Karila Palma Silva

ANÁLISE DE VALOR PARA DETERMINAÇÃO DO TEMPO DE EXECUÇÃO NO PIOR CASO (WCET) DE TAREFAS EM SISTEMAS DE TEMPO REAL

Dissertação submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas para a obtenção do Grau de Mestre em Engenharia de Automação e Sistemas.
Orientador: Prof. Rômulo Silva de Oliveira, Dr. Eng.

Florianópolis

2015

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Silva, Karila Palma
Análise de valor para determinação do tempo de execução no pior caso (WCET) de tarefas em sistemas de tempo real / Karila Palma Silva ; orientador, Rômulo Silva de Oliveira - Florianópolis, SC, 2015.
99 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Inclui referências

1. Engenharia de Automação e Sistemas. 2. Sistemas de tempo real. 3. WCET. 4. Análise estática. 5. Análise de valor. I. Oliveira, Rômulo Silva de. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas. III. Título.

ANÁLISE DE VALOR PARA DETERMINAÇÃO DO TEMPO DE EXECUÇÃO NO PIOR CASO (WCET) DE TAREFAS EM SISTEMAS DE TEMPO REAL

Karila Palma Silva

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisistos para a obtenção do grau de Mestre em Engenharia de Automação e Sistemas.

Prof. Rômulo Silva de Oliveira, Dr. Eng.
Orientador

Prof. Rômulo Silva de Oliveira, Dr. Eng.
Coordenador do Programa de Pós-Graduação
em Engenharia de Automação e Sistemas

Banca Examinadora:

Prof. Rômulo Silva de Oliveira, Dr. Eng.
Presidente

Prof. Fabiano Hessel, Dr. - PUCRS/Porto Alegre

Prof. Marcelo Ricardo Stemmer, Dr. - UFSC/Florianópolis

Prof. Patricia Della Méa Plentz, Dra. - UFSC/Florianópolis

Este trabalho é dedicado aos meus pais e minha irmã. Que todo o meu esforço posso retribuir a dedicação e confiança que sempre depositaram em mim.

AGRADECIMENTOS

Primeiramente agradeço Deus, que sempre iluminou meu caminho, dando-me força, sabedoria, saúde e proteção.

Agradeço aos meus pais Carlos Antônio Tanajura da Silva e Marcia Terezinha Palma Silva pelo auxílio em todas as horas, sempre me apoiando nas decisões e incentivando buscar meus sonhos, além de me proporcionar uma educação de boa qualidade. E a minha irmã Kauana Palma Silva, que esteve ao meu lado durante todo o período do mestrado transmitindo fé, amor, alegria, força, determinação, paciência, confiança para buscar meus sonhos e objetivos.

Agradeço ao professor Rômulo Silva de Oliveira, que orientou a execução deste trabalho, pela colaboração, dedicação, incentivo, paciência e por suas sugestões imensamente construtivas.

À Universidade Federal de Santa Catarina, principalmente ao Programa de Pós-Graduação de Engenharia de Automação e Sistemas, por oferecer um mestrado com docentes qualificados.

À CAPES pelo apoio financeiro possibilitando a realização deste trabalho e contribuindo para um melhor aprendizado.

E a todos que colaboraram diretamente e indiretamente para que fosse possível a realização deste trabalho. Os amigos do LTIC: Renan, Andreu, Luís, Sydnei, Carlos, Leonardo Martinz, Leonardo, Fernando, Alexandre.

A tarefa não é tanto ver aquilo que ninguém viu, mas pensar o que ninguém pensou sobre aquilo que todo mundo vê.
Arthur Schopenhauer

O limite de nossos avanços é a nossa própria criatividade.
(Autor desconhecido)

RESUMO

A utilização de sistemas computacionais na sociedade tem se expandido e as aplicações com requisitos de tempo real são mais comuns, variando em relação à complexidade e às necessidades de garantia no atendimento de restrições temporais (*deadlines*). Uma propriedade importante na definição do comportamento temporal de uma tarefa é o tempo de computação, que é o tempo necessário para a execução completa da tarefa. Um dos grandes problemas de obtê-lo está ligado à análise da microarquitetura do processador. Considerando um processador que possui memória de dados com latência variada, é necessário a análise de valor para identificar a região de memória que a instrução acessa (memória principal ou *ScrathPad Memory*), para que o pior tempo de execução dos programas não seja consideravelmente superestimado. O objetivo deste trabalho é usar a análise de valor para determinar o tempo correto de acesso à memória, através da identificação da região de memória que cada instrução acessa, com a finalidade de obter um limite superior do *WCET* menos pessimista.

