P2L - Uma ferramenta de profiling ao nível de instrução para o processador softcore LEON3

Carlos Roberto Pereira Almeida Júnior

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Carlos Roberto Pereira Almeida Júnior

P2L - Uma ferramenta de profiling ao nível de instrução para o processador softcore LEON3

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

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Eduardo Marques

USP – São Carlos
Julho de 2016

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

A447p Almeida Júnior, Carlos Roberto Pereira
P2L - Uma ferramenta de profiling ao nível
de instrução para o processador softcore LEON3 /
Carlos Roberto Pereira Almeida Júnior; orientador
Eduardo Marques. - São Carlos - SP, 2016.
102 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, 2016.

1. profiling. 2. Computação Reconfigurável.
3. FPGA. 4. Desempenho. 5. LEON3. I. Marques,
Eduardo, orient. II. Título.

Carlos Roberto Pereira Almeida Júnior

P2L - A instruction level profiling tool for LEON3 softcore

Master dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Master Program in Computer Science and Computational Mathematics. FINAL VERSION

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Eduardo Marques

USP – São Carlos
July 2016

Dedico este trabalho aos meus pais, Carlos e Terezinha, e a minha esposa, Anne.

AGRADECIMENTOS

Aos meus pais Carlos e Terezinha pelo amor, apoio e esforço sem tamanho para que eu pudesse me dedicar aos estudos desde o princípio. Sem eles, indubitavelmente, não teria chegado até esta etapa.

Ao meu orientador, Prof. Eduardo Marques, que me aceitou como aluno e situou-me diversas vezes em meu projeto de pesquisa. A sua contribuição técnica e de valores humanos são inestimáveis.

Aos Professores Vanderlei Bonato e Maikon Adilles pelas dicas valiosas e que contribuíram de forma significativa para a conclusão deste projeto.

À minha esposa, Anne, por ter tornado minha vida mais feliz e manter-me são nos momentos de maior dificuldade.

Ao Rubem Saldanha da Intel por ceder o kit de desenvolvimento utilizado neste trabalho e também ao Magnus Hjorth e Jan Andersson da Cobham Gaisler AB pelo suporte ao LEON3.

Aos colegas e funcionários do ICMC-USP.

“ It is paradoxical, yet true, to say,
that the more we know, the more ignorant
we become in the absolute sense, for it is
only through enlightenment that we become
conscious of our limitations. Precisely
one of the most gratifying results of
intellectual evolution is the continuous
opening up of new and greater prospects.”
(Nikola Tesla)

RESUM O

ALMEIDA JÚNIOR, CARLOS R. P. P2L - Uma ferramenta de profiling ao nível de instrução para o processador softcore LEON3. 2016. 102 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

A maioria dos sistemas embarcados hoje desenvolvidos utilizam complexos sistemas eletrônicos integrados em um único chip, os Systems-on-a-Chip (SoC). A análise do comportamento de uma aplicação em execução, ou seja, o profiling nesses sistemas não é uma tarefa trivial em virtude da complexidade dos SoCs e pela restrição de ferramentas de profiling adequadas. Neste contexto, este trabalho apresenta o P2L, uma ferramenta de profiling que se baseia em métricas de nível de instrução e função para o processador LEON3. O P2L fornece estatísticas detalhadas de uso do processador, memórias e barramento de programas em execução sem uso de instrumentação. A ferramenta é composta por um componente em hardware e drivers e aplicativos em software. Os resultados mostram que o P2L fornece medidas com erro inferior a 1% e overhead desprezível quando comparado ao tempo de execução nativa do programa e ao do profiler GNU gprof.

Palavras-chave: profiling, Computação Reconfigurável, FPGA, Desempenho, LEON3.

ABSTRACT

ALMEIDA JÚNIOR, CARLOS R. P. P2L - Uma ferramenta de profiling ao nível de instrução para o processador softcore LEON3. 2016. 102 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Most embedded systems developed today use complex electronic systems integrated into a single chip, the Systems-on-a-Chip (SoC). The analysis of the behavior of a running application or profiling in these systems is not a trivial task due to the complexity of the SoC and the restriction of appropriate profiling tools. In this context, this work presents P2L - a profiling tool that is based on instruction and function level metrics for the LEON3 processor. P2L provides detailed usage statistics of the processor, memories, and bus of running programs without the use of instrumentation. The tool consists of a component in hardware, drivers and applications software. The results show that P2L provides measures with an error less than 1% and negligible overhead compared to native runtime program and the GNU profiler gprof.

Key-words: profiling, Reconfigurable Computing, FPGA, Performance, LEON3.

LISTA DE ILUSTRAÇÕES

Figura 1 – Alternativas de implementação	26
Figura 2 – Uma abordagem para Co-Projeto	27
Figura 3 – Contextualização do trabalho	27
Figura 4 – Instrumentação - possíveis etapas para inserção de probes	35
Figura 5 – Resultado do profiling ao nível de instruções de um software hipotético	42
Figura 6 – Arquitetura genérica da SnoopP	43
Figura 7 – Arquitetura da ferramenta ACAD	44
Figura 8 – Arquitetura da ferramenta MPPA	45
Figura 9 – Placa de desenvolvimento Terasic DE2i-150	48
Figura 10 – Arquitetura da Terasic DE2i-150	48
Figura 11 – Diagrama interno do processador LEON3	51
Figura 12 – Formato ELF	53
Figura 13 – Configuração do LEON3: principal	58
Figura 14 – Configuração do LEON3: barramento, memórias e periféricos	59
Figura 15 – Fluxo de desenvolvimento com ferramentas GRLIB e Altera	60
Figura 16 – Arquitetura simplificada do LEON3 com módulo P2L	61
Figura 17 – Estágios de Pipeline do LEON3	62
Figura 18 – Análise do Pipeline - Modelsim	63
Figura 19 – Arquivos editados para interface de sinal da Unidade de Inteiros	63
Figura 20 – L3STAT - Registro de controle de contadores	66
Figura 21 – Análise de formas de onda na captura de eventos relacionados à instruções . .	67
Figura 22 – Diagramas de blocos do módulo P2L	69
Figura 23 – Configuração de segmentos de código e eventos no P2L	71
Figura 24 – P2L - Fluxo completo de um profiling	72
Figura 25 – P2L - Etapa 1 de configuração de um profiling	73
Figura 26 – P2L - Etapa 2 de configuração de um profiling	74
Figura 27 – P2L - Etapa 3 de configuração de um profiling	75
Figura 28 – Coremark - Tempo de execução	81
Figura 29 – Dhrystone - Tempo de execução	85
Figura 30 – Mi Bench Susan - Tempo de execução	87
Figura 31 – Mi Bench Dijkstra - Tempo de execução	89
Figura 32 – CPU SPEC2006 Bzip2 - Tempo de execução	91

LISTA DE QUADROS

Quadro 1 – Algoritmo para computação de um número fatorial	30
Quadro 2 – Contagem de ciclos de clock das instruções assembly do fatorial de 3	31
Quadro 3 – Resultado do teste de assertividade	56
Quadro 4 – Unidade de estatísticas - Eventos disponíveis	65
Quadro 5 – Unidade de estatísticas - Mapeamento no barramento AHB/APB	66
Quadro 6 – Análise da função de exemplo - Definição de segmentos de código	66
Quadro 7 – Espaço de endereçamento do P2L no barramento AHB/APB	69
Quadro 8 – Análise da função de exemplo - Definição de segmentos de código	70
Quadro 9 – Análise da função de exemplo - Definição de eventos e precisão	70
Quadro 10 – Configuração da Plataforma de Testes	78
Quadro 11 – Eventos selecionados para os experimentos	79
Quadro 12 – Coremark - Geral	79
Quadro 13 – Dhrystone - Geral	83
Quadro 14 – Mi Bench Susan - Geral	86
Quadro 15 – Mi Bench Dijkstra - Geral	86
Quadro 16 – CPU SPEC2006 Bzip2 - Geral	91

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Teste de assertividade para Modelsim - Código em C	55
Código-fonte 2 – Teste de assertividade para Modelsim - Código em Assembly	56
Código-fonte 3 – Exemplo: instância de um componente em Leon3mp.vhd	58
Código-fonte 4 – Modificações na Unidade de Inteiros	63
Código-fonte 5 – Modificações na Unidade de estatística	67
Código-fonte 6 – Conexão de estatísticas e	68
Código-fonte 7 – Exemplo de uma aplicação hipotética para configuração da ferramenta de profiling	69

LISTA DE TABELAS

Tabela 1 – Recursos disponíveis nas PMUs	39
Tabela 2 – Exemplo da visualização de dados gerados por profiling	42
Tabela 3 – Coremark - Tempo médio de execução	80
Tabela 4 – Coremark - Métricas de nível de instrução	82
Tabela 5 – Dhrystone - Tempo médio de execução	83
Tabela 6 – Dhrystone - Métricas de nível de instrução	84
Tabela 7 – Mi Bench Susan - Tempo médio de execução	87
Tabela 8 – Mi Bench Susan - Métricas de nível de instrução	88
Tabela 9 – Mi Bench Dijkstra - Tempo médio de execução	89
Tabela 10 – Mi Bench Dijkstra - Métricas de nível de instrução	90
Tabela 11 – CPU SPEC2006 Bzip2 - Tempo médio de execução	92
Tabela 12 – CPU SPEC2006 Bzip2 - Métricas de nível de instrução	93
Tabela 13 – Ferramentas de profiling	97

SUMÁRIO

1	INTRODUÇÃO	25
1.1	Contextualização	25
1.2	Motivação e Objetivos	27
1.3	Objetivos	28
1.3.1	Objetivo Geral	28
1.3.2	Objetivos específicos	28
1.3.3	Justificativa	29
1.4	Organização do Trabalho	30
2	FUNDAMENTAÇÃO TEÓRICA	33
2.1	Profiling	33
2.1.1	Instrumentação	34
2.1.2	Amostragem	36
2.1.3	Abordagens sem intrusividade e baixo overhead	37
2.1.4	Monitoramento	37
2.1.4.1	Monitoramento exato e estatístico	38
2.1.4.2	Monitoramento com suporte por contadores de hardware	39
2.1.5	Análise	40
2.1.5.1	Análise Post-Mortem	41
2.1.5.2	Análise On-the-fly	41
2.1.6	Visualização	41
2.2	Trabalhos relacionados	42
2.2.1	SnoopP	42
2.2.2	ACAD	43
2.2.3	MPPA	44
3	MATERIAIS E MÉTODOS	47
3.1	Plataforma de Desenvolvimento	47
3.2	GRLIB	49
3.3	GRMON	50
3.4	Processador LEON3	50
3.5	L3STAT	51
3.6	Ferramentas EDA	52

3.7	Sistema Operacional Linux	52
3.8	Fomato ELF	52
3.9	Métodos de validação	53
4	IMPLEMENTAÇÃO DA ARQUITETURA PROPOSTA	57
4.1	Fluxo de desenvolvimento do Hardware	57
4.2	Implementação do core P2L	60
4.2.1	Unidade de Inteiros	61
4.2.2	Unidade de estatísticas	64
4.2.3	Integração de dados obtidos na Unidade de Inteiros e de Estatísticas com o P2L	68
4.3	Preparação da aplicação para profiling	71
4.4	Driver e Escalonador para Linux	72
4.5	Ferramentas de controle e relatório	74
5	RESULTADOS	77
5.1	Benchmarks	77
5.2	Experimentos	78
5.2.1	Coremark	78
5.2.2	Dhrystone	83
5.2.3	MiBench	86
5.2.3.1	Controle Automotivo e Industrial: Susan	86
5.2.3.2	Telecomunicações - Djikstra	86
5.2.4	SPEC CPU2006	91
5.2.5	Considerações	94
6	CONCLUSÃO	95
6.0.6	Trabalhos Futuros	95
	Referências	99

INTRODUÇÃO

1.1 Contextualização

Os sistemas embarcados podem ser definidos como sistemas de processamento incorporados a produtos dos quais a maioria são pouco visíveis e perceptíveis para os humanos. Esses sistemas comandam robôs no chão de fábrica, coordenam motores, processos químicos e satélites. Auxiliam a vida doméstica, controlando máquinas de lavar, micro-ondas e lava louças. Equipam brinquedos inteligentes, smartphones e tablets. Por fim, estão presentes em diversos ambientes e compõem uma área de aplicação importante da tecnologia da informação (LEE; SESHIA, 2015; MARWEDEL, 2006). O nível de complexidade das aplicações embarcadas determina uma ampla abordagem no desenvolvimento desses sistemas. Os mesmos podem ser concebidos com circuitos integrados off-the-shelf; circuitos integrados personalizados, como os Application-Specific Integrated Circuits (ASIC); e circuitos integrados reconfiguráveis, os Field Programmable Gate Arrays (FPGAs). Em virtude dessas opções, uma aplicação simples pode ter seus requisitos atendidos por um modesto microcontrolador disponível comercialmente. Contudo, aplicações embarcadas mais complexas, como a de um sistema autônomo em um automóvel, exigem inúmeras unidades de processamento, memórias, periféricos e softwares que geralmente estão integrados em componentes dentro de Systems-on-a-Chip (SoC) ou Multiprocessors System-on-Chip (MPSoC) sintetizados em FPGA ou fabricados em ASIC.

A [Figura 1](#) ilustra as alternativas de implementação e as restrições das tecnologias que estão na essência do processo de decisão do Co-Projeto de Hardware/Software (SCHAUMONT, 2012). Esse campo de pesquisa em Engenharia de Computação tem em uma de suas etapas o Particionamento de Hardware/Software, cuja importância é descrita por Dufour e Radetzki (2006) em:

While pure hardware synthesis tools like Hardware Description Languages allow the implementation of systems on a chip from high-level specification, the resulting cost of implementation is a considerable drawback. On the other hand, off-the-shelf generic processors, and implementation in software, are much cheaper, but seldom suit the hard time constraints imposed on embedded systems. Thus, cost-effective designs should use a mixture of hardware and software to accomplish their goals.

The problem with the design-oriented approach, where allocation of tasks to hardware and software precedes the synthesis, is that it is not known whether the final system will meet its requirements. This suggests a synthesis-oriented approach, where constraints on performance and cost of the systems are specified, and a systematic constraint-driven exploration of the design space is done. The model must first be given in terms of a specification for hardware (often in a Hardware Description Language) and for software. Then it is partitioned, and the specifications allow run-time estimations, simulations as well as cost estimations, to test whether the resulting system meets the constraints. (DUFOR; RADETZKI, 2006)

Assim, o objetivo do particionamento de hardware/software é encontrar uma composição coerente de componentes de hardware e software que não apenas satisfaça as restrições de desempenho, mas que também atenda a outros propósitos desejáveis, como segurança, baixo custo de comunicação, consumo de energia e etc (GEGUANG et al., 2004). Como resultado, esta etapa é dependente de ferramentas precisas de avaliação durante o desenvolvimento do sistema, conforme ilustra a [Figura 2](#). Uma delas é a de profiling, que é responsável por analisar o comportamento de um programa em execução identificando regiões ineficientes ou de gargalo, padrões de acesso à memória, Worst Case Execution Time (WCET) e etc. É desejável que o profiling seja preciso e ofereça baixo overhead para tornar o particionamento efetivo, pois em posse das informações obtidas por essa ferramenta, os segmentos críticos de programa podem ser otimizados por versões reescritas do software, guiarem modificações na arquitetura de processadores ou serem convertidas em aceleradores de hardware a fim de se obter o cumprimento de um requisito de projeto ou melhorar o desempenho de todo sistema (MISHRA et al., 2012).

Figura 1 – Alternativas de implementação

Fonte: Elaborada pelo autor.

Contudo, as técnicas de profiling utilizadas pelas ferramentas existentes variam em

Figura 2 – Uma abordagem para Co-Projeto

Fonte: Elaborada pelo autor.

relação ao nível de precisão, overhead adicional no tempo execução da aplicação e intrusividade, que pode causar modificações no comportamento da aplicação analisada. O impacto desses aspectos varia de acordo com a plataforma alvo na qual as aplicações são executadas e da presença de um ambiente multitarefa. No contexto deste trabalho, conforme ilustra a figura 3, as ferramentas existentes são insuficientes para lidar a complexidade dos SoCs/MPSoCs e exigem novas abordagens, sendo a do P2L uma proposta neste sentido.

Figura 3 – Contextualização do trabalho

Fonte: Elaborada pelo autor.

1.2 Motivação e Objetivos

Um SoC avançado é composto por inúmeras unidades funcionais. Analisar o comportamento de uma aplicação em um sistema tão complexo não é trivial e leva a algumas questões: Como é possível executar um profiling dentro dessas arquiteturas? É necessário o uso de uma ou várias ferramentas? Como obter o desempenho máximo em uma plataforma se não houver boas técnicas de profiling?

Nesse sentido, é desejável não só conhecer o tempo de execução da aplicação ou de partes

dela, mas também o comportamento do processador, acesso a memórias e barramentos. Essas preocupações levam a necessidade de ferramentas de profiling dedicadas e não intrusivas. Nesse contexto, a computação reconfigurável oferece flexibilidade e condições de desenvolvimento de ferramentas baseadas em hardware e software. Baseando-se nesses princípios, o processador soft core LEON3 foi escolhido como plataforma alvo. O mesmo é amplamente configurável e adequado para projetos de SoCs. Possui código fonte completo disponibilizado com boa documentação e é baseado na arquitetura consagrada do processador SPARC V8, herdando dela um ambiente de desenvolvimento de software maduro. Além disso, é utilizado em inúmeros projetos industriais com destaque aos da Agência Espacial Europeia.

