragmentação.

Os espaços que foram alocados apresentam uma área contendo *bytes* de controle, onde ficam informações tais como o tamanho do *chunk* alocado e um *flag* indicador de uso ou não uso do espaço em questão. Esses *bytes* reservados para efetuar esse controle representam um *overhead*, cujo valor absoluto pode ser obtido a partir da fórmula anteriormente apresentada.

Por outro lado, conforme a técnica de busca adotada pelo compilador, formas variadas de fragmentação podem ocorrer, implicando em distintas configurações (tamanhos) de elementos das áreas livres. E os espaços que sobram e retornam à lista de endereços livres podem ser tão reduzidos para a aplicação em questão que não se consiga, em futuras requisições, que sejam aproveitados.

As figuras 4.4 e 4.5 ilustram uma situação em que isso ocorre. A título de exemplo, foi criada uma rotina que aloca, inicialmente, 32 MB de memória RAM, num total de 64 MB. Esse espaço é liberado e, logo em seguida, são alocados 24 MB, os quais podem ocupar tanto parte da área liberada quanto da área fragmentada. A opção depende de como essa questão foi implementada no compilador. Posteriormente, também esses 24 MB são liberados e, após isso, requisita-se a alocação de 36 MB. Em princípio, como os 64 MB de memória estão liberados, há espaço suficiente para essa última solicitação. Entretanto, devido à fragmentação ocasionada, é apontado um erro de alocação.

Não se implementou neste trabalho nenhuma técnica para detecção desse problema. Esta é uma questão que deve ser futuramente melhor estudada e analisada. Uma possível solução seria fornecer um índice de fragmentação que indique um estado crítico de uso e desperdício de memória *heap* em relação a estatísticas efetuadas sobre o conjunto de espaços requisitados pela aplicação. Os resultados obtidos neste tipo de **Profiling** podem levar o projetista a alterar as técnicas de obtenção dos *chunks* mais apropriados à alocação de memória feita, caso ele tenha acesso a reconfigurar a técnica de busca utilizada pelo compilador, ou a lançar mão de processos de reorganização da memória dinâmica que possam corrigir o problema da fragmentação em tempo de execução, embora estes demandem tempo de máquina para serem executados e, portanto, afetem o desempenho dos sistemas em geral, quando adotados. Mas isso tudo consiste em um outro tipo de análise de perfil de uso de memória de um sistema cujo estudo, como dito, pode ser aprofundado em trabalhos futuros.

Figura 4.4: Mensagem apresentada quando há erro de alocação

Figura 4.5: Relatório após apresentada mensagem de erro de alocação

E, por fim, o quadro da **Figura 4.6** apresenta uma especificação genérica relativa ao programa `nios2-elf-HipReport.c` (cujo código é mostrado no Apêndice E), que imprime o relatório analítico sobre o uso de memória *heap* por função do programa fonte C verificado.

Para poder executar os processos acima, os passos abaixo descritos foram seguidos.

1. Copiar `nios2-elf-HipReport.exe` e `nios2-elf-HipReport.exe` para `C:\Alter\11.1\nios2eds\bin\gnu\H-i686-min` (Figura 4.7).
2. Criar uma pasta (`<benchmark>`) para onde deverão ser copiados os arquivos de *benchmark* e o arquivo `InstCode.c` que contém o código de instrumentação (Figura 4.8).

Programa analisador do uso de memória *heap* - `nios2-elf-HipReport.c`:

- Abre arquivo `HipReport.out`;
- Lê os registros e os insere em uma lista duplamente encadeada, rastreando nessa lista a quantidade de memória alocada anteriormente quando houver uma liberação de memória com o uso de uma função `realloc` ou `free`.
- A cada registro lido, atualiza uma variável de acúmulo global que deverá conter, ao final do processamento, a maior quantidade de memória alocada em todo o processamento, bem como cria uma lista duplamente encadeada de acumulados contendo o nome da função onde foi chamada a alocação dinâmica, a maior quantidade de *bytes* alocados por ela durante todos os seus processamentos e a maior quantidade global acumulada quando de seu uso.
- Ordena, por Quick Sort, os registros da lista de acumulados por ordem de:
 - maior quantidade de *bytes* alocados pela função
 - maior quantidade global de *bytes* alocados quando da execução da função
 - nome da função chamadora
- Lê a lista duplamente encadeada de acumulados do fim ao início, apresentando os resultados em ordem da função que mais alocou memória para a que menos alocou.
- Fecha o arquivo `HipReport.out`;

Figura 4.6: Arquivo `nios2-elf-HipReport.c`.

3. Alterar os códigos fonte dos *benchmarks* e do `InstCode.c` para que os arquivos de entrada e de saída tenham seus nomes endereçados respectivamente para `/mnt/rozipfs` e `/mnt/host`.
4. No Command Shell do Nios2 (Figura 4.8), na pasta `<benchmark>` criada no passo 2, executar
...\$ `nios2-elf-Hippie.exe <nome_arqfonte_benchmark>`
5. No EDS (IDE do Nios2), criar um projeto para executar o arquivo a ser analisado (Figura 4.9).
6. Se houver arquivos de entrada usados no *benchmark*, criar arquivo `.ZIP` no modo Armazenar contendo os devidos arquivos e copiá-lo para a pasta principal do projeto criado no passo 5.
7. Copiar o arquivo `HipInstr.c` gerado no passo 4 para a pasta principal do projeto criado (Figura 4.10).

8. No EDS, em propriedades da pasta 'syslib' do projeto criado no passo 5, na opção 'System Library', clicar no botão 'Software Components' para configurar os *file systems* de entrada e de saída (Figura 4.11).
9. No EDS, para 'Altera Zip Read-Only File System' (arquivos de entrada), especificar (Figura 4.12):
 - Marcar *check-box* 'Add this software component'
 - Flash memory device: `cfi_flash`
 - Offset: `0x1000`
 - Mount-point: `/mnt/rozipfs`
 - Zip file: Indicar arquivo .ZIP gerado no passo 6, que deve estar na pasta principal do projeto criado no passo 5.
10. No EDS, para 'Altera Host Based File System' (arquivos de saída), especificar (Figura 4.13):
 - Marcar *check-box* 'Add this software component'
 - Mount-point: `/mnt/host`
11. No EDS, efetuar o 'build' da pasta 'syslib' do projeto criado no passo 5 (Figura 4.14).
12. No EDS, efetuar o 'build' da pasta principal do projeto criado no passo 5. Neste momento, os arquivos fonte do projeto estarão sendo compilados pelo programa `nios2-elf-gcc` (Figura 4.15).
13. No menu do EDS, opção Tools/Flash Programmer, criar uma nova configuração relativa ao projeto em questão para programação da memória *flash*. Nessa configuração, especificar o arquivo .ZIP que contém os arquivos de entrada usados pelos programas fonte do projeto e o *offset* (por exemplo, `0x1000`). Feito isso, confirmar a programação da *flash* (Figura 4.16).
14. No EDS, executar o projeto principal no modo *debug* (Debug As Nios II Hardware) (Figura 4.17).
15. A perspectiva do EDS será automaticamente alterada. O programa `HipInstr.elf` criado pela compilação do projeto no passo 12 será executado (Figura 4.18), gerando seus resultados e o arquivo `HipReport.out` (Figura 4.19) com o registro do uso de memória *heap* para análise.

- [illegible]

Figura 4.7: Copiar nios2-elf-Hippie.exe e nios2-elf-HipReport para .../bin

Figura 4.8: Criar uma pasta para os <benchmark>

Figura 4.9: Criar um projeto para executar o <benchmark>

Figura 4.10: Copiar HiplInstr.c para pasta principal do projeto

Figura 4.11: Configurar os *file systems* de entrada e de saída

Figura 4.12: Configurar o *file system* de entrada

Figura 4.13: Configurar o *file system* de saída

Figura 4.14: Efetuar o 'build' da pasta 'syslib' do projeto

Figura 4.15: Efetuar o 'build' da pasta principal do projeto

Figura 4.16: Programação da memória *flash*

Figura 4.17: Executar o projeto principal no modo *debug*

Figura 4.18: Execução do programa HipIntr.elf

Figura 4.19: Gerado o arquivo HipReport.out

Figura 4.20: Execução do gerador do relatório de análise

4.3 Resultados

Concluídas as etapas anteriores, foram submetidos a ela diversos programas escritos em linguagem C que utilizam-se de ou foram adaptados para utilizarem alocação dinâmica de

Figura 4.21: Arquivo de relatório de análise

memória. Ressalta-se que em muitos dos casos em que houve a adaptação, os programas alterados adquiraram a vantagem da flexibilidade de se definir em tempo de execução o tamanho dos vetores e matrizes utilizados.

Nos testes, foram submetidos à instrumentação descrita na seção 4.2 os programas alvo abaixo relacionados. Após cada um são apresentados os resultados obtidos tanto na execução em um PC quanto no processador *soft core* Nios II da Altera.

- Max Element (figura 4.22)
- Factorial (figura 4.23)
- Sequential Search (figura 4.24)
- Binary Search (figura 4.25)
- Bubble Sort (figura 4.26)
- Insertion Sort (figura 4.27)
- Selection Sort (figura 4.28)
- Merge Sort (figura 4.29)
- Quick Sort (figura 4.30)
- Determinant (figura 4.31)
- Matrix Multiplication (figura 4.32)
- Matrix Transposition (figura 4.33)
- Fibonacci (figura 4.34)
- Gauss Jordan (figura 4.35)
- Jacobi sequence (figura 4.36)

- Sobel (figura 4.37)
- BMP image processing (figura 4.38)

Os resultados podem ser contrastados entre si conforme o maior ou menor uso de alocação dinâmica de memória em diferentes situações. Dessa forma, é possível conhecer melhor os problemas de uso de memória que podem ocorrer nos mais variados casos, permitindo um aprofundamento na compreensão do funcionamento dos processos relacionados ao acesso a memórias e, assim, concorrendo para a especificação de um conjunto de propostas de solução para os diversos casos inerentes, principalmente no que se refere a arquiteturas de Sistemas Embarcados baseados em Computação Reconfigurável.

O presente trabalho pretende consistir em um *ponta-pé* inicial para a construção de uma ferramenta de **Profiling** de memória voltada à Computação Reconfigurável. O uso apropriado de memórias em SoCs pode contribuir muito para a otimização desses sistemas e para a redução de seus custos e de seus consumos energéticos. Espera-se que a contribuição aqui oferecida tenha, num futuro próximo, seus frutos para o desenvolvimento de novas ferramentas de elaboração otimizada de sistemas eletrônicos.

Figura 4.22: Resultados Elemento Máximo

Figura 4.23: Resultados Fatorial

```

03 - Sequential Search

Informe o tamanho do vetor randomico: 11
Informe o numero aproximado (0 a 999) que deseja encontrar no vetor randomico:
432

Vetor:
a[0] = 938
a[1] = 403
a[2] = 945
a[3] = 860
a[4] = 836
a[5] = 314
a[6] = 134
a[7] = 381
a[8] = 451
a[9] = 372
a[10] = 453

Valor 451 encontrado em 8:

EXECUTAR BEM-SUCEDIDO (tempo total: 3s)

-----
|Ord| Function Name          | Max Function |*(Fnc/Glb)| Global Accum|
-----
| 1| main                    |              | 48 | 100.000 |         48 |
-----

Remarks: Results in bytes.
          (Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR BEM-SUCEDIDO (tempo total: 73ms)

NIOS2

-----
|Ord| Function Name          | Max Function |*(Fnc/Glb)| Global Accum|
-----
| 1| main                    |              | 48 | 100.000 |         48 |
-----

Remarks: Results in bytes.
          (Fnc/Glb) = bytes Function / bytes greatest Global Accum.

```

Figura 4.24: Resultados Sequential Search

```
04 - Binary Search
Informe o tamanho do vetor randomico: 11
Informe o numero aproximado (0 a 999) que deseja encontrar no vetor randomico:
249

Vetor nao ordenado:
a[0] = 135
a[1] = 630
a[2] = 727
a[3] = 725
a[4] = 551
a[5] = 870
a[6] = 859
a[7] = 11
a[8] = 634
a[9] = 967
a[10] = 566

Vetor ordenado:
a[0] = 11
a[1] = 135
a[2] = 551
a[3] = 566
a[4] = 630
a[5] = 634
a[6] = 725
a[7] = 727
a[8] = 859
a[9] = 870
a[10] = 967
Number 249 would be in between 1 and 2.

EXECUTAR SEM-SUCESSO (tempo total: 3s)

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1| main | | 48 | 100.000 | 48 |
-----
Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR SEM-SUCESSO (tempo total: 86ms)

NIOSE

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1| main | | 44 | 100.000 | 44 |
-----
Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.
```

Figura 4.25: Resultados Binary Search

```
05 - Bubble Sort
Informe o tamanho do vetor randomico: 11

Vetor nao ordenado:
a[0] = 901
a[1] = 415
a[2] = 667
a[3] = 127
a[4] = 520
a[5] = 825
a[6] = 692
a[7] = 696
a[8] = 678
a[9] = 489
a[10] = 4

Vetor ordenado:
a[0] = 4
a[1] = 127
a[2] = 489
a[3] = 520
a[4] = 625
a[5] = 696
a[6] = 667
a[7] = 678
a[8] = 692
a[9] = 825
a[10] = 901

EXECUTAR SEM-SUCESSO (tempo total: 3s)

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1| main | | 48 | 100.000 | 48 |
-----
Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR SEM-SUCESSO (tempo total: 41ms)

NIOSE

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1| main | | 48 | 100.000 | 48 |
-----
Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.
```

Figura 4.26: Resultados Bubble Sort

```
06 - Insertion Sort
Informe o tamanho do vetor randomico: 12

Vetor nao ordenado:
a[0] = 171
a[1] = 824
a[2] = 649
a[3] = 429
a[4] = 165
a[5] = 420
a[6] = 210
a[7] = 929
a[8] = 74
a[9] = 618
a[10] = 156
a[11] = 232

Vetor ordenado:
a[0] = 74
a[1] = 156
a[2] = 165
a[3] = 171
a[4] = 210
a[5] = 232
a[6] = 420
a[7] = 429
a[8] = 618
a[9] = 649
a[10] = 824
a[11] = 929

EXECUTAR SEM-SUCESSO (tempo total: 2s)

-----
|Ord| Function Name | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| main | 56 | 100.000 | 56 |
-----
Remarks: Results in Bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR SEM-SUCESSO (tempo total: 107ms)

NIOSE

-----
|Ord| Function Name | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| main | 56 | 100.000 | 56 |
-----
Remarks: Results in Bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.
```

Figura 4.27: Resultados Insertion Sort

```
07 - Selection Sort
Informe o tamanho do vetor randomico: 10

Vetor nao ordenado:
a[0] = 722
a[1] = 649
a[2] = 492
a[3] = 465
a[4] = 411
a[5] = 731
a[6] = 27
a[7] = 241
a[8] = 386
a[9] = 823

Vetor ordenado:
a[0] = 27
a[1] = 241
a[2] = 386
a[3] = 411
a[4] = 465
a[5] = 492
a[6] = 649
a[7] = 722
a[8] = 731
a[9] = 823

EXECUTAR SEM-SUCESSO (tempo total: 1s)

-----
|Ord| Function Name | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| main | 45 | 100.000 | 45 |
-----
Remarks: Results in Bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR SEM-SUCESSO (tempo total: 32ms)

NIOSE

-----
|Ord| Function Name | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| main | 45 | 100.000 | 45 |
-----
Remarks: Results in Bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.
```

Figura 4.28: Resultados Selection Sort

Figura 4.29: Resultados Merge Sort

Figura 4.30: Resultados Quick Sort

```

10 - Determinante
** Calculando o determinante de matrizes reais. **
Digite a ordem desejada da matriz: 4
Digite os valores da matriz (da direita para a esquerda, de cima para baixo):
A matriz eh:
[ 1.00 0.00 -1.00 3.00 ]
[ 2.00 3.00 4.00 2.00 ]
[ 0.00 2.00 5.00 1.00 ]
[ 4.00 1.00 0.00 0.00 ]
O Determinante eh: -54.000000
EXECUTAR SEM-SUCESSO (tempo total: 5s)
-----
|Ord| Function Name | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| allocarMQ | 264 | 100.000 | 264 |
-----
Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR SEM-SUCESSO (tempo total: 55ms)

NIOS2
-----
|Ord| Function Name | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| allocarMQ | 264 | 100.000 | 264 |
-----
Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

```

Figura 4.31: Resultados Determinante

```

11 - Matrix Multiplication
** Calculando o produto de matrizes reais. **
Digite a ordem desejada da matriz: 4
Digite os valores das matrizes (da direita para a esquerda, de cima para baixo):
A matriz1 eh:
[ 1.00 0.00 -1.00 3.00 ]
[ 2.00 3.00 4.00 2.00 ]
[ 0.00 2.00 5.00 1.00 ]
[ 4.00 1.00 0.00 0.00 ]
A matriz2 eh:
[ 2.00 0.00 -1.00 3.00 ]
[ 0.00 1.00 4.00 1.00 ]
[ 0.00 -2.00 5.00 2.00 ]
[ 4.00 1.00 3.00 2.00 ]
A matriz3 = Matriz1 * Matriz2 eh:
[ 14.00 5.00 3.00 7.00 ]
[ 12.00 -3.00 36.00 21.00 ]
[ 4.00 -7.00 36.00 14.00 ]
[ 8.00 1.00 0.00 13.00 ]
EXECUTAR SEM-SUCESSO (tempo total: 22s)
-----
|Ord| Function Name | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| allocarMQ | 360 | 100.000 | 360 |
-----
Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR SEM-SUCESSO (tempo total: 72ms)

NIOS2
-----
|Ord| Function Name | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| allocarMQ | 360 | 100.000 | 360 |
-----
Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

```

Figura 4.32: Resultados Multiplicação de Matrizes

```
12 - Matrix Transposition

** Transpondo uma matriz real. **

Digite a ordem desejada da matriz: 4
Digite os valores das matrizes (da direita para a esquerda, de cima para baixo):

A matriz eh:
[ 1.00 0.00 -1.00 3.00 ]
[ 2.00 3.00 4.00 2.00 ]
[ 0.00 2.00 5.00 1.00 ]
[ 4.00 1.00 0.00 0.00 ]

A matriz transposta = eh:
[ 1.00 2.00 0.00 4.00 ]
[ 0.00 3.00 2.00 1.00 ]
[ -1.00 4.00 5.00 0.00 ]
[ 3.00 2.00 1.00 0.00 ]

EXECUTAR BEM-SUCEDIDO (tempo total: 13s)

-----
|Ord| Function Name          | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| allocarrMQ              |             | 240 | 100.000 | 240 |
-----

Remarks: Results in bytes.
          (Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR BEM-SUCEDIDO (tempo total: 34ms)

NIOS2

-----
|Ord| Function Name          | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| allocarrMQ              |             | 240 | 100.000 | 240 |
-----

Remarks: Results in bytes.
          (Fnc/Glb) = bytes Function / bytes greatest Global Accum.
```