Palavras-chave: Sistemas de tempo real. *Worst-Case Execution Time*. Análise estática. Análise de valor.

ABSTRACT

The use of computer systems in our society has expanded and applications with real-time requirements are getting more usual, varying in relation to the complexity and the necessity of guaranting deadlines. An important restriction in defining the temporal behavior of a task is the computation time, i.e., the time necessary to complete the task. A major problem in obtaining WCET is the processor microarchitecture analysis. Considering a processor with a data memory that has varying latency, value analysis is necessary to identify the memory region that each instruction accesses (main memory or ScrathPad Memory), so the worst execution time of programs are not considerably overestimated. The objective of this work is to use value analysis to obtain the correct memory access time by identifying the region of memory each instruction accesses, obtaining WCET upper bounds that are less pessimistic.

Keywords: Real-time. Worst-Case Execution Time. Static analysis. Value analysis.

LISTA DE FIGURAS

Figura 1 Análise temporal de sistema 26
Figura 2 Representação de condicionais em *CFG* 32
Figura 3 Análise estática de código 33
Figura 4 Infraestrutura de experimentação existente 40
Figura 5 Diagrama do processador para tempo real deste trabalho 43
Figura 6 Exemplo de composição de tempo de dois blocos básicos sucessivos 51
Figura 7 Resultado do IPET 54
Figura 8 CFG com as instruções de máquina 60
Figura 9 Classificação das instruções de acesso à memória 62
Figura 10 CFG com as instruções de máquina do programa *swap*. 65
Figura 11 CFG com as instruções de máquina do programa *array-pointer* 69
Figura 12 Validação do BB: 0 do *benchmark swap* 78
Figura 13 Validação do BB: 1 do *benchmark swap* 79
Figura 14 Validação do BB: 2 do *benchmark swap* 80
Figura 15 Comparativo do simulador com a análise *WCET* 83
Figura 16 Comparativo do simulador com a análise *WCET* 83
Figura 17 Comparativo da análise *WCET* com e sem a análise de valor 85
Figura 18 Comparativo da análise *WCET* com e sem a análise de valor 85
Figura 19 Comparativo do simulador com a análise *WCET* 87
Figura 20 Comparativo do simulador com a análise *WCET* 87
Figura 21 Comparativo do simulador com as análises *WCET* 88
Figura 22 Comparativo do simulador com as análises *WCET* 89
Figura 23 Analise temporal de sistema 93

LISTA DE TABELAS

Tabela 1 Especificação do processador 42
Tabela 2 Formato das instruções 45
Tabela 3 Representação das instruções 45
Tabela 4 Instruções de acesso à memória 46
Tabela 5 Especificação das instruções de memória 46
Tabela 6 Endereçamento de memória e sua penalidade 47
Tabela 7 Classificação da memória 61
Tabela 8 Propagação das constantes do programa *swap* 66
Tabela 9 Todas as instruções do programa *swap* 66
Tabela 10 Classificação das instruções do programa *swap* 67
Tabela 11 Propagação das constantes do programa *array-pointer* . 70
Tabela 12 Todas as instruções do programa *array-pointer* 70
Tabela 13 Classificação das instruções do *array-pointer* 71
Tabela 14 Instrução de escrita na memória do programa *swap* 72
Tabela 15 Instrução de leitura na memória do programa *swap* 73
Tabela 16 Instrução de escrita na memória do programa *array-pointer* 73
Tabela 17 Instrução de leitura na memória do programa *array-pointer* 73
Tabela 18 Considerando a implementação da memória para o *array-pointer* 74
Tabela 19 Análise do *WCET* 82
Tabela 20 Análise do *WCET* com e sem análise de valor 84
Tabela 21 Análise do *WCET* considerando o conteúdo da memória 86
Tabela 22 Número de blocos básicos e instruções 90