1.3 Objetivos

1.3.1 Objetivo Geral

Este projeto tem como objetivo desenvolver uma ferramenta de profiling ao nível de instrução de alta precisão e baixa intrusividade para o processador soft-core LEON3. A essa ferramenta é dado o nome P2L, um acrônimo para Profiler to LEON. O P2L tem como papel fornecer estatísticas detalhadas de uso do processador, memórias e barramento de uma aplicação em execução com capacidade de fornecer informações sobre funções ou segmentos de código cujo desenvolvedor deseja saber o comportamento. Essa ferramenta se baseia na implementação de um componente em hardware, que mantém em bancos de registradores as informações de profiling extraídas da CPU e de um componente de estatísticas desenvolvido pela (GAISLER, 2016).

1.3.2 Objetivos específicos

Os objetivos específicos deste trabalho são:

- Desenvolver um componente em hardware para extrair informações do registrador Program Counter no estágio de pipeline adequado dentro da Unidade de Inteiros (UI) do processador LEON3 e, também, eventos gerados pela unidade de estatísticas desenvolvida pela (GAISLER, 2016);
- Desenvolver um driver para o sistema operacional Linux capaz de gerenciar as funcionalidades do hardware desenvolvido;
- Desenvolver um aplicativo para extrair informações do programa alvo de análise e configurar o hardware de profiling;
- Desenvolver um aplicativo para análise, monitoramento e geração de relatório.

1.3.3 Justificativa

Localizar regiões críticas de uma aplicação é o propósito de uma ferramenta de profiling. Todavia, além de identifica-los com métricas associadas ao tempo de execução, determinar quais recursos de hardware estão associados ao gargalo torna-se necessário para se explorar melhor o desempenho de um SoC ou servir de parâmetro a um melhor particionamento durante o desenvolvimento de um sistema. Uma visão geral dessa questão é exemplificada com o auxílio do seguinte exemplo: a computação de números fatoriais é descrita por um simples algoritmo descrito em linguagem C e o correspondente em assembly do conjunto de instruções da arquitetura do LEON3 (Sparc V8) é descrita no quadro 1.

Observando o código assembly, podemos notar que a estrutura de repetição é desempenhada após o offset 0x4c e as rotinas que computam o fatorial, no segmento de offset 0x1c a 0x4c. Dentro deste último segmento, a multiplicação é exercida pela biblioteca .umul. Ela não é levada em consideração neste exemplo, entretanto, o custo de cada operação varia de 25 a 45 ciclos de clock. A partir da informação que descreve o número de ciclos necessários para execução das instruções (descrito no manual (COBHAM-GAISLER, 2015)) é possível extrair algumas métricas de profiling de forma offline, ou seja, sem que seja necessária a execução do código. A partir da análise do fluxo de instruções supondo a execução do fatorial do inteiro 3, chega-se a conclusão que a quantidade de instruções a serem processadas é de 63. No caso do cômputo de ciclos de clock, a contagem é auxiliada pelo quadro 2.

Os cabeçalhos Quantidade e Ciclos de processamento indicam, respectivamente, o número de instruções* contabilizados no fluxo de execução total do algoritmo do fatorial do número 3 e quantos ciclos de processamento são necessários para executar uma dessas instruções*.

Analisando o quadro anterior, conclui-se que as 63 instruções seriam executadas em 78 ciclos de clock. Todavia, essa contagem não leva em conta os aspectos intrínsecos da arquitetura do processador LEON3. Entretanto, quando se considera o overhead no ciclo de execução, os stalls no pipeline implicam em um tempo de execução diferente do estimado. O código do fatorial quando executado em um LEON3 na sua configuração padrão¹ demanda em média² 175 ciclos de clock. Assim, conhecer as razões que implicam na diferença do tempo de processamento se torna desejável em uma ferramenta de profiling. Esta foi uma preocupação considerada na implementação do P2L. Logo, utilizando-se dessa ferramenta, constatou-se que as causas do stall estão relacionadas com o cache miss de dados e de instrução que ocorrem, respectivamente, 4 e 2 vezes. Deste modo, estimar o tempo de processamento analisando apenas o fluxo de execução offline não é ideal visto que a variação do valor estimado para o extraído na plataforma real é de cerca de 220%. Tal resultado legitima a importância das ferramentas de profiling.

¹ Um núcleo de processamento, 128 MB de memória SDRAM, MMU fast e cache dados e instrução de 4*4 kB

² A média foi obtida após 10 execuções do código

Quadro 1 – Algoritmo para computação de um número fatorial

<pre> 1 // Código C 2 int fatorial(int n) { 3 int i = 0, f = 1; 4 for (i; i < n; i++) { 5 f = f*(n-i); 6 } 7 return f; 8 } </pre>	<pre> 1 # Assembly 2 <fatorial>: 3 save %sp, -104, %sp 4 st %i0, [%fp + 0x44] 5 clr [%fp + -8] 6 mov 1, %g1 7 st %g1, [%fp + -4] 8 b fatorial+0x4c 9 nop 10 .fatorial+0x1c: 11 ld [%fp + 0x44], %g2 12 ld [%fp + -8], %g1 13 sub %g2, %g1, %g1 14 ld [%fp + -4], %o0 15 mov %g1, %o1 16 call <.umul> 17 nop 18 mov %o0, %g1 19 st %g1, [%fp + -4] 20 ld [%fp + -8], %g1 21 inc %g1 22 st %g1, [%fp + -8] 23 .fatorial+0x4c: 24 ld [%fp + -8], %g2 25 ld [%fp + 0x44], %g1 26 cmp %g2, %g1 27 bl fatorial+0x1c 28 nop 29 ld [%fp + -4], %g1 30 mov %g1, %i0 31 restore 32 retl 33 nop </pre>
---	---

1.4 Organização do Trabalho

Este documento encontra-se organizado da seguinte forma: o Capítulo 2 apresenta a revisão bibliográfica que aborda o tema profiling. O capítulo 3 apresenta materiais e métodos que foram utilizados para implementação e validação do projeto. O Capítulo 4 apresenta a implementação da arquitetura proposta. Por fim, o Capítulo 5 apresenta os resultados obtidos e o Capítulo 6, a conclusão.

Quadro 2 – Contagem de ciclos de clock das instruções assembly do fatorial de 3

Instrução*	Quantidade	Ciclos de processamento	Subtotal
save	1	1	1
st	8	2	16
clr	1	1	1
mov	8	1	8
b	1	1	1
bl	4	1	4
nop	9	1	9
ld	21	1	21
sub	3	1	3
call	3	1	3
inc	3	1	3
cmp	4	1	4
restore	1	1	1
retl	1	3	3
Total			78

FUNDAMENTAÇÃO TEÓRICA

Este capítulo discute as técnicas empregadas pelos Profilers e quais são suas restrições para uso em aplicações em sistemas embarcados.

2.1 Profiling

O processo que envolve a análise de um programa sobre uma arquitetura de hardware a fim de se obter informações sobre o seu comportamento ou aspectos de sua execução é o que define um profiling. O resultado pode auxiliar na identificação de seções críticas que permitem guiar o desenvolvedor na modificação de algoritmos ou estruturas de dados, configuração/projeto de arquiteturas de hardware ou na conversão de trechos de programa em aceleradores de hardware (MISHRA et al., 2012). Além disso, um profiler, ou seja, uma ferramenta de profiling, permite que um desenvolvedor de hardware avalie se uma nova arquitetura atinge os requisitos de projeto, um desenvolvedor do software obter os pontos de gargalo ou hotspot de seu programa e, um desenvolvedor de compilador, verificar se o funcionamento de uma nova política de escalonamento de instruções conduz à melhoria do processo de compilação (SRIVASTAVA; EUSTACE, 1994).

Associado ao profiling estão os termos overhead e precisão, que possuem uma relação direta com as diversas técnicas utilizadas para medir desempenho e com a arquitetura de hardware no qual é executado. Traçando um paralelo da implicação desses termos, na física, o princípio da incerteza de Heisenberg diz que a posição e o momento do elétron não são possíveis de serem simultaneamente determinadas com precisão. A observação do momento implica no uso de radiação para aferição, que interfere na velocidade do elétron e torna essa medida impossível de ser obtida. Nesse sentido, um profiling pode causar perturbações no objeto da aferição, o programa. Logo, entender as técnicas utilizadas por um profiler é uma importante parte do profiling (WHITE, 2011).

Essas ferramentas podem ser classificadas naquelas que simulam um conjunto de instruções e as que monitoram o programa na arquitetura de hardware na qual são executados. Os profilers baseados em Instruction Set Simulators (ISS) dispõem de um maior nível de detalhes e maior precisão ao custo da velocidade de execução. Esse tipo de ferramenta é geralmente utilizada por desenvolvedores de arquiteturas de hardware para determinar o custo-benefício de uma arquitetura apoiados por conjunto de benchmarks. Esse tipo de profiling chega a ser 10 mil vezes mais lento que a execução nativa (LANCASTER et al., 2011). Dessa forma, tornam-se inviáveis para uso com softwares grandes e com SoCs, cuja natureza heterogênea constitui uma enorme dificuldade para a concepção de modelos de simulação (TONG; KHALID, 2008). Em razão dessas dificuldades, as ferramentas que se baseiam na execução do programa no ambiente em que são executadas são as mais difundidas.

Os autores Tong e Khalid (2008) e Patel e Rajawat (2011) realizaram esforços para estruturar as diferentes técnicas de profiling no contexto de sistemas embarcados, que resultaram na categorização de três abordagens. A primeira, profilers baseados em software, são aqueles que podem envolver simulação ou a inserção de códigos de monitoramento dentro dos programas, sem dependência de um hardware especial. Porém, causam considerável overhead. A segunda, as baseadas em hardware, são as que utilizam recursos presentes nesse meio, como registradores contadores especiais contidos em Performance Monitor Units (PMU) com o intuito de fornecer medidas mais precisas de tempo de execução e outros eventos com menor overhead. Por último, abordagens que levaram ao desenvolvimento de hardwares específicos para profiling em Field-Programmable Gate Array (FPGA) que interagem com soft processors e tem o objetivo de causar o mínimo overhead e oferecer alta precisão. Essa classificação não foi identificada em mais artigos dentro desse contexto e, por essa razão, o enfoque das subseções seguintes será baseada nas técnicas e na cisão do profiling como sendo o processo de monitorar, analisar e visualizar os dados de desempenho definida por Lessard (2005).

2.1.1 Instrumentação

Os profilers podem exigir algum nível de modificação do código do programa ao qual se deseja analisar o desempenho. A quantidade de código inserida define o nível de intrusividade do profiler, sendo que os mais comuns se utilizam da instrumentação. Este é um processo de adicionar pequenos pedaços de código ou probes em um programa a fim de se analisar o seu comportamento e consiste de três etapas (MILLER; CALLAGHAN, 1995):

- Ponto de instrumentação: local no código do programa onde o probe é inserido. Geralmente são localizados na entrada e saída de métodos (edge profiling) e declarações de chamada (instruções call);
- Primitiva de instrumentação: operação que atua na computação de uma métrica, ou seja, uma medida de algum aspecto do desempenho. As métricas são baseadas em contadores

desenvolvidos em hardware ou software e timers implementados em hardware;

- Predicado de instrumentação: expressões booleanas que controlam a execução de uma primitiva de instrumentação (essencialmente uma declaração `if`).

Figura 4 – Instrumentação - possíveis etapas para inserção de probes

Fonte: Elaborada pelo autor.

A inserção de probes pode ser realizada de forma estática ou dinâmica. Na primeira, a inserção do código e dados adicionais são feitas em algum passo do processo de compilação, antes da execução do programa. Essa abordagem tem a vantagem de gerar executáveis mais eficientes quando comparados a abordagem dinâmica em virtude de se resumir apenas a inserção do código da instrumentação. Também permite determinar maior cobertura do programa, aferindo com precisão quantas vezes um segmento de código foi executado. Contudo, não pode ser removida sem a recompilação e pode exigir o uso do código fonte da aplicação. Além disso, requer que o

programa seja compilado de forma estática para permitir a instrumentação de bibliotecas ou que estas sejam instrumentadas separadamente se o executável fizer uso de bibliotecas compartilhadas. Lembrando que neste caso, o executável precisa ser gerado baseado nas bibliotecas modificadas. Mesmo com essas restrições, o autor Laurenzano et al. (2010) descreve que esta abordagem pode ser eficiente em aplicações no domínio dos High Performance Computing (HPC), que podem abranger centenas ou milhares de processadores e funcionarem por horas. Neste caso, o código instrumentado tende a ser executado com overhead tolerável na coleta informações de desempenho, mas deve respeitar os prazos de deadlines da aplicação. Os profilers GNU Gprof (GRAHAM; KESSLER; MCKUSICK, 1982), Pebil (LAURENZANO et al., 2010) e Cobi (Forschungszentrum Jülich GmbH, 2016) utilizam-se desta técnica.

Na forma dinâmica ou just-in-time (JIT), a instrumentação é realizada durante a execução do programa em um processo que gera overhead adicional por compreender tarefas de parsing, disassembly, geração de código e outras decisões em tempo de execução. Este não é um problema nos profilers de instrumentação estática, pois todas as decisões e ações são tomadas antes do tempo de execução. Todavia, oferece a vantagem de não modificar o programa permanentemente e nem exigir o código fonte. Os profilers Pin (LUECK; PATIL; PEREIRA, 2012), Dyninst (BUCK, 2000) e Valgrind (NETHERCOTE; SEWARD, 2007) são exemplos de ferramentas que se baseiam nesse tipo de instrumentação.

A intrusividade causada pelo probe introduzido de forma estática ou dinâmica causa inerente overhead no tempo de execução do programa. Em sistemas embarcados, principalmente os reativos, cujas aplicações tem rigorosas restrições de cumprimento de tempo real, é comum que o mínimo aumento do overhead não seja tolerável (SHANKAR, 2010). Em virtude desses problemas que a instrumentação pode causar, alguns profilers evitam ou se utilizam da "instrumentação leve" (METZ; LENCEVICIUS; GONZALEZ, 2005) a fim de causar baixo ou nenhum overhead durante a execução do programa analisado.

2.1.2 Amostragem

Apenas com uso de instrumentação não é possível em sistemas multitarefa que um profiler consiga determinar algumas métricas de maneira precisa como, por exemplo, a medição de tempo de execução das rotinas de um programa. Por essa razão, se um probe captura o tempo real no início de uma rotina, determinar o tempo total de execução por meio de outro probe no fim dela, considerando as inúmeras trocas de contexto que ocorreram, é bastante impreciso. Uma forma de evitar esse problema é utilizar a amostragem, que se baseia na observação do programa em execução em determinados períodos de tempo. Essa abordagem tem como vantagem principal menor overhead em relação à instrumentação, entretanto, pode ser menos precisa. Porém, é baseada no preceito de que a maior parte do tempo de execução se concentra nas regiões críticas de um programa. No caso das aplicações de sistemas embarcados, é observado que 90% do tempo de execução tende a estar em 10% do código (SURESH et al., 2003).

Os sistemas operacionais oferecem recursos para amostragem de programas. Dessa forma, um profiler pode, por exemplo, realizar a contagem de tempo com auxílio da syscall `profil()`, que é implementada nos sistemas UNIX-like e tem como função registrar um vetor no espaço de endereçamento do kernel em conjunto com um fator de escala que determina o mapeamento do endereçamento do programa no vetor. Cada elemento do vetor pode armazenar valores de escala de endereçamento de 2 a 8 bytes. Durante a execução do programa em profiling, o valor do registrador program counter é examinado e o valor correspondente no item do vetor é incrementado a cada tick do sistema de clock. Uma vez que isso ocorre no espaço do kernel, que precisa interromper o processo de qualquer forma para tratar a interrupção de clock, o overhead adicional é considerado pequeno.

Uma outra abordagem para sistemas operacionais que não oferecem a chamada de sistema `profil()` é o kernel gerenciar o envio de um sinal periodicamente para o processo. Isso pode ser implementado pelo profiler utilizando chamadas de sistemas semelhantes a `settimer()`, que então realiza a mesma operação de examinar o program counter e incrementar um elemento do vetor em memória. Uma vez que este método requer que um sinal seja enviado para o espaço de usuário para cada amostra, o overhead é consideravelmente maior e o atraso envolvido no processo de sinalização torna o método bem menos preciso.

Os autores Gang Ren et al. (2010) descrevem o trabalho de profiling que o Google realiza de forma contínua em seus data centers com a consideração de que as ferramentas baseadas em amostragem propiciam níveis aceitáveis de overhead e imprecisão, sendo as únicas qualificadas para o monitoramento de desempenho no data center, um contraponto a Laurenzano et al. (2010).

2.1.3 Abordagens sem intrusividade e baixo overhead

O overhead causado pela intrusividade, a preocupação com a precisão das técnicas que se utilizam de amostragem e ausência de meios para avaliar os complexos projetos de arquiteturas levaram ao surgimento de diversas iniciativas de sistemas de profiling específicos para os sistemas embarcados, com alguns deles relatados nos projetos relacionados. Estes sistemas, que lidam com restrições de diversas naturezas ficam à margem da compatibilidade das ferramentas de profiler existentes, que se baseiam em arquiteturas mais difundidas no mercado. Por essa razão, o desenvolvimento de hardwares para profiling tornaram-se uma opção a ser considerada, principalmente pelas etapas da concepção do produto em ASIC se basearem em ambientes de desenvolvimento em plataforma reconfigurável (RIZZATTI, 2016).