Figura 4.33: Resultados Transposição de Matriz

```
13 - Fibonacci

Informe o numero limite para os quais deseja calcular Fibonacci: 20
Fibonacci em 0 eh: 0
Fibonacci em 1 eh: 1
Fibonacci em 2 eh: 1
Fibonacci em 3 eh: 2
Fibonacci em 4 eh: 3
Fibonacci em 5 eh: 5
Fibonacci em 6 eh: 8
Fibonacci em 7 eh: 13
Fibonacci em 8 eh: 21
Fibonacci em 9 eh: 34
Fibonacci em 10 eh: 55
Fibonacci em 11 eh: 89
Fibonacci em 12 eh: 144
Fibonacci em 13 eh: 233
Fibonacci em 14 eh: 377
Fibonacci em 15 eh: 610
Fibonacci em 16 eh: 987
Fibonacci em 17 eh: 1597
Fibonacci em 18 eh: 2584
Fibonacci em 19 eh: 4181
Fibonacci em 20 eh: 6765

EXECUTAR BEM-SUCEDIDO (tempo total: 5s)

-----
|Ord| Function Name          | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| malloc                  |             | 88 | 100.000 | 88 |
-----

Remarks: Results in bytes.
          (Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR BEM-SUCEDIDO (tempo total: 83ms)

NIOS2

-----
|Ord| Function Name          | Max Function |(Fnc/Glb)| Global Accum|
-----
| 1| malloc                  |             | 88 | 100.000 | 88 |
-----

Remarks: Results in bytes.
          (Fnc/Glb) = bytes Function / bytes greatest Global Accum.
```

Figura 4.34: Resultados Fibonacci

```

14 - Gauss-Jordan

** Calculando a matriz inversa por Gauss-Jordan. **

Digite a ordem desejada da matriz: 4
Digite os valores das matrizes (da direita para a esquerda, de cima para baixo):

A matriz eh:

[ 1.00 0.00 -1.00 3.00 ]
[ 2.00 3.00 4.00 2.00 ]
[ 0.00 2.00 5.00 1.00 ]
[ 4.00 1.00 0.00 0.00 ]

A matriz inversa eh:

[ 0.11 -0.30 0.26 0.37 ]
[ -0.44 1.19 -1.04 -0.46 ]
[ 0.11 -0.46 0.59 0.20 ]
[ 0.32 -0.06 0.11 -0.06 ]

A matriz identidade eh:

[ 1.00 -0.00 -0.00 0.00 ]
[ -0.00 1.00 0.00 -0.00 ]
[ 0.00 0.00 1.00 -0.00 ]
[ 0.00 0.00 0.00 1.00 ]

EXECUTAR BEM-SUCEDIDO (tempo total: 2s)

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1 | alocarMQ | | 560 | 100.000 | | 560 |
-----

Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR BEM-SUCEDIDO (tempo total: 157ms)

WQS2

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1 | alocarMQ | | 560 | 100.000 | | 560 |
-----

Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

```

Figura 4.35: Resultados Gauss Jordan

```

15 - Jacobi Sequence

INPUT:
[ 57.0000, 0.0000, 1.0000, 2.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000 ] - [-4.0000]
[ 0.0000, 58.0000, 1.0000, 2.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000 ] - [ 1.0000]
[ 1.0000, 1.0000, 61.0000, 2.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000 ] - [ 6.0000]
[ 2.0000, 2.0000, 2.0000, 63.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000 ] - [-11.0000]
[ 3.0000, 3.0000, 3.0000, 3.0000, 65.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000 ] - [-16.0000]
[ 4.0000, 4.0000, 4.0000, 4.0000, 4.0000, 67.0000, 5.0000, 6.0000, 7.0000, 8.0000 ] - [-21.0000]
[ 5.0000, 5.0000, 5.0000, 5.0000, 5.0000, 5.0000, 69.0000, 6.0000, 7.0000, 8.0000 ] - [-11.0000]
[ 6.0000, 6.0000, 6.0000, 6.0000, 6.0000, 6.0000, 6.0000, 71.0000, 7.0000, 8.0000 ] - [-14.0000]
[ 7.0000, 7.0000, 7.0000, 7.0000, 7.0000, 7.0000, 7.0000, 7.0000, 73.0000, 8.0000 ] - [-17.0000]
[ 8.0000, 8.0000, 8.0000, 8.0000, 8.0000, 8.0000, 8.0000, 8.0000, 8.0000, 75.0000 ] - [-20.0000]

Result[0] = -4.0072; B[0] = -4.0000
Result[1] = 0.9928; B[1] = 1.0000
Result[2] = 5.9927; B[2] = 6.0000
Result[3] = 10.9823; B[3] = 11.0000
Result[4] = 15.9919; B[4] = 16.0000
Result[5] = 20.9912; B[5] = 21.0000
Result[6] = 10.9905; B[6] = 11.0000
Result[7] = 13.9895; B[7] = 14.0000
Result[8] = 16.9884; B[8] = 17.0000
Result[9] = 19.9872; B[9] = 20.0000

EXECUTAR BEM-SUCEDIDO (tempo total: 365ms)

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1 | alocMatriz | | 1056 | 81.481 | | 1056 |
| 2 | alocVector | | 240 | 18.519 | | 1296 |
-----

Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR BEM-SUCEDIDO (tempo total: 45ms)

WQS2

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1 | alocMatriz | | 1056 | 81.481 | | 1056 |
| 2 | alocVector | | 240 | 18.519 | | 1296 |
-----

Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

```

Figura 4.36: Resultados Sequência de Jacobi


```
16 - Sobel

** Processando filtro de Sobel. **

Digite a definicao desejada (linha, coluna): 10, 10
-1 -1 -16 -1 112 -1 -1 -1 -1 -1 -1
-128 -48 -96 -48 -1 -1 -1 -1 -1 -1 -64
-48 -96 -48 -1 -1 -1 112 -1 -1 96 -1
-1 -16 -64 -1 -32 -1 -1 -48 -1 -1 -64 80 -32
-1 0 -1 -32 80 -64 -1 -1 -48 -1 -1 -32 -1
-64 -16 -1 112 -1 -1 96 -1 -1 112 -1 -1 -1

EXECUTAR BEM-SUCEDIDO (tempo total: 101ms)

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1| main | 208 | 100.000 | 208 |
-----

Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR BEM-SUCEDIDO (tempo total: 167ms)

NIO$2

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1| main | 208 | 100.000 | 208 |
-----

Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.
```

Figura 4.37: Resultados Filtro de Sobel

```
17 - BMP Image Processing

Height=241 - Width=512

EXECUTAR BEM-SUCEDIDO (tempo total: 6s)

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1| smoth | 527872 | 50.000 | 1055744 |
| 2| main | 527872 | 50.000 | 527872 |
-----

Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.

EXECUTAR BEM-SUCEDIDO (tempo total: 184ms)

NIO$2

-----
|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|
-----
| 1| smoth | 527872 | 50.000 | 1055744 |
| 2| main | 527872 | 50.000 | 527872 |
-----

Remarks: Results in bytes.
(Fnc/Glb) = bytes Function / bytes greatest Global Accum.
```

Figura 4.38: Resultados Processamento de Imagem BMP

Conclusão

Como dito anteriormente, o desenvolvimento do presente trabalho para análise de *Profiling* de uso de memória *heap* é apenas a etapa inicial do desenvolvimento de um produto voltado a sistemas embarcados, primeiramente aplicado ao processador *soft core* Nios II, da Altera, podendo, posteriormente, ter seu uso estendido a outros processadores dessa natureza, como o Micro Blaze, da Xilinx, e outros.

No início, procurou-se desenvolver uma ferramenta de *Profiling* baseada no código fonte do GCC, o que acabou sendo uma tarefa extremamente complexa. Esta tarefa foi, sem dúvida nenhuma, subestimada no princípio deste Mestrado. O código do GCC possui milhares de linhas desenvolvidas por diferentes programadores, o que dificulta sobremaneira seu entendimento. Entretanto, com a ferramenta ora desenvolvida em operação, será possível começar a atacar o grande problema de alocação de memória em Sistemas Embarcados, mais especificamente naqueles que utilizam o processador *soft core* Nios II.

5.1 Contribuições deste Trabalho

Conforme já mencionado no tópico 1.4, conhecer em detalhes o padrão de acesso às memórias pode ser de grande auxílio na fase de co-projeto, visando tanto o desempenho do sistema quanto a economia de energia. Assim, esta ferramenta será de imensa utilidade para os projetos em an-

damento no LCR, principalmente aqueles que envolvem grandes movimentações de dados em processamento de imagem.

Um pré-processador que possa ser chamado de dentro do ambiente de desenvolvimento de aplicações para o Nios II para instrumentalizar o programa alvo abre oportunidade para a implementação de novos recursos de análise. Com o presente trabalho, verifica-se o uso de memória alocada dinamicamente, com a possibilidade de, além disso, com poucas implementações, conseguir-se detectar vazamentos de memória (*memory leaks*).

Embora o escopo do trabalho atual só se limite a analisar o uso de memória de forma restrita à alocação dinâmica da memória *heap*, a partir dele poder-se-á, futuramente, estudar formas de se obter com maior precisão informações sobre o uso global da memória, bem como sobre outros usos específicos, como o da memória *stack* e de memórias *cache* (quanto à detecção de *cache-misses*). Também a detecção de *memory leaks* e outros problemas associados ao uso de memória em sistemas embarcados poderão derivar do presente trabalho.

5.2 Trabalhos Futuros

Concluída a primeira etapa, preveem-se outras implementações e estudos, entre os quais:

- Alteração dos programas fonte GCC/GProf, inserindo-se neles comandos para obtenção do uso total de memória em cada amostragem obtida (embora não tão preciso e confiável quanto o princípio de registro de cada comando de alocação dinâmica de memória). Os valores coletados devem ser armazenados nos registros do arquivo gmon.out gerado durante a execução do programa. O GProf deverá efetuar uma análise da memória utilizada pelo princípio da maior quantidade de memória registrada para cada função. O analisador deverá fazer constar no relatório essa quantidade de memória registrada, permitindo ao usuário ordenar esse relatório por tempo de execução/quantidade de amostras ou por quantidade máxima de memória utilizada por função.
- Análise de uso memória especificamente em relação à *stack*.
- Instrumentação do fonte via aplicativo *source-to-source* Rose Compiler.
- Detecção de *memory leaks*.
- Análise de faltas em acessos a memórias *cache* (*cache misses*).
- Influência do compartilhamento simultâneo do mesmo espaço de memória por várias aplicações em termos da complicação da avaliação do uso de memória.

- Otimizações relacionadas à redução de consumo de energia.
- Otimizações relacionadas à redução da área do circuito.
- Análise de situações de paralisações de ciclos (*stall cycles*).
- Análise de problemas de contenção de unidade funcional (*functional unit contention*).
- Análise de falhas na previsão de ramificações (*branch mispredictions*).

Referências Bibliográficas

Diagramas. 2010.

Disponível em http://commons.wikimedia.org/wiki/File:4008_Gate-level_Diagram.svg, <http://asic-soc.blogspot.com.br/2007/12/asynchronous-fifo-clock-generation.html>, <http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm>

Bluespec. 2011.

Disponível em <http://www.bluespec.com/> [LastaccessedinDec28, 2011]

Valgrind. 2011.

Disponível em <http://valgrind.org/> [LastaccessedinNov12, 2011]

Altera. 2012.

Disponível em <http://www.altera.com/> [LastaccessedinMay14, 2012]

Gcc. 2012.

Disponível em <http://gcc.gnu.org/> [LastaccessedinApr25, 2012]

Gprof. 2012.

Disponível em <http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html> [LastaccessedinApr25, 2012]

A memory allocator. 2012.

Disponível em <http://gee.cs.oswego.edu/dl/html/malloc.html> [LastaccessedinNov5, 2012]

AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. *Compilers principles, techniques, and tools*. Pearson Addison Wesley, 2007.

ALTERACORPORATION *Profiling nios ii - application note 391, ver. 1.3*. 2008.

Disponível em <http://www.altera.com/literature/an/an391.pdf> [LastaccessedinDez01, 2009]

- ANDERSON, J. M.; BERC, L. M.; DEAN, J.; GHEMAWAT, S.; HENZINGER, M. R.; LEUNG, S.-T. A.; SITES, R. L.; VANDEVOORDE, M. T.; WALDSPURGER, C. A.; WEIHL, W. E. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, v. 15, n. 4, p. 357–390, 1997.
- BALL, T.; MATAGA, P.; SAGIV, M. Edge profiling versus path profiling: the showdown. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '98*, New York, NY, USA: ACM, 1998, p. 134–148 (POPL '98, v.). Disponível em <http://doi.acm.org/10.1145/268946.268958>
- BERGER, A. *Embedded systems design: an introduction to processes, tools, and techniques*. Embedded Systems Series. CMP Books, 2002.
- BONATO, V.; MARQUES, E. Roboarch: A component-based tool proposal for developing hardware architecture for mobile robots. In: *SIES*, 2009, p. 249–252.
- CARLONI, L. P.; DE BERNARDINIS, F.; PINELLO, C.; SANGIOVANNI-VINCENTELLI, A. L.; SGROI, M. Platform-based design for embedded systems. <http://www.cs.columbia.edu/luca/research/papersIndexByYear.shtml>, 2005.
Disponível em <http://www.cs.columbia.edu/~luca/research/pbdes.pdf>
- CONTE, T.; MENEZES, K. N.; HIRSCH, M. A. Accurate and practical profile-driven compilation using the profile buffer. In: *In Proceedings of the 29th Annual International Symposium on Microarchitecture*, ACM Press, 1996, p. 36–45.
- DEAN, J.; HICKS, J. E.; WALDSPURGER, C. A.; WEIHL, W. E.; CHRYSOS, G. Profileme: Hardware support for instruction-level profiling on out-of-order processors. 1997, p. 292–302.
- GHENASSIA, F. *Transaction-level modeling with systemc: Tlm concepts and applications for embedded systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.
- GUPTA, R. *Co-synthesis of hardware and software for digital embedded systems*. Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 1995.
- HÜBERT, H.; STABERNACK, B. Profiling-based hardware/software co-exploration for the design of video coding architectures. *IEEE Trans. Cir. and Sys. for Video Technol.*, v. 19, p. 1680–1691, 2009.
Disponível em <http://dx.doi.org/10.1109/TCSVT.2009.2031522>
- JAMES, T. B.; LARUS, J. R. Efficient path profiling. In: *In Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996, p. 46–57.
- LARMAN, C. *Utilizando uml e padrões*. BOOKMAN COMPANHIA ED, 2008.
- LUK, C.-K.; COHN, R.; MUTH, R.; PATIL, H.; KLAUSER, A.; LOWNY, G.; WALLACE, S.; REDDI, V. J.; HAZELWOOD, K. Pin: Building customized program analysis tools with

- dynamic instrumentation. 2005.
Disponível em <http://ursuletz.com/people/faculty/pdfs/p190-luk.pdf>
- MCDONALD, J. How esl becomes a business imperative. <http://www.eetimes.com/showArticle.jhtml?articleID=57701411> [Last accessed in Mar 09, 2010], 2005.
Disponível em <http://www.design-reuse.com/news/9481/how-esl-becomes-business-imperative.html>
- NIKHIL, R. Tutorial bluespec systemverilog: efficient, correct rtl from high-level specifications. 2010.
Disponível em <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01459818> [LastaccessedinFeb05, 2010]
- NOERGAARD, T. *Embedded systems architecture: a comprehensive guide for engineers and programmers*. Embedded technology series. Elsevier/Newnes, 2005.
- RAVAGNANI, G. S. Projeto de um sistema para monitoramento de hardware/software on-chip baseado em computação reconfigurável. *Master's dissertation: Universidade de São Paulo - USP*, 2007.
- ROCHA, E. A. *Princípios de economia*. Companhia Editora Nacional, 1975.
- SANGIOVANNI-VINCENTELLI, A. Quo vadis sld: Reasoning about trends and challenges of system-level design. *Proceedings of the IEEE*, v. 95, n. 3, p. 467–506, 2007.
Disponível em <http://chess.eecs.berkeley.edu/pubs/263.html>
- SOURCEWARE Gnu prof - table of contents - chapt. 9 - details of profiling - subchapt. 9.1 - implementation of profiling. 2010.
Disponível em <http://sourceware.org/binutils/docs/gprof/Implementation.html> [LastaccessedinFeb05, 2010]
- SPIRITCONSORTIUM Spirit consortium - ipxact. 2010.
Disponível em <http://www.spiritconsortium.org/home/> [LastaccessedinFeb05, 2010]
- SYSTEMC Open systemc initiative. 2010.
Disponível em <http://www.systemc.org/home/> [LastaccessedinJan28, 2010]
- SYSTEMVERILOG Open systemc initiative. 2010.
Disponível em <http://www.systemverilog.org/> [LastaccessedinJan28, 2010]
- TABBARA, B.; KASHMI, K. Transaction-level modelling and debug of socs. 2004.
Disponível em <http://www.design-reuse.com/articles/10626/transaction-level-modelling-and-debug-of-socs.html> [LastaccessedinJan28, 2010]

TENSILICA Esl design. 2009.

Disponível em <http://www.tensilica.com/methodology/esl-design.htm>[LastaccessedinSep05, 2009]

TONG, J. G.; KHALID, M. A. S. *A comparison of profiling tools for fpga-based embedded systems* p. 1687–1690, 2007.

Disponível em <http://www.scopus.com/inward/record.url?eid=2-s2.0-48749120511&partnerID=40&md5=71ee6a6b00ad218dfde007160e08c5e8>

VAHID, F.; GIVARGIS, T. *Embedded system design - a unified hardware / software introduction*. Wiley-VCH, I-XXI, 1-324 p., 2002.

WOLF, W. Computers as components - principles of embedded computing system design. 2005, p. I-XXVI, 1–656.

XILINX Xilinx bluespec. 2010a.

Disponível em http://www.xilinx.com/products/design_tools/logic_design/advanced/esl/bluespec.htm#tech[LastaccessedinFeb05, 2010]

XILINX, I. *Microblaze processor reference guide - embedded development kit - edk 12.2*. 2010b.

Disponível em http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/mb_ref_guide.pdf[LastaccessedinMay14, 2012]

ZHU, Y. Register-transfer level (rtl) design (a simple datapath and control unit). 2010.