LISTA DE ABREVIATURAS E SIGLAS

AV	Análise de valor
BB	Bloco Básico
BCET	*Best-Case Execution Time* – melhor caso tempo de execução
CFG	*Control Flow Graph* – grafo de controle de fluxo
CPI	*Cycles Per Instruction* – ciclos por instrução
CPU	*Central Processing Unit* – processador
DI	*Instruction Decode* – estágio de decodificação de instrução
DMA	*Direct Memory Access* – acesso direto à memória
DRAM	*Dynamic Random Access Memory* – memória dinâmica de acesso aleatório
EX	*Instruction Execution* – estágio de execução de instrução
IF	*Instruction Fetch* – estágio de busca de instrução
ILP	*Integer Linear Programming* – programação linear inteira
IPET	*Implicit Path Enumeration Technique* – técnica de enumeração implícita de caminhos
ISA	*Instruction Set Architecture* – arquitetura de conjunto de instruções
LT	Estágio de Leitura dos registradores
MEM	Estágio de acesso à Memória
MIPS	*Microprocessor without Interlocked Pipeline Stages* – microprocessador sem estágios encaixado pipeline
RA	Endereço de retorno
RAM	*Random Access Memory* – memória de acesso aleatório
REG	Estágio de Escrita dos Registradores
RTL	*Register-transfer Level Design*
SRAM	*Static Random Access Memory* – memória estática de acesso aleatório
ULA	Unidade Lógica e Aritmética
WB	*Write-back* – gravação dos resultados
WCET	*Worst-Case Execution Time* – pior tempo de computação de uma tarefa

LISTA DE SÍMBOLOS

C	Pior tempo de computação ($WCET$) de uma tarefa
D	*Deadline* de uma tarefa
E	Aresta que conectam dois vértices
G	Grafo direcionado
lb_i	Limite superior de iterações do laço
n_c	Número de faltas de *cache*
n_m	Número de acessos à memória
T	Intervalo mínimo de chegadas ou período de uma tarefa
t_{bb}	Tempo de execução dos blocos básicos
t_c	Tempo de penalidade da *cache*
t_{entity}	Coeficiente temporal
t_p	Tempo do *pipeline* para execução das instruções do bloco básico
t_m	Tempo de acesso à memória
V	Vértice que representam os blocos básicos
x_{entity}	Variável de contagem

SUMÁRIO

1 INTRODUÇÃO 25
1.1 CONCEITOS BÁSICOS 27
1.2 CONTEXTO DO TRABALHO 28
1.3 OBJETIVO 29
1.4 ORGANIZAÇÃO DO TEXTO 29
2 ANÁLISE DO TEMPO DE EXECUÇÃO NO PIOR CASO (*WCET*) 31
2.1 CONSIDERAÇÕES INICIAIS 31
2.2 ANÁLISE TEMPORAL ESTÁTICA DE CÓDIGO 31
2.2.1 Análise de valor 33
2.2.2 Análise de fluxo de controle 34
2.2.3 Análise da microarquitetura 34
2.2.4 Obtenção do pior caminho 35
2.3 ANÁLISE DE VALOR 36
2.3.1 Trabalhos relacionados 37
2.4 CONSIDERAÇÕES FINAIS 38
3 CONTEXTO DO TRABALHO: PROCESSADOR E A FERRAMENTA DE ANÁLISE *WCET* 39
3.1 CONSIDERAÇÕES INICIAIS 39
3.2 CONTEXTO DO TRABALHO E ONDE ESTARÁ A CONTRIBUIÇÃO 39
3.3 PROCESSADOR 41
3.4 OPERANDOS DO *HARDWARE* 44
3.5 INSTRUÇÕES DE ACESSO À MEMÓRIA 45
3.6 ENDEREÇAMENTO DE MEMÓRIA E SUA PENALIDADE 47
3.7 ANÁLISE *WCET* 47
3.7.1 Análise da memória cache de instruções 49
3.7.2 Análise do pipeline 50
3.7.3 Obtenção do pior caminho 51
3.8 CONSIDERAÇÕES FINAIS 54
4 EMPREGO DA ANÁLISE DE VALOR NO CONTEXTO DO TRABALHO 57
4.1 CONSIDERAÇÕES INICIAIS 57
4.2 IMPLEMENTAÇÃO 57
4.3 EXEMPLOS 63
4.3.1 Classificação das instruções de memória do programa *swap* 64