2.1.4 Monitoramento

O monitoramento, exato ou estatístico, é a etapa do profiling responsável pela coleta, filtragem e armazenagem de dados de desempenho (LESSARD, 2005). Esta etapa pode ser realizada em diferentes níveis de abstração, sendo as principais baseadas em níveis de sistema,

processo, função e instrução. No nível de sistemas, o mais alto deles, observa-se o comportamento externo do sistema e/ou a interação entre sistemas diferentes. No nível de processo é analisada a comunicação entre o processo e o sistema com foco em métricas que relacionem o escalonamento, a criação e finalização do processo, mudanças de estado, as operações de I/O e a comunicação e a sincronização entre os processos. No nível de função, a preocupação se concentra na análise de métricas de funções ou procedimentos de um programa ou processo. No nível de instrução, o mais baixo analisável, o monitoramento se baseia nos eventos ocorridos no processador ou hardware dedicado durante o profiling. Nesse nível, tem-se acesso a informações detalhadas em relação às instruções executadas como as de branch, call, load/store e outras da arquitetura alvo (RAVAGNANI, 2007).

2.1.4.1 Monitoramento exato e estatístico

No monitoramento exato, o monitor de profiling atua de forma passiva, aguardando as primitivas de instrumentação e gerando métricas a partir de todos os eventos ocorridos. Tem como vantagem oferecer precisão nas medidas, mas tende a criar perturbação no comportamento do programa analisado, pois cada evento detectado gera um overhead e um número elevado deles tende a gerar uma quantidade muito grande de dados em um curto período de tempo, entretanto, o controle do predicado pode atenuar essa situação.

O monitoramento exato em software é utilizado geralmente para reunir o rastreamento de eventos de forma precisa em aplicações com baixo tempo de execução ou segmentos pequenos de programas de longo tempo de execução analisadas por meio de instrumentação dinâmica.

Esse tipo de monitoramento, concebido originalmente em abordagens de profilers em software, tem como foco a computação de propósito geral. Todavia, a lógica desse monitor pode ser desenvolvida em hardware eliminando as desvantagens da implementação em software. Nesse contexto, o hardware se baseia em um conjunto de contadores e registradores que armazenam o espaço de endereçamento de segmentos de interesse do programa do qual se deseja analisar o comportamento. Para o seu funcionamento, entretanto, precisa ser configurado antes da execução do programa. Porém, torna-se ideal para uso em sistemas embarcados, pois elimina a necessidade por instrumentação e pode ter a precisão configurada baseando-se na necessidade do projeto. Os trabalhos citados em projetos relacionados fazem uso de monitoramento exato.

O monitoramento estatístico é uma alternativa ao método anterior e é capaz de reduzir o overhead quando comparado ao método sem suporte a hardware dedicado. A quantidade de dados coletadas é inferior à da forma exata e, por essa razão, pode ser menos precisa. Neste caso, o monitor de profiling atua de forma ativa, realizando a amostragem do programa em execução em intervalos específicos. Geralmente, esse processo é amparado por recursos do sistema operacional, como chamadas de sistema ou hardwares dedicados à contagem existentes nos processadores mais populares.

2.1.4.2 Monitoramento com suporte por contadores de hardware

Integrados aos processadores modernos, as Performance Monitor Units (PMU) disponibilizam para profiling um conjunto de contadores de hardware capazes de medir diversos eventos que ocorrem no processador durante a execução de um programa. A quantidade de eventos relacionadas as métricas de performance são extensas, como o número de cache miss, pipeline stalls, instruções de acesso à memória, número de ciclos de CPU durante a execução. Contudo, essa quantidade é variável dependendo do fabricante e do modelo do processador.

Os contadores dentro da PMU podem ser utilizados de dois modos: por contagem e por amostragem. A contagem produz dados durante toda a execução sem oferecer qualquer informação a respeito de funções ou instruções que geraram os eventos. Por outro lado, por amostragem, obtém-se uma correlação dos eventos capturados em conjunto com amostras do Program Counter (PC). O sistema operacional pode configurar o processador para gerar uma interrupção quando a contagem de evento exceder um limite. Essa interrupção é tratada pelo kernel e os dados da amostragem, incluindo o valor do PC, são armazenados em um buffer circular. Este buffer sofre amostragens periódicas por um profiler em espaço de usuário, escrevendo o conteúdo em disco. Após a execução, os endereços armazenados do PC são comparados aos dos obtidos nos arquivos binários, que são traduzidos para os nomes de funções e outros símbolos.

A [Tabela 1](#) mostra a quantidade média de contadores existentes nas arquiteturas de processador mais populares. Estes são implementados em número limitado devido ao custo adicional associado ao tamanho no processador em termos de área que ocupam. Por esse motivo, o monitoramento de mais eventos que o número de contadores disponíveis exige o uso de multiplexação. Esse processo efetua o rodízio do conjunto de eventos ativos em períodos, causando overhead já que depende do sistema operacional. O custo de monitorar mais eventos que a capacidade do hardware causa, em média, 25% de overhead, sendo tão custoso quanto o monitoramento em software. No entanto, a contagem, que se utiliza, geralmente, de contadores fixos tem um overhead de até 2% (NOWAK; BITZES, 2014).

Tabela 1 – Recursos disponíveis nas PMUs

Fabricante	Quantidade de contadores
Intel (Core i7)	4 programáveis e 3 fixos
AMD (Opteron)	4 programáveis e 1 fixo
ARM (Cortex A-9)	6 programáveis

Fonte: Dados da pesquisa.

Devido aos longos pipelines e o fluxo rápido de instruções executadas nos processadores, no momento em que a interrupção é emitida, o ponteiro do Program Counter pode ter avançado e a localização da instrução, e respectiva identificação da rotina, que disparou o evento pode ser tratada incorretamente. Esse problema, conhecido por counter-skidding, obrigou os fabricantes a implementar novos recursos para evitar o tratamento da interrupção, como Precise Event-

Based Sampling (PEBS) disponível nos processadores da Intel (CORPORATION, 2010) ou Instruction-Based Sampling (IBS) nos da AMD (DRONGOWSKI; CENTER, 2007). Apoiado por um hardware adicional e disponibilizado apenas para alguns eventos, o processador armazena em um buffer em memória o valor de PC e informações adicionais, sem utilizar interrupções. Contudo, não garante 100% de precisão, é limitado a alguns eventos e presente apenas nos processadores mais recentes. Esse comportamento causa problemas especialmente em loops, onde a recorrência pode mascarar uma quantidade grande de eventos. Outro problema que afeta a precisão na contagem de eventos está relacionada à variação de frequência de operação dos processadores baseados em políticas de economia de energia ou da carga de trabalho. Assim, as métricas relacionadas a ciclo de clock podem ser comprometidas já que dependem da informação da frequência de operação do processador para ser determinado o tempo de processamento (KLEEN, 2011).

As informações de grão fino (em níveis de instrução), tornam-se difíceis de serem obtidas sem o auxílio das PMUs nas plataformas de hardware de computação de propósito geral. Além disso, análises ao nível de função permitem menor intrusividade quando se utiliza contadores em hardware. Porém, por serem oferecidos em quantidade limitada, o monitoramento de muitos eventos simultâneos deve ser evitado dada a perturbação que a multiplexagem causa, podendo comprometer as medidas de desempenho.

Nos sistemas embarcados, onde estão presentes inúmeras opções de soft processors, a presença de PMUs é limitada. Altera e Xilinx, as principais fabricantes de hardware reconfigurável, fornecem, respectivamente, os processadores Nios II e MicroBlaze. Apenas no NIOS II é presente nativamente um conjunto de contadores de desempenho, entretanto, não é fornecido suporte para uso em sistemas operacionais e seu uso se restringe à instrumentação de segmentos de código em aplicações standalone apenas (Altera Corporation, 2014).

2.1.5 Análise

A etapa seguinte do processo de profiling é analisar os dados de desempenho extraído deles informações úteis sobre o comportamento do programa. Segundo Lessard (2005), a análise envolve:

- A remoção de informações dispensáveis e reduzir os dados relevantes para um subconjunto gerenciável e apropriado;
- Realizar a computação de dados de performance brutos para as informações se tornarem acessíveis;
- Se possível relacionar a informação obtida com o código fonte do programa;
- Preparar a informação para visualização, que pode incluir sumário, informações detalhadas com grafos e gráficos.

A filtragem das informações pode ocorrer nesta etapa e assumem um caráter global baseado em todos os dados de desempenho disponíveis. Por essa razão, essa etapa pode ser opcional no monitoramento e dada a complexidade que pode assumir, permite ser realizada de duas formas: post-mortem e on-the-fly.

2.1.5.1 Análise Post-Mortem

A análise Post-Mortem não oferece riscos de causar perturbação na execução do programa porque é realizada ao fim da execução. Contudo, o processamento de uma grande quantidade de dados pode ser custosa e o ajuste dinâmico do profiling impossível de ser feita.

2.1.5.2 Análise On-the-Fly

Neste tipo de análise, a filtragem é realizada no momento da execução do programa e possibilita evitar que uma grande quantidade de dados seja coletada. Além disso, permite ao profiler ou desenvolvedor, analisar se a configuração do profiling necessita ser ajustada, focando em pontos com computação mais intensiva ou, caso o software tenha sido instrumentado, desativar por meio de predicados o monitoramento de pontos irrelevantes durante a execução. Contudo, essa análise pode ter um custo computacional elevado e gerar um considerável overhead, tornando o ajuste do profiling impraticável.

As duas formas de análise podem ser executadas concorrentemente de forma a permitir ajustes no profiling em tempo de execução e ao fim, uma análise Post-Mortem dos dados do tempo de execução.

2.1.6 Visualização

A etapa de visualização compreende o mais importante passo no processo de profiling, pois guia o desenvolvedor (ou equipe de desenvolvimento) à tomar decisões sobre como deve gerenciar os elementos de desempenho do sistema a fim de atingir um ou mais objetivos. Os dados de desempenho devem ser claros, concisos e capazes de transmitir os pontos chave sem esforço para compreensão (JAIN, 2011). Esses dados devem ser produzidos de forma a serem visualizados em tabelas, gráficos e grafos. Uma adaptação de Harris (2011) ilustra a visualização de um profiling e as considerações que as informações podem gerar.

Considere um programa e uma arquitetura de hardware hipotéticas cujo profiling é realizado a partir da amostragem de eventos a cada 10 segundos. A frequência do processador é de 100 Mhz e o resultado é exibido na [Tabela 2](#):

Observando-se a tabela é possível concluir que o pipeline estava em stall, ou seja, paralisado à espera do fim de uma operação em 10% do número de ciclos totais. E também, que o número de cache miss de dados foi de 2% e de instruções de 50% do total de ciclos de clock. A inferência da cache miss com o stall do pipeline poderia ser o alvo da análise do desenvolvedor,

Tabela 2 – Exemplo da visualização de dados gerados por profiling

Contador	Porcentagem	Valor
Stalls de Pipeline	10%	100.000
Cache miss de dados	2%	20.000
Cache miss de instrução	50%	500.000
Número total de ciclos	100%	1.000.000

Fonte: Dados da pesquisa.

Figura 5 – Resultado do profiling ao nível de instruções de um software hipotético

Fonte: Harris (2011).

que a partir das constatações pode repensar a estrutura do tamanho físico das memórias cache ou de suas políticas de substituição de dados. Porém, a análise pode partir do software e chegar-se a conclusão da necessidade de um profiling a nível de instrução que permita analisar se o uso das estruturas de dados estão bem definidas para o uso apropriado da arquitetura de hardware, por exemplo. A [Figura 5](#) indica com mais objetividade os eventos que causaram maior impacto no desempenho da arquitetura de forma gráfica.

2.2 Trabalhos relacionados

2.2.1 SnoopP

O SnoopP (SHANNON; CHOW, 2004) é uma ferramenta de profiling não intrusiva que analisa a métrica de tempo de execução de seções de código em aplicações com alvo em processadores soft core Microblaze da empresa Xilinx. O programa alvo do profiling tem de seu binário extraída todo endereçamento de funções. O número delas define a quantidade de núcleos de hardware que analisam o programa durante sua execução. Isso ocorre por meio de comparadores, que analisam o valor corrente do Program Counter (PC) e cada núcleo de

hardware verifica entre os endereços especificados em low e high se o valor está nos limites de comparação. Se estiver, um contador de 64 bits é incrementado.

Figura 6 – Arquitetura genérica da SnoopP

Fonte: Shannon e Chow (2004).

Os autores comparam o desempenho e precisão do SnoopP com a ferramenta GNU Gprof utilizando o benchmark Dhrystone. Nele é definido execuções de 100 e 1 milhão de iterações e os resultados obtidos pelo SnoopP apontam 0.06% de variação onde o Gprof não foi capaz de reproduzir qualquer informação na execução de 100 iterações. Isso ocorre porque o benchmark é executado em menos de 10 ms, valor abaixo do tempo de amostragem definido pela rotina de interrupção do Gprof.

Esse profiler tem algumas limitações, como a de só executar a análise de códigos que não tenham subfunções. Além dela, não funciona em ambiente com Sistema Operacional e necessita realizar a síntese do hardware para cada aplicação avaliada.

2.2.2 ACAD

A ferramenta de Análise, Configuração e Aquisição de Dados (ACAD) (RAVAGNANI, 2007) foi desenvolvida por Guilherme Ravagnani durante o seu mestrado no Laboratório de Computação Reconfigurável (LCR), do qual o autor deste trabalho também faz parte. A ferramenta tem como foco monitorar o barramento Avalon em sistemas projetados com processador soft core NIOS II da Altera a fim do trace e captura de eventos baseados em ciclos de clock e números de execução funções de softwares que são executados de forma standalone. Para isso, utiliza-se de instrumentação do código alvo. Tem como destaque atuar remotamente, colhendo os dados capturados por meio de uma interface serial.

Figura 7 – Arquitetura da ferramenta ACAD

Fonte: Ravagnani (2007).

2.2.3 MPPA

A MultiProcessor Profiling Architecture (MPPA) (CHEN et al., 2009) tem como objetivos monitorar eventos que ocorrem em processadores baseados no soft core LEON3. Esse profiler é composto por duas partes: uma responsável pela detecção do evento e outra para contagem. O módulo de hardware realiza diversas alterações na arquitetura do sistema, introduzindo interfaces para capturas de eventos dentro do processador e interface com barramento AHB. Os eventos disponíveis não são relatados no artigo, entretanto, os resultados se baseiam em métrica associadas a ciclos de clock, cache miss de dados e instruções. A ferramenta tem suporte implementado no sistema operacional LINUX, mas não detalha de forma clara como a interação com o profiler é realizada. Os autores do trabalho alegam que a precisão da ferramenta é de 94%.

O profiler tem limitações em não detectar overflow dos contadores e não esclarecer detalhes do projeto da ferramenta.

Figura 8 – Arquitetura da ferramenta MPPA

Fonte: Chen et al. (2009).

MATERIAIS E MÉTODOS

Este capítulo descreve as ferramentas de desenvolvimento em hardware e software e os métodos para elaboração deste projeto. No contexto do hardware é apresentado a plataforma de desenvolvimento DE2i-150. Em relação ao software, a síntese e depuração do hardware tem auxílio das EDAs (Eletronic Design Automation) da Altera (ALTERA, 2015) e da Cobham Gaisler (GAISLER, 2016).

3.1 Plataforma de Desenvolvimento

A plataforma de desenvolvimento Terasic DE2i-150 (TERASIC, 2014) foi utilizada para a implementação do projeto proposto. Este kit contempla uma solução híbrida composta por uma placa com processador Intel x86 (INTEL, 2014) e um chip de tecnologia Field-Programmable Gate Array (FPGA) da família Cyclone IV da Altera (ALTERA, 2015). A plataforma é definida por uma placa de desenvolvimento que contempla um processador Intel Atom Dual Core de 1.6 Ghz, 2GB de memória DDR3 SDRAM, 60GB de memória SSD, conectividade VGA, HDMI, USB, Wireless 802.11b/g/n, Ethernet 1Gbps/s e outros dispositivos que a permite ser operacionada como um computador pessoal, o lado PC. Na mesma placa há o circuito que contempla a FPGA Cyclone IV modelo EP4CGX150DF31C7N, que possui 149.760 Elementos Lógicos, 720 blocos de memória M9K, 6480 Kbits de memória embarcadas e 8 PLLs. Conectado a esta FPGA há memórias de 128MB (32MBx32bit) SDRAM, 4MB (1Mx32) SSRAM e 64MB (4Mx16) Flash com funcionamento em modo 16-bit. Além disso, conectividade por Ethernet 1Gbps, HSMC, VGA, RS-232 e 40 pinos de I/O. O clock gerado pelo oscilador da placa à FPGA é de 50MHz, mas permite conexão a um gerador de clock externo por meio de conector SMA. A figura [Figura 10](#) descreve uma visão geral da arquitetura da plataforma de desenvolvimento.

O lado PC e FPGA estão interligados por meio do barramento PCI-express e permitem a exploração de aplicações de alto desempenho agregando as duas tecnologias. Neste contexto, por exemplo, o FPGA poderia atuar como um acelerador de aplicações que são executadas

Figura 9 – Placa de desenvolvimento Terasic DE2i-150

Fonte: Terasic (2014).

em plataforma x86. Exemplos de como construir aplicações envolvendo as tecnologias são fornecidas pela Terasic e limitam-se ao uso do softcore NIOS II da Altera ((ALTERA, 2015)).