Disponível em http://www.eece.maine.edu/~zhu/ece473/project/project-2/project_2.pdf[LastaccessedinFeb05, 2010]

ZYS, M.; VAUMORIN, E.; SOBANSKI, I.; ZHU, Y. Straightforward ip integration with ip-xact rtl-tlm switching. 2010.

Disponível em http://www.ipsupermarket.com/whitepapers/IP_Integration_RTL_TLM_Switching.pdf[LastaccessedinJan10, 2010]

IPXAct

Definido pelo consórcio SPIRIT, organização sem fins lucrativos da qual participam empresas como ARM, Cadence, Freescale, Mentor Graphics, Nokia, Altera, Xilinx, Synopsys, Texas Instruments, Samsung, entre dezenas de outras, o IP-XACT (A.1) é um padrão aberto (independente de linguagens e de fabricantes) de documentação de IPs utilizando metadados baseado em um conjunto unificado de especificações no formato XML (Zys *et al.*, 2010). Os metadados são definidos no padrão IP-XAct de forma compatível com técnicas de integração automatizada, através do quê diferentes desenvolvedores podem interconectar (bem como configurar e verificar) seus IPs entre si e entre os IPs de diversos outros fabricantes. Os metadados também podem ser usados por ferramentas de integração que utilizem APIs para integração entre ferramentas diversas, tais como as APIs LGI (Loose Generator Interface) e TGI (Tight Generator Interface), para consulta, manipulação e gravação de dados da base de metadados.

O padrão IP-XAct permite, nas ferramentas EDA, um acesso aos metadados dos componentes e sistemas de hardware/software representados - independentemente do fabricante e da implementação do componente - de forma a propiciar o reúso de IPs, bem como a automação do processo de desenvolvimento de projetos. Como resultado, obtém-se a utilização de qualquer IP em qualquer ferramenta EDA que ofereça o recurso de reconhecimento desse padrão.

Figura A.1: (SPIRITConsortium, 2010) - IPXAct

Níveis de Representação de Hardware

B.1 RTL

Design em nível RT (RTL - Register Transfer Level) corresponde à elaboração de circuitos digitais eletrônicos em termos de fluxo de dados entre registradores, em que as informações são armazenadas em circuitos digitais a cada ciclo de clock, bem como em termos das operações lógicas efetuadas sobre os sinais gerados. É largamente usado para implementar algoritmos em hardware, seja através de linguagens de descrição de hardware como, por exemplo, Verilog e VHDL, seja através de ferramentas que permitem a edição de esquemáticos em nível RT. As primitivas ou componentes em RTL compreendem registradores, lógica de direcionamento de dados (multiplexadores e demultiplexadores), barramentos e blocos funcionais (Zhu, 2010).

B.2 ESL

O projeto de sistemas em um nível mais alto de abstração que RTL (Verilog e VHDL) tem-se tornado uma forma mais avançada de elaboração de projeto de sistemas, que pode atuar tanto na redução dos esforços aplicados quanto na dos riscos envolvidos. Entretanto, há vários métodos para se elaborar projetos em níveis mais altos (normalmente chamados de projetos em nível de sistema eletrônico - "electronic system level", ou ESL), o que inclui síntese em alto nível, montagem e simulação de arquiteturas SoC (tanto para projetos baseados em mono ou multiprocessadores) e projeto de algoritmos funcionais (Tensilica, 2009).

Diversas linguagens têm sido especificadas e desenvolvidas com o intuito de permitir o desenvolvimento de sistemas eletrônicos em alto nível de abstração. Em sua grande maioria, baseiam-se nas linguagens C/C++, a cujos conjuntos de instruções são agregadas bibliotecas de classes padronizadas para modelagem de hardware. Entre os recursos incorporados por essas

classes encontram-se novas estruturas de dados, instruções dedicadas ao paralelismo, mecanismos de sincronização e interfaces para comunicação com módulos externos de hardware/software (Bonato e Marques, 2009). Abaixo, são destacadas as principais linguagens adotadas pelo mercado de fabricantes de sistemas eletrônicos.

1) HandelC

O HandelC é uma linguagem proprietária, originada na Universidade de Oxford, Inglaterra, e que passou, em um dado momento, a ser comercializada pela Celoxica, a qual, após um processo de fusão com outra empresa, foi adquirida pela Mentor Graphics. HandelC é baseada no ANSI C, e suporta a maioria dos tipos de dados e instruções do C, fornecendo novos recursos para exploração de paralelismo em hardware, alocação de recursos de hardware, comunicação e interface. Apesar de possuir um grande potencial para descrição das arquiteturas e de permitir a sua compilação para verificação funcional com compiladores tradicionais de C, HandelC não apresenta recursos de verificação e de orientação a objetos, e os sintetizadores para hardware, apesar de serem automáticos, são proprietários (Bonato e Marques, 2009).

2) SystemC

O SystemC, definido pelo padrão IEEE 1666-2005 (IEEE SYSTEMC, 2010) é promovido pelo OSCI - Open SystemC Initiative (SystemC, 2010) -, portanto, seu código é aberto e livre. A linguagem, que é composta de classes estendidas do C++ definidas em bibliotecas específicas (systemc.h), a partir de sua versão 2.0 tem sido associada a projetos ESL e, também, a TLM (Ghenassia, 2006). O paradigma de orientação a objetos de SystemC é uma característica que beneficia a exploração de paralelismo e o reúso de código, bem como propicia a diminuição na complexidade de modelagem. Além do mais, o código fonte em SystemC pode ser compilado para verificação funcional por compiladores tradicionais de C++ (Visual C++, GNU) (Bonato e Marques, 2009).

3) System Verilog

SystemVerilog, definida pelo padrão IEEE 1800-2005, é uma linguagem para descrição e verificação de hardware estendida de Verilog, esta definida pelo padrão IEEE-1364 (SystemVerilog, 2010). Assim como SystemC, SystemVerilog também suporta orientação a objetos, além de possuir interfaces para comunicação com módulos externos descritos em C/C++, SystemC e RTL. Considerada uma das linguagens mais completas, pois, além de possuir recursos de implementação, também oferece recursos avançados para verificação, SystemVerilog permite, ainda, que os modelos possam ser automaticamente sintetizados para hardware. Entretanto, uma grande desvantagem apontada pelos desenvolvedores que a utilizam é o fato de os compiladores para verificação funcional e síntese de hardware serem proprietários.

4) Bluespec/System Verilog

Bluespec ((Blu, 2011)), linguagem de descrição de hardware baseada em System Verilog, consiste em um conjunto de ferramentas EDL (Event Driven Language) para a elaboração de projetos de ASICs e de FPGAs.

Por sua semântica ser fundamentada em uma linguagem já familiar aos profissionais projetistas de hardware, sua curva de aprendizado tende a apresentar uma grande aceleração em um curto espaço de tempo (Xilinx, 2010a). E, apesar dessa similaridade, Bluespec implementa uma especificação comportamental de hardware em alto nível (semelhantemente aos padrões de softwares), o que propicia maior produtividade na elaboração dos projetos (Nikhil, 2010).

O paradigma seguido na definição de Bluespec é, como na maioria dos casos, o modelo baseado nas linguagens C e C++, porém, com alguns incrementos de classes para permitir representar paralelismos de processos, de canais e de primitivas de sincronização. Quando da tradução da especificação em alto nível, elementos de estado, data path e lógica de controle de hardware são construídos de forma estática, durante o processo de compilação, através de uma combinação de escalonamento e mapeamento (ou multiplexação) dos recursos compartilhados que utiliza Control Data Flow Graph (CDFG).

Já o comportamento de um componente especificado em Bluespec é expresso em termos de Guarded Atomic Actions (GAA), que são máquinas de estados finitos concorrentemente colaborativas que encerram em si a propriedade da atomicidade. Utilizam-se, no caso, regras (que são transações atômicas sequenciadas) que expressam operações concorrentes para descrever como devem ser efetuadas as atualizações dos elementos de estado.

Consequentemente, obtém-se uma grande simplificação na elaboração da lógica de controle em projetos especificados em Bluespec e, com isso, um alto controle sobre suas arquiteturas e o desempenho de cada sistema implementados sob os recursos dessa linguagem. Segundo a (Xilinx, 2010a), o uso de Bluespec proporciona cerca de 50% de redução de tempo na elaboração de projetos, além de uma redução de mais de 50% nos erros constatados, comparando-se com projetos RTL.

B.3 TLM

Segundo (Ghenassia, 2006), TLM (Transaction-level Modeling) corresponde a uma abordagem de modelagem fundamentada em transações e baseada em linguagens de programação de alto nível, como, por exemplo, SystemC. Nele, enfatiza-se o conceito de separação entre comunicação e computação em um sistema. Os componentes são especificados como módulos que contêm um conjunto de processos concorrentes que representam e processam seus respectivos comportamentos. Esses módulos comunicam-se entre si na forma de transações através de um canal abstrato. Em TLM, as interfaces são implementadas nos canais, encapsulando protocolos de comunicação. Para estabelecer uma comunicação, um processo necessita apenas acessar essas interfaces através das portas dos módulos.

TLM define uma transação como uma transferência de dados (ou seja, comunicação) ou uma sincronização entre dois módulos, em um dado instante (ou seja, evento de SoC), determinada pela especificação do sistema de hardware/software.

Programa nios2-elf-Hippie.c

C.1 Código fonte do programa nios2-elf-Hippie.c

```

1  /* ===== */
2  /*                      HIPPIE.C                      */
3  /* ===== */
4
5  /*
6   * File:   Hippie.c
7   * Author: Luiz Henrique Kiehn
8   *
9   */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>
14
15 /* ===== */
16 /*                      GLOBALS                      */
17 /* ===== */
18
19 #define CR 13
20 #define LF 10
21 #define SPACE ' '
22 #define NULL_STRING ""
23 #define NULL_ZERO '\0'
24 #define APOSTROPHE "\'"
25 #define QUOTE "\""
26 #define COMMA ", "
27 #define SCOPE_PROGRAM '0'
28 #define SCOPE_FUNCTION '1'
29 #define SIM 'S'
30 #define NAO 'N'
31 #define FALSE 0
32 #define TRUE 1
33 #define COMMENT_NO 0
34 #define COMMENT_SLASH_SLASH 1
35 #define COMMENT_SLASH_ASTERISK 2
36 #define MAXLEN 256 // Valor arbitrario. Assume-se que nao ha linhas com mais de 256 caracteres

```

```

37
38 const char strCR = (char) CR;
39 const char strLF = (char) LF;
40
41 /*
42 * -----
43 * | target | | InstrCode.c |
44 * -----
45 * | |
46 * | V | V
47 * -----
48 * | Hippie.c |
49 * -----
50 * |
51 * | V
52 * -----
53 * | HipInstr.c |
54 * -----
55 * |
56 * | V
57 * -----
58 * | GCC |
59 * -----
60 * |
61 * | V
62 * -----
63 * | HipInstr.exe |
64 * -----
65 * |
66 * | V
67 * -----
68 * | HipReport.out |
69 * -----
70 * |
71 * | V
72 * -----
73 * | HipReport.c |
74 * -----
75 * |
76 * | V
77 * -----
78 * | report |
79 * -----
80 */
81
82 //char fileNameTarget[19] = "greatest_element.c";
83 //char fileNameTarget[11] = "fatorial.c";
84 //char fileNameTarget[20] = "sequential_search.c";
85 //char fileNameTarget[16] = "binary_search.c";
86 //char fileNameTarget[14] = "bubble_sort.c";
87 //char fileNameTarget[17] = "insertion_sort.c";
88 //char fileNameTarget[17] = "selection_sort.c";
89 //char fileNameTarget[13] = "merge_sort.c";
90 //char fileNameTarget[13] = "quick_sort.c";
91 //char fileNameTarget[15] = "determinante.c";
92 //char fileNameTarget[24] = "matrix_multiplication.c";
93 //char fileNameTarget[13] = "transposta.c";
94 //char fileNameTarget[12] = "fibonacci.c";
95 //char fileNameTarget[9] = "gaussj.c";
96 //char fileNameTarget[13] = "jacobi_seq.c";
97 //char fileNameTarget[8] = "sobel.c";
98 char fileNameTarget[12] = "BMP_Image.c"; // BMP image processing
99 char fileNameInstCode[11] = "InstCode.c";
100 char fileNameFinalCode[25] = "../FinalCode/HipInstr.c";
101
102 FILE *fileHProfTarget ,
103      *fileInstrumentationCode ,

```

```

104     *fileHProfFinalCode;
105
106 typedef struct {
107     char dynAllocFunctionType;
108     int  dynAllocCommandInitial;
109     int  dynAllocParamInitial;
110     int  dynAllocParamFinal;
111     char dynAllocStrParam[MAXLEN];
112 } structDynAlloc;
113
114 char programInitial    = SPACE;
115 char programScope     = SPACE;
116 char streamLine[MAXLEN] = NULL_STRING;
117 char functionName[30]  = NULL_STRING;
118 int  commentType       = COMMENT_NO;
119 int  blockCounter      = 0;
120
121 structDynAlloc regDynAlloc;
122
123 /* ===== */
124 /* File HProf - Target file to be profiled */
125 /* ===== */
126
127 /* Open File HProfTarget */
128 /* ----- */
129 int fileHProfTarget_Open(char fileNameTarget[]) {
130     if ((fileHProfTarget = fopen(fileNameTarget, "r")) == NULL) {
131         printf("Error opening file %s.\n", fileNameTarget);
132         exit(1);
133         //return 1;
134     } else {
135         printf("Abriu arquivo %s.\n", fileNameTarget);
136         return 0;
137     }
138 }
139
140 /* Close File HProfTarget */
141 /* ----- */
142 int fileHProfTarget_Close() {
143     fclose(fileHProfTarget);
144 }
145
146 /* Read File HProfTarget */
147 /* ----- */
148 int fileHProfTarget_Read() {
149
150     int lenStr = 0;
151     int ch     = SPACE;
152
153     ch = fgetc(fileHProfTarget);
154     while (!feof(fileHProfTarget) && ch != LF) {
155         streamLine[lenStr] = (char) ch;
156         lenStr++;
157         ch = fgetc(fileHProfTarget);
158     }
159     if (!feof(fileHProfTarget)) {
160         streamLine[lenStr] = (char) LF;
161         lenStr++;
162     }
163     streamLine[lenStr] = NULL_ZERO;
164     return lenStr;
165 }
166
167 /* ===== */
168 /* File HProf - Final Code */
169 /* ===== */
170

```

```

171 /* Open File HProfFinalCode (Heap Profile Instrumented) */
172 /* ----- */
173 int fileHProfFinalCode_Open(char fileNameFinalCode[]) {
174     if ((fileHProfFinalCode = fopen(fileNameFinalCode, "w")) == NULL) {
175         printf("Error opening file %s.\n", fileNameFinalCode);
176         return TRUE;
177     } else {
178         printf("Abriu arquivo %s.\n", fileNameFinalCode);
179         return FALSE;
180     }
181 }
182
183 /* Close File HProfFinalCode (Heap Profile Instrumented) */
184 /* ----- */
185 int fileHProfFinalCode_Close() {
186     fclose(fileHProfFinalCode);
187 }
188
189 /* Write File HProfFinalCode (Heap Profile Instrumented) */
190 /* ----- */
191 int fileHProfFinalCode_Write(char *streamLine) {
192
193     char strLine[MAXLEN] = NULL_STRING;
194     char auxStr[2] = NULL_STRING;
195
196     auxStr[0] = regDynAlloc.dynAllocFunctionType;
197     if (regDynAlloc.dynAllocFunctionType == SPACE) {
198         strcpy(strLine, streamLine);
199     } else {
200         strncpy(strLine, streamLine, regDynAlloc.dynAllocCommandInitial);
201         strcat(strLine, "dynAlloc_HProf(");
202         strcat(strLine, QUOTE);
203         strcat(strLine, functionName);
204         strcat(strLine, QUOTE);
205         strcat(strLine, COMMA);
206         strcat(strLine, APOSTROPHE);
207         strcat(strLine, auxStr);
208         strcat(strLine, APOSTROPHE);
209         strcat(strLine, COMMA);
210         switch (regDynAlloc.dynAllocFunctionType) {
211             case 'm':
212                 strcat(strLine, "NULL, ");
213                 strcat(strLine, regDynAlloc.dynAllocStrParam);
214                 strcat(strLine, ", 0");
215                 break;
216             case 'c':
217                 strcat(strLine, "NULL, ");
218                 strcat(strLine, regDynAlloc.dynAllocStrParam);
219                 break;
220             case 'r':
217                 strcat(strLine, regDynAlloc.dynAllocStrParam);
222                 strcat(strLine, ", 0");
223                 break;
224             case 'f':
225                 strcat(strLine, regDynAlloc.dynAllocStrParam);
226                 strcat(strLine, ", 0, 0");
227                 break;
228             default:
229                 break;
230         }
231         strcat(strLine, &streamLine[regDynAlloc.dynAllocParamFinal]);
232         // printf("%s\n", strLine);
233     }
234 }
235 fwrite(strLine, strlen(strLine), 1, fileHProfFinalCode);
236 }
237

```

```

238 /* ===== */
239 /* File Instrumentation Code - Input */
240 /* ===== */
241
242 /* Open File InstCode (Instrumentation code to be inserted) */
243 /* ===== */
244 int fileInstrumentationCode_Open(char fileNameInstCode[]) {
245     if ((fileInstrumentationCode = fopen(fileNameInstCode, "r")) == NULL) {
246         printf("Error opening file %s.\n", fileNameInstCode);
247         // exit(1);
248         return TRUE;
249     } else {
250         printf("Abriu arquivo %s.\n", fileNameInstCode);
251         return FALSE;
252     }
253 }
254
255 /* Close File InstCode (Instrumentation code to be inserted) */
256 /* ===== */
257 int fileInstrumentationCode_Close() {
258     fclose(fileInstrumentationCode);
259 }
260
261 /* Insert Instrumentation Code */
262 /* ===== */
263 void fileInstrumentationCode_Insert() {
264
265     char strInstCode[MAXLEN];
266
267     rewind(fileInstrumentationCode);
268     fgets(strInstCode, MAXLEN, fileInstrumentationCode);
269     while (!feof(fileInstrumentationCode)) {
270         fwrite(strInstCode, strlen(strInstCode), 1, fileHProfFinalCode);
271         fgets(strInstCode, MAXLEN, fileInstrumentationCode);
272     }
273 }
274
275 /* ===== */
276 /* Sets and Gets */
277 /* ===== */
278
279 void setProgramInitial(char ch) {
280     programInitial = ch;
281 }
282
283 char getProgramInitial() {
284     return programInitial;
285 }
286
287 int isProgramInitial(char ch) {
288     if (programInitial == ch) {
289         return TRUE;
290     } else {
291         return FALSE;
292     }
293 }
294
295 void setProgramScope(char ch) {
296     programScope = ch;
297 }
298
299 char getProgramScope() {
300     return programScope;
301 }
302
303 int isValidName(char ch) {
304     int validName = FALSE;