4.3.2 Classificação das instruções de memória do programa *array-pointer* 68
4.3.3 Inclusão do conteúdo da memória na análise 71
4.4 CONSIDERAÇÕES FINAIS 74
5 VALIDAÇÃO E AVALIAÇÃO DOS RESULTADOS DA ANÁLISE DE VALOR 77
5.1 CONSIDERAÇÕES INICIAIS 77
5.2 VALIDAÇÃO DOS RESULTADOS DA ANÁLISE DE VALOR 77
5.3 AVALIAÇÃO DOS RESULTADOS DA ANÁLISE DE VALOR 81
5.3.1 WCET com a análise de valor 81
5.3.2 WCET com a análise de valor considerando o conteúdo da memória 86
5.3.3 Análise *WCET* 88
5.4 CONSIDERAÇÕES FINAIS 90
6 CONCLUSÃO E TRABALHOS FUTUROS 93
REFERÊNCIAS 97

1 INTRODUÇÃO

Os sistemas computacionais de tempo real são identificados como sistemas submetidos a requisitos de natureza temporal, onde os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto. Esses sistemas possuem requisitos que variam com relação ao tamanho, complexidade e criticalidade. As falhas de natureza temporal nesses sistemas são, em alguns casos, consideradas críticas no que diz respeito às suas consequências (FARINES et al., 2000).

No contexto da automação industrial, são muitas as possibilidades de empregar sistemas com requisitos de tempo real, por exemplo, sistemas de controle embutidos em equipamentos industriais, sistema de supervisão e controle de células de manufatura, sistemas responsáveis pela supervisão e controle de plantas industriais completas, sistemas automotivos, entre outros.

A principal característica de sistemas de tempo real é que eles possuem restrições que aparecem na forma de prazos (*deadlines*). Para o escalonamento de sistemas de tempo real, é necessário que as restrições temporais sejam mapeadas em termos destes prazos. Em testes mais sofisticados, são estabelecidas relações matemáticas mais complexas entre os parâmetros das tarefas. O escalonamento de forma geral é um problema combinatório e difícil de resolver, pertencendo à classe de problemas NP-difícil, conforme a teoria da complexidade computacional. Diante da inclusão de sofisticações no modelo, torna-se mais complexa a análise, demandando muitos recursos computacionais (para resolver o problema) e humanos (para formular o problema) (FARINES et al., 2000).

Os testes de escalonabilidade provam geralmente a viabilidade da escala determinando se todas as tarefas do sistema podem ou não ser escalonadas cumprindo todos os prazos individualmente. A análise é realizada utilizando os seguintes parâmetros de cada tarefa (BUTTAZZO, 2011):

- C_i: *Worst-Case Execution Time*;
- T_i: período ou tempo mínimo entre chegadas;
- D_i: *deadline*.

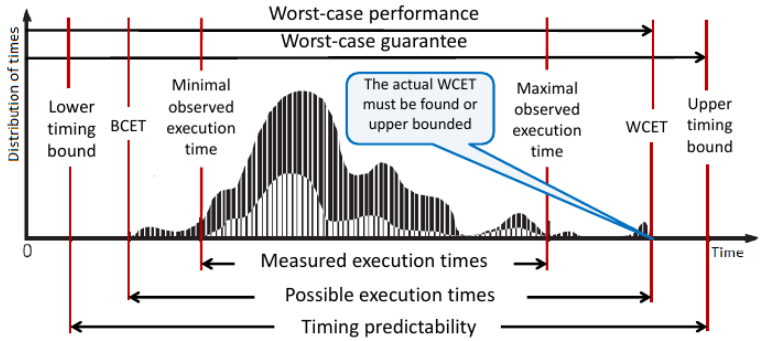
Neste contexto, a determinação do tempo de execução no pior caso (*Worst-Case Execution Time* $-$ *WCET*) é um elemento essencial

para a análise de escalonabilidade e garantia temporal de sistemas de tempo real, especialmente em sistemas de tempo real críticos.