Figura 10 – Arquitetura da Terasic DE2i-150

Fonte: Terasic (2014).

3.2 GRLIB

O projeto LEON nasceu da necessidade da construção de um processador para uso em ambientes críticos sob supervisão da European Spacial Agency. A baixa escalabilidade inicial desse projeto levou ao surgimento da biblioteca GRLIB, que fornece toda a infraestrutura necessária, organização e suporte a ferramentas para o desenvolvimento de System-On-Chip (SOC). A GRLIB integra um conjunto de cores IP e pode ser utilizada por diferentes ferramentas Computer-Aided Design (CAD) e dispositivos FPGA de diferentes fabricantes, concentrando um conjunto de recursos que atuam na geração automática de síntese e scripts de simulação.

Todos os componentes ou core IP da biblioteca são desenvolvidos em VHDL e um único método plug&play é utilizado para configurar e conectar os componentes sem a necessidade de modificar qualquer recurso global. Este método se baseia na instância do componente no arquivo do top-level do projeto. Por padrão, os projetos de sistemas ficam armazenados em diretórios com o nome da plataforma de desenvolvimento antecedido da palavra "leon3-" em design, conforme a árvore de diretórios descrita a seguir.

Recursos para integração de plataformas de desenvolvimento compatíveis estão disponíveis no diretório boards e caso uma plataforma esteja indisponível, a Cobham-Gaisler fornece templates para a tentativa de integração à GRLIB.

A plataforma de desenvolvimento deste trabalho, que contém uma FPGA da família Cyclone IV, não tem um template compatível à GRLIB. Neste caso, a solução foi consultar o manual da GRLIB para acrescentar uma nova plataforma para ela. Esta tarefa não é trivial e o suporte pleno à plataforma não foi alcançado devido a netlist da Cyclone IV disponível não ser capaz de permitir a interligação de todos os recursos da plataforma, como o uso das memórias FLASH e SRAM que compartilham o mesmo barramento de endereçamento e exigem outras modificações na GRLIB para total compatibilidade.

3.3 GRMON

O GRMON é um monitor para depuração para o processador LEON3 e projetos SOC baseados na biblioteca GRLIB. Tem como principais recursos:

- Acesso de leitura/escrita em todos registradores do sistema e memórias;
- Incorpora um disassembler e o gerenciamento de trace buffer;
- Download e execução de aplicações;
- Gerenciamento de watchpoint e breakpoint;
- Suporte a conexão remota ao GNU debugger (GDB) para depuração em nível de código fonte;
- Suporte à comunicação por USB, seral, JTAG, Ethernet, PCI, USB, Spacewire;
- Prover um único modelo para síntese e simulação.

O GRMON é imprescindível para uso do LEON3, mas deve ser desacoplado quando se realiza medições ou se deseja extrair o máximo de performance do sistema em FPGA. Todavia, em algumas situações pode não ser possível a desacoplagem desse monitor, como a execução de aplicações standalone salva em memória volátil.

3.4 Processador LEON3

Para este projeto é utilizado o processador LEON3 desenvolvido pela Cobham Gaisler AB (GAISLER, 2016) na Suécia. Este processador, do tipo soft core de 32 bits, é uma implementação HDL, escrito em linguagem Verilog Hardware Description Language (VHDL), com conformidade reconhecida (SPARC International Inc., 2005) da arquitetura SPARC V8 (SPARC International Inc., 1992). O código fonte completo deste processador é disponível sob duas licenças, uma comercial e outra sob GNU GPL para uso gratuito e irrestrito para propósitos de pesquisa e educacionais. O LEON3 é um processador de reconhecida confiabilidade, modular, portátil, com interfaces bem definidas e suporte a diversas plataformas, incluindo LINUX, VxWorks, eCos, RTEMS etc. Tem importância destacada pelo o seu emprego em projetos de aplicações aeroespaciais críticas (GAISLER, 2003), incluindo projetos da European Space Agency (ESA) (ESA, 2016). A escolha do LEON3 neste projeto é devida a ampla capacidade de configuração do processador, documentação detalhada disponível e suporte técnico eficiente. Complementa-se, a experiência de uso deste processador por outros pesquisadores do Laboratório de Computação Reconfigurável (LCR). Taís requisitos foram fundamentais para o sucesso deste trabalho visto que foi necessário alterar aspectos avançados do processador.

Figura 11 – Diagrama interno do processador LEON3

Fonte: Gaisler (2016).

A ampla flexibilidade de configurações do LEON3 permite em sua configuração mínima contar apenas com um pipeline de inteiros, uma cache de instrução e dados. Em versões mais elaboradas pode conter unidade de gerenciamento da memória (MMU), de multiplicação e divisão de 32 bits, de soma-multiplicação de 16 bits, e interface de debug e trace buffers. Tem suporte a interfaces de extensão que podem ser utilizadas para unidade de ponto flutuante e um co-processador. Pode ser implementação com até 16 processadores, tendo desempenho máximo na versão em FPGA de 125 Mhz e em ASIC, 400 Mhz. Neste projeto, a frequência de operação do processador é de 50 Mhz.

Em sistemas baseados em processadores sobre FPGA, os mais utilizados são os produzidos pelas empresa Altera e Xilinx sendo, respectivamente de nomes, NIOS II e MicroBlaze/PicoBlaze. As duas empresas são maiores fabricantes de chips com tecnologia FPGA do mundo e oferecem ferramentas compatíveis para uso desses processadores, entretanto, ambos são produtos comerciais e de código fechado. Sendo assim, a flexibilidade de uso de qualquer um deles é bastante limitada e impediria a execução deste trabalho.

3.5 L3STAT

Era previsto no início deste projeto a construção de um módulo de estatísticas, que havia sido parcialmente desenvolvido em outra pesquisa do LCR. Todavia, em abril de 2015, a partir da versão 1.4.0-b4154 da GRLIB, foi acrescentada o bloco Intellectual Property(IP) L3STAT, uma unidade de estatística para eventos do processador LEON3 e do barramento AHB. Esta unidade assemelha-se a uma Performance Monitor Unit (PMU) de um processador de propósito

geral e tem como característica suporte a até 32 contadores de 32-bits. Tais contadores podem ser utilizados com até 16 cores e monitorar até 32 eventos simultaneamente. Os contadores podem retornar a 0 quando alcançam o seu limite e/ou serem zerados após a leitura, contribuindo para a construção de estatísticas por longos períodos.

A L3STAT fornece uma interface bem definida com o(s) processador(es) e barramento com um baixo footprint, não impactando significativamente no número de elementos lógicos utilizados em FPGA. Esse componente em hardware pode ser configurado baseado nas necessidades de desenvolvimento do projeto, permitindo a configuração da quantidade de contadores e do suporte ao monitoramento do barramento, implicando no número de elementos lógicos ocupados. Também define parâmetros de configuração e eventos e permite o desenvolvimento de eventos personalizados. Em virtude dessas características, decidiu-se a adoção deste componente em detrimento do desenvolvimento de um próprio.

3.6 Ferramentas EDA

O termo Electronic Design Automation (EDA) compreende uma categoria de ferramentas que trabalham em conjunto em um fluxo de desenvolvimento cujo objetivo é projetar circuitos eletrônicos, atuando desde esta etapa à concepção e, posteriormente, produção. Este projeto foi realizado com o suporte de ferramentas de configuração do LEON3 fornecidas pela Cobham Gaisler e a síntese por responsabilidade do software Quartus Prime 15.1 da Altera. A simulação do código VHDL do LEON3 foi feita com a ferramenta ModelSim Starter Edition 10.4b da Mentor Graphics.

3.7 Sistema Operacional Linux

A Cobham-Gaisler fornece em sua página uma versão da ferramenta Buildroot (BUILDROOT, 2016) com o kernel Linux na versão 3.10 com scripts adaptados para uso com LEON3. A Buildroot é uma ferramenta que auxilia na escolha de bibliotecas, aplicativos e opções avançadas de configuração para geração do kernel e todo ambiente para uso do Linux. É também a responsável por gerar a imagem contendo o kernel e o bootloader. Além deles, Cobham-Gaisler fornece a toolchain para compilação cruzada, que é essencial para uso da Buildroot. Por padrão, o Buildroot define o escalonador Completely Fair Schedule (CFS) para o Linux.

3.8 Formato ELF

O Executable and Linking Format (ELF) define um formato para a representação binária de um executável ou código objeto para execução em um processador. A organização é realizada por meio de um cabeçalho e seções, que armazenam informações sobre instruções, dados, tabela

de símbolos, links a bibliotecas e etc. O ELF foi desenvolvido no UNIX System Laboratories (USL) como parte da Application Binary Interface (ABI) do System V e tornou-se um padrão aceito por diversas arquiteturas de hardware e sistemas operacionais (COMMITTEE, 2001).

Figura 12 – Formato ELF

File Offset	File	Virtual Address
0	ELF header	
Program header table		
	Other information	
0x100	Text segment	0x8048100
	...	
	0x2be00 bytes	0x8073eff
0x2bf00	Data segment	0x8074f00
	...	
	0x4e00 bytes	0x8079cff
0x30d00	Other information	
	...	

Fonte: Committee (2001).

Os binários gerados neste formato podem sofrer disassemble, permitindo a análise da alocação de memória virtual do segmento de instruções para análises envolvendo tracing, debug ou profiling.

3.9 Métodos de validação

O método mais comum para o desenvolvimento de modelos sintetizáveis em VHDL é o conhecido por *ad hoc* ou *dataflow style*. Neste método um número grande de declarações concorrentes em VHDL e processos pequenos conectados através de sinais são utilizados para implementar uma funcionalidade desejada. Não há distinção entre sinais combinacionais e sequenciais. Ler e compreender códigos concebidos desta forma é difícil e são comparados aos diagramas esquemáticos desenvolvidos para projetos de circuitos eletrônicos. Dada a complexidade do projeto de um processador e falta de escalabilidade do *ad hoc*, a Cobram-Gaisler propôs um novo método para o desenvolvimento do código do LEON, chamado de "dois processos" ou *two-process*. O objetivo foi suplantar as limitações do método *ad hoc*, sendo aplicados à projetos de sistemas com um único clock de sincronismo, cuja representação é a da maioria dos projetos desenvolvidos (GAISLER, 2000). Os objetivos da técnica de "dois processos" são:

- Prover codificação uniforme ao algoritmo;
- Aumentar o nível de abstração;
- Melhorar a legibilidade;

- Identificar claramente a lógica sequencial;
- Simplificar a depuração;
- Melhorar a velocidade de simulação;
- Prover um único modelo para síntese e simulação.

Alcançar os objetivos desta técnica em VHDL implica em:

- Utilizar tipos record em declaração de sinais e portas;
- Somente utilizar dois processos por entidade;
- Utilizar declarações sequenciais em alto nível para codificar o algoritmo.

Em resumo, o modelo arquitetural em VHDL do LEON3 é projetado para ter boa legibilidade. Geralmente, um módulo contém apenas dois processos: um combinacional descrevendo toda a funcionalidade e um processo implementando registros, sendo este um processo orientado à clock. Estruturas do tipo record são utilizadas em demasia para agrupar sinais de acordo com sua funcionalidade e a comunicação entre módulos/processos são realizadas por meio dos records (CENTRE; TECHNOLOGY, 2000). Contudo, a complexidade do código fonte se dá por sua extensão. A compreensão de funcionamento dos módulos não é imediata e exige ter demasiado conhecimento de todo o processador. Assim, há uma enorme dificuldade em mapear as consequências de modificação do código do LEON3 ou blocos acoplados a ele. Sendo necessária a realização de exaustivos testes a fim de se detectar falhas na execução do processador.

Para a validação deste projeto foram realizadas:

- Simulação comportamental do processador por meio da ferramenta ModelSim;
- Depuração do código assembly com funções buscando correspondência entre etapas executadas no processador e eventos computados pelo módulo de estatística e P2L utilizando GNU Debugger (GDB) e GRMON;
- Exaustivos testes de funções de assertividade para garantir que o funcionamento com correteza do módulo de hardware;

Para o teste de assertividade se definiu um conjunto de códigos como o que calcula o fatorial de inteiros sobre duas condições: uma função executa um teste rápido e outra lento, indicando um número de argumentos diferentes ao invocar a função fatorial do número. O código é mostrado em 1 e os valores esperados de retorno são descritos na tabela Quadro 3. O código em assembly em 2 exemplifica o corpo da função a() e permite comparar o número de instruções real com o capturado pelo P2L. Esta é uma versão adaptada do teste para uso com Modelsim e

tem com objetivo descobrir se a execução desse teste transcorreria bem após as modificações no LEON3, permitindo assim analisar o comportamento do hardware durante a execução do código fonte por meio da observação dos registradores PC e nPC da CPU, contadores de eventos registrados pelo módulo de estatísticas, como também, dos registradores do P2L que fazem a leitura dos dados obtidos do processador e do L3STAT. A simulação comportamental, por ser lenta, não permite avaliar aplicações em um sistema operacional, mas valida o funcionamento de todo o sistema e do sincronismo dos blocos de hardware.

No ambiente real, os testes envolvendo o hardware sintetizado em FPGA se deram por meio da depuração do código fonte em assembly utilizando o depurador GNU GDB. A cada passo de execução do código, ou seja, a cada incremento do registrador PC, analisou-se o tipo de instrução e o evento que geraria. Assim, através do monitoramento oferecido pelo GRMON, obteve-se o status do hardware e registradores, permitindo detalhar o comportamento pormenorizado do hardware durante a execução dos testes.

Código-fonte 1: Teste de assertividade para Modelsim - Código em C

```
1 #include <stdio.h>
2
3 void __attribute__((noinline)) fatorial(int n) {
4     volatile int i = 0;
5     double f = 1;
6
7     for (i; i < n; i++) {
8         f = f*(n-i);
9     }
10    printf("\nFatorial de %d : %lf\n", n, f);
11 }
12
13 void __attribute__((noinline)) a() {
14     fatorial(4);
15 };
16
17 void __attribute__((noinline)) b() {
18     fatorial(9);
19 };
20
21 void profile();
22
23 int main() {
24     profile();
25     a();
26     b();
27
28     return 0;
29 }
```

```

30
31 void profile() {
32     // #Habilita escrita no modulo P2L
33     LEON3_BYPASS_STORE_PA(ADDR_P2L + 0x8, 0x1);
34
35     // # Descreve intervalo da função a() - dentro do executavel (ELF)
36     LEON3_BYPASS_STORE_PA(0x8000050c, 0x1);
37     LEON3_BYPASS_STORE_PA(0x80000518, 0x400019d0);
38     LEON3_BYPASS_STORE_PA(0x8000051c, 0x400019e8);
39
40     ...
41     // #Enable P2L
42     LEON3_BYPASS_STORE_PA(ADDR_P2L, 0x1);
43 }

```

Código-fonte 2: Teste de assertividade para Modelsim - Código em Assembly

```

1 ..
2 400019d0 <a>:
3 400019d0:  9d e3 bf a0      save  %sp, -96, %sp
4 400019d4:  90 10 20 04      mov   4, %o0
5 400019d8:  7f ff ff d3      call  40001924 <fatorial>
6 400019dc:  01 00 00 00      nop
7 400019e0:  81 e8 00 00      restore
8 400019e4:  81 c3 e0 08      retl
9 400019e8:  01 00 00 00      nop
10 ...

```

Quadro 3 – Resultado do teste de assertividade

Função	Arg	Retorno	# Instruções	# Chamadas	Tempo de execução
a()	4	24	7	1	1c
b()	9	362880	7	1	1c
fatorial	-	-	84c9	12eb	a7cc

IMPLEMENTAÇÃO DA ARQUITETURA PROPOSTA

Este capítulo apresenta todas as etapas da implementação da ferramenta de profiling, cujo core é baseado em hardware reconfigurável e o software, em um driver para o Sistema Operacional Linux e aplicativos para configuração e análise de dados gerados.

4.1 Fluxo de desenvolvimento do Hardware

O desenvolvimento de um novo sistema dentro do ambiente da Altera geralmente se inicia com a criação de um projeto dentro da ferramenta Quartus, que oferece uma interface amigável e recursos para adicionar e criar componentes, configurá-los e interconectá-los com objetivo da síntese de hardware. Contudo, o fluxo de desenvolvimento de um novo projeto que envolva o LEON3 ocorre de maneira distinta. O primeiro passo é conhecer a estrutura da biblioteca GRLIB e entender a sua organização. Todo esse processo não envolve uma ferramenta visual, mas sim um conjunto de scripts e explorar a estrutura de diretórios e arquivos dos quais compõe essa biblioteca. Dentro do diretório `$local1/grib/lib` encontra-se a estrutura:

```
/
├── cont r i b
├── cypr ess
├── esa
├── et h
├── f m i
├── gai sl er
├── gsi
└── libs.txt
```

¹ `$local` é uma variável de ambiente que indica um diretório arbitrário onde o usuário instalou a biblioteca GRLIB

```

|_ m i c r o n
|_ o p e n c o r e s
|_ s p a n s i o n
|_ s p w
|_ s y n p l i f y
|_ t e c h
|_ t e c h n a p
|_ w o r k

```

```

$> cat $local/glib/libs.txt
synplify technap spw eth opencor es gai sl er esa f nfi spansi on gsi

```

Essa estrutura é rígida e padroniza um conjunto de regras em scripts que atuam na compilação e simulação de projetos. Por essa razão, quando um novo componente de hardware for criado, este deve respeitar alguns critérios para ser reconhecido dentro das bibliotecas de cores IP, que estão listados em `libs.txt`. Em cada diretório descrito nesse arquivo há um outro, o `dirs.txt`, que contém os caminhos de sub-diretórios onde são armazenados os arquivos em VHDL que implementam os componentes. Além deles, em cada sub-diretório devem existir os arquivos `vhdl syn.txt` e `vhdl sim.txt` que, respectivamente, contém os nomes dos arquivos que devem ser compilados para síntese e/ou simulação e, o último, a ordem no qual os arquivos devem ser compilados.