```

```

305     if ((ch >= 'A' && ch <= 'Z') ||
306         (ch >= 'a' && ch <= 'z') ||
307         (ch >= '0' && ch <= '9') ||
308         (ch == '_')) {
309         validName = TRUE;
310     }
311     return validName;
312 }
313
314 void setFunctionName(char *streamLine, int posLine) {
315     int posBegin,
316         posEnd,
317         indLine,
318         indName;
319     if (posLine == 0) {
320         strcpy(functionName, NULL_STRING);
321     } else {
322         indLine = posLine - 1;
323         while (indLine > 0 && streamLine[indLine] == SPACE) indLine--;
324         posEnd = indLine;
325         while (indLine > 0 && isValidName(streamLine[indLine]) == TRUE) indLine--;
326         posBegin = indLine + 1;
327         indName = 0;
328         for (indLine = posBegin; indLine <= posEnd; indLine++, indName++) {
329             functionName[indName] = streamLine[indLine];
330         }
331         functionName[indName] = NULL_ZERO;
332     }
333 }
334
335 void setRegDynAlloc(structDynAlloc paramDynAlloc) {
336     regDynAlloc = paramDynAlloc;
337 }
338
339 /* ===== */
340 /*      PROCESS FILE      */
341 /* ===== */
342
343 /* ----- */
344 /* Find Dynamic Allocation Command */
345 /* ----- */
346 /* The source code can not have the strings malloc, calloc, realloc and free, */
347 /* except as the corresponding C function names. */
348 structDynAlloc parseDynAlloc(char *streamLine) {
349
350     int indLine = 0,
351         indParam = 0,
352         countParent = 0;
353     char *auxPtr = NULL;
354
355     structDynAlloc regDynAlloc;
356     regDynAlloc.dynAllocFunctionType = SPACE;
357     regDynAlloc.dynAllocCommandInitial = 0;
358     regDynAlloc.dynAllocParamInitial = 0;
359     regDynAlloc.dynAllocParamFinal = 0;
360     strcpy(regDynAlloc.dynAllocStrParam, NULL_STRING);
361
362     /* Return the corresponding address of the location of malloc/calloc/realloc/free commands */
363     auxPtr = strstr(streamLine, " malloc(");
364     if (auxPtr != NULL) {
365         regDynAlloc.dynAllocFunctionType = 'm';
366     } else {
367         auxPtr = strstr(streamLine, " calloc(");
368         if (auxPtr != NULL) {
369             regDynAlloc.dynAllocFunctionType = 'c';
370         } else {
371             auxPtr = strstr(streamLine, " realloc(");

```

```

372         if (auxPtr != NULL) {
373             regDynAlloc.dynAllocFunctionType = 'r';
374         } else {
375             auxPtr = strstr(streamLine, " free(");
376             if (auxPtr != NULL) {
377                 regDynAlloc.dynAllocFunctionType = 'f';
378             }
379         }
380     }
381 }
382
383 /* Return the command and its parameters initial and final positions in the streamline */
384 if (regDynAlloc.dynAllocFunctionType != SPACE) {
385     auxPtr++;
386     regDynAlloc.dynAllocCommandInitial = auxPtr - &streamLine[0];
387     indLine = regDynAlloc.dynAllocCommandInitial;
388     while (streamLine[indLine] != '(') indLine++;
389     indLine++;
390     regDynAlloc.dynAllocParamInitial = indLine;
391     indParam = 0;
392     countParent = 0;
393     while (streamLine[indLine] != ')' || countParent > 0) {
394         regDynAlloc.dynAllocStrParam[indParam] = streamLine[indLine];
395         switch (streamLine[indLine]) {
396             case '(':
397                 countParent++;
398                 break;
399             case ')':
400                 countParent--;
401                 break;
402         }
403         indParam++;
404         indLine++;
405     }
406     regDynAlloc.dynAllocStrParam[indParam] = NULL_ZERO;
407     regDynAlloc.dynAllocParamFinal = indLine;
408 }
409 return regDynAlloc;
410 }
411
412
413 /* ----- */
414 /* Parse Line */
415 /* ----- */
416 void parseLine(char *streamLine) {
417
418     if (blockCounter == 0) {
419         //Fechou bloco de funcao
420         setProgramScope(SCOPE_PROGRAM);
421         setFunctionName(streamLine, 0);
422     }
423
424     /*** AVISO *** ==> Falta prever prototipos sem abre/fecha bloco
425     int posLine = 0;
426     while (posLine < MAXLEN && streamLine[posLine] != strLF) { //Percorre a linha inteira
427
428         if (commentType != COMMENT_SLASH_SLASH) {
429             if (commentType == COMMENT_SLASH_ASTERISK) {
430                 if (posLine > 0 && streamLine[posLine-1] == '*' && streamLine[posLine] == '/') {
431                     commentType = COMMENT_NO;
432                 }
433             } else {
434                 if (posLine > 0 && streamLine[posLine-1] == '/' && streamLine[posLine] == '*') {
435                     commentType = COMMENT_SLASH_ASTERISK;
436                 } else if (posLine > 0 && streamLine[posLine-1] == '/' && streamLine[posLine] == '/') {
437                     commentType = COMMENT_SLASH_SLASH;
438                 } else if (programScope == SCOPE_FUNCTION && streamLine[posLine] == '{') {

```

```

439         //Encontrou inicio de bloco dentro de funcao
440         blockCounter++;
441     } else if (programScope == SCOPE_FUNCTION && streamLine[posLine] == '}') {
442         //Encontrou fim de bloco dentro de funcao
443         blockCounter--;
444     } else if (programScope == SCOPE_PROGRAM && streamLine[posLine] == '(') {
445         if (isProgramInitial(SIM)) {
446             fileInstrumentationCode_Insert();
447             setProgramInitial(NAO);
448         }
449         //Encontrou declaracao de funcao
450         setProgramScope(SCOPE_FUNCTION);
451         setFunctionName(streamLine, posLine);
452     }
453 }
454 }
455 posLine++;
456
457 }
458 if (commentType == COMMENT_SLASH_SLASH) {
459     //Termino de linha => termino de comentario tipo //
460     commentType = COMMENT_NO;
461 }
462 }
463 }
464
465 /* ===== */
466 /*      MAIN      */
467 /* ===== */
468 int main(int argc, char *argv[]) {
469     /*
470     {
471         if(argc == 2)
472         {
473             process_file(argv[1]);
474             process_file("teste.c");
475         }
476         else
477         {
478             printf("Erro: Número de argumentos inválido!\n");
479             process_file("teste.c");
480         }
481         return 0;
482     }
483     */
484
485     //char *nomeFile = argv[1];
486     //
487     fileHProfTarget_Open(fileNameTarget);
488     fileHProfFinalCode_Open(fileNameFinalCode);
489     fileInstrumentationCode_Open(fileNameInstCode);
490
491     setProgramInitial(SIM);
492     setProgramScope(SCOPE_PROGRAM);
493     setFunctionName(streamLine, 0);
494     commentType = 0;
495     blockCounter = 0;
496     while (fileHProfTarget_Read() > 0) {
497         parseLine(streamLine);
498         setRegDynAlloc(parseDynAlloc(streamLine));
499         fileHProfFinalCode_Write(streamLine);
500     }
501
502     fileHProfTarget_Close();
503     fileHProfFinalCode_Close();

```

```
506     fileInstrumentationCode_Close ();
507
508     // system ( "PAUSE" );
509     return (EXIT_SUCCESS);
510 }
```

Programa InstCode.c

D.1 Código fonte do programa InstCode.c

```

1  /* ===== */
2  /*                               INSTCODE.C                               */
3  /* ===== */
4
5  #define CR 13
6  #define LF 10
7  const char strCR = (char) CR;
8  const char strLF = (char) LF;
9
10 char fileNameOut[24] = "/mnt/host/HipReport.out";
11 char openType[3] = "wb";
12
13 FILE *fileHProfOut;
14
15 typedef struct {
16     char    HProfOutFunctionName[32];
17     void    *HProfOutPtrOrig;
18     void    *HProfOutPtrTarget;
19     long int HProfOutSize;
20     char    HProfOutFiller1[2];
21     char    HProfOutCR;
22     char    HProfOutLF;
23 } structHProfOut;
24
25 #define sizeHProfOut sizeof(structHProfOut)
26
27 typedef union union_HProfOut {
28     char    unionStr[sizeHProfOut];
29     structHProfOut unionReg;
30 } unionHProfOut;
31
32 structHProfOut regHProfOut;
33
34
35 /* ===== */
36 /*      File HProf - Out      */

```

```

37 /* ===== */
38
39 /* Open File HProf (Heap Profile) */
40 /* ===== */
41 int fileHProfOut_Open(char fileNameOut[]) {
42     if ((fileHProfOut = fopen(fileNameOut, openType)) == NULL) {
43         printf("Error opening file %s.\n", fileNameOut);
44         return 1;
45     } else {
46         //printf("Abriu arquivo %s.\n", fileNameOut);
47         if (openType[0] == 'w') {
48             openType[0] = 'a';
49         }
50         return 0;
51     }
52 }
53
54 /* Close File HProfOut */
55 /* ===== */
56 int fileHProfOut_Close() {
57     fclose(fileHProfOut);
58 }
59
60 /* Write File HProfOut */
61 /* ===== */
62 int fileHProfOut_Write(structHProfOut *paramHProfOut) {
63     fwrite(paramHProfOut, sizeof(regHProfOut), 1, fileHProfOut);
64 }
65
66
67 /* ===== */
68 /* Dynamic Allocation Type Selection */
69 /* ===== */
70
71 void *dynAlloc_HProf(char *functionName, char functionType, void *paramPtr, unsigned int nbytes, unsigned int nsize) {
72     fileHProfOut_Open(fileNameOut);
73
74     void *returnPtr = NULL;
75     void *ptrAuxVoid = NULL;
76     char descrFunction[8] = "";
77     int memSize = 0;
78
79     switch (functionType) {
80         case 'm':
81             strcpy(descrFunction, "MALLOC");
82             memSize = nbytes;
83             returnPtr = malloc(nbytes);
84             break;
85         case 'c':
86             strcpy(descrFunction, "CALLOC");
87             memSize = nbytes * nsize;
88             returnPtr = calloc(nbytes, nsize);
89             break;
90         case 'r':
91             strcpy(descrFunction, "REALLOC");
92             memSize = nbytes;
93             returnPtr = realloc(paramPtr, nbytes);
94             break;
95         case 'f':
96             strcpy(descrFunction, "FREE");
97             memSize = 0;
98             free(paramPtr);
99             break;
100     }
101
102     memSize = (int) ((floor((memSize / 8.0) + 0.49) + 1.0) * 8.0);

```

```
104     if (memSize < 16) memSize = 16;
105
106     if (returnPtr == NULL && strcmp(descrFunction, "FREE") != 0) {
107         printf ("Allocation error: %s\n", descrFunction);
108         exit (EXIT_FAILURE);
109     } else {
110         strcpy (regHProfOut.HProfOutFunctionName, functionName);
111         regHProfOut.HProfOutPtrOrig = paramPtr;
112         regHProfOut.HProfOutPtrTarget = returnPtr;
113         regHProfOut.HProfOutSize = memSize;
114         strcpy (regHProfOut.HProfOutFiller1, "00");
115         regHProfOut.HProfOutCR = strCR;
116         regHProfOut.HProfOutLF = strLF;
117         ptrAuxVoid = &regHProfOut;
118         fileHProfOut_Write (ptrAuxVoid);
119     }
120
121     fileHProfOut_Close ();
122
123     return returnPtr;
124
125 }
126
127 //=====
```

Programa nios2-elf-HipReport.c

E.1 Código fonte do programa nios2-elf-HipReport.c

```

1  /* ===== */
2  /*          HIPREPORT.C          */
3  /* ===== */
4
5  /*
6   * Author: Luiz Henrique Kiehn
7   *
8   */
9
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <math.h>
14 #include <malloc.h>
15
16 /* ===== */
17 /*          GLOBALS          */
18 /* ===== */
19
20 #define CR 13
21 #define LF 10
22 #define SPACE ' '
23 #define NULL_STRING ""
24 #define NULL_STRING10 "          "
25 #define NULL_ZERO '\0'
26 #define QUOTE "\""
27 #define COMMA ", "
28 #define SCOPE_PROGRAM '0'
29 #define SCOPE_FUNCTION '1'
30 #define SIM 'S'
31 #define NAO 'N'
32 #define FALSE 0
33 #define TRUE 1
34 #define MAXLEN 256 // Valor arbitrario. Assume-se que nao ha linhas com mais de 256 caracteres
35
36 //int sizeUnsignLongInt = sizeof(unsigned long int);

```

```

37
38 const char strCR = (char) CR;
39 const char strLF = (char) LF;
40 char strNumber[11] = NULL_STRING10;
41 int sizePtrOrig = 0,
42     sizePtrTarget = 0;
43 int regFound = 0;
44
45 /* Struct HProfOut - Arquivo de Resultados Coletados */
46 /* ----- */
47 char fileNameOut[24] = "../RHProf/HipReport.out";
48 FILE *fileHProfOut;
49
50 typedef struct struct_HProfOut {
51     char HProfOutFunctionName[32];
52     void *HProfOutPtrOrig;
53     void *HProfOutPtrTarget;
54     long int HProfOutSize;
55     char HProfOutFiller1[2];
56     char HProfOutCR;
57     char HProfOutLF;
58 } structHProfOut;
59
60 const int regOutSize = sizeof(structHProfOut);
61
62 structHProfOut regHProfOut,
63                 *ptrHProfOut;
64
65 /* Struct HProfList - Lista duplamente encadeada de registro para
66    rastreo das alocações efetuadas
67 ----- */
68 typedef struct struct_HProfList {
69     char HProfListFunctionName[32];
70     void *HProfListPtr;
71     int HProfListSize;
72     char HProfListProcessed;
73     struct struct_HProfList *HProfListPrevious;
74     struct struct_HProfList *HProfListNext;
75 } structHProfList;
76
77 const int regListSize = sizeof(structHProfList);
78
79 structHProfList *ptrHProfList = NULL,
80                 *ptrHProfListNext = NULL,
81                 *ptrListPrevious = NULL,
82                 *ptrListFirst = NULL,
83                 *ptrListLast = NULL;
84
85 /* Struct AcumKey - Estrutura de definição de campo chave de ordenação
86    para impressão dos resultados apurados em HProfAcum
87 ----- */
88 typedef struct struct_acumKey {
89     char HProfAcumMaxFunctionSize[11];
90     char HProfAcumMaxGlobalSize[11];
91     char HProfAcumFunctionName[32];
92 } structAcumKey;
93
94 typedef union union_acumKey {
95     struct structAcumKey varUnionAcumKey;
96     char varStrAcumKey[ sizeof(structAcumKey) ];
97 } unionAcumKey;
98
99 unsigned long int acumGlobalSize = 0,
100                  acumMaxGlobalSize = 0;
101
102 void *ptrCmpAcumKey1 = NULL,
103       *ptrCmpAcumKey2 = NULL;

```



```

104
105 /* Struct HProfAcum — Lista duplamente encadeada de registro de valores
106      acumulados por funcao e globalmente
107      _____ */
108 typedef struct struct_HProfAcum {
109     structAcumKey          HProfAcumKey;
110     unsigned long int      HProfAcumFunctionSize;
111     struct struct_HProfAcum *HProfAcumPrevious;
112     struct struct_HProfAcum *HProfAcumNext;
113 } structHProfAcum;
114
115 const int regAcumSize = sizeof(structHProfAcum);
116
117 structHProfAcum *ptrHProfAcum = NULL,
118     *ptrAcumFirst = NULL,
119     *ptrAcumLast = NULL;
120
121 /* ===== */
122 /*      File HProf — Out      */
123 /* ===== */
124
125 /* Open File HProf (Heap Profile) */
126 /* _____ */
127 int fileHProfOut_Open(char *fileNameOut) {
128     char outFileName[11];
129     strcpy(outFileName, fileNameOut);
130     if ((fileHProfOut = fopen(fileNameOut, "rb")) == NULL) {
131         printf("Error opening file %s.\n", outFileName);
132         return 1;
133     } else {
134         //printf("Abriu arquivo %s.\n", outFileName);
135         return 0;
136     }
137 }
138
139 /* Close File HProfOut */
140 /* _____ */
141 int fileHProfOut_Close() {
142     fclose(fileHProfOut);
143 }
144
145 /* Read File HProfOut (List) */
146 /* _____ */
147 // structHProfList *fileHProfOut_Read() {
148 void fileHProfOut_Read(void *ptrHProfOutRead) {
149     fread(ptrHProfOutRead, regOutSize, 1, fileHProfOut);
150 }
151
152 /* ===== */
153 /* Convert Number to String */
154 /* ===== */
155 void longToString(unsigned long int intNumber, char *strNumber) {
156     strcpy(strNumber, NULL_STRING10);
157     int indNumber = 9,
158         rest = 0;
159     while (indNumber >= 0 && intNumber != 0) {
160         rest = intNumber % 10;
161         strNumber[indNumber] = (char) (48 + rest);
162         intNumber = (intNumber - rest) / 10;
163         indNumber--;
164     }
165 }
166
167 /* ===== */
168 /* Customized String Compare */
169 /* ===== */
170 /* Usada para comparacao entre as chaves de ordenacao (AcumKey) */