Na figura 1 é ilustrada a problemática da obtenção do *WCET*. O limite esquerdo representa o melhor tempo de execução (*Best-Case Execution Time*−*BCET*), enquanto que o limite direito, o pior tempo de execução (*WCET*). É notório que existe uma diferença entre estes e as medições de execução. Obter esses valores extremos exatos é muito difícil devido ao imenso número de caminhos de execuções possíveis, tornando inviável a exploração exaustiva de todos eles (WILHELM et al., 2008).

Existem duas abordagens para encontrar e avaliar valores aproximados dos tempos extremos de execução. Uma abordagem é baseada em estimação dinâmica (ou simulação), e os valores resultantes são chamados de estimativas. E a outra abordagem é baseada em análise temporal estática, e os valores resultantes são chamados de limites ou garantias (WILHELM et al., 2008).

Figura 1: Análise temporal de sistema



Fonte: Wilhelm et al. (2008)

Para obter o parâmetro *WCET* é necessário conhecer o *hardware* (processador) e o *software* (tarefas). Os processadores modernos utilizam técnicas de *pipelining*, *cache* de instruções e dados, *branch prediction* dinâmico, execução fora de ordem, execução especulativa e *multithreading* de granulação fina (instruções de várias *threads* em execução no *pipeline*). O comportamento temporal de um *software* que executa em processador com estas técnicas é difícil de modelar para a análise *WCET*, devido ao fato que o tempo de execução de uma instrução pode depender do histórico de execução. Por exemplo, na previsão de fluxo baseada em histórico é feito a previsão da direção do

salto, como assumido um caminho e não outro, baseado nas direções observadas anteriormente, consequentemente, podem degradar a escalonabilidade devido possíveis superestimações. Busca-se fornecer um tempo de execução seguro, mas não extremamente pessimista, o que dificultaria sua utilização prática (STARKE, 2013).

1.1 CONCEITOS BÁSICOS

Com a disseminação do uso de sistemas computacionais na sociedade, aplicações com requisitos de tempo real tornam-se mais comuns, variando em relação à complexidade e à necessidade de garantia das restrições temporais. Os programas sofrem variações do tempo de execução que dependem dos dados de entrada, do estado do *hardware* e das interferências do ambiente. Para uma busca exaustiva e completa de todas as possibilidades de execução, o espaço de estados, resultado da combinação dos dados de entrada e possíveis estados iniciais é muito grande. Desta forma, obtêm-se os limites superiores de tempo de execução de blocos básicos e constrói-se o limite superior do sistema (STARKE, 2012).

Para a análise ser viável são realizadas abstrações da plataforma. Com isso há perda de informação e esta é uma das características que dificultam a obtenção dos limites exatos dos tempos de execução. A quantidade de perda de informação depende dos métodos de análise e das propriedades dos sistemas, como arquitetura e analisabilidade do código (STARKE, 2012). A redução de sobrestimação melhora a precisão e é um elemento indicativo da qualidade da análise que obtém o *WCET*.

Com base em pesquisas dos últimos anos, existe um modelo padrão para a análise temporal estática de código. Esse modelo possui quatro blocos principais: análise de valor; reconstrução do fluxo de controle e análise estática para fluxo de controle e dados; análise da microarquitetura; obtenção do pior caminho de execução dentre todos possíveis caminhos do programa (CULLMANN et al., 2010).