Para o desenvolvimento de um novo componente, portanto, o desenvolvedor deve configurá-lo na GRLIB seguindo as regras descritas acima. O passo seguinte é configurar o processador LEON3 dentro do diretório top-level do projeto que está associado a plataforma de desenvolvimento. Esse processo pode ser realizado de duas formas: por meio de um ambiente gráfico ou editando os arquivos de configuração manualmente.

Figura 13 – Configuração do LEON3: principal

Fonte: Elaborada pelo autor.

A configuração gráfica, vistas nas figuras [Figura 13](#) e [Figura 14](#), é limitada e não lista todos os componentes disponíveis na GRLIB e nem os que forem criados posteriormente. Por essa razão, caso seja necessário, a instância de um novo componente deve ser feita no arquivo top-level do projeto, `leon3mp.vhd`, como exemplificado abaixo.

Figura 14 – Configuração do LEON3: barramento, memórias e periféricos

Fonte: Elaborada pelo autor.

Código-fonte 3: Exemplo: instância de um componente em leon3mp.vhd

```

341 -- ...
342 -- MEMORY ONCHIP: DUAL PORT RAM - Size 512 Kb
343 ahbdpram0: ahbdpram
344 generic map (hindex => 7, haddr => CFG_AHBRADDR, tech => CFG_MEMTECH,
               abits => 17)
345 port map ( rstn, clk, ahbsi, ahbso(7), clk, p2l_dp_addr, p2l_dp_in,
               p2l_dp_out, p2l_dp_enable, p2l_dp_write);
346 --
347 -- ...

```

A organização de parâmetros de configuração do componente com o uso de constantes, como a CFG_AHBRADDR e CFG_MEMTECH do exemplo anterior, podem ser feitas no arquivo conf i g. vhd que se localiza no diretório top-level do projeto.

A configuração do processador pela ferramenta gráfica deve ser feita sempre antes de se instanciar um componente em virtude dessa ferramenta reescrever os arquivos principais do projeto, leon3mp.vhd e conf i g. vhd, desconsiderando modificações anteriores. Depois das configurações do processador e componente estiverem prontas, a síntese do hardware pode ser iniciada. Esse processo é dependente das ferramentas fornecidas pelos fabricantes de FPGA, entretanto, a GRLIB simplifica o acesso à elas fornecendo suporte para acioná-las por linha de comando de forma simplificada.

Neste projeto é utilizada uma FPGA da Altera e, por essa razão, o uso de scripts

Figura 15 – Fluxo de desenvolvimento com ferramentas GRLIB e Altera

Fonte: Elaborada pelo autor.

da GRLIB se baseiam no ambiente definido por esse fabricante. Dessa forma, executar o comando `make quartus` no diretório top-level inicia o processo de compilação do projeto até a etapa de geração do arquivo bitstream (ver figura 15). O último passo é programar a FPGA, passo que pode também ser realizado por scripts da GRLIB, executando o comando `make quartus-program-fpga`.

Na seção a seguir será apresentado detalhes da implementação do core P2L descrevendo as modificações necessárias para extrair informações da unidade de inteiros do processador e de estatística e sua interligação com o barramento APB.

4.2 Implementação do core P2L

O core P2L é desenvolvido totalmente em VHDL e respeita os padrões definidos pela Cobham-Gaisler de se utilizar apenas 2 processos dentro da arquitetura (GAISLER, 2000). Essa metodologia simplifica o desenvolvimento, mas envolve uma curva de aprendizagem maior que o processo ad-hoc, visto que o uso dos sinais recebidos fora do componente e os manipulados internamente precisam estar garantidamente estáveis. Durante o desenvolvimento do P2L, percebeu-se que o fluxo sequencial dentro do processo combinacional tinha instabilidade não detectada pela simulação comportamental. A imprevisibilidade manifestada no processo só foi corrigida quando se reconheceu a necessidade de tratar sinais que apenas tenham sido obtidos de processos síncronos.

Figura 16 – Arquitetura simplificada do LEON3 com módulo P2L

Fonte: Elaborada pelo autor.

A figura 16 ilustra o core P2L integrado ao LEON3 e a sua dependência de componentes do sistema. Esse processo é analisado nas subseções seguintes.

4.2.1 Unidade de Inteiros

O LEON3 possui uma unidade de inteiros (UI) que implementa integralmente o conjunto de instruções estabelecido pelo SPARC V8, incluindo as de multiplicação e divisão. Nessa unidade, as interfaces de dados e instruções da cache são implementadas de forma disjunta (arquitetura Harvard). Além disso, é constituída por um pipeline de 7 estágios, ilustrado na Figura 17, que realiza as seguintes operações:

1. Fetch (FE): busca a instrução dentro da cache de instruções, mas a prossegue pelo barramento AHB caso não seja encontrada ou esta cache não tenha sido implementada na arquitetura. A instrução é válida no fim deste estágio e é armazenada dentro da UI;
2. Decode (DE): a instrução é decodificada e os endereços de Call e Branch são gerados;
3. Execute (EX): operações na unidade lógica aritmética (ULA) e de deslocamento são realizadas neste estágio;
4. Register Access (RA): os operandos são lidos a partir de arquivos de registros ou de dados internos;

Figura 17 – Estágios de Pipeline do LEON3

Fonte: Cobham-Gaisler (2016).

5. Memory (ME): estágio onde a cache de dados é lida ou escrita;
6. Exception (XC): interrupções do fluxo normal de controle (traps) e outras interrupções são tratadas;
7. Write (WR): o resultado gerado pela ULA e de deslocamento ou operações sobre cache são escritas de volta no arquivo de registro.

O registrador Program Counter (PC) na arquitetura LEON3/SPARC V8 (SPARC International Inc., 1992) é implementado até o penúltimo estágio do pipeline. Por essa razão, analisando a implementação em código fonte VHDL do processador e por meio de simulação comportamental na ferramenta Modelsim, verificou-se que os dados sobre fim da execução da instrução estão disponíveis após o estágio Memory, na entrada de Exception. É nesse ponto do processo combinacional que o valor de PC é exportado até o arquivo top-level do projeto para

Figura 18 – Análise do Pipeline - Modelsim

Fonte: Elaborada pelo autor.

ser conectado a um pino de entrada do P2L. Esse processo, por mais simples que possa parecer, envolve inúmeras camadas e estão relacionados aos arquivos ilustrados na [Figura 19](#).

Figura 19 – Arquivos editados para interface de sinal da Unidade de Inteiros

Fonte: Elaborada pelo autor.

Em \$local/lib/gaisler/leon3v3 se encontra o arquivo que implementa a unidade de inteiros do LEON3, o iu3.vhd. O [Código-fonte 4](#) mostra as modificações necessárias para captura do dados gerados pelo PC.

Código-fonte 4: Modificações na Unidade de Inteiros

```

1 entity iu3 is
2   generic (...);
3   port (
4     ...

```

```

5 +(100)  pc_ex : out std_logic_vector(31 downto 2)
6    );

8 ...
9 architecture rtl of iu3 is
10 +(109)  -- P2L: adicionando um sinal de saída
11 +(109)  signal p2l_pc_excep : std_logic_vector(31 downto 2) := zero32
        (31 downto 2);
12    ...

14 comb : process(ico, dco, rfo, r, wpr, ir, dsur, rstn, holdn, irqi,
        dbgi, fpo, cpo, tbo, tbo_2p, mulo, divo, dummy, rp, BPRED,
        BLOCKBPMISS)
15 +(2904) -- P2L: atribuindo PC em exception
16 +(2905)  l2p_pc_excep <= r.x.ctrl.pc(31 downto 2);
17 ...
18  reg : process (clk)
19  begin
20      if rising_edge(clk) then
21 +(3525)  -- P2L: repassando sinal com atraso em 1 ciclo de clock
22 +(3526)  pc_out_x <= l2p_pc_excep;
23 ...

```

As modificações nos outros arquivos se resumem a repassagem do sinal do pino `pc_out_x` até o arquivo top-level do projeto. Em virtude da análise de eventos ser feita em conjunto com a unidade de estatísticas, manter o sincronismo exigiu que o sinal de PC fosse lido com um atraso de um ciclo de clock.

4.2.2 Unidade de estatísticas

A unidade de estatísticas L3STAT é um componente responsável por capturar eventos relacionados ao processador, barramento AHB e pode ser estendida com eventos personalizados pelo desenvolvedor da arquitetura. É composta por 64 registradores de 32 bits, sendo que metade deles são responsáveis pela configuração e o restante pela armazenagem dos eventos. O quadro 4 demonstra parte dos eventos disponíveis, sendo que a descrição completa pode ser obtida no manual GRRIP Cobham-Gaisler (2015).

O quadro 5 lista de que forma esse componente é mapeado no espaço de endereçamento do barramento AHB/APB para uso em um projeto com LEON3. Esse mapeamento possibilita a configuração dos eventos nos registradores de controle a partir do endereço base somado a 0x80. A configuração segue um padrão definido delimitado por intervalos de bits conforme a Figura 20 e cuja interpretação é dada no quadro 6.

Os dados delimitados pelos bits 31:19 servem apenas de referência e não como argumentos de configuração. Indicam o número de núcleos de processamento especificados na arquitetura

Quadro 4 – Unidade de estatísticas - Eventos disponíveis

ID	Descrição do evento
Eventos relacionados ao processador:	
0x00	Instruction cache miss
0x01	Instruction MMU TLB miss
0x02	Instruction cache hold
0x03	Instruction MMU hold
0x08	Data cache (read) miss
0x09	Data MMU TLB miss
0x0A	Data cache hold
0x0B	Data MMU hold
0x10	Data write buffer hold
0x11	Total instruction count
0x12	Integer instructions
0x13	Floating-point unit instruction count
0x14	Branch prediction miss
0x15	Execution time, excluding debug mode
0x17	AHB utilization (per AHB master) (implementation dependent)
0x18	AHB utilization (total, master/CPU selection is ignored) (implementation dependent)
0x22	Integer branches
0x28	CALL instructions
0x30	Regular type 2 instructions
0x38	LOAD and STORE instructions
0x39	LOAD instructions
0x3A	STORE instructions
Eventos relacionados ao barramento AHB:	
0x40	AHB IDLE cycles
0x41	AHB BUSY cycles
0x42	AHB NON-SEQUENTIAL transfers. Filtered on CPU/AHBM if SU(1) = '1'
0x43	AHB SEQUENTIAL transfers. Filtered on CPU/AHBM if SU(1) = '1'
0x44	AHB read accesses. Filtered on CPU/AHBM if SU(1) = '1'
0x45	AHB write accesses. Filtered on CPU/AHBM if SU(1) = '1'
0x51-0x5F	Reserved

e configurações definidas na instância do componente L3STAT no top-level do projeto. Para detecção de eventos do barramento é necessário que a unidade de suporte a depuração (DSU) seja acrescentada ao projeto.

O P2L definiu o uso da unidade de estatísticas em detrimento de construir uma nova devido a integração da L3STAT com LEON3 ser nativa e também pela flexibilidade de uso e a possibilidade da criação de novos eventos. O componente em hardware do P2L se utiliza apenas da funcionalidade de detecção de eventos gerados pela L3STAT, não considerando os dados de contagem armazenados nessa unidade. Porém, esses dados foram utilizados apenas no princípio para validação do P2L.

Quadro 5 – Unidade de estatísticas - Mapeamento no barramento AHB/APB

Endereço APB (offset)	Registrador
0x00	Valor do registrador de contagem 0
0x04	Valor do registrador de contagem 1
4* n	Valor do registrador de contagem n
0x80	Registrador de controle 0
0x84	Registrador de controle 1
0x80 + (4* n)	Registrador de controle n

Figura 20 – L3STAT - Registro de controle de contadores

31	28	27	23	22	21	20	19	18	17	16	15	14	13	12	11	4	3	0
NCPU		NCNT	MC	IA	DS	EE	AE	EL	CD	SU	CL	EN				EVENT ID		CPU/AHB

Fonte: Cobham-Gaisler (2015).

Quadro 6 – Análise da função de exemplo - Definição de segmentos de código

Bits	Descrição
31: 28	(NCPU) indica o número de núcleos de CPU suportados pelo sistema
27: 23	(NCNT) indica o número de contadores implementados na L3STAT
22	(MC) se definido '1', mantém o máximo valor de contagem computado
21	(IA) contagem interna do barramento AHB. Se definido '1', indica suporte ao eventos 0x17 e 0x18
20	(DSU) indica suporte a Unidade de Suporte a Depuração do LEON. Se definido '1', permite usar os eventos 0x40 a 0x5F
19	(EE) se definido '1', indica suporte a eventos customizados de 0x60 a 0x6F
18	(AE) eventos de trace para o barramento AHB
17	(EL) se CD for definido '1', conta o tempo entre as asserções do evento. Caso contrário, os ciclos de clock envolvendo o evento
16	(CD) contagem de máxima duração (CD) - Se definido '1', contará o máximo de tempo computado para um evento definido em (EL). Só é possível ativá-lo se MC for '1'
15: 14	(SU) filtro de modo Supervisor/Usuário - "01" efetua a contagem de eventos apenas em modo supervisor; "10", apenas em modo usuário; e, outros valores, ambos. Atuante apenas nos eventos 0x0 a 0x3A
13	(CD) limpa o contador a leitura
12	(EN) habilita o contador
11: 4	ID do evento a ser contado
3: 0	Define a seleção específica de uma CPU ou um barramento AHB.

Para a utilização da unidade de estatísticas foi necessário uma nova simulação comportamental para verificar o momento da ocorrência do evento na execução da instrução em pipeline. Esta análise permitiu verificar que a geração do evento ocorre um ciclo de clock após o fim da instrução. Por essa razão, como descrito anteriormente, é realizado o atraso em um ciclo de clock dos dados obtidos da unidade de inteiros com a finalidade de obtê-los coerentemente no P2L. As formas de onda ilustradas na [Figura 21](#) exemplificam a sequência real em que os dados são

Figura 21 – Análise de formas de onda na captura de eventos relacionados à instruções

Fonte: Elaborada pelo autor.

gerados pela Unidade de Inteiros e de estatística, onde a instrução A (inst_A) e B (inst_B) geram um determinado evento sinalizado a um ciclo de clock de atraso.

Assim como na Unidade de Inteiros, a de estatística precisou ser modificada para que fossem extraídos os eventos gerados. A lógica para captura desses eventos é mais sofisticada que a realizada dentro do processador, tendo exigido compreender o funcionamento aprofundado da L3STAT. As modificações realizadas no arquivo l3stat.vhd (localizado em \$local/lib/gaisler/leon3v3), que representa o núcleo da unidade de estatística, é demonstrado no [Código-fonte 5](#).

Código-fonte 5: Modificações na Unidade de estatística

```

1 entity l3stat is
2   generic (...);
3   port (
4     ...
5 + (56)   l3s\_cnt : out std\_logic\_vector (ncnt-1 downto 0);
6 + (57)   l3s\_en  : out std\_logic
7   );

9 + (276)  comb : process(r, rc, mrc, rstn, apbi, dbg, astat)
10  ...
11 + (287)  variable l3inc : std\_logic\_vector(ncnt-1 downto 0);
12 + (288)  variable l3en  : std\_logic := '0';

14 + (471)  -- Interface para L2
15 + (472)  l3en  := '0';
16 + (473)  l3inc := zero32(ncnt-1 downto 0);
17 + (474)  for i in 0 to ncnt-1 loop
18 + (475)    if rc(i).inc = '1' and rv(i).cnt(0) /= rc(i).cnt(0) and rv(
19 + (476)      l3inc(i) := '1';
20 + (477)    end if;
21 + (478)
22 + (479)  if rc(i).en = '1' then
23 + (480)    l3en := '1'; end if;
24 + (481)  end loop;
25 + (482)
26 + (483)  l3s\_cnt <= l3inc;
27 + (484)  l3s\_en  <= l3en;

```

4.2.3 Integração de dados obtidos na Unidade de Inteiros e de Estatísticas com o P2L

As duas subseções anteriores demonstraram o funcionamento e de que forma foram modificadas para terem seus dados analisados pelo P2L. No arquivo top-level é realizada a etapa de conectividade entre os componentes, sendo exibido pelo [Código-fonte 6](#).