```

```

171 /* dos resultados apurados em HProfAcum */
172 /* ----- */
173 /* varCmp1 > varCmp2 => resultComp > 0 */
174 /* varCmp1 = varCmp2 => resultComp = 0 */
175 /* varCmp1 < varCmp2 => resultComp < 0 */
176 /* ----- */
177 int custStrCmp(unionAcumKey *paramCmp1, unionAcumKey *paramCmp2, int paramSize) {
178     int indCmp = 0,
179         resultComp = 0;
180     unsigned char varCmp1 = 0,
181                 varCmp2 = 0;
182     while (indCmp <= paramSize) {
183         varCmp1 = (unsigned char) paramCmp1->varStrAcumKey[indCmp];
184         varCmp2 = (unsigned char) paramCmp2->varStrAcumKey[indCmp];
185         if (varCmp1 > varCmp2) {
186             return 1;
187         } else {
188             if (varCmp1 < varCmp2) {
189                 return -1;
190             } else {
191                 indCmp++;
192             }
193         }
194     }
195     return resultComp;
196 }
197
198 /* ===== */
199 /* Quick Sort */
200 /* ===== */
201
202 void swap(structHProfAcum* structA, structHProfAcum* structB) {
203
204     structHProfAcum tmpHProfAcum;
205
206     // Swap entre structA e structB
207     strcpy(tmpHProfAcum.HProfAcumKey.HProfAcumMaxGlobalSize, structA->HProfAcumKey.HProfAcumMaxGlobalSize);
208     strcpy(tmpHProfAcum.HProfAcumKey.HProfAcumMaxFunctionSize, structA->HProfAcumKey.HProfAcumMaxFunctionSize);
209     strcpy(tmpHProfAcum.HProfAcumKey.HProfAcumFunctionName, structA->HProfAcumKey.HProfAcumFunctionName);
210     tmpHProfAcum.HProfAcumFunctionSize = structA->HProfAcumFunctionSize;
211
212     strcpy(structA->HProfAcumKey.HProfAcumMaxGlobalSize, structB->HProfAcumKey.HProfAcumMaxGlobalSize);
213     strcpy(structA->HProfAcumKey.HProfAcumMaxFunctionSize, structB->HProfAcumKey.HProfAcumMaxFunctionSize);
214     strcpy(structA->HProfAcumKey.HProfAcumFunctionName, structB->HProfAcumKey.HProfAcumFunctionName);
215     structA->HProfAcumFunctionSize = structB->HProfAcumFunctionSize;
216
217     strcpy(structB->HProfAcumKey.HProfAcumMaxGlobalSize, tmpHProfAcum.HProfAcumKey.HProfAcumMaxGlobalSize);
218     strcpy(structB->HProfAcumKey.HProfAcumMaxFunctionSize, tmpHProfAcum.HProfAcumKey.HProfAcumMaxFunctionSize);
219     strcpy(structB->HProfAcumKey.HProfAcumFunctionName, tmpHProfAcum.HProfAcumKey.HProfAcumFunctionName);
220     structB->HProfAcumFunctionSize = tmpHProfAcum.HProfAcumFunctionSize;
221 }
222
223 /* -----
224          Partition
225 ----- */
226 structHProfAcum *partition(structHProfAcum *ptrLeft, structHProfAcum *ptrRight) {
227
228     structHProfAcum *ptr_I,
229                 *ptr_J;
230     void *ptrCmp_J,
231         *ptrCmp_Left;
232
233     ptr_I = ptrLeft;
234     for (ptr_J = ptrLeft->HProfAcumNext; (ptr_J != NULL && ptr_J != ptrRight->HProfAcumNext); ptr_J = ptr_J->HProfAcumNext) {
235         ptrCmp_J = &ptr_J->HProfAcumKey;
236         ptrCmp_Left = &ptrLeft->HProfAcumKey;
237         if (custStrCmp(ptrCmp_J, ptrCmp_Left, sizeof(structAcumKey)) < 0) {

```

```

238     ptr_I = ptr_I->HProfAcumNext;
239     swap(ptr_I, ptr_J);
240 }
241 }
242 swap(ptrLeft, ptr_I);
243
244 return ptr_I;
245 }
246
247 /* -----
248                Quick Sort Main
249 ----- */
250 void quickSort(structHProfAcum *ptrLeft, structHProfAcum *ptrRight) {
251
252     structHProfAcum *ptrPivot;
253
254     ptrCmpAcumKey1 = &ptrLeft->HProfAcumKey;
255     ptrCmpAcumKey2 = &ptrRight->HProfAcumKey;
256     if (custStrCmp(ptrCmpAcumKey1, ptrCmpAcumKey2, sizeof(structAcumKey)) > 0) {
257         ptrPivot = partition(ptrLeft, ptrRight);
258         if (ptrPivot->HProfAcumPrevious != NULL) {
259             quickSort(ptrLeft, ptrPivot->HProfAcumPrevious);
260         }
261         if (ptrPivot->HProfAcumNext != NULL) {
262             quickSort(ptrPivot->HProfAcumNext, ptrRight);
263         }
264     }
265 }
266
267 /* ===== */
268 /*                MAIN                */
269 /* ===== */
270
271 /* ----- */
272 /* Atualiza ACUM a partir dos valores apurados (sizePtrOrig/sizePtrTarget) */
273 /* ----- */
274 void atualizaAcum (int sizePtr) {
275
276     // Atualiza acumulador geral (soma valor positivo/negativo);
277     acumGlobalSize += sizePtr;
278     if (acumGlobalSize > acumMaxGlobalSize) {
279         acumMaxGlobalSize = acumGlobalSize;
280     }
281
282     // Procura HProfListFunctionName em ACUM; se nao encontrar, cria registro;
283     regFound = FALSE;
284     if (ptrHProfList != NULL) {
285         ptrHProfAcum = ptrAcumFirst;
286         ptrCmpAcumKey1 = &ptrHProfList->HProfListFunctionName;
287     } else {
288         ptrHProfAcum = NULL;
289     }
290     while((ptrHProfAcum != NULL) && (regFound == FALSE)) {
291         ptrCmpAcumKey2 = &ptrHProfAcum->HProfAcumKey.HProfAcumFunctionName;
292         if (strcmp(ptrCmpAcumKey1, ptrCmpAcumKey2) == 0) {
293             regFound = TRUE;
294         } else {
295             ptrHProfAcum = ptrHProfAcum->HProfAcumNext;
296         }
297     }
298     if (!regFound) {
299         // functionName nao encontrado em ACUM: cria entrada para ele
300         ptrHProfAcum = malloc(regAcumSize);
301         // strcpy(ptrHProfAcum->HProfAcumKey.HProfAcumFunctionName, regHProfOut.HProfOutFunctionName);
302         strcpy(ptrHProfAcum->HProfAcumKey.HProfAcumFunctionName, ptrHProfList->HProfListFunctionName);
303         strcpy(ptrHProfAcum->HProfAcumKey.HProfAcumMaxGlobalSize, NULL_STRING10);
304         strcpy(ptrHProfAcum->HProfAcumKey.HProfAcumMaxFunctionSize, NULL_STRING10);

```

```

305     ptrHProfAcum->HProfAcumFunctionSize = 0;
306     ptrHProfAcum->HProfAcumPrevious      = ptrAcumLast;
307     ptrHProfAcum->HProfAcumNext          = NULL;
308     if (ptrAcumLast != NULL) {
309         ptrAcumLast->HProfAcumNext = ptrHProfAcum;
310     }
311     if (ptrAcumFirst == NULL) {
312         ptrAcumFirst = ptrHProfAcum;
313     }
314     ptrAcumLast = ptrHProfAcum;
315 }
316 // Soma PTR2.size a AcumSize ou subtrai PTR1.size de AcumSize
317 ptrHProfAcum->HProfAcumFunctionSize += sizePtr;
318
319 if (sizePtr > 0) {
320     // Verifica AcumMaxGlobalSize
321     if (acumGlobalSize > atol(ptrHProfAcum->HProfAcumKey.HProfAcumMaxGlobalSize)) {
322         longToString(acumGlobalSize, strNumber);
323         strcpy(ptrHProfAcum->HProfAcumKey.HProfAcumMaxGlobalSize, strNumber);
324     }
325     // Verifica AcumMaxFunctionSize
326     if (ptrHProfAcum->HProfAcumFunctionSize > atol(ptrHProfAcum->HProfAcumKey.HProfAcumMaxFunctionSize)) {
327         longToString(ptrHProfAcum->HProfAcumFunctionSize, strNumber);
328         strcpy(ptrHProfAcum->HProfAcumKey.HProfAcumMaxFunctionSize, strNumber);
329     }
330 }
331 }
332
333
334 /* ----- */
335 /*           Main           */
336 /* ----- */
337 int main(int argc, char** argv) {
338
339     /* Ler sequencialmente os registros de Resultados Coletados (HProfOut)
340        computando, a partir dos tamanhos de memoria alocada,
341        para cada resultado:
342        - o maior acumulo da funcao e
343        - o maior acumulo geral.
344     */
345
346     /*
347        Situacoes possiveis de alocao de memoria e respectivos resultados coletados:
348
349        |           | malloc | calloc | realloc | free |
350
351        | OutPtrOrig | NULL   | NULL    | PTR1     | PTR1   |
352        | OutPtrTarget | PTR2   | PTR2     | PTR2     | NULL   |
353        | OutSize (PTR2) | >0     | >0       | >0        | =0     |
354        | AcumFunctionSize |       |         | -PTR1.size | -PTR1.size |
355        |           | +PTR2.size | +PTR2.size | +PTR2.size |         |
356
357        Nota: PTR1.size jah deverah ter sido inserido anteriormente na LIST
358        (como PTR2), devendo ser procurado para uso de seu functionSize.
359     */
360
361     // Abrir arquivo de resultados
362     fileHProfOut_Open(fileNameOut);
363
364     // Le registro de resultados para memoria alocada dinamicamente
365     rewind(fileHProfOut);
366     ptrHProfOut = &regHProfOut;
367     fileHProfOut_Read(ptrHProfOut);
368
369     // Loop de leitura dos resultados e processamento
370     ptrListFirst = NULL;
371     ptrListLast  = NULL;

```

```

372     ptrAcumFirst      = NULL;
373     ptrAcumLast       = NULL;
374     acumGlobalSize    = 0;
375     acumMaxGlobalSize = 0;
376     while (!feof(fileHProfOut)) {
377
378         sizePtrOrig    = 0;
379         sizePtrTarget  = 0;
380
381         /* ----- */
382         /* Atualiza LIST a partir dos resultados (Out) */
383         /* . ptrHProfOut->HProfOutFunctionName */
384         /* . ptrHProfOut->HProfOutPtrTarget */
385         /* ----- */
386
387         /* Se PTR2 nao nulo, procura functionName/PTR2 nao processado em LIST
388          * para acrescentar memoria alocada (malloc, calloc, realloc)
389          */
390         ptrHProfList = ptrListFirst;
391         while (ptrHProfList != NULL &&
392               ptrHProfOut->HProfOutPtrTarget != NULL &&
393               ((strcmp(ptrHProfList->HProfListFunctionName, ptrHProfOut->HProfOutFunctionName) != 0) ||
394                ptrHProfList->HProfListPtr != ptrHProfOut->HProfOutPtrTarget ||
395                ptrHProfList->HProfListProcessed != '0')) {
396             ptrHProfList = ptrHProfList->HProfListNext;
397         }
398         if (ptrHProfOut->HProfOutPtrTarget != NULL) {
399             // Se PTR2 nao nulo, atualiza LIST
400             if (ptrHProfList == NULL) {
401                 // Se nao encontrou functionName/PTR2 em LIST, cria-a
402                 ptrHProfList = malloc(regListSize);
403                 strcpy(ptrHProfList->HProfListFunctionName, ptrHProfOut->HProfOutFunctionName);
404                 ptrHProfList->HProfListPtr = ptrHProfOut->HProfOutPtrTarget;
405                 ptrHProfList->HProfListSize = 0;
406                 ptrHProfList->HProfListProcessed = '0';
407                 ptrHProfList->HProfListPrevious = ptrListLast;
408                 ptrHProfList->HProfListNext = NULL;
409                 if (ptrListLast != NULL) {
410                     ptrListLast->HProfListNext = ptrHProfList;
411                 }
412                 if (ptrListFirst == NULL) {
413                     ptrListFirst = ptrHProfList;
414                 }
415                 ptrListLast = ptrHProfList;
416             }
417             // Atribui Out a LIST
418             sizePtrTarget = ptrHProfOut->HProfOutSize;
419             ptrHProfList->HProfListSize += ptrHProfOut->HProfOutSize;
420         }
421
422         // Atualiza HProfAcum para HProfListFunctionName/PTR2
423         if (sizePtrTarget != 0) {
424             atualizaAcum(sizePtrTarget);
425         }
426
427         /* Se PTR1 nao nulo, procura PTR1 nao processado em LIST
428          * para abater memoria liberada (free/realloc)
429          */
430         int c=0;
431         ptrHProfList = ptrListFirst;
432         while (ptrHProfList != NULL &&
433               ptrHProfOut->HProfOutPtrOrig != NULL &&
434               (ptrHProfList->HProfListPtr != ptrHProfOut->HProfOutPtrOrig ||
435                ptrHProfList->HProfListProcessed != '0')) {
436             ptrHProfList = ptrHProfList->HProfListNext;
437         }
438         if (ptrHProfOut->HProfOutPtrOrig != NULL) {

```

```

439
440     // Se PTR1 nao nulo, atualiza LIST
441     if (ptrHProfList == NULL) {
442         // Se nao encontrar, erro
443         printf("Erro: ponteiro origem %p da funcao %s nao encontrado!\n", ptrHProfOut->HProfOutPtrOrig, p
444     } else {
445         // Se nao erro, armazena ListSizePTR1 e marca como processado
446         sizePtrOrig = ptrHProfList->HProfListSize;
447         ptrHProfList->HProfListProcessed = '1';
448     }
449 }
450
451 // Atualiza HProfAcum para HProfListFunctionName/PTR1
452 if (sizePtrOrig != 0) {
453     atualizaAcum(sizePtrOrig * -1);
454 }
455
456 // Le proximo registro de resultados para memoria alocada dinamicamente
457 fileHProfOut_Read(ptrHProfOut);
458 }
459
460 // Ordenar tabela de acumulos por maior acumulo geral/maior acumulo funcao/nome funcao
461 ptrHProfAcum = ptrAcumFirst;
462 while (ptrHProfAcum != NULL) {
463     ptrHProfAcum = ptrHProfAcum->HProfAcumNext;
464 }
465 if (ptrAcumFirst != NULL && ptrAcumLast != NULL) {
466     quickSort(ptrAcumFirst, ptrAcumLast);
467 }
468
469 // Imprimir relatorio (Report Heap Profile - RHProf)
470 int order = 0;
471 printf("\n");
472 printf("-----\n");
473 printf("|Ord| Function Name | Max Function |%(Fnc/Glb)| Global Accum|\n");
474 printf("-----\n");
475 ptrHProfAcum = ptrAcumLast;
476 while (ptrHProfAcum != NULL) {
477     order++;
478     printf("|%3i", order);
479     printf(" | ");
480     printf("%-32s", ptrHProfAcum->HProfAcumKey.HProfAcumFunctionName);
481     printf(" | ");
482     printf("%10s", ptrHProfAcum->HProfAcumKey.HProfAcumMaxFunctionSize);
483     printf(" | ");
484     printf("%7.3f", (float) (atol(ptrHProfAcum->HProfAcumKey.HProfAcumMaxFunctionSize) * 100.00 / acumMaxGlo
485     printf(" | ");
486     printf("%11s |\n", ptrHProfAcum->HProfAcumKey.HProfAcumMaxGlobalSize);
487     ptrHProfAcum = ptrHProfAcum->HProfAcumPrevious;
488 }
489 printf("-----\n");
490 printf("Remarks: Results in bytes.\n");
491 printf(" (Fnc/Glb) = bytes Function / bytes greatest Global Accum.\n");
492 printf("\n");
493
494 // Fechar arquivo de resultados
495 fileHProfOut_Close();
496
497 return (EXIT_SUCCESS);
498 }

```

Benchmarks

F.1 Códigos fonte dos programas usados como Benchmark

```

1  /* ===== */
2  /*          GREATEST ELEMENT          */
3  /* ===== */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <time.h>
9
10 int main () {
11     int i, indMaior, maior, N;
12     int *ptrA;
13     printf("Informe o tamanho do vetor randomico: ");
14     scanf("%d", &N);
15     ptrA = malloc(N * sizeof(int));
16
17     srand(time(NULL));
18     printf("\nINPUT:\n");
19     for (i = 0; i < N; i++) {
20         ptrA[i] = rand() % 1000;
21         printf("a[%d] = %4d\n", i, ptrA[i]);
22     }
23     indMaior = 2;
24     maior = ptrA[indMaior];
25     for(i = 0; i < N; i++){
26         if(ptrA[i] > maior) {
27             maior = ptrA[i];
28             indMaior = i;
29         }
30     }
31     printf("\nMaior elemento: [%d] %d\n", indMaior, maior);
32     return 0;
33 }

```

```
1 /* ===== */
2 /*          FATORIAL          */
3 /* ===== */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <time.h>
9
10 int main()
11 {
12     int i;
13     int N;
14
15     int *ptrM;
16     printf("Informe o numero limite para os quais deseja calcular seus fatoriais: ");
17     scanf("%d", &N);
18     ptrM = malloc((N+1) * sizeof(int));
19
20     for (i = 0; i <= N; i++) {
21         if (i == 0) {
22             ptrM[0] = 1;
23         } else {
24             ptrM[i] = ptrM[i-1] * i;
25         }
26         printf("O fatorial de %d e: %d\n", i, ptrM[i]);
27     }
28
29     return 0;
30 }
```

```
1  /* ===== */
2  /*          SEQUENTIAL SEARCH          */
3  /* ===== */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <time.h>
9
10 int procura(int vector[], int tamanho, char elementoProcurado) {
11     int i, diff, diffMin, foundInd;
12     foundInd = 0;
13     diffMin = abs(vector[0] - elementoProcurado);
14     for (i = 1; i < tamanho; i++) {
15         diff = abs(vector[i] - elementoProcurado);
16         if (diff < diffMin) {
17             diffMin = diff;
18             foundInd = i;
19         }
20     }
21     printf("Valor %d encontrado em %d:\n", vector[foundInd], foundInd);
22 }
23
24 int main() {
25
26     int i, vectorSize, nearNumber;
27     int *ptrA;
28
29     printf("Informe o tamanho do vetor randomico: ");
30     scanf("%d", &vectorSize);
31     printf("Informe o numero aproximado (0 a 999) que deseja encontrar no vetor randomico: ");
32     scanf("%d", &nearNumber);
33     ptrA = malloc(vectorSize * sizeof(int));
34
35     srand(time(NULL));
36     printf("\nVetor:\n");
37     for (i = 0; i < vectorSize; i++) {
38         ptrA[i] = rand() % 1000;
39         printf("a[%d] = %4d\n", i, ptrA[i]);
40     }
41     printf("\n");
42
43     procura(ptrA, vectorSize, nearNumber);
44
45     return 0;
46 }
```

```

1  /* ===== */
2  /*          BINARY SEARCH          */
3  /* ===== */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <time.h>
9
10 /* ===== */
11 /*    Quick Sort    */
12 /* ===== */
13
14 void swap(int* ptrA , int* ptrB) {
15     int tmp;
16     tmp = *ptrA;
17     *ptrA = *ptrB;
18     *ptrB = tmp;
19 }
20
21 }
22
23 /* -----
24          Partition
25 ----- */
26 int* partition(int vec[], int* left , int* right) {
27     int *i , *j;
28
29     i = left;
30     for (j = left + 1; j <= right; ++j) {
31         //if (vec[j] < vec[left]) {
32             if (*j < *left) {
33                 ++i;
34                 //swap(&vec[i] , &vec[j]);
35                 swap(i , j);
36             }
37         }
38     }
39     //swap(&vec[left] , &vec[i]);
40     swap(left , i);
41
42     return i;
43 }
44
45 /* -----
46          Quick Sort Main
47 ----- */
48 void quickSort(int vec[], int* left , int* right) {
49
50     int *r;
51
52     if (right > left) {
53         r = partition(vec , left , right);
54         if (r > left)
55             quickSort(vec , left , r - 1);
56         if (r < right)
57             quickSort(vec , r + 1 , right);
58     }
59 }
60
61 /* ===== */
62 /*          Main          */
63 /* ===== */
64
65 int main() {
66
67     int i , vectorSize , nearNumber;