Pretende-se nesta pesquisa focar na análise de valor. Ela é utilizada para determinar os valores dos registradores e dos endereços efetivos de memória acessados, os quais estão disponíveis somente na execução do programa. Seus resultados são utilizados na análise da microarquitetura em conjunto com o modelo de hierarquia de memória do sistema. A análise da microarquitetura computa o tempo de execução de cada instrução do processador (WILHELM et al., 2008; HECKMANN; FERDINAND, 2005; STARKE, 2012; ALVAREZ, 2013).

Após analisado o sistema, os limites dos tempos de execução são calculados formulando um problema de otimização baseado nos tempos dos blocos básicos, informação de chamadas de função e fluxos de dados das tarefas. Com isso, busca-se a obtenção analítica do pior tempo de execução do programa executando no modelo de processador e hierarquia de memória em questão, conforme os vários estados obtidos na análise de valor, fluxo e microarquitetura. Esta é uma etapa importante no desenvolvimento e validação de sistemas de tempo real críticos (ALVAREZ, 2013; STARKE, 2012).

1.2 CONTEXTO DO TRABALHO

O desenvolvimento deste trabalho de mestrado faz parte de um projeto maior, composto pelo projeto de *hardware* e o projeto do compilador (ambos trabalhos de doutorado executados em paralelo), onde já existe uma infraestrutura de *software* capaz de compilar, analisar e executar programas compilados em linguagem C. No Capítulo 3 serão apresentados mais detalhes.

Com o objetivo de obter desempenho e determinismo, o processador, o compilador e a análise *WCET* são integradas na mesma ferramenta, possibilitando a reutilização de informações importantes (grafo de fluxo de controle e a modelagem do processador).

As ferramentas *WCET* geralmente reconstroem o fluxo de controle do programa em análise partindo do arquivo executável. Um diferencial do projeto é a construção do grafo de fluxo de controle pelo compilador. Como o compilador necessita internamente deste grafo para gerar o código objeto, a reutilização desta informação elimina um passo na análise *WCET*.

Dada a arquitetura de memória utilizada é necessário o reconhecimento dos acessos à memória com o objetivo de determinar sua latência. Existem duas situações a serem classificadas:

- Acessos à memória principal;
- Acessos à *ScrathPad Memory*.

A distinção estática destes acessos é importante devido a diferença no tempo de acesso da memória principal e da *ScratchPad Memory*. Neste contexto, verificou-se a necessidade da análise de valor para classificar os acessos à memória.

1.3 OBJETIVO

Assume-se neste trabalho um processador que acessa memórias com latências diferentes, sendo uma a *SPRAM* (*ScratchPad Memory*) uma memória rápida porém com conteúdo gerenciado pelo programa e não por *hardware*, e outra a *RAM* (memória principal) uma memória mais lenta de acesso aleatório. Em função dessa característica é necessário classificar os acessos à *RAM* e à *SPRAM* durante a análise *WCET*. O objetivo deste trabalho é realizar análise de valor para determinar qual região de memória cada instrução acessa e, consequentemente, utilizar o tempo correto de execução, considerando o tempo de acesso a cada região de memória.

Com o propósito de investigar o impacto gerado na obtenção do *WCET* com a implementação da análise de valor, será desenvolvida uma ferramenta protótipo para classificação das instruções que acessam a memória, técnica conhecida como análise de valor, visando o uso em ferramentas *WCET*.

1.4 ORGANIZAÇÃO DO TEXTO

Este documento está organizado em seis capítulos começando com o atual, composto por conceitos básicos, contexto do trabalho e os objetivos.

No Capítulo 2 apresenta-se a problemática de obter o *WCET*, os conceitos básicos para entendimento da complexidade do seu cálculo. Neste contexto, é enfatizada a técnica para determinar o comportamento dos dados e da memória do processador, conhecida como análise de valor, e o estado da arte.

No Capítulo 3 é apresentado o contexto do trabalho, abordando o processador utilizado, as instruções suportadas e a análise *WCET* existente.

No Capítulo 4 são apresentadas as técnicas implementadas e a classificação das instruções de acesso à memória.

No Capítulo 5 são apresentados métodos de validação e a avaliação da ferramenta de análise de valor.

No Capítulo 6 é apresentada a conclusão.