Código-fonte 6: Conexão de estatísticas e

```

1  -- P2L
2  p2l0: p2l -- 128 address configured with pmask 16\#FFE\#
3      generic map (pindex => 5, paddr => 5, pmask => 16\#FFF\#, nreg\
        _prog => CFG\_P2L\_SEGMENTS, nreg\_r32 => CFG\_P2L\_R32, nreg\_r64
        => CFG\_P2L\_R64, abits => 17, ncnt => CFG\_L3S\_CNT)
4      port map (rst => rstn, clk => clkm, apbi => apbi, apbo => apbo
        (5), pc\_in => pc\_excep\_s, l3s\_cnt => l3s\_cnt\_s, l3s\_en =>
        l3s\_en\_s);

```

A constante `CFG_P2L_SEGMENTS` define o número máximo de segmentos de código analisáveis e a `CFG_P2L_R32` e `CFG_P2L_R64`, respectivamente, de registradores de 32 e de 64 bits disponíveis para coleta de eventos. Esses valores tem impacto no número de elementos lógicos consumidos na síntese do hardware e por serem configuráveis, dão flexibilidade de uso em sistemas com mais ou menos recursos.

O diagrama [22](#) ilustra os blocos que compõem o P2L com a representação da memória da aplicação. O bloco de controle é o responsável por gerenciar o funcionamento do P2L, atuando na configuração, análise de dados recebidos dos outros componentes e fornecer os dados armazenados de profiling para o usuário. Além disso, gerencia o uso dos bancos de registradores de 32/64 bits e a filtragem dos eventos baseados nas seções de código. Essas seções são armazenadas na memória da aplicação, que agrega informações adicionais, como o mapeamento dos bancos de memória, o número de vezes que o segmento foi executado e caso tenha ocorrido overflow, indica em qual evento isso ocorreu.

O componente P2L ocupa um espaço de endereçamento do bloco AHB/APB (vide [7](#)) e define por padrão a análise de 64 segmentos de um programa em profiling. Para cada segmento é possível configurar a análise de até 32 eventos simultâneos, limite da unidade de estatísticas. O banco de registradores contemplam por padrão 128 registradores de 32 bits e 32 de 64 bits, mas podem ter tamanho modificados pelo usuário. O projeto do L2P se preocupou em evitar a utilização de registradores em elevada quantidade, pois causam um aumento elevado no uso de elementos lógicos da chip FPGA pelo aumento de caminhos críticos que diminuem a

Figura 22 – Diagramas de blocos do módulo P2L

Fonte: Elaborada pelo autor.

performance do hardware. Sendo assim, a lógica de uso desses registradores foi racionalizada de forma a maximizar a eficiência dos recursos disponíveis na FPGA.

Quadro 7 – Espaço de endereçamento do P2L no barramento AHB/APB

APB offset	Função do registrador
0x00	Ativar/Desativar core
0x04	Reset
0x08	Habilitar escrita
0x0c	Seleciona #segmento
0x10	Seleciona #evento
0x14	Define precisão (32/64 bits)
0x18	Segmento de código: limitante inferior (Leitura/Escrita)
0x1c	Segmento de código: limitante superior (Leitura/Escrita)
0x20	Configura registrador de evento (SETEV)
0x24	Detector de erro

Para ilustrar a lógica de configuração do hardware P2L será utilizado um exemplo de uma aplicação hipotética, que teria sido obtida pela extração do segmento de texto de um código objeto que respeita o formato ELF.

Código-fonte 7: Exemplo de uma aplicação hipotética para configuração da ferramenta de profiling

```
1 000156c4 <crc16fail>:
2 156c4: 88 10 00 08    mov    %o0, %g4
3 156c8: 86 1a 40 04    xor    %o1, %g4, %g3
```

```

4   156cc: 09 00 00 10   sethi  %hi(0x4000), %g4
5   156d0: 80 88 e0 01   btst  1, %g3
6   156d4: 86 11 20 02   or    %g4, 2, %g3
7   156d8: 83 2a 60 10   sll   %o1, 0x10, %g1
8   156dc: 92 1a 40 03   xor   %o1, %g3, %o1
9   156e0: 07 3f ff e0   sethi  %hi(0xffff8000), %g3
10  156e4: 84 0a 20 ff   and   %o0, 0xff, %g2
11  1065c: 81 c3 e0 08   retl
12  10660: 01 00 00 00   nop

14 00011d20 <main>:
15  11d20: 9d e3 b7 80   save  %sp, -2176, %sp
16  11d24: 94 10 00 19   mov   %i1, %o2
17  11d28: 92 07 a0 44   add   %fp, 0x44, %o1
18  11d2c: f0 27 a0 44   st    %i0, [ %fp + 0x44 ]
19  11d30: b0 07 bf fe   add   %fp, -2, %i0
20  11d34: 40 00 11 95   call  10570 <core\_rem>
21  11d38: 90 10 00 18   mov   %i0, %o0
22  11d3c: 40 00 0c b0   call  10598 <core\_add>
23  11d40: 90 10 20 01   mov   1, %o0
24  11d44: 81 c3 e0 08   retl
25  11d48: 01 00 00 00   nop

```

Neste exemplo, os segmentos de código que se deseja analisar da aplicação são descritos no quadro 8, os eventos e precisão de interesse, no quadro 9.

Quadro 8 – Análise da função de exemplo - Definição de segmentos de código

#	Função	limitante inferior	limitante superior
0	main()	0x11d20	0x11d48
1	crc16fail()	0x156c4	0x10660

Quadro 9 – Análise da função de exemplo - Definição de eventos e precisão

#Evento	Evento_ID	main()	crc16fail()
0	0x11	32 bits	32 bits
1	0x08	64 bits	–
2	0x28	32 bits	64 bits
3	0x39	32 bits	32 bits

Para a configuração do core P2L é necessário que o pino de escrita, wren (write enable) esteja ativo (estado lógico '1'). Após esse passo, é necessário definir a posição do segmento em memória através do pino sel_segment e enviar para os pinos seg_binf e seg_bsup os limites dos segmentos de código que se deseja analisar. Os eventos podem ser configurados um a um no módulo L3STAT por meio do endereço dos registradores de controle.

Figura 23 – Configuração de segmentos de código e eventos no P2L

A alocação do primeiro e segundo segmento resulta na ocupação de registradores na forma descrita pela figura 23. Os bancos de registradores não tem nenhum registrador de controle para ocupação, pois toda a lógica se apoia no vetor de alocação de eventos para cada segmento e pelo deslocamento orientado pelos registradores de alocação a cada iteração da configuração.

A figura 24 ilustra o fluxo completo do profiling de uma aplicação. A seguir serão descritos as etapas desse processo.

4.3 Preparação da aplicação para profiling

O estágio inicial do P2L se inicia com o usuário definindo o tipo de aplicação que será analisada: standalone ou em ambiente multi-tarefa (com Linux). Esse passo é facilitado por ferramentas desenvolvidas em shell script e uma gráfica em QT, que analisa um arquivo binário e oferece ao usuário a chance de selecionar cada uma das funções/métodos encontrados. Essa ferramenta analisa o segmento de texto do arquivo binário por meio do software objdump, filtra as informações relevantes encontradas e as disponibiliza para o usuário na etapa anterior a execução do profiling. Um fluxograma desta etapa é mostrado na figura ??.

Na etapa de configuração do profiling, o usuário deve selecionar os eventos de interesse e os segmentos da aplicação que deseja analisar. Após a seleção, descrita pela figura 26, um

Figura 24 – P2L - Fluxo completo de um profiling

Fonte: Elaborada pelo autor.

arquivo no formato XML é gerado com as especificações dos eventos e segmentos de código para o profiler P2L. Caso seja uma aplicação standalone, neste passo é gerado um executável para a plataforma do LEON3 para a configuração do profiling.

Na última etapa deste processo, o usuário deve realizar o upload do arquivo de configuração de profiling em conjunto com a aplicação para plataforma do LEON3 e executar o daemon do P2L, caso não seja uma aplicação standalone.

Por fim, na etapa 3 (27) o profiling é realizado de fato. Uma etapa sequente é a da extração dos dados armazenados do bloco P2L. Isso é feito com dois aplicativos, um para o modo standalone e outro, quando o aplicativo é executado em SO. Neste último há a geração de um arquivo em CSV.

4.4 Driver e Escalonador para Linux

Para utilização do Profiler em ambiente Linux foi desenvolvido um driver do tipo dispositivo caracter que mapeia o endereçamento de hardware do componente P2L. Assim,

Figura 25 – P2L - Etapa 1 de configuração de um profiling

Fonte: Elaborada pelo autor.

é possível que o hardware seja configurado e tenha os valores de profiling lidos com muita facilidade. A interface escolhida se baseia em mecanismo que manipulam SYSFS. Assim, a partir do momento em que o driver é carregado, torna-se possível a direta interação por meio do diretório /sys/profiling/p2l.

Para que o módulo de hardware do P2L seja possível de ser utilizado no Linux é necessário que atue apenas quando o programa de interesse esteja em execução. A abordagem para realizar esse controle levou a modificação do escalonador em uso no Sistema Operacional, o CFS. O driver quando recebe parâmetros de configuração, define o valor do PID do programa em uma variável global em espaço do kernel. Essa não é a abordagem mais segura, contudo, a mais conveniente para evitar o overhead de uma syscall. A partir do momento que o PID é configurado, uma rotina do escalonador define que ativa o hardware P2L quando o software está em execução e o desativa, quando é interrompido ou finalizado. Essas modificações foram realizadas no arquivo `linux/include/linux/sched.c` e nas rotinas `dequeue_task`, `__sched_for_k` e `__sched_scheduled`.

Para auxiliar na configuração do profiling, foi desenvolvida uma ferramenta que é operacionalizada em conjunto do programa que se deseja realizar o profiling. Essa ferramenta executa o programa alvo, mas em estado de espera, configura o driver e escalonador e só então, através do PID, inicializa o programa.

Figura 26 – P2L - Etapa 2 de configuração de um profiling

Fonte: Elaborada pelo autor.

4.5 Ferramentas de controle e relatório

O estágio final do P2L é a geração do relatório de desempenho. Esse processo é realizado por uma ferramenta que faz a leitura dos bancos de registradores por meio da interface sysfs disponibilizada pelo hardware. Esses dados, em formato CSV, são enviados ao host sendo interpretados com auxílio do MS Excel ou OpenOffice Spreadsheet. Uma ferramenta gráfica encontra-se em desenvolvimento e será disponibilizada em <http://almeidajr.com/leon>

Figura 27 – P2L - Etapa 3 de configuração de um profiling

Fonte: Elaborada pelo autor.

RESULTADOS

Neste capítulo são apresentados os resultados obtidos durante a execução de benchmarks, que foram utilizados como comparação do P2L com a ferramenta GNU Gprof. O objetivo dessa comparação é mensurar as diferenças entre as técnicas utilizadas entre as ferramentas. Houve a tentativa de se adicionar mais uma ferramenta de profiling no comparativo, a oprofile. Todavia, a falta de compatibilidade das bibliotecas da ferramenta, que não permite compilação cruzada, inviabilizou que participasse do ensaio.

5.1 Benchmarks

A plataforma de desenvolvimento explorada neste trabalho é limitada em recursos computacionais quando comparada às FPGAs mais modernas disponíveis no mercado. Por essa razão, as ferramentas utilizadas na avaliação do profiling foram escolhidas levando em conta essa restrição. Contudo, foi possível utilizar algumas das mais tradicionais para avaliações de sistemas embarcados, que são listadas abaixo:

- Coremark;
- Dhrystone;
- Mi Bench - Automotive/Susan;
- Mi Bench - Network/Dijkstra;
- SPEC CPU 2006 - Bzip2.

Além delas, houve a tentativa de avaliar uma parte do módulo de química do Brazilian Regional Atmospheric Modeling System BRAMS (INPE, 2016). Esse é um dos maiores algoritmos em estudo no Laboratório de Computação Reconfigurável, sendo composto por mais de

200 mil linhas de código, em sua maior parte, em fortran. Todavia, a complexidade em estudar e segmentar esse código em um curto espaço de tempo impossibilitou essa avaliação.

5.2 Experimentos

O P2L não utiliza instrumentação para avaliação de programas, mas necessita ser configurado com os segmentos de interesse para análise das regiões do código antes da execução do profiling. Nos benchmarks, essas regiões são as funções/métodos que os compõem. No Gprof, por outro lado, utiliza-se a instrumentação do código para conseguir extrair dados de desempenho. Essa instrumentação causa intrusividade e pode gerar mudanças no comportamento do código avaliado. Com a instrumentação, o Gprof obtém de forma relativamente precisa o número de chamadas e o tracing de execução, entretanto, necessita de amostragens a cada 10 ms para determinar o tempo de execução.

Quadro 10 – Configuração da Plataforma de Testes

Descrição	Valor
Número de núcleos de processamento	1
Unidade de Gerenciamento de Memória (MMU)	Sim
Frequência de Operação	50 MHz
Memória	128 MB SD-RAM
Cache de Instrução	4 * 4 kB
Cache de Dados	4 * 4 kB
Unidade de Ponto Flutuante	em software
Sistema Operacional	Buildroot Linux 3.10.58

Cada benchmark avaliado foi executado dez vezes e sobre esses dados foram determinados um intervalo de confiança de 95%. Os resultados foram comparados em termos de tempo total de processamento entre P2L e GProf. O tempo individual de cada função não é levado em conta neste comparativo devido a baixa resolução da amostragem do Gprof e também do tempo de execução dos algoritmos. Além disso, devido ao número de execuções das funções detectadas pelo Gprof ser igual ao do P2L em todas as execuções, este dado é omitido na comparação de ambas as ferramentas. Uma outra observação se dá pela diferença obtida com diferentes diretivas de compilação, sendo elas descritas a cada benchmark.

A configuração do ambiente de execução é descrita no [Quadro 10](#) e os eventos avaliados para cada benchmark no [Quadro 11](#). Devido a interferência que causa nos resultados, o monitor GRMON é desconectado a partir do momento em que o boot do sistema é iniciado.

5.2.1 Coremark

O Coremark é um dos benchmarks fruto de um consórcio formado por várias empresas do ramo de microeletrônica, a EEMBC (EEMBC, 1997). A preocupação desse consórcio é o de

Quadro 11 – Eventos seleccionados para os experimentos

#	Evento_ID	Descrição
1	0x00	Cache Miss de Instrução
2	0x08	Cache Miss de Dados
3	0x11	Contagem Total de Instruções
4	0x15	Tempo de execução
5	0x22	Desvios (branches)
6	0x28	Instruções de chamada (Call)
7	0x39	Instruções de LOAD
8	0x3A	Instruções de STORE

definir benchmarks capazes de oferecer dados confiáveis que simulam uso real das arquiteturas de processador avaliadas sem risco de ser manipulado, como ocorria com o Dhrystone. Assim, o Coremark é um benchmark não susceptível a otimizações de compilação. É composto por listas, strings e arrays (matrizes) nas quais as atribuições de variável são realizadas apenas em tempo de execução, não faz chamadas a bibliotecas externas e o uso de ponteiros tem acesso à memória em padrões não sequenciais. Além disso, o benchmark verifica todas as computações executadas através de cyclic redundancy check CRC de 16 bits comparando os resultados com dados armazenados em uma lista (EMBEDDED, 2011).

O Coremark é distribuído gratuitamente mediante cadastro prévio tem código compatível com ANSI C e define ao final da execução um valor de unidade Coremark/Mhz.

Quadro 12 – Coremark - Geral

Métrica	P2L	Gprof	Diferença (em módulo)
Tempo de Execução (em seg)	21.02	33.5	37.2%
Tempo do Sistema (user)	21.610	39.620	58.6%
Índice Coremark	92.63	50.46	54.47%
Função crítica	core_state_transition	core_state_transition	–
Flags de compilação	-O3 -funroll-loops -fgcse-sm -msoft-float -mcpu=v8		

Tabela 3 – Coremark - Tempo médio de execução

	P2L		Gprof	
	Tempo	Desvio	Tempo	Desvio
core_list_reverse	0.0000	0%	-	-
core_list_init	0.0002	0%	-	-
cmp_complex	0.0000	0%	-	-
core_bench_list	6.2217	0%	6.6290	12%
iterate	0.0066	0%	-	-
main	0.0016	0%	0.0050	1%
core_init_matrix	0.0000	0%	-	-
matrix_sum	0.0000	0%	-	-
matrix_mul_matrix	0.0009	0%	-	-
matrix_mul_matrix_bitextract	0.0000	0%	-	-
matrix_test	4.0476	0%	5.2090	20%
core_bench_matrix	0.0030	0%	0.0020	0%
core_init_state	0.0002	0%	0.0560	3%
core_state_transition	6.9868	0%	7.5080	15%
core_bench_state	0.7629	0%	0.8760	9%
get_seed_32	0.0000	0%	-	-
crcu8	0.0000	0%	-	-
check_data_types	0.0000	0%	-	-
crcu16	0.3150	0%	0.3090	4%
crc16	1.3721	0%	1.3970	12%
crcu32	1.3073	0%	1.4120	7%
get_time	0.0000	0%	-	-
time_in_secs	0.0002	0%	-	-
portable_init	0.0000	0%	-	-
portable_fini	0.0000	0%	-	-
stop_time	0.0000	0%	-	-
start_time	0.0000	0%	0.6400	8%
Total	21.0263	0%	33.5450	19%

Figura 28 – Coremark - Tempo de execução

Fonte: Elaborada pelo autor.