```

```
68     int *ptrA;
69
70     printf("Informe o tamanho do vetor randomico: ");
71     scanf("%d", &vectorSize);
72     printf("Informe o numero aproximado (0 a 999) que deseja encontrar no vetor randomico: ");
73     scanf("%d", &nearNumber);
74     ptrA = malloc(vectorSize * sizeof(int));
75
76     srand(time(NULL));
77     printf("\nVetor:\n");
78     for (i = 0; i < vectorSize; i++) {
79         ptrA[i] = rand() % 1000;
80         printf("a[%d] = %4d\n", i, ptrA[i]);
81     }
82     printf("\n");
83
84     quickSort(ptrA, &ptrA[0], &ptrA[(vectorSize - 1)]);
85     for (i = 0; i < vectorSize; i++) {
86         printf("a[%d] = %d\n", i, ptrA[i]);
87     }
88
89
90     int first, last, middle;
91
92     first = 0;
93     last = vectorSize - 1;
94     middle = (first+last)/2;
95
96     while( first <= last )
97     {
98         if (ptrA[middle] < nearNumber)
99             first = middle + 1;
100        else if (ptrA[middle] == nearNumber)
101        {
102            printf("%d found at location %d.\n", nearNumber, middle+1);
103            break;
104        }
105        else
106            last = middle - 1;
107
108        middle = (first + last)/2;
109    }
110    if (first > last && ptrA[middle] != nearNumber)
111        //printf("Not found! %d is not present in the list.\n", nearNumber);
112        printf("Number %d would be in between %d and %d.\n", nearNumber, last, first);
113
114    return 0;
115 }
```

```

1  /* ===== */
2  /*          BUBBLE SORT          */
3  /* ===== */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <time.h>
9
10 #define true 1
11 #define false 0
12
13 inline void troca(int* a, int* b)
14 {
15     int aux = *a;
16     *a = *b;
17     *b = aux;
18 }
19
20 void bubbleSort (int *primeiro, int *ultimo)
21 {
22     int naoTrocou;
23     int *posAtual;
24     for (; ultimo > primeiro; —ultimo)
25     {
26         naoTrocou = true;
27         for (posAtual = primeiro; posAtual < ultimo; ++posAtual)
28         {
29             if (*posAtual > *(posAtual+1))
30             {
31                 troca (posAtual, posAtual+1);
32                 naoTrocou = false;
33             }
34         }
35         if (naoTrocou) return;
36     }
37 }
38
39 int main(){
40     int i, vectorSize;
41     int *ptrA;
42
43     printf("Informe o tamanho do vetor randomico: ");
44     scanf("%d", &vectorSize);
45     ptrA = calloc(vectorSize, sizeof(int));
46
47     srand(time(NULL));
48     printf("\nVetor nao ordenado:\n");
49     for (i = 0; i < vectorSize; i++) {
50         ptrA[i] = rand() % 1000;
51         printf("a[%d] = %4d\n", i, ptrA[i]);
52     }
53     printf("\n");
54
55     bubbleSort(&ptrA[0], &ptrA[vectorSize - 1]);
56
57     printf("\nVetor ordenado:\n");
58     for (i = 0; i < vectorSize; i++) {
59         printf("a[%d] = %d\n", i, ptrA[i]);
60     }
61
62     printf("\n");
63     return 0;
64 }

```

```
1  /* ===== */
2  /*          INSERTION SORT          */
3  /* ===== */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <time.h>
9
10 void insertionSort(int V[], int tam)
11 {
12     int i, j, aux;
13
14     for(i = 1; i < tam; i++){
15         j = i;
16
17         while(V[j] < V[j - 1]) {
18
19             aux = V[j];
20             V[j] = V[j - 1];
21             V[j - 1] = aux;
22             j--;
23
24             if(j == 0)break;
25         }
26     }
27 }
28
29 int main(){
30     int i, vectorSize;
31     int *ptrA;
32
33     printf("Informe o tamanho do vetor randomico: ");
34     scanf("%d", &vectorSize);
35     ptrA = calloc(vectorSize, sizeof(int));
36
37     srand(time(NULL));
38     printf("\nVetor nao ordenado:\n");
39     for (i = 0; i < vectorSize; i++) {
40         ptrA[i] = rand() % 1000;
41         printf("a[%d] = %4d\n", i, ptrA[i]);
42     }
43     printf("\n");
44
45     insertionSort(ptrA, vectorSize);
46
47     printf("\nVetor ordenado:\n");
48     for (i = 0; i < vectorSize; i++) {
49         printf("a[%d] = %d\n", i, ptrA[i]);
50     }
51
52     printf("\n");
53     return 0;
54 }
```

```
1 /* ===== */
2 /*                SELECTION SORT                */
3 /* ===== */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <time.h>
9
10 void selection_sort(int num[], int tam)
11 {
12     int i, j, min;
13     for (i = 0; i < (tam-1); i++)
14     {
15         min = i;
16         for (j = (i+1); j < tam; j++) {
17             if (num[j] < num[min]) {
18                 min = j;
19             }
20         }
21         if (i != min) {
22             int swap = num[i];
23             num[i] = num[min];
24             num[min] = swap;
25         }
26     }
27 }
28
29 int main() {
30     int i, vectorSize;
31     int *ptrA;
32
33     printf("Informe o tamanho do vetor randomico: ");
34     scanf("%d", &vectorSize);
35     ptrA = calloc(vectorSize, sizeof(int));
36
37     srand(time(NULL));
38     printf("\nVetor nao ordenado:\n");
39     for (i = 0; i < vectorSize; i++) {
40         ptrA[i] = rand() % 1000;
41         printf("a[%d] = %4d\n", i, ptrA[i]);
42     }
43     printf("\n");
44
45     selection_sort(ptrA, vectorSize);
46
47     printf("\nVetor ordenado:\n");
48     for (i = 0; i < vectorSize; i++) {
49         printf("a[%d] = %d\n", i, ptrA[i]);
50     }
51
52     printf("\n");
53     return 0;
54 }
```

```

1  /* ===== */
2  /*          MERGE SORT          */
3  /* ===== */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <time.h>
9
10 void merge(int vec[], int vecSize) {
11     int mid;
12     int i, j, k;
13     int* tmp;
14
15     tmp = (int*) malloc(vecSize * sizeof(int));
16     if (tmp == NULL) {
17         exit(1);
18     }
19
20     mid = vecSize / 2;
21
22     i = 0;
23     j = mid;
24     k = 0;
25     while (i < mid && j < vecSize) {
26         if (vec[i] < vec[j]) {
27             tmp[k] = vec[i];
28             ++i;
29         }
30         else {
31             tmp[k] = vec[j];
32             ++j;
33         }
34         ++k;
35     }
36
37     if (i == mid) {
38         while (j < vecSize) {
39             tmp[k] = vec[j];
40             ++j;
41             ++k;
42         }
43     }
44     else {
45         while (i < mid) {
46             tmp[k] = vec[i];
47             ++i;
48             ++k;
49         }
50     }
51
52     for (i = 0; i < vecSize; ++i) {
53         vec[i] = tmp[i];
54     }
55
56     free(tmp);
57 }
58
59 void mergeSort(int vec[], int vecSize) {
60     int mid;
61
62     if (vecSize > 1) {
63         mid = vecSize / 2;
64         mergeSort(vec, mid);
65         mergeSort(vec + mid, vecSize - mid);
66         merge(vec, vecSize);
67     }

```

```
68 }
69
70 int main() {
71     int i, vectorSize;
72     int *ptrA;
73
74     printf("Informe o tamanho do vetor randomico: ");
75     scanf("%d", &vectorSize);
76     ptrA = calloc(vectorSize, sizeof(int));
77
78     srand(time(NULL));
79     printf("\nVetor nao ordenado:\n");
80     for (i = 0; i < vectorSize; i++) {
81         ptrA[i] = rand() % 1000;
82         printf("a[%d] = %4d\n", i, ptrA[i]);
83     }
84     printf("\n");
85
86     mergeSort(ptrA, vectorSize);
87
88     printf("\nVetor ordenado:\n");
89     for (i = 0; i < vectorSize; i++) {
90         printf("a[%d] = %d\n", i, ptrA[i]);
91     }
92
93     printf("\n");
94     return 0;
95 }
```

```

1  /* ===== */
2  /*          QUICK SORT          */
3  /* ===== */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <time.h>
9
10 void swap(int* a, int* b) {
11     int tmp;
12     tmp = *a;
13     *a = *b;
14     *b = tmp;
15 }
16
17 int* partition(int vec[], int* left, int* right) {
18     int *i, *j;
19
20     i = left;
21     for (j = left + 1; j <= right; ++j) {
22         if (*j < *left) {
23             ++i;
24             swap(i, j);
25         }
26     }
27     swap(left, i);
28
29     return i;
30 }
31
32 void quickSort(int vec[], int* left, int* right) {
33
34     int *r;
35
36     if (right > left) {
37         r = partition(vec, left, right);
38         if (r > left)
39             quickSort(vec, left, r - 1);
40         if (r < right)
41             quickSort(vec, r + 1, right);
42     }
43 }
44
45 int main(){
46     int i, vectorSize;
47     int *ptrA;
48     printf("Informe o tamanho do vetor randomico: ");
49     scanf("%d", &vectorSize);
50     ptrA = malloc(vectorSize * sizeof(int));
51
52     srand(time(NULL));
53     printf("\nVetor nao ordenado:\n");
54     for (i = 0; i < vectorSize; i++) {
55         ptrA[i] = rand() % 1000;
56         printf("a[%d] = %4d\n", i, ptrA[i]);
57     }
58     printf("\n");
59
60     quickSort(ptrA, &ptrA[0], &ptrA[(vectorSize - 1)]);
61     printf("\nVetor ordenado:\n");
62     for (i = 0; i < vectorSize; i++) {
63         printf("a[%d] = %d\n", i, ptrA[i]);
64     }
65     return 0;
66 }

```

```

1  /* ===== */
2  /*          DETERMINANTE          */
3  /* ===== */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <math.h>
9
10 float MAT[10][10] =
11     {{ 1.0, 0.0, -1.0, 3.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0},
12      { 2.0, 3.0, 4.0, 2.0, 4.0, 2.0, 3.0, 2.0, 1.0, 5.0},
13      { 0.0, 2.0, 5.0, 1.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0},
14      { 4.0, 1.0, 0.0, 0.0, 4.0, 3.0, 1.0, 2.0, 1.0, 1.0},
15      {-4.0, 0.0, 3.0, 1.0, 2.0, 3.0, 1.0, 3.0, 2.0, 1.0},
16      { 1.0, 3.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0},
17      { 2.0, 1.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0},
18      { 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0},
19      { 3.0, 3.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0},
20      { 1.0, 4.0, 1.0, 1.0, 1.0, 4.0, 4.0, 1.0, 1.0, 4.0}};
21
22 // http://forum.imasters.com.br/topic/311774-codigo-calcular-determinante/
23
24 // Prototypes
25 float det(float**, int);
26 float** matrizMenor(float**, float**, int, int);
27 float** alocarMQ(int);
28 float** freeMQ(float**, int);
29 void printMQ(float**, int);
30
31 int main(int argc, char *argv[]){
32
33     int ordem, i, j;
34     float **matriz;
35
36     //Recuperando a ordem da matriz
37     printf("** Calculando o determinante de matrizes reais. **\n\n");
38     ordem = 4;
39     printf("Digite a ordem desejada da matriz: %d", ordem);
40     //scanf("%d", &ordem);
41
42     //Alocando espaço da matriz quadrada
43     matriz = alocarMQ(ordem);
44
45     //Preenchendo a matriz
46     printf("\nDigite os valores da matriz (da direita para a esquerda,\n
47 de cima para baixo):\n");
48     for(i=0; i<ordem; i++)
49         for(j=0; j<ordem; j++)
50             //scanf("%d", &matriz[i][j]);
51             matriz[i][j] = MAT[i][j];
52
53     //Imprimindo a matriz e seu determinante
54     printf("\nA matriz eh:\n\n");
55     printMQ(matriz, ordem);
56     printf("\nO Determinante eh: %f\n", det(matriz, ordem));
57
58     getchar();
59     getchar();
60     return 0;
61 }
62
63 //Retorna o determinante da matriz m de ordem n por Laplace
64 float det(float **m, int n){
65     //Caso fundamental
66     if(n == 1)
67         return **m;

```

```

68
69      //Criando a matriz do cofator
70      int c;
71      float d=0, cof, **m2 = alocarMQ(n-1);
72
73      //Para cada coluna
74      for(c=0; c<n; c++){
75          //Calculando o cofator
76          cof = (c%2?-1:1) * det(matrizMenor(m, m2, c, n), n-1);
77          //Somando a parcela ao determinante final
78          d += m[0][c]*cof;
79      }
80
81      //Liberando o espaço alocado da matriz do cofator
82      freeMQ(m2, n-1);
83
84      //Retornando o determinante
85      return d;
86  }
87
88  //Retira a primeira linha a coluna c de m e armazena em m2. n Ã a ordem de m
89  float **matrizMenor(float **m, float **m2, int c, int n){
90      int i, j;
91      float **p=m2, *p2;
92      for(i = 1; i < n; i++){
93          p2 = *p++;
94          for(j = 0; j < n; j++){
95              if(j==c) continue;
96              else *p2++ = m[i][j];
97          }
98      }
99      return m2;
100  }
101  //Aloca espaço para uma matriz quadrada
102  float **alocarMQ(int n){
103      float **m;
104      int i;
105      m = (float**) calloc(n, sizeof(float*));
106      if(m == NULL) exit(1);
107      for(i=0; i<n; i++){
108          *(m+i) = (float*) calloc(n, sizeof(float));
109          if(*(m+i) == NULL) exit(1);
110      }
111      return m;
112  }
113
114  //Desaloca espaço de uma matriz quadrada
115  float **freeMQ(float **m, int n){
116      int i;
117      if(m == NULL) return NULL;
118      for(i=0; i<n; i++) free(m[i]);
119      free(m);
120      return NULL;
121  }
122
123  //Imprime uma matriz quadrada
124  void printMQ(float **m, int n){
125      int i, j;
126      for(i = 0; i < n; i++){
127          printf("[");
128          for(j = 0; j < n; j++){
129              printf(" %5.2f", m[i][j]);
130          }
131          printf(" ]\n");
132      }

```

```

1  /* ===== */
2  /*          MULTIPLICACAO DE MATRIZES          */
3  /* ===== */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8
9  float MAT1[10][10] =
10     {{ 1.0, 0.0, -1.0, 3.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0},
11      { 2.0, 3.0, 4.0, 2.0, 4.0, 2.0, 3.0, 2.0, 1.0, 5.0},
12      { 0.0, 2.0, 5.0, 1.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0},
13      { 4.0, 1.0, 0.0, 0.0, 4.0, 3.0, 1.0, 2.0, 1.0, 1.0},
14      {-4.0, 0.0, 3.0, 1.0, 2.0, 3.0, 1.0, 3.0, 2.0, 1.0},
15      { 1.0, 3.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0},
16      { 2.0, 1.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0},
17      { 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0},
18      { 3.0, 3.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0},
19      { 1.0, 4.0, 1.0, 1.0, 1.0, 4.0, 4.0, 1.0, 1.0, 4.0}};
20
21 float MAT2[10][10] =
22     {{ 2.0, 0.0, -1.0, 3.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0},
23      { 0.0, 1.0, 4.0, 1.0, 4.0, 2.0, 3.0, 2.0, 1.0, 5.0},
24      { 0.0, -2.0, 5.0, 2.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0},
25      { 4.0, 1.0, 3.0, 2.0, 4.0, 3.0, 1.0, 2.0, 1.0, 1.0},
26      { 2.0, 3.0, 0.0, -1.0, 2.0, 3.0, 1.0, 3.0, 2.0, 1.0},
27      { 1.0, 3.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0},
28      { 2.0, 1.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0},
29      { 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0},
30      { 3.0, 3.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0},
31      { 1.0, 4.0, 1.0, 1.0, 1.0, 4.0, 4.0, 1.0, 1.0, 4.0}};
32
33 //Aloca espaco para uma matriz quadrada
34 float **alocarMQ(int n){
35     float **m;
36     int i;
37     m = (float**) calloc(n, sizeof(float*));
38     if(m == NULL) exit(1);
39     for(i=0; i<n; i++){
40         *(m+i) = (float*) calloc(n, sizeof(float));
41         if(*(m+i) == NULL) exit(1);
42     }
43     return m;
44 }
45
46 //Desaloca espaco de uma matriz quadrada
47 float **freeMQ(float **m, int n){
48     int i;
49     if(m == NULL) return NULL;
50     for(i=0; i<n; i++) free(m[i]);
51     free(m);
52     return NULL;
53 }
54
55 //Imprime uma matriz quadrada
56 void printMQ(float **m, int n){
57     int i, j;
58     for(i = 0; i < n; i++){
59         printf("[");
60         for(j = 0; j < n; j++)
61             printf(" %5.2f", m[i][j]);
62         printf(" ]\n");
63     }
64 }
65
66 void multiplica_mat(float **A, float **B, float **C, int n) {
67     int i, j, k;