2 ANÁLISE DO TEMPO DE EXECUÇÃO NO PIOR CASO (*WCET*)

2.1 CONSIDERAÇÕES INICIAIS

A análise do tempo de execução no pior caso no desenvolvimento de sistemas embarcados de tempo real tem como finalidade proporcionar informações a priori sobre o pior tempo de execução possível de um trecho de *software* antes deste ser executado por um processador (ALVAREZ, 2013).

O *WCET* corresponde ao máximo tempo de execução de um determinado programa em um processador específico. Para o cálculo da estimativa do *WCET*, algumas suposições são assumidas, conforme Engblom et al. (2003):

- O programa específico executa isoladamente;
- A execução ocorre em um determinado processador (*Central Processing Unit − CPU*) e em uma frequência de *clock*;
- O programa é compilado com um determinado compilador;
- Não existe interferência de atividades de fundo (*background*), como acesso direto à memória (*Direct memory access − DMA*) e *refresh* da *Dynamic random access memory (DRAM)*;
- Não existem trocas de contextos (preempções) pelo escalonador.

Visando a obtenção de estimativas do tempo de execução, foram desenvolvidas várias técnicas que podem ser classificadas em: técnicas baseadas em estimativas dinâmicas e técnicas baseadas em análise estática. As técnicas baseadas em estimativas dinâmicas não garantem que o maior tempo de execução medido seja o verdadeiro tempo de execução no pior caso, devido ao fato que o *WCET* pode ocorrer com pouca frequência e as condições para que ele aconteça são normalmente desconhecidas. O foco principal deste trabalho está nas técnicas de análise estática (ALVAREZ, 2013).

2.2 ANÁLISE TEMPORAL ESTÁTICA DE CÓDIGO

Para realizar a análise estática, os programas são representados por um grafo de controle de fluxo, composto por blocos básicos, laços

e caminhos (COPELAND, 2003; STARKE, 2013).

Grafo de Fluxo de Controle (*Control Flow Graph – CFG*): É um grafo direcionado $G = (V, E, i)$, onde os vértices V representam os blocos básicos e as arestas $E \subseteq VxV$ conectam dois vértices v_i e v_j se, e somente se v_j é imediatamente executado depois de v_i. O vértice de entrada do *CFG* é representado por i, ou seja, i não possui arestas de entrada ($\neg\exists v \epsilon V : (v, i)\epsilon E$).

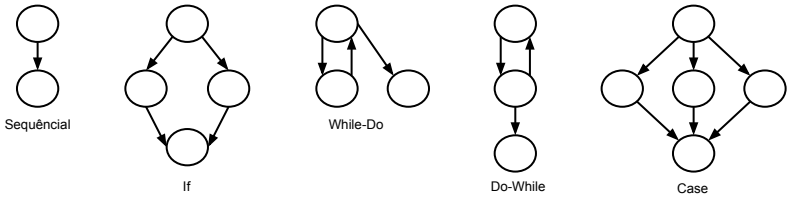
Bloco Básico (BB): É uma sequência de instruções, pode ser alcançado somente pela primeira instrução e a única saída é pela última instrução.

Laço: É um componente fortemente conexo do grafo G que representa o *CFG*. Um laço é composto por um único cabeçalho (*header*) que é o seu ponto de entrada. Podem existir diferentes arestas que retornam de vértices internos de um laço para o cabeçalho, bem como diferentes arestas de saída do laço.

Caminho: Representa o fluxo de controle entre blocos básicos.

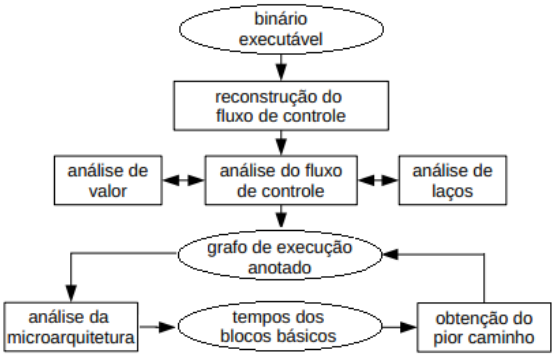
Para representar as funções ou métodos de um programa como um *CFG*, podem-se utilizar as construções conforme a Figura 2:

Figura 2: Representação de condicionais em *CFG*



Com o objetivo de estimar ou calcular o *WCET* de um programa, existe um modelo padrão para análise temporal estática de código, conforme apresentado na Figura 3.