Tabela 4 – Coremark - Métricas de nível de instrução

	Cache Miss (Instrução)	Cache Miss (Dados)	Total Instruções	Tempo de execução	Branch	Call	Load	Store
core_list_init	58	23	6362	0.0422	1208	0	947	537
core_bench_list	203238.6	1613	200354620	6.2793	45444476	300000	54047086	14864196
main	2504.3	40.2	41477	0.0439	3048	8047	4057	4021
core_init_matrix	17	0	1301	0.0000	58	0	0	166
matrix_test	110520.3	2382.5	177916000	4.0452	19056000	32000	29184000	3328000
core_bench_matrix	4174.1	35.2	112000	0.0037	8000	8000	32000	0
core_init_state	32	20	5584	0.0002	1157	0	617	567
core_state_transition	38700.9	1931	242352000	6.9833	53200000	0	26832000	6896000
core_bench_state	18068	744.7	27708979.2	1.2230	2488000	2176000	6672000	2584000
get_seed_32	7	4	60	0.0000	5	0	10	0
check_data_types	0	0	2	0.0000	0	0	0	0
crc16	57891.1	0	15056538	0.3177	1408269	0	0	0
crc16	70137.8	0	67251022	1.3754	6289103	0	0	0
crc32	95978.5	0	63931630.8	0.0187	6077815	0	0	0
get_time	2	1	6	0.3355	0	0	2	0
time_in_secs	5.1	3.7	45	0.0002	0	10	5	0
portable_init	3	0	3	0.3355	0	0	0	1
portable_fini	1.2	0	2.1	0.0000	0	0	0	1
stop_time	1.9	0	7.9	0.0000	0	1	0	1
start_time	2	0	7	0.0000	0	1	0	1
Total	601633.5	6831.2	794997756.5	21.0039	133978504.1	2524070	116772889	27677653
Desvio Padrão	1.29%	3.15%	0%	0%	0%	0%	0%	0%

Tabela 5 – Dhrystone - Tempo médio de execução

	P2L		Gprof	
	P2L	Desvio P2L	Gprof	Desvio
main	2.166182198	0.000304009	3.181	0.193703668
Proc_6	0.112931516	1.10302E-05	0.128	0.028205594
Proc_7	0.142201218	7.61428E-06	0.205	0.042752518
Proc_8	0.514100242	0.000213928	0.381	0.097348172
Func_1	0.240019326	5.68961E-06	0.296	0.046951512
Func_3	0.02000017	6.3421E-08	0	0
Func_2	0.190205644	2.58623E-05	0.174	0.027968236
.div	0.034995532	0.000980492	0.174	0.071987653
ready_to_divide	0.095187568	0.000974423	0.284	0.051033758
do_single_div	0.020001664	9.58694E-07	0	0
not_really_big	0.100017072	2.91412E-06	0.244	0.073212021
divloop	0.18030686	3.03988E-05	0.489	0.092430154
end_regular_divide	0.0200354	1.11347E-05	0.168	0.04802777
got_result	0.290310384	1.71609E-05	0.249	0.069673843
total	4.126507614	0.000597749	5.973	0.366759079

5.2.2 Dhrystone

O Dhrystone é um benchmark que consiste em um conjunto de trechos de códigos padronizados envolvendo aritmética de inteiros, operações sobre strings, decisões lógicas e de acesso à memória de forma a representar as atividades de um processador nas mais comuns aplicações de computação. O resultado do Dhrystone é determinado pela medida média de tempo que o processador leva para executar iterações sobre um loop contendo uma sequência fixa de instruções. O resultado é um valor de unidade Dhrystone MIPS/MHz ou DMIPs (YORK, 2002).

Esse benchmark não executa operações sobre ponto flutuante e é muito influenciado por diretivas utilizadas no momento da compilação. Como o código é baseado em comparações numéricas, a otimização gerada pelo compilador pode causar um aparente aumento de performance DMIPs. Por essa razão, mesmo sendo utilizado como referência em vários testes de performance, tem sido preterido em razão de outras ferramentas de benchmark.

Neste ensaio, o Dhrystone foi avaliado com 500 mil iterações.

Quadro 13 – Dhrystone - Geral

Métrica	P2L	Gprof	Diferença (em módulo)
Tempo de Execução (em seg)	4.12	5.973	31%
Tempo do Sistema (user)	7.3	17.5	58.6%
Índice Dhrystone	34013	14318	57.9%
Função crítica	main	main	–
Flags de compilação	-msoft-float -O3		

Tabela 6 – Dhrystone - Métricas de nível de instrução

	Cache Miss (Instrução)	Cache Miss (Dados)	Total Instruções	Tempo de Execução	Branch	Call	Load	Store
main	7645.7	248.6	5354.2968976963	2.235762424	3903134.4	5500114	15999765.4	12000081.2
Proc_6	46.8	0	32.5414784456	0.122941398	500000	0	0	1000000
Proc_7	5.1	0	3.8283269719	0.194181672	524288	0	131072	1500000
Proc_8	34329	48.6	24039.0071436158	0.456909034	0	131072	82496	4000000
Func_1	25.7	0	18.1368749187	0.272598458	1000000	0	0	0
Func_2	2246.3	46.8	1578.3635941584	0.260632096	1000001	500000	1500000	0
.div	26.9	0	18.6324412873	0.034202648	500000	0	0	0
ready_to_divide	765.1	0	535.1825288952	0.100245052	1500000	0	0	0
not_really_big	27.7	0	19.1913594362	0.178090534	1000000	0	0	0
divloop	2966.6	0	2073.5593446743	0.180949502	2499999.2	0	0	0
end_regular_divide	0.1	0	0.1416227766	0.050003882	1209716	0	0	0
got_result	749.6	0	525.8377350412	0.113700218	500000	0	0	0
Total	48852	131416	47308.5246794951	4.200229714	14268292.6	6131189	17713347.4	18500094.2
Desvio	0.46%	0.02%	0.76%	6.29%	0.00%	0.00%	0.00%	0.00%

Figura 29 – Dhrystone - Tempo de execução

Fonte: Elaborada pelo autor.

5.2.3 MiBench

O MiBench é um benchmark formado por um conjunto de 35 aplicações divididas em seis conjuntos com cada um deles representando uma área específica do mercado de sistemas embarcados. São elas: Controle automotivo e industrial, Dispositivos de consumo, Automação de escritório, Segurança e Telecomunicações. É distribuído de forma gratuita tendo o código de todas as aplicações disponíveis em linguagem C (GUTHAUS et al., 2001).

Dentro desta suíte foram escolhidos as seguintes aplicações:

5.2.3.1 Controle Automotivo e Industrial: Susan

A Susan é um pacote de reconhecimento de imagens que foi desenvolvido para reconhecer cantos e bordas de imagens capturadas do cérebro por ressonância magnética. É um programa explorado no mundo real para melhorar a qualidade das imagens, podendo suavizar e ajustar brilho, limiares de cor e controle espacial. Há disponível neste benchmark dois arquivos de entrada: o pequeno possui uma imagem em preto e branco de um retângulo e o grande, uma imagem complexa.

Quadro 14 – Mi Bench Susan - Geral

Métrica	P2L	Gprof	Diferença (em módulo)
Tempo de Execução (em seg)	34.51	36.016	4.18%
Tempo do Sistema (user)	35.620	36.230	1.68%
Função crítica	mul_shortway	ready_to_divide	–
Flags de compilação	-O4 -lm		

5.2.3.2 Telecomunicações - Dijkstra

O algoritmo de Dijkstra é uma solução conhecida para o problema do caminho mais curto, tem complexidade computacional $O(n^2)$ e é definido por um grafo representado por uma matriz de adjacência que tem seus pares analisados em busca do menor caminho.

Quadro 15 – Mi Bench Dijkstra - Geral

Métrica	P2L	Gprof	Diferença (em módulo)
Tempo de Execução (em seg)	7.11	7.75	8.26%
Tempo do Sistema (user)	8.28	8.53	2.93%
Função crítica	dijkstra	dijkstra	–
Flags de compilação	-static -O3 -msoft-float		

Tabela 7 – Mi Bench Susan - Tempo médio de execução

	P2L	Desvio	Gprof	Desvio
enlarge	0.001375428	4.33495E-05	0.004	0.006992059
put_image	0.00000024	5.58032E-23	0.001	0.003162278
getint	0.000059328	2.05956E-07	0	0
get_image	0.00000008	1.39508E-23	0	0
susan_smoothing	7.52915249	0.008327016	0	0
main	0.005301988	4.25493E-05	10.025	0.271098588
.umul	7.854931618	0.000471807	9.263	0.388731555
mul_shortway	17.06487317	0.000384369	0	0
ready_to_divide	0.022438712	0.0002172	16.498	0.358726141
do_single_div	0	0	0.025	0.01779513
not_really_big	0.027691766	0.000126124	0	0
divloop	0.082376902	0.001406911	0.033	0.011595018
end_regular_divide	0.017112802	0.000103439	0.086	0.027568098
got_result	0.024506216	0.000101511	0.025	0.009718253
__pack_f	0.003110732	2.95993E-06	0.015	0.009718253
__unpack_f	0.003340244	1.36871E-05	0.001	0.003162278
Total	34.51305023	0.011286534	36.016	0.068345527

Figura 30 – Mi Bench Susan - Tempo de execução

Fonte: Elaborada pelo autor.

Tabela 8 – Mi Bench Susan - Métricas de nível de instrução

	Cache Miss (instrução)	Cache Miss (dados)	Total de Instruções	Tempo de execução	Branch	Call	Load	Store
susan_edges	0	0	0	0	0	0	0	0
setup_brightness_lut	0	0	0	0	0	0	0	0
enlarge	219.4	902.4	47796	0.001507	2426	5134	11473	4231
put_image	0	0	0	0.0000001	0	0	0	0
getint	17	0	192	0.0000124	35	15	0	0
get_image	0	0	0	0	0	0	0	0
susan_smoothing	18057.5	38615	340857659	7.5122632	28423109	50099779	76310874	666152
main	647.9	12.8	12483	0.0005510	521	4636	513	514
.umul	4949.7	0	367599965	7.8552640	49999601	0	0	0
mul_shortway	415.1	0	839511810	17.0593748	0	0	0	0
ready_to_divide	3615.5	0	1105900	0.02334663	331770	0	0	0
not_too_big	0	0	0	0	0	0	0	0
do_single_div	0	0	0	0	0	0	0	0
single_divloop	0	0	0	0	0	0	0	0
end_single_divloop	0	0	0	0	0	0	0	0
not_really_big	105.3	0	1192630	0.0275793	316589	0	0	0
divloop	15628.5	0	1361961	0.0361632	453987	0	0	0
end_regular_divide	98.2	0	783964	0.0168169	316589	0	0	0
got_result	3528.7	0	443386	0.0254096	110590	0	0	0
_fpadd_parts	0	0	0	0	0	0	0	0
__pack_f	312.1	3.5	106603	0.002697432	24116	0	12063	0
__unpack_f	141.9	0.6	99728.5	0.002951798	10530	0	5609	19568
Total	59464.8	39614.1	15560.2	34.5108	80787824	50109572	76378877	724453
Desvio	0.4542%	0.0001%	0.0018%	0.0016%	0.00%	0.00%	0.00%	0.00%

Figura 31 – Mi Bench Dijkstra - Tempo de execução

Fonte: Elaborada pelo autor.

Tabela 9 – Mi Bench Dijkstra - Tempo médio de execução

	P2L	Desvio	Gprof	Desvio
print_path	0.001646282	4.76067E-06	0	0
dijkstra	7.102907628	0.000458394	7.747	0.073794008
main	0.00442597	0.000155757	0.008	0.007888106
divide	0.000381008	1.60886E-07	0	0
divloop	0.000002678	8.55386E-07	0	0
end_regular_divide	0.00000282	6.7915E-07	0	0
got_result	0.00004403	3.91578E-08	0	0
Total	7.109414876	0.000515573	7.757	0.071344236

Tabela 10 – Mi Bench Dijkstra - Métricas de nível de instrução

	Cache Miss (instrução)	Cache Miss (dados)	Total de Instruções	Tempo de execução	Branch	Call	Load	Store
qcount	0	0	100	0.000002	0	0	0	32
dequeue	100	0	0	0.000002	0	0	0	0
enqueue	0	0	0	0	0	0	0	0
print_path	0	0	0	0	0	0	0	0
dijkstra	8447.7	1586702.8	267257708	7.084140066	47334386	151942	40281818	16224394
main	471.3	10.4	91917	0.00223349	10200	10202	10000	10000
divide	200	0	604	0.000162998	201	0	0	0
not_too_big	0	0	0	0	0	0	0	0
do_single_div	0	0	0	0	0	0	0	0
single_divloop	0	0	0	0	0	0	0	0
end_single_divloop	0	0	0	0	0	0	0	0
not_really_big	1	0	7	0.00000046	2	0	0	0
divloop	2	0	6	0.00000072	2	0	0	0
end_regular_divide	0	0	4	0.0000001	2	0	0	0
got_result	100	0	400	0.000044016	100	0	0	0
Total	16423.8	1586723.1	268028498.2	7.110742782	47346503	162163	40293641	16234427
Desvio	0.006848112	0.0005887132	2.05710E-08	3.7299465E-05	6.2178199E-08	0	0	0

5.2.4 SPEC CPU2006

O SPEC CPU2006 é uma suite de benchmarks cujo objetivo é fornecer uma medida de desempenho a partir de atividade de computação intensiva. As aplicações utilizam cargas de trabalho que representam problemas do mundo real, como compressão de dados, vídeo, processamento gráfico, algoritmos científicos e etc. É composto por dois grupos de benchmark: um focado em aplicações com aritmética de inteiros e outro de ponto-flutuante, o SPECint e o SPECfp, respectivamente. É um benchmark comercial que utiliza como medidas de desempenho o tempo necessário para uma única tarefa ser completada e quantas podem ser realizadas em um determinado intervalo de tempo.

Neste projeto foi utilizado o benchmark bzip2, que é baseado no algoritmo original desse compressor de dados, entretanto, realiza a compressão de arquivos que são transferidos primeiramente para memória, minimizando o uso de I/O de disco e tornando essa aplicação com característica Memory-Bound.

Quadro 16 – CPU SPEC2006 Bzip2 - Geral

Métrica	P2L	Gprof	Diferença (em módulo)
Tempo de Execução (em seg)	23.28	24.56	5.21%
Tempo do Sistema (user)	23.960	24.230	%
Função crítica	BZ2_compressBlock	BZ2_compressBlock	–
Flags de compilação	-O3 -msoft-float		

Figura 32 – CPU SPEC2006 Bzip2 - Tempo de execução

Fonte: Elaborada pelo autor.

Tabela 11 – CPU SPEC2006 Bzip2 - Tempo médio de execução

	P2L	desvio	Gprof	desvio
mainSort	6.26350625	0.003421409	6.859	0.009944289
BZ2_blockSort	0.182907386	8.51562E-06	0.193	0.006749486
copyFileName	0.00001057	3.27719E-07	0	0
myMalloc	0.000005106	3.36657E-07	0	0
snocString	0.000020142	2.11545E-07	0	0
addFlagsFromEnvVar	0.00000838	8.27312E-08	0	0
T.271	0.000001496	8.43274E-09	0	0
T.272	0.000168868	2.9306E-07	0	0
compressStream	0.000897406	6.57541E-07	0.002	0.00421637
compress	0.000058058	7.17678E-07	0	0
main	0.000102526	4.19741E-07	0	0
BZ2_bzCompressInit	0.00002723	6.48074E-08	0	0
default_bzfree	0.000008928	5.59365E-08	0	0
BZ2_bzWriteOpen	0.00001079	4.54606E-08	0	0
default_bzalloc	0.000035072	4.09357E-07	0	0
handle_compress	1.793599352	4.93949E-05	1.768	0.304149086
BZ2_bzCompress	0.001127648	1.52115E-06	0.005	0.009718253
BZ2_bzWrite	0.000952154	8.18999E-07	0.003	0.004830459
BZ2_bzwrite	0	0	0	0
BZ2_bzWriteClose64	0.000778094	2.63914E-06	0.001	0.003162278
BZ2_compressBlock	14.95025718	0.000664723	15.637	0.016363917
BZ2_hbAssignCodes	0.002092654	9.89323E-07	0.003	0.004830459
BZ2_hbMakeCodeLengths	0.08619029	1.53858E-05	0.089	0.00875595
total	23.28277156	0.003834497	24.56	0.293636207

Tabela 12 – CPU SPEC2006 Bzip2 - Métricas de nível de instrução

	Cache Miss (instrução)	Cache Miss (dados)	Total de Instruções	Tempo de execução	Branch	Call	Load	Store
mainSort	94991.3	4379524.2	765801627.6	23.282771562	93919846	0	0	94650
BZ2_blockSort	51969.4	2839803.6	113335543.6	6.26350625	8942454	0	0	50458
copyFileName	37.3	162662.2	4562705	0.182907386	1303622	0	0	32
myMalloc	16.9	0	89	0.00001057	7	0	0	17
snocString	12.9	0	41.1	0.000005106	5	0	0	13
addFlagsFromEnvVar	23.4	1.5	75.1	0.000020142	5	0	0	23
T.271	12.5	0	24.2	0.00000838	3	0	0	13
T.272	4.5	0	15.8	0.000001496	1	0	0	4
compressStream	277	3	1079	0.000168868	5	0	0	277
compress	1479.4	138	4718	0.000897406	542	0	0	1479
main	95.9	12.4	309	0.000058058	45	0	0	96
BZ2_bzCompressInit	160.2	32	977	0.000102526	129	0	0	160
default_bzfree	21	10	369	0.00002723	78	0	0	21
BZ2_bzWriteOpen	14	0	43	0.000008928	9	0	0	14
default_bzalloc	14	0	69	0.00001079	10	0	0	14
handle_compress	65.4	0	505	0.000035072	106	0	0	65
BZ2_bzCompress	9254.7	95994.5	50169440	1.793599352	6534416	0	0	9336
BZ2_bzWrite	2287.7	650.9	9128	0.001127648	2607	0	0	2296
BZ2_bzwrite	1446.9	265.7	8777	0.000952154	1703	0	0	1446
BZ2_bzWriteClose64	0	0	0	0	0	0	0	0
BZ2_compressBlock	1204.1	175.1	5225	0.000778094	676	0	0	1204
BZ2_hbAssignCodes	26166.3	1277374.2	594444796.8	14.950257178	76682024	0	0	27232
BZ2_hbMakeCodeLengths	6.4	0	63764	0.002092654	19690	0	0	5
total	289.9	2401.1	3193934	0.08619029	431709	0	0	314
Desvio	5.61%	2.01%	0.00%	0.02%	0	0	0	0

5.2.5 Considerações

O Gprof utiliza instrumentação do código fonte e amostragem para fornecer dados de desempenho. A intrusividade é percebida pelo aumento do tempo de execução dos benchmarks, entretanto, essa ferramenta oferece insights sobre os pontos de gargalo na maioria dos casos. Porém, não mais que as métricas relacionadas a tempo de execução. Por outro lado, o P2L oferece uma análise detalhada de eventos que ocorrem no processador e barramento. Além disso, não se utiliza de instrumentação, o overhead adicional na execução se mostrou desprezível e não necessita do código fonte da aplicação. O baixo desvio padrão nas medidas demonstra integridade nas medidas realizadas em hardware, tornando-se uma referência importante para ajustes da arquitetura ou do software.