```

```
68
69     for (i = 0; i<n; i++)
70         for (j=0; j<n; j++)
71             {
72                 C[i][j]=0;
73                 for (k=0; k<n; k++)
74                     C[i][j] += A[i][k] * B[k][j];
75             }
76 }
77
78 int main() {
79
80     int ordem, i, j;
81     float **matriz1,
82           **matriz2,
83           **matriz3;
84
85     //Recuperando a ordem da matriz
86     printf("** Calculando o produto de matrizes reais. **\n\n");
87     ordem = 4;
88     printf("Digite a ordem desejada da matriz: %d", ordem);
89     //scanf("%d", &ordem);
90
91     //Alocando espaco para as matrizes quadradas
92     matriz1 = alocarMQ(ordem);
93     matriz2 = alocarMQ(ordem);
94     matriz3 = alocarMQ(ordem);
95
96     //Preenchendo a matriz
97     printf("\nDigite os valores das matrizes (da direita para a esquerda,\n
98 de cima para baixo):\n");
99     for(i=0; i<ordem; i++) {
100         for(j=0; j<ordem; j++) {
101             //scanf("%d", &matriz[i][j]);
102             matriz1[i][j] = MAT1[i][j];
103             matriz2[i][j] = MAT2[i][j];
104         }
105     }
106
107     //Imprimindo as matrizes
108     printf("\nA matriz1 eh:\n\n");
109     printMQ(matriz1, ordem);
110     printf("\nA matriz2 eh:\n\n");
111     printMQ(matriz2, ordem);
112     multiplica_mat(matriz1, matriz2, matriz3, ordem);
113     printf("\nA matriz3 = matriz1 x matriz2 eh:\n\n");
114     printMQ(matriz3, ordem);
115
116     //Liberando as matrizes
117     freeMQ(matriz1, ordem);
118     freeMQ(matriz2, ordem);
119     freeMQ(matriz3, ordem);
120
121     getchar();
122     getchar();
123     return 0;
124 }
```

```

1  /* ===== */
2  /*          MATRIZ TRANSPOSTA          */
3  /* ===== */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8
9  float MAT[10][10] =
10     {{ 1.0, 0.0, -1.0, 3.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0},
11      { 2.0, 3.0, 4.0, 2.0, 4.0, 2.0, 3.0, 2.0, 1.0, 5.0},
12      { 0.0, 2.0, 5.0, 1.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0},
13      { 4.0, 1.0, 0.0, 0.0, 4.0, 3.0, 1.0, 2.0, 1.0, 1.0},
14      {-4.0, 0.0, 3.0, 1.0, 2.0, 3.0, 1.0, 3.0, 2.0, 1.0},
15      { 1.0, 3.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0},
16      { 2.0, 1.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0},
17      { 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0},
18      { 3.0, 3.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0},
19      { 1.0, 4.0, 1.0, 1.0, 1.0, 4.0, 4.0, 1.0, 1.0, 4.0}};
20
21 //Aloca espaço para uma matriz quadrada
22 float **alocarMQ(int n){
23     float **m;
24     int i;
25     m = (float**) calloc(n, sizeof(float*));
26     if(m == NULL) exit(1);
27     for(i=0; i<n; i++){
28         *(m+i) = (float*) calloc(n, sizeof(float));
29         if(*(m+i) == NULL) exit(1);
30     }
31     return m;
32 }
33
34 //Desaloca espaço de uma matriz quadrada
35 float **freeMQ(float **m, int n){
36     int i;
37     if(m == NULL) return NULL;
38     for(i=0; i<n; i++) free(m[i]);
39     free(m);
40     return NULL;
41 }
42
43 //Imprime uma matriz quadrada
44 void printMQ(float **m, int n){
45     int i, j;
46     for(i = 0; i < n; i++){
47         printf("[");
48         for(j = 0; j < n; j++)
49             printf(" %5.2f", m[i][j]);
50         printf(" ]\n");
51     }
52 }
53
54 void transposta(float **A, float **B, int n) {
55     int i, j;
56
57     for (i = 0; i<n; i++)
58         for (j=0; j<n; j++)
59             {
60                 B[i][j] = A[j][i];
61             }
62 }
63
64 int main() {
65
66     int ordem, i, j;
67     float **matriz1,

```

```
68         **matriz2;
69
70         //Recuperando a ordem da matriz
71         printf("** Transpondo uma matriz real. **\n\n");
72         ordem = 4;
73         printf("Digite a ordem desejada da matriz: %d", ordem);
74         //scanf("%d", &ordem);
75
76         //Alocando espaço para as matrizes quadradas
77         matriz1 = alocarMQ(ordem);
78         matriz2 = alocarMQ(ordem);
79
80         //Preenchendo a matriz
81         printf("\nDigite os valores das matrizes (da direita para a esquerda,\n
82 de cima para baixo):\n");
83         for(i=0; i<ordem; i++) {
84             for(j=0; j<ordem; j++) {
85                 //scanf("%d", &matriz[i][j]);
86                 matriz1[i][j] = MAT[i][j];
87             }
88         }
89
90         //Imprimindo as matrizes
91         printf("\nA matriz eh:\n\n");
92         printMQ(matriz1, ordem);
93         // Calcula matriz transposta
94         transposta(matriz1, matriz2, ordem);
95         printf("\nA matriz transposta = eh:\n\n");
96         printMQ(matriz2, ordem);
97
98         //Liberando as matrizes
99         freeMQ(matriz1, ordem);
100        freeMQ(matriz2, ordem);
101
102        getchar();
103        getchar();
104        return 0;
105 }
```

```
1 /* ===== */
2 /*          FIBONACCI          */
3 /* ===== */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8
9 int main()
10 {
11     int i;
12     int N;
13
14     int *ptrM;
15     printf("Informe o numero limite para os quais deseja calcular Fibonacci: ");
16     scanf("%d", &N);
17     ptrM = malloc((N+1) * sizeof(int));
18
19     for (i = 0; i <= N; i++) {
20         if (i == 0 || i == 1) {
21             ptrM[i] = i;
22         } else {
23             ptrM[i] = ptrM[i-1] + ptrM[i-2];
24         }
25         printf("Fibonacci em %d eh: %d\n", i, ptrM[i]);
26     }
27
28     return 0;
29 }
```

```

1  /* ===== */
2  /*          GAUSS-JORDAN          */
3  /* ===== */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <math.h>
9
10 float MAT1[10][10] =
11     {{ 1.0, 0.0, -1.0, 3.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0},
12      { 2.0, 3.0, 4.0, 2.0, 4.0, 2.0, 3.0, 2.0, 1.0, 5.0},
13      { 0.0, 2.0, 5.0, 1.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0},
14      { 4.0, 1.0, 0.0, 0.0, 4.0, 3.0, 1.0, 2.0, 1.0, 1.0},
15      {-4.0, 0.0, 3.0, 1.0, 2.0, 3.0, 1.0, 3.0, 2.0, 1.0},
16      { 1.0, 3.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0},
17      { 2.0, 1.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0},
18      { 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0},
19      { 3.0, 3.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0},
20      { 1.0, 4.0, 1.0, 1.0, 1.0, 4.0, 4.0, 1.0, 1.0, 4.0}};
21
22 float MAT2[10][10] =
23     {{ 2.0, 0.0, -1.0, 3.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0},
24      { 0.0, 1.0, 4.0, 1.0, 4.0, 2.0, 3.0, 2.0, 1.0, 5.0},
25      { 0.0, -2.0, 5.0, 2.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0},
26      { 4.0, 1.0, 3.0, 2.0, 4.0, 3.0, 1.0, 2.0, 1.0, 1.0},
27      { 2.0, 3.0, 0.0, -1.0, 2.0, 3.0, 1.0, 3.0, 2.0, 1.0},
28      { 1.0, 3.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0},
29      { 2.0, 1.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0},
30      { 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0},
31      { 3.0, 3.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0},
32      { 1.0, 4.0, 1.0, 1.0, 1.0, 4.0, 4.0, 1.0, 1.0, 4.0}};
33
34 float **alocarMQ(int n);
35 float **freeMQ(float **A, int n);
36 void printMQ(float **A, int n);
37 void multiplica_mat(float **A, float **B, float **C, int n);
38 void transposta(float **A, float **B, int n);
39 void matrizcomplementar(float **A, float **B, int n);
40 void submatriz(float **A, float **B, int n, int ci, int cj);
41 float **matrizMenor(float **A, float **A2, int c, int n);
42 float determinante(float **A, int n);
43
44 //Aloca espaço para uma matriz quadrada
45 float **alocarMQ(int n){
46     float **A;
47     int i;
48     A = (float**) calloc(n, sizeof(float*));
49     if(A == NULL) exit(1);
50     for(i=0; i<n; i++){
51         *(A+i) = (float*) calloc(n, sizeof(float));
52         if(*(A+i) == NULL) exit(1);
53     }
54     return A;
55 }
56
57 //Desaloca espaço de uma matriz quadrada
58 float **freeMQ(float **A, int n){
59     int i;
60     if(A == NULL) return NULL;
61     for(i=0; i<n; i++) free(A[i]);
62     free(A);
63     return NULL;
64 }
65
66 //Imprime uma matriz quadrada
67 void printMQ(float **A, int n){

```

```

68     int i, j;
69     for(i = 0; i < n; i++){
70         printf("[");
71         for(j = 0; j < n; j++)
72             printf(" %5.2f", A[i][j]);
73         printf(" ]\n");
74     }
75 }
76
77 void multiplica_mat(float **A, float **B, float **C, int n) {
78     int i, j, k;
79
80     for (i = 0; i < n; i++)
81         for (j = 0; j < n; j++)
82         {
83             C[i][j] = 0;
84             for (k = 0; k < n; k++)
85                 C[i][j] += A[i][k] * B[k][j];
86         }
87 }
88
89 void transposta(float **A, float **B, int n) {
90     int i, j;
91
92     for (i = 0; i < n; i++)
93         for (j = 0; j < n; j++)
94         {
95             B[i][j] = A[j][i];
96         }
97 }
98
99 void matrizcomplementar(float **A, float **B, int n)
100 {
101     int i, j;
102     float **C;
103     C = alocarMQ(n);
104
105     for (i = 0; i < n; i++)
106         for (j = 0; j < n; j++)
107         {
108             // Calcula o cofator:  $A_{ij} = (-1)^{(i+j)} \det(C_{ij})$ ;
109             submatriz(A, C, n, i, j);
110             B[i][j] = determinante(C, n-1);
111             if ((i+j)%2 != 0)
112                 B[i][j] *= -1;
113         }
114 }
115
116 void submatriz(float **A, float **B, int n, int ci, int cj)
117 {
118     int ai, aj, i, j;
119
120     for (i = 0; i < n; i++)
121         for (j = 0; j < n; j++)
122         {
123             if (i < ci) { ai = i; } else { ai = (i+1) % n; }
124             if (j < cj) { aj = j; } else { aj = (j+1) % n; }
125
126             B[i][j] = A[ai][aj];
127         }
128 }
129
130 //Retira a primeira linha a coluna c de m e armazena em m2. n Ã a ordem de m
131 float **matrizMenor(float **A, float **A2, int c, int n){
132     int i, j;
133     float **p = A2, *p2;
134     for(i = 1; i < n; i++){

```

```

135         p2 = *p++;
136         for(j = 0; j < n; j++)
137             if(j==c) continue;
138             else *p2++ = A[i][j];
139     }
140     return A2;
141 }
142
143 //Retorna o determinante da matriz m de ordem n por Laplace
144 float determinante(float **A, int n){
145     //Caso fundamental
146     if(n == 1)
147         return **A;
148
149     //Criando a matriz do cofator
150     int c;
151     float d=0, cof, **A2 = alocarMQ(n-1);
152
153     //Para cada coluna
154     for(c=0; c<n; c++){
155         //Calculando o cofator
156         cof = (c%2?-1:1) * determinante(matrizMenor(A, A2, c, n), n-1);
157         //Somando a parcela ao determinante final
158         d += A[0][c]*cof;
159     }
160
161     //Liberando o espaço alocado da matriz do cofator
162     freeMQ(A2, n-1);
163
164     //Retornando o determinante
165     return d;
166 }
167
168 int main() {
169
170     int ordem, i, j;
171     float **matriz1, // MAT
172           **matriz2, // INV
173           **matriz3; // AUX
174     float det;
175     float inv_det;
176
177     //Recuperando a ordem da matriz
178     printf("*** Calculando a matriz inversa por Gauss-Jordan. **\n\n");
179     ordem = 4;
180     printf("Digite a ordem desejada da matriz: %d", ordem);
181     //scanf("%d", &ordem);
182
183     //Alocando espaço para as matrizes quadradas
184     matriz1 = alocarMQ(ordem);
185     matriz2 = alocarMQ(ordem);
186     matriz3 = alocarMQ(ordem);
187
188     //Preenchendo a matriz
189     printf("\nDigite os valores das matrizes (da direita para a esquerda,\n
190 de cima para baixo):\n");
191     for(i=0; i<ordem; i++) {
192         for(j=0; j<ordem; j++) {
193             //scanf("%d", &matriz[i][j]);
194             matriz1[i][j] = MAT1[i][j];
195         }
196     }
197
198     //Imprimindo as matrizes
199     printf("\nA matriz1 eh:\n\n");
200     printMQ(matriz1, ordem);
201

```

```
202 // Calcula o determinante
203 det = determinante(matriz1, ordem);
204
205 if (det == 0)
206     printf("\n\n**** A matriz nao tem inversa (determinante = 0). ****\n");
207 else
208 {
209     // Calcula a matriz complementar (matriz de cofatores)
210     inv_det = 1/det;
211     matrizcomplementar(matriz1, matriz3, ordem);
212
213     // Calcula a matriz adjunta (transposta da matriz complementar)
214     transposta(matriz3, matriz2, ordem);
215
216     // Calcula a matriz inversa
217     for (i=0; i<ordem; i++)
218         for (j=0; j<ordem; j++)
219             matriz2[i][j] *= inv_det;
220
221     // Calcula a matriz identidade a partir das matrizes original e inversa
222     multiplica_mat(matriz1, matriz2, matriz3, ordem);
223
224     //Imprimindo as matrizes
225     printf("\nA matriz inversa eh:\n\n");
226     printMQ(matriz2, ordem);
227     printf("\nA matriz identidade eh:\n\n");
228     printMQ(matriz3, ordem);
229 }
230
231 //Liberando as matrizes
232 freeMQ(matriz1, ordem);
233 freeMQ(matriz2, ordem);
234 freeMQ(matriz3, ordem);
235
236 return 0;
237 }
```

```

1  /* ===== */
2  /*                                JACOBI SEQUENCE                                */
3  /* ===== */
4
5  #include <stdlib.h>
6  #include <stdio.h>
7
8  /* SHARED VARIABLES */
9  int N;
10 float** A;
11 float** A1;
12 float* B;
13 float* B1;
14 float* X;
15
16 // Verificar necessidade de ser global no algoritmo paralelo
17 float Mr; // valor inicial para entrar no loop. pode ser qquer valor > 0.001
18 float maxDif;
19 float maxX;
20 float* Dif;
21 float* somat;
22
23 /* INITIALIZER AND FINALIZER FUNCTIONS*/
24 void start1(int argc, char *argv[]);
25 void finish();
26
27 /* JACOBI-RICHARDSON FUNCTIONS */
28 void JRInit();
29 void JRConvergenceTest();
30 void JRExecute();
31 void JRVerification();
32
33 /* UTILITY FUNCTIONS */
34 float MAIOR(float a, float b);
35 float ABS(float x);
36 float *allocVector(int n);
37 float *freeVector(int n, float *v);
38 float **allocMatrix(int m, int n);
39 float **freeMatrix(int m, int n, float **v);
40
41 /* MAIN FUNCTION */
42 int main(int argc, char *argv[]){
43     start1(argc, argv);
44
45     JRInit();
46     JRConvergenceTest();
47     JRExecute();
48     JRVerification();
49
50     finish();
51     return 0;
52 }
53
54 // =====
55 /* FUNCTION'S IMPLEMENTATION */
56
57 void start1(int argc, char *argv[]){
58     int i,j;
59     //FILE * linearSystem;
60
61     //if(argc != 2){
62     //    printf("\njacobi <file_name.txt>\n\n");
63     //    exit(1);
64     //}
65
66     //if((linearSystem = fopen(argv[1], "r")) == NULL){
67     //    printf("Error opening file.\n");
68     //}
69 }

```



```

135
136      // COMUNICAÇÃO/SINCRONIZAÇÃO: Espera a leitura de B[i]
137      // e A[i][i] (lido anteriormente, se este passo for sequencial)
138      // para calcular B1 e X
139      B1[i] = B[i] / A[i][i];
140      X[i] = B1[i];
141      somat[i] = 0;
142  }
143
144  }
145
146  void JRExecute(){
147
148      Mr = 1; // valor inicial para entrar no loop. pode ser qquer valor > 0.001
149      int i, j;
150      int k = 1;
151
152      // Calcula x_{K+1} até atingir o erro determinado
153      // printf("Calculating...\n");
154      while(Mr > 0.001){
155          // printf("[iteration %d]\n",k);
156          maxDif = 0;
157          maxX = 0;
158
159          // calcula soma das colunas de A1 para cada linha i de 0 a N e armazena em somat[i]
160
161          // É possível dividir este loop entre as threads onde cada uma é responsável por
162          // calcular N/NUM_THREADS iterações
163          for(i = 0; i < N; i++){
164              somat[i] = 0;
165
166              // É possível dividir este loop entre as threads onde cada uma é responsável por
167              // calcular sua parcela de somat[i]
168              for(j = 0; j < N; j++){
169                  somat[i] += A1[i][j] * X[j];
170              }
171              // COMUNICAÇÃO/SINCRONIZAÇÃO: Join para calcular valor total de somat[i]
172          }
173
174
175          // COMUNICAÇÃO/SINCRONIZAÇÃO: Se este loop não for sequencial em relação ao anterior
176          // Deve-se usar uma barreira aqui para garantir que
177          // todos os somatórios foram calculados
178
179          // X[i] = B1[i] - somat(A1[i][j] * X[j]), para i <> j
180
181          // É possível dividir este loop por iterações/dados, neste caso,
182          // cada tarefa calcularia N/NUM_THREADS valores de X, Dif, maxDif e maxX
183          for(i = 0; i < N; i++){
184              float oldX = X[i]; // Guarda valor da iteração anterior
185              if(i != j) X[i] = B1[i] - somat[i];
186
187              Dif[i] = ABS(X[i] - oldX);
188              maxDif = MAIOR(maxDif, Dif[i]);
189              maxX = MAIOR(maxX, ABS(X[i]));
190          }
191
192          // COMUNICAÇÃO/SINCRONIZAÇÃO: Barreira/Join para calcular Mr
193          // e iniciar seguramente a nova iteração com X totalmente atualizado
194          // calcula Mr
195          Mr = maxDif / maxX;
196
197          k++;
198      }
199  }
200
201  void JRConvergenceTest(){

```

```

269     return (NULL);
270 }
271 return (v);    /* retorna o ponteiro para o vetor */
272 }
273
274 float *freeVector(int n, float *v){
275     if (v == NULL) return (NULL);
276     if (n < 1) { /* verifica parametros recebidos */
277         //printf ("** Erro: Parametro invalido **\n");
278         return (NULL);
279     }
280     free(v);    /* libera o vetor */
281     return (NULL); /* retorna o ponteiro */
282 }
283
284 float **allocMatrix(int m, int n){
285     float **v; /* ponteiro para a matriz */
286     int i;    /* variavel auxiliar */
287     if (m < 1 || n < 1) { /* verifica parametros recebidos */
288         //printf ("** Erro: Parametro invalido **\n");
289         return (NULL);
290     }
291     /* aloca as linhas da matriz */
292     v = (float **) calloc(m, sizeof(float *));
293     if (v == NULL) {
294         //printf ("** Erro: Memoria Insuficiente **");
295         return (NULL);
296     }
297     /* aloca as colunas da matriz */
298     for ( i = 0; i < m; i++ ) {
299         v[i] = (float*) calloc(n, sizeof(float));
300         if (v[i] == NULL) {
301             //printf ("** Erro: Memoria Insuficiente **");
302             return (NULL);
303         }
304     }
305     return (v); /* retorna o ponteiro para a matriz */
306 }
307
308 float **freeMatrix(int m, int n, float **v){
309     int i; /* variavel auxiliar */
310     if (v == NULL) return (NULL);
311     if (m < 1 || n < 1) { /* verifica parametros recebidos */
312         //printf ("** Erro: Parametro invalido **\n");
313         return (v);
314     }
315     for (i=0; i<m; i++) free(v[i]); /* libera as linhas da matriz */
316     free(v); /* libera a matriz */
317     return (NULL); /* retorna um ponteiro nulo */
318 }

```

```

1 /* ===== */
2 /*                      SOBEL                      */
3 /* ===== */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8
9 int main() {
10
11     int rows, cols, N;
12     int H, O, V, i, l=0;
13     int i00, i01, i02;
14     int i10,      i12;
15     int i20, i21, i22;
16
17     //Recuperando a ordem da matriz
18     printf("** Processando filtro de Sobel. **\n\n");
19     rows = 10;
20     cols = 10;
21     printf("Digite a definicao desejada (linha, coluna): %d, %d\n", rows, cols);
22     //scanf("%d", &rows);
23     //scanf("%d", &cols);
24     N = cols*rows;
25
26     char *in = malloc(N);
27     char *out = malloc(N);
28
29     for (i = 0; i < N; i++) {
30         in[i] = i*i;
31     }
32
33     for (i = 0; i < cols*(rows-2)-2; i++) {
34         i00=in[i      ]; i01=in[i      +1]; i02=in[i      +2];
35         i10=in[i+  cols];                i12=in[i+  cols+2];
36         i20=in[i+2*cols]; i21=in[i+2*cols+1]; i22=in[i+2*cols+2];
37         H = - i00 - 2*i01 - i02 +
38             + i20 + 2*i21 + i22;
39         V = - i00          + i02
40             - 2*i10          + 2*i12
41             - i20          + i22;
42         if (H<0)
43             H = -H;
44         if (V<0)
45             V = -V;
46         O = H + V;
47         if (O > 255) O = 255;
48         out[i + 1] = (char)O;
49         printf(" %4d", out[i + 1]);
50         if (l++>(cols+1)) {
51             printf("\n");
52             l=0;
53         }
54     }
55
56     return 0;
57 }

```

```

1  /* ===== */
2  /*          BMP IMAGE PROCESSING          */
3  /* ===== */
4
5  /*
6   * Author: Luiz Henrique Kiehn, Thadeu Antonio Ferreira Melo
7   */
8
9  /*
10 *
11 */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <string.h>
16 #include <math.h>
17
18 // int noiseGenerator(SINGLE_PIXEL **, unsigned long, unsigned long, int){
19 // int noiseStats(SINGLE_PIXEL **, unsigned long, unsigned long, double *, double *, double *, double *)
20 // int thresh(SINGLE_PIXEL **, unsigned long, unsigned long, char)
21 // int smoth(SINGLE_PIXEL **, unsigned long, unsigned long, int)
22 // int print_header(BITMAPFILEHEADER, BITMAPINFOHEADER)
23 // int read_header(FILE *, BITMAPFILEHEADER *, BITMAPINFOHEADER *)
24 // int write_bmp(FILE *, BITMAPFILEHEADER *, BITMAPINFOHEADER *, SINGLE_PIXEL **)
25 // int main(int, char *)
26
27 typedef struct {
28     unsigned short type;
29     unsigned long size;
30     unsigned short reserved1;
31     unsigned short reserved2;
32     unsigned long offsetbits;
33 } BITMAPFILEHEADER;
34
35 typedef struct {
36     unsigned long size;
37     unsigned long width;
38     unsigned long height;
39     unsigned short planes;
40     unsigned short bitcount;
41     unsigned long compression;
42     unsigned long sizeimage;
43     long xpelspermeter;
44     long ypelspermeter;
45     unsigned long colorsused;
46     unsigned long colorsimportant;
47 } BITMAPINFOHEADER;
48
49 typedef struct {
50     unsigned char blue;
51     unsigned char green;
52     unsigned char red;
53 } SINGLE_PIXEL;
54
55 /* essas funções funcionam para imagens BMP 24bit */
56
57 int noiseGenerator(SINGLE_PIXEL **pixel, unsigned long width, unsigned long height, int quant){
58     int i, j, k, cont;
59
60     for(i = 0; i < height; i++){
61         for(j = 0; j < width; j++){
62             cont = (pixel[i][j].blue + pixel[i][j].red + pixel[i][j].green)/3;
63             k = rand()%quant + cont;
64             if(k >= 255){
65                 k = rand()%(255-(char)quant);
66             }
67         }

```

```

68         pixel[i][j].blue = k;
69         pixel[i][j].red = k;
70         pixel[i][j].green = k;
71     }
72 }
73 }
74
75 }
76
77 int noiseStats(SINGLE_PIXEL **pixel, unsigned long width, unsigned long height, double *Imean, double *Isd, double *Istd, double *Nmean, double *Nsd) {
78     int i, j, k, cont;
79     float x, y, z;
80     double IMGmean, IMGsd, Noisesd, Noisemean;
81
82     /* Measure */
83     x = y = 0.0;
84     for(i = 0; i < height; i++){
85         for(j = 0; j < width; j++){
86             cont = (pixel[i][j].blue + pixel[i][j].red + pixel[i][j].green)/3;
87             x += (float)cont;
88             y += (float)cont*(float)cont;
89             k +=1;
90         }
91     }
92
93     /* Compute estimate - mean noise is 0 */
94     IMGsd = (double)(y - x*x/(float)k)/(float)(k-1);
95     IMGmean = (double)(x/(float)k);
96     IMGsd = sqrt(IMGsd);
97
98     x = y = z = 0.0;
99     for(i = 0; i < height; i++){
100         for(j = 0; j < width; j++){
101             cont = (pixel[i][j].blue + pixel[i][j].red + pixel[i][j].green)/3;
102             z = (float)(cont - IMGmean);
103             x += z;
104             y += z*z;
105         }
106     }
107
108     Noisesd = (double)(y - x*x/(float)k)/(float)(k-1);
109     Noisemean = (double)(x/(float)k);
110     Noisesd = sqrt(Noisesd);
111
112     *Imean = IMGmean;
113     *Isd = IMGsd;
114     *Nmean = Noisemean;
115     *Nsd = Noisesd;
116
117 }
118
119 int thresh(SINGLE_PIXEL **pixel, unsigned long width, unsigned long height, char threshHold) {
120     int cont, i, j;
121     for(i = 0; i < height; i++){
122         for(j = 0; j < width; j++){
123             cont = (pixel[i][j].blue + pixel[i][j].red + pixel[i][j].green)/3;
124
125             if (cont >= threshHold){
126                 pixel[i][j].blue = 0;
127                 pixel[i][j].green = 0;
128                 pixel[i][j].red = 0;
129             } else {
130                 pixel[i][j].blue = 255;
131                 pixel[i][j].green = 255;
132                 pixel[i][j].red = 255;
133             }
134         }
135     }
136 }

```

```

135     }
136 }
137
138 int color2gray(SINGLE_PIXEL **pixel, unsigned long width, unsigned long height){
139     int cont, i, j;
140     for(i = 0; i < height; i++){
141         for(j = 0; j < width; j++){
142             cont = (pixel[i][j].blue + pixel[i][j].red + pixel[i][j].green)/3;
143             pixel[i][j].blue = cont;
144             pixel[i][j].green = cont;
145             pixel[i][j].red = cont;
146         }
147     }
148 }
149
150 int smoth(SINGLE_PIXEL **pixel, unsigned long width, unsigned long height, int level){
151     int i, j, k;
152     int cont;
153     SINGLE_PIXEL **temp;
154     temp = calloc(height, sizeof(SINGLE_PIXEL *));
155     for (i = 0; i < height; ++i){
156         temp[i] = calloc(width, sizeof(SINGLE_PIXEL));
157         //printf("%d\n", i);
158     }
159
160     for(k = 0; k < level; k++){
161         for(i=0; i < height; i++ ){
162             for(j=0; j < width; j++){
163                 temp[i][j].blue = pixel[i][j].blue;
164                 temp[i][j].green = pixel[i][j].green;
165                 temp[i][j].red = pixel[i][j].red;
166             }
167         }
168         for(i = 1; i < height - 1; i++){
169             cont = 0;
170             for(j = 1; j < width - 1; j++){
171                 cont += (temp[i-1][j-1].blue + temp[i-1][j-1].red + temp[i-1][j-1].green)/3;
172                 cont += (temp[i-1][j].blue + temp[i-1][j].red + temp[i-1][j].green)*2/3;
173                 cont += (temp[i-1][j+1].blue + temp[i-1][j+1].red + temp[i-1][j+1].green)/3;
174                 cont += (temp[i][j-1].blue + temp[i][j-1].red + temp[i][j-1].green)*2/3;
175                 cont += (temp[i][j].blue + temp[i][j].red + temp[i][j].green)*3/3;
176                 cont += (temp[i][j+1].blue + temp[i][j+1].red + temp[i][j+1].green)*2/3;
177                 cont += (temp[i+1][j-1].blue + temp[i+1][j-1].red + temp[i+1][j-1].green)/3;
178                 cont += (temp[i+1][j].blue + temp[i+1][j].red + temp[i+1][j].green)*2/3;
179                 cont += (temp[i+1][j+1].blue + temp[i+1][j+1].red + temp[i+1][j+1].green)/3;
180
181                 cont = cont/15;
182                 if(cont>255) {
183                     //printf("%d\n", cont);
184                     cont = 255;
185                 }
186
187                 pixel[i][j].blue = (char)cont;
188                 pixel[i][j].red = (char)cont;
189                 pixel[i][j].green = (char)cont;
190             }
191         }
192     }
193
194 }
195
196 int print_header(BITMAPFILEHEADER fileHeader, BITMAPINFOHEADER infoHeader){
197
198     printf("type %c\n", fileHeader.type);
199     printf("tamanho %d\n", fileHeader.size);
200     printf("reservado 1 %d\n", fileHeader.reserved1);
201     printf("reservado 2 %d\n", fileHeader.reserved2);

```

```

202     printf("tamanho %d\n", infoHeader.size);
203     printf("largura %d\n", (infoHeader.width));
204     printf("altura %d\n", (infoHeader.height));
205     printf("planes %d\n", (infoHeader.planes));
206     printf("bitcount %d\n", (infoHeader.bitcount));
207     printf("compression %d\n", (infoHeader.compression));
208     printf("sizeimage %d\n", (infoHeader.sizeimage));
209     printf("xpelspermeter %d\n", (infoHeader.xpelspermeter));
210     printf("ypelspermeter %d\n", (infoHeader.ypelspermeter));
211     printf("colorsused %d\n", (infoHeader.colorsused));
212     printf("colorsimportant %d\n", (infoHeader.colorsimportant));
213
214     return 0;
215 }
216
217 int read_header(FILE *fp, BITMAPFILEHEADER *fileHeader, BITMAPINFOHEADER *infoHeader){
218
219     fseek(fp,0,0);
220
221     fread(&fileHeader->type, sizeof(fileHeader->type),1,fp);
222     fread(&fileHeader->size, sizeof(fileHeader->size),1,fp);
223     fread(&fileHeader->reserved1, sizeof(fileHeader->reserved1),1,fp);
224     fread(&fileHeader->reserved2, sizeof(fileHeader->reserved2),1,fp);
225     fread(&fileHeader->offsetbits, sizeof(fileHeader->offsetbits),1,fp);
226
227     fread(&infoHeader->size, sizeof(infoHeader->size),1,fp);
228     fread(&infoHeader->width, sizeof(infoHeader->width),1,fp);
229     fread(&infoHeader->height, sizeof(infoHeader->height),1,fp);
230     fread(&infoHeader->planes, sizeof(infoHeader->planes),1,fp);
231     fread(&infoHeader->bitcount, sizeof(infoHeader->bitcount),1,fp);
232     fread(&infoHeader->compression, sizeof(infoHeader->compression),1,fp);
233     fread(&infoHeader->sizeimage, sizeof(infoHeader->sizeimage),1,fp);
234     fread(&infoHeader->xpelspermeter, sizeof(infoHeader->xpelspermeter),1,fp);
235     fread(&infoHeader->ypelspermeter, sizeof(infoHeader->ypelspermeter),1,fp);
236     fread(&infoHeader->colorsused, sizeof(infoHeader->colorsused),1,fp);
237     fread(&infoHeader->colorsimportant, sizeof(infoHeader->colorsimportant),1,fp);
238 }
239
240 int write_bmp(FILE *ft, BITMAPFILEHEADER *fileHeader, BITMAPINFOHEADER *infoHeader, SINGLE_PIXEL **pixel){
241
242     fseek(ft,0,0);
243
244     fwrite(&fileHeader->type, sizeof(fileHeader->type),1,ft);
245     fwrite(&fileHeader->size, sizeof(fileHeader->size),1,ft);
246     fwrite(&fileHeader->reserved1, sizeof(fileHeader->reserved1),1,ft);
247     fwrite(&fileHeader->reserved2, sizeof(fileHeader->reserved2),1,ft);
248     fwrite(&fileHeader->offsetbits, sizeof(fileHeader->offsetbits),1,ft);
249
250     fwrite(&infoHeader->size, sizeof(infoHeader->size),1,ft);
251     fwrite(&infoHeader->width, sizeof(infoHeader->width),1,ft);
252     fwrite(&infoHeader->height, sizeof(infoHeader->height),1,ft);
253     fwrite(&infoHeader->planes, sizeof(infoHeader->planes),1,ft);
254     fwrite(&infoHeader->bitcount, sizeof(infoHeader->bitcount),1,ft);
255     fwrite(&infoHeader->compression, sizeof(infoHeader->compression),1,ft);
256     fwrite(&infoHeader->sizeimage, sizeof(infoHeader->sizeimage),1,ft);
257     fwrite(&infoHeader->xpelspermeter, sizeof(infoHeader->xpelspermeter),1,ft);
258     fwrite(&infoHeader->ypelspermeter, sizeof(infoHeader->ypelspermeter),1,ft);
259     fwrite(&infoHeader->colorsused, sizeof(infoHeader->colorsused),1,ft);
260     fwrite(&infoHeader->colorsimportant, sizeof(infoHeader->colorsimportant),1,ft);
261
262     int i;
263     int j;
264
265     for(i=0; i < infoHeader->height; i++){
266         for(j=0; j < infoHeader->width; j++){
267             fwrite(&pixel[i][j].blue, sizeof(pixel[i][j].blue), 1, ft);
268             fwrite(&pixel[i][j].green, sizeof(pixel[i][j].green), 1, ft);

```

```

269             fwrite(&pixel[i][j].red, sizeof(pixel[i][j].red), 1, ft);
270         }
271     }
272 }
273
274 //int main(int argc, char** argv) {
275
276 int main(int argc, char *argv[]) {
277
278     FILE *fp, *ft;
279     char *file1;
280     BITMAPINFOHEADER infoHeader;
281     BITMAPFILEHEADER fileHeader;
282
283     file1 = "img7.bmp";
284
285     /* open the file */
286     if ((fp = fopen(file1, "rb")) == NULL) {
287         printf("Error opening file %s.\n", file1);
288         exit(1);
289     }
290     /* check to see if it is a valid bitmap file */
291     if (fgetc(fp) != 'B' || fgetc(fp) != 'M') {
292         fclose(fp);
293         printf("%s is not a bitmap file.\n", file1);
294         exit(1);
295     }
296
297     read_header(fp, &fileHeader, &infoHeader);
298
299     SINGLE_PIXEL    **pixel;
300
301     int i;
302     int j;
303
304     pixel = calloc(infoHeader.height, sizeof(SINGLE_PIXEL *));
305     for (i = 0; i < infoHeader.height; ++i)
306         pixel[i] = calloc(infoHeader.width, sizeof(SINGLE_PIXEL));
307
308     for(i=0; i < infoHeader.height; i++){
309         for(j=0; j < infoHeader.width; j++){
310             pixel[i][j].blue = fgetc(fp);
311             pixel[i][j].green = fgetc(fp);
312             pixel[i][j].red = fgetc(fp);
313         }
314     }
315
316     for(i=0; i < infoHeader.height; i++){
317         for(j=0; j < infoHeader.width; j++){
318             }
319         }
320
321     //double lmean, lsd, nmean, nsd;
322     //print_header(fileHeader, infoHeader);
323     //printf("matriz %d\n", sizeof(pixel[1][1]));
324     //noiseStats(pixel, infoHeader.height, infoHeader.width, &lmean, &lsd, &nmean, &nsd);
325     //printf("Sem:\nMedia Imagem = %f\nDesvio Imagem = %f\nMedia Ruido = %f\nDesvio Ruido = %f\n", lmean, lsd,
326     //color2gray(pixel, infoHeader.height, infoHeader.width);
327     ft = fopen("cinza2.bmp", "wb");
328
329     fclose(ft);
330     //thresh(pixel, infoHeader, 64);
331
332
333     //ft = fopen("thresh.bmp", "wb");
334     //write_bmp(ft, &fileHeader, &infoHeader, pixel);
335

```

```
336 //fclose(ft);
337
338 ft = fopen ( "smoth3.bmp", "wb" ) ;
339 smoth(pixel, infoHeader.width, infoHeader.height, 10);
340 // noiseStats(pixel, infoHeader.height, infoHeader.width, &Imean, &Isd, &Nmean, &Nsd);
341 // printf("Ruido:\nMedia Imagem = %f\nDesvio Imagem = %f\nMedia Ruido = %f\nDesvio Ruido = %f\n",Imean, Isd, Nmean, Nsd);
342 write_bmp(ft, &fileHeader, &infoHeader, pixel);
343
344 fclose(ft);
345
346 //printf();
347 fclose(fp);
348
349 printf("Height=%d - Width=%d\n", infoHeader.height, infoHeader.width);
350
351 return (EXIT_SUCCESS);
352 }
```