CONCLUSÃO

O projeto de um sistema eficiente para aplicações que necessitem de alto desempenho requer a escolha de componentes de hardware e software para se definir uma arquitetura apropriada. Para esta decisão, o conhecimento detalhado das exigências computacionais de uma aplicação é obrigatório. O profiling pode auxiliar nessa missão, desde que forneça dados precisos da aplicação em execução sobre unidades funcionais que compõem uma arquitetura.

A oferta de ferramentas de profiling é grande. Porém, poucas delas dão suporte a sistemas embarcados e quando existem, estão restritas, em sua maioria, a projetos acadêmicos ou confinadas aos ambientes de teste, vide o [Tabela 13](#). A GNU Gprof, por essa razão, é bastante utilizada nesse contexto dada a sua fácil utilização e suporte a inúmeras arquiteturas. Contudo, dada a realidade na qual foi concebida, não se mostra adequada para os sistemas embarcados avançados.

Essa necessidade motivou o desenvolvimento deste trabalho. Assim, um dos objetivos para o desenvolvimento do P2L foi o de escolher um processador robusto com condições de evolução do projeto. O LEON3 oferece elevada capacidade computacional, conta com o suporte de ferramentas que vem sendo construídas há décadas. Além disso, está preparado para Symmetric multiprocessing (SMP), ampla quantidade de aplicações open source e, recentemente, disponibilizou uma unidade de aquisição de dados de performance, a L3STAT, que possibilita ser expandida para medição de novos eventos no futuro. Por fim, a utilidade de um processador soft core para a computação reconfigurável é considerável.

Por fim, o P2L foi construído nessa plataforma visando ser não intrusivo por utilizar o monitoramento em hardware, código otimizado de driver e poucas modificações no Linux, aspectos que contribuíram para um baixo overhead. Conforme mostrado no capítulo de resultados, métricas relacionadas a ciclos de instrução foram obtidas com precisão. Todavia, muitas otimizações ainda são possíveis. Mesmo assim, esperamos ter contribuído de forma significativa nesse área de profiling, que é tão promissora.

A documentação desta ferramenta e todos os códigos serão disponibilizados em <http://almeidajr.com/leon>.

6.0.6 Trabalhos Futuros

Alguns dos trabalhos futuros que podem ser realizados para continuidade desta pesquisa são:

- Implementar suporte a múltiplos núcleos de processamento;
- Tornar esta ferramenta apta ao ensino na disciplina de Co-Projeto de Hardware/Software;
- Desenvolvimento de um daemon que possibilite a configuração automática e gerencie dinamicamente o profiling das aplicações;
- Suporte ao R na geração de gráficos;

Tabela 13 – Ferramentas de profiling

Característica	Gprof	SimpleScalar	VTune	Valgrind	ARM Profiler	SnoopP	MPAA	P2L
Instrumentação	+	-	+	+	-	+	-	-
Amostragem	+	-	+	+	+	-	-	-
Nível de ciclos de clock	-	+	+	-	+	+	+	+
Nível de função	+	-	+	+	+	+	+	+
Nível de linha	+	+	+	+	+	-	-	+
Baseada em Hardware	-	-	-	-	-	+	+	+
Acesso à Memória	-	+	+	+	+	-	+	+
Callgraph	+	-	+	+	+	-	-	-
Baseada em hardware	-	-	-	-	-	+	+	+
Arquitetura	Diversas	Sintética	x86	x86, ARM, MIPS e PPC	ARM	Microblaze (Xilinx)	LEON3	LEON3

REFERÊNCIAS

ALTERA. Altera - Quartus Prime Design Software Overview. 2015. Disponível em: <<https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html>>. Acesso em: 02/01/2016. Citado 2 vezes nas páginas 47 e 48.

Altera Corporation. Embedded Peripheral IP User Guide. 2014. 336 p. Acesso em: 10/11/2015. Citado na página 40.

BUCK, B. An API for Runtime Code Patching. International Journal of High Performance Computing Applications, v. 14, n. 4, p. 317–329, 2000. ISSN 1094-3420. Citado na página 36.

BUILDROOT. Tool to generate embedded Linux. 2016. Disponível em: <<https://buildroot.org/>>. Acesso em: 11/12/2015. Citado na página 52.

CENTRE, E. E. S. R.; TECHNOLOGY. LEON/AMBA VHDL model description. Power, p. 1–58, 2000. Disponível em: <<http://www.iuma.ulpgc.es/~nunez/clases-micros-para-com/sparc-leon-esa-2.2/leonvhdl.p>>. Acesso em: 07/08/2015. Citado na página 54.

CHEN, P. H.; KING, C. T.; CHANG, Y. Y.; TSENG, S. Y. Multiprocessor system-on-chip profiling architecture: Design and implementation. Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS, p. 519–526, 2009. ISSN 15219097. Citado 2 vezes nas páginas 44 e 45.

COBHAM-GAISLER. GRLIB IP Core User's Manual. [s.n.], 2015. 828–835 p. Disponível em: <<http://www.gaisler.com/products/grlib/grip.pdf>>. Acesso em: 14/11/2015. Citado 3 vezes nas páginas 29, 64 e 66.

_____. GRLIB IP Library User's Manual. [s.n.], 2016. 1–76 p. Disponível em: <<http://www.gaisler.com/products/grlib/grlib.pdf>>. Acesso em: 14/11/2015. Citado na página 62.

COMMITTEE, T. I. S. Executable and Linkable Format (ELF). Specification, Unix System Laboratories, 2001. Disponível em: <[http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Executable+and+Linkable+Format+\(+E](http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Executable+and+Linkable+Format+(+E)>. Citado na página 53.

CORPORATION, I. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A : System Programming Guide, Part 1. System, v. 3, n. 253666, p. 1807–1814, 2010. ISSN 15222594. Disponível em: <<http://www.intel.com/products/processor/manuals/>>. Citado na página 40.

DRONGOWSKI, P.; CENTER, B. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. Advanced Micro Devices, Inc, p. 1–14, 2007. Citado na página 40.

DUFOUR, F.; RADETZKI, A. M. Seminar “Embedded Systems” - Hardware / Software Partitioning. Stuttgart: [s.n.], 2006. Disponível em: <www.iti.uni-stuttgart.de/~radetzki/Seminar06/05/_repor>. Citado 2 vezes nas páginas 25 e 26.

EEMBC. About. 1997. Disponível em: <<http://www.eembc.org/about/index.php>>. Acesso em: 01/02/2016. Citado na página 78.

EMBEDDED. CoreMark - A-realistic-way-to-benchmark-CPU-performance. 2011. Disponível em: <<http://www.embedded.com/design/other/4212735/CoreMark--A-realistic-way-to-benchmark-CPU-performance>>. Acesso em: 14/02/2016. Citado na página 79.

ESA. European Space Agency - Microprocessor LEON. 2016. Disponível em: <http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Microprocessors>. Acesso em: 10/02/2016. Citado na página 50.

Forschungszentrum Jülich GmbH. Configurable Binary Instrumenter (Cobi). 2016. Disponível em: <<http://www.scalasca.org/software/cobi>>. Acesso em: 10/03/2016. Citado na página 36.

GAISLER. Cobham Gaisler AB. 2016. Disponível em: <<http://www.gaisler.com/>>. Acesso em: 01/02/2016. Citado 4 vezes nas páginas 28, 47, 50 e 51.

GAISLER, J. A structured VHDL design method. p. 41–50, 2000. Acesso em: 22/06/2015. Citado 2 vezes nas páginas 53 e 60.

_____. LEON SPARC Processor: The past, present and future. 2003. Disponível em: <<http://www.gaisler.com>>. Citado na página 50.

Gang Ren; TUNE, E.; Tipp Moseley; Yixin Shi; Silvius Rus; HUNDT, R. Google-Wide Profiling: a continuous profiling infrastructure for data centers. p. 65–79, 2010. Citado na página 37.

GEQUANG, P.; Van Hung, D.; JIFENG, H.; YI, W. An Optimal Approach to Hardware/Software Partitioning for Synchronous Model. [S.l.]: Springer Berlin Heidelberg, 2004. 363–381 p. ISBN 978-3-540-21377-2. Citado na página 26.

GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. gprof: a Call Graph Execution Profiler. ACM SIGPLAN Notices, v. 17, n. 6, p. 120–126, 1982. ISSN 03621340. Disponível em: <<http://dl.acm.org/citation.cfm?id=806987&delimiter=026E30F&nhttp://portal.acm.org/citation.cfm?doid=872726.806987>>. Citado na página 36.

GUTHAUS, M.; RINGENBERG, J.; ERNST, D.; AUSTIN, T.; MUDGE, T.; BROWN, R. MiBench: A free, commercially representative embedded benchmark suite. Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), p. 3–14, 2001. Citado na página 86.

HARRIS, I. Using the Performance Monitor Unit on the e200z760n3 Power Architecture Core. 2011. 1–10 p. Citado 2 vezes nas páginas 41 e 42.

INPE. 2016. Disponível em: <<http://brams.cptec.inpe.br/>>. Citado na página 77.

INTEL. Intel® Atom™ Processor N2600 (1M Cache, 1.6 GHz). 2014. Disponível em: <http://ark.intel.com/products/58916/Intel-Atom-Processor-N2600-1M-Cache-1{\\}_6->. Acesso em: 20/03/2016. Citado na página 47.

JAIN, R. The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling. In: . [S.l.: s.n.], 2011. p. 120–122. ISBN 978-0-471-50336-1. Citado na página 41.

KLEEN, A. Modern CPU Performance Analysis on Linux. Yokohama: [s.n.], 2011. Disponível em: <<http://halobates.de/modern-pmus-yokohama.pdf>>. Citado na página 40.

LANCASTER, A. J.; CHAMBERLAIN, R.; BUHLER, J.; LANCASTER, J. M. Low-Impact Profiling of Streaming, Heterogeneous Applications. Tese (Doutorado) — Washington University, 2011. Citado na página 34.

LAURENZANO, M. A.; TIKIR, M. M.; CARRINGTON, L.; SNAVELY, A. PEBIL: Efficient static binary instrumentation for linux. ISPASS 2010 - IEEE International Symposium on Performance Analysis of Systems and Software, p. 175–183, 2010. Citado 2 vezes nas páginas 36 e 37.

LEE, E. A.; SESHIA, S. A. Introduction to Embedded Systems—A Cyber-Physical Systems Approach. [s.n.], 2015. 589 p. ISBN 978-1-312-42740-2. Disponível em: <<http://leeseshia.org/>>. Citado na página 25.

LESSARD, J. Profiling concurrent programs using hardware counters. Tese (Doutorado) — University of Waterloo, 2005. Disponível em: <<http://plg.uwaterloo.ca/usystem/theses/LessardThesis.pdf>>. Citado 3 vezes nas páginas 34, 37 e 40.

LUECK, G.; PATIL, H.; PEREIRA, C. PinADX - An Interface for Customizable Debugging with Dynamic Instrumentation. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization - CHO '12. New York, New York, USA: ACM Press, 2012. p. 114. ISBN 9781450312066. Citado na página 36.

MARWEDEL, P. Embedded System Design. [s.n.], 2006. 258 p. ISBN 9780387300870. Disponível em: <<http://books.google.com.br/books?id=DZEoizXV1swC>>. Citado na página 25.

METZ, E.; LENCEVICIUS, R.; GONZALEZ, T. F. Performance data collection using a hybrid approach. ACM SIGSOFT Software Engineering Notes, v. 30, n. 5, p. 126, 2005. ISSN 01635948. Disponível em: <<http://portal.acm.org/citation.cfm?doi=1095430.1081729>>. Citado na página 36.

MILLER, B.; CALLAGHAN, M. The Paradyn parallel performance measurement tool. Computer, p. 1–23, 1995. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=471178>. Citado na página 34.

MISHRA, A.; GARG, K.; ASATI, A. R.; RAJU, K. S. Hardware software co-design using profiling and clustering. Proceedings - 2012 International Conference on Communication, Information and Computing Technology, ICCICT 2012, p. 1–4, 2012. Citado 2 vezes nas páginas 26 e 33.

NETHERCOTE, N.; SEWARD, J. Valgrind - a framework for heavyweight dynamic binary instrumentation. Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07, p. 89, 2007. ISSN 03621340. Disponível em: <<http://portal.acm.org/citation.cfm?doi=1250734.1250746>>. Citado na página 36.

NOWAK, A.; BITZES, G. The overhead of profiling using PMU hardware counters. n. CERN Openlab Report 2014, 2014. Disponível em: <<http://dx.doi.org/10.5281/zenodo.10800>>. Citado na página 39.

PATEL, R.; RAJAWAT, A. A Survey of Embedded Software Profiling Methodologies. International Journal of Embedded Systems and Applications, v. 1, n. 2, p. 19–40, 2011. ISSN 18395171. Disponível em: <<http://arxiv.org/abs/1312.2949>>. Citado na página 34.

RAVAGNANI, G. Projeto de um sistema para monitoramento de hardware/software on-chip baseado em computação reconfigurável. Tese (Doutorado) — Universidade de São Paulo, 2007. Disponível em: <<http://www.teses.usp.br/teses/disponiveis/55/55134/tde-21062007-165154/>>. Citado 3 vezes nas páginas 38, 43 e 44.

RIZZATTI, L. Hardware emulation to debug embedded system software. 2016. Disponível em: <<http://embedded-computing.com/articles/hardware-emulation-to-debug-embedded-system-software/>>. Citado na página 37.

SCHAUMONT, P. A Practical Introduction to Hardware/Software Codesign. Springer US, 2012. 17 p. (SpringerLink : B{ü}cher). ISBN 9781461437376. Disponível em: <<https://books.google.com.br/books?id=dgTx92SrFo0C>>. Citado na página 25.

SHANKAR, K. V. Non-Intrusive Dynamic Application Profiling And Error Detection For Embedded Systems In the Graduate College. Tese (MASTER OF SCIENCE) — THE UNIVERSITY OF ARIZONA, 2010. Citado na página 36.

SHANNON, L.; CHOW, P. Using reconfigurability to achieve real-time profiling for hardware/software codesign. Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, p. 190–199, 2004. Disponível em: <[http://dl.acm.org/citation.cfm?id=968308%delimater"026E30F\\$nhhttp://portal.acm.org/citation.cfm?doid=968280.968308](http://dl.acm.org/citation.cfm?id=968308%delimater)>. Citado 2 vezes nas páginas 42 e 43.

SPARC International Inc. The SPARC Architecture Manual V8. p. 295, 1992. Citado 2 vezes nas páginas 50 e 62.

_____. SPARC V8 - Declaration of Compliance. 2005. Disponível em: <<http://www.gaisler.com/images/leon3val.gif>>. Citado na página 50.

SRIVASTAVA, A.; EUSTACE, A. ATOM: A System for Building Customized Program Analysis Tools. p. 196–205, 1994. ISSN 03621340. Citado na página 33.

SURESH, D. C.; VAHID, F.; VILLARREAL, J. R.; STITT, G. Profiling Tools for Hardware / Software Partitioning of Embedded Applications. n. June, 2003. Citado na página 36.

TERASIC. DE2i-150 FPGA Development Kit. 2014. Disponível em: <<http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=139&No=529>>. Citado 2 vezes nas páginas 47 e 48.

TONG, J. G.; KHALID, M. A. S. Profiling tools for FPGA-based embedded systems: Survey and quantitative comparison. Journal of Computers, v. 3, n. 6, p. 1–14, 2008. ISSN 1796203X. Citado na página 34.

WHITE, E. Speed. In: Making Embedded Systems. [S.l.]: O'Reilly, 2011. cap. Doing More, p. 229–243. ISBN 978-1-449-30214-6. Citado na página 33.

YORK, B. R. Benchmarking in context : Dhrystone. n. March, p. 1–7, 2002. Citado na página 83.