Figura 3: Análise estática de código



Fonte: Cullmann et al. (2010)

Esse modelo possui quatro técnicas principais: **reconstrução do fluxo de controle** que faz a análise do binário da tarefa formando o grafo de fluxo de controle; **análise de valor** que computa os estados dos registradores do processador que serão utilizados na análise da microarquitetura em conjunto com o modelo de hierarquia de memória do sistema; **análise da microarquitetura** computa o tempo de execução de cada instrução do processador; e **obtenção do pior caminho** onde, após analisado todo o sistema, é formado um problema de otimização com os vários estados obtidos nas análises anteriores, cuja função é encontrar o *WCET* do programa executado em um determinado modelo de processador e hierarquia de memória (CULLMANN et al., 2010).

2.2.1 Análise de valor

A análise de valor é utilizada para determinar os valores dos registradores e consequentemente os endereços efetivos de memória acessados. Esses endereços efetivos estão disponíveis somente na execução do programa. Dependendo da implementação da análise de valor, é possível determinar muitos dos endereços se estes forem acessos estáticos e o código gerado pelo compilador for disciplinado. Computam-se limites para os valores nos registradores do processador e das variáveis locais a cada ponto do programa. É também útil para determinar limites de iterações de laços e caminhos de execução inviáveis (WILHELM et al., 2008; STARKE, 2012). Seus resultados são utilizados como entrada para análises mais avançadas, o que acaba por produzir informações sobre o

uso da pilha e do comportamento temporal (HECKMANN; FERDINAND, 2005).

Como o foco principal deste trabalho está na análise de valor, são apresentados mais detalhes e trabalhos relacionados na Seção 2.3.

2.2.2 Análise de fluxo de controle

Na análise do fluxo de controle determina-se o comportamento dinâmico do programa, com o objetivo de coletar informações sobre os possíveis caminhos de execuções. É necessário o conhecimento de informações fornecidas por anotações manuais ou pela análise automática do fluxo sobre o número de iterações de *loops*, profundidade das recursões, dependências de dados de entrada, caminhos inviáveis (*infeasible paths*) e instâncias de funções (RUFINO, 2008).

A entrada para a análise de fluxo é um grafo de fluxo de controle onde os nós representam os blocos básicos e as arestas representam o fluxo. Os métodos diferem no tipo de código que analisam, isto é, código fonte, intermediário ou código de máquina (RUFINO, 2008).

A análise de fluxo de controle é considerada mais fácil no código fonte ao invés do binário, mas é difícil mapear os resultados que o compilador incorpora à estrutura do fluxo de controle devido à otimização e ao ligamento (*linking*) de código. Geralmente, as ferramentas de análise estática temporal utilizam o código binário já compilado (STARKE, 2012; WILHELM et al., 2008).

2.2.3 Análise da microarquitetura

A análise da microarquitetura descreve estaticamente o *hardware* do sistema e define a análise temporal de uma sequência de instruções.

Um processador é composto por vários componentes (como memórias *caches*, *pipelines* e previsores especulativos) que fazem o tempo de execução dependente do contexto, ou seja, o tempo de execução de instruções individuais pode mudar em várias ordens de grandezas, dependendo do estado do processador. O tempo de execução de instruções individuais e até acessos à memória podem depender de estados anteriores (WILHELM et al., 2008; CULLMANN et al., 2010; STARKE, 2012).

Na medida em que os processadores se tornam mais complexos, a modelagem completa é intratável devido a grande quantidade de estados. A modelagem imprecisa aumenta o pessimismo, tornando-se um

fator dominante na análise.

O conhecimento estático sobre os conteúdos da *cache* (classificar referências à memória em acertos e faltas), ocupação das unidade funcionais do processador (incluir ou excluir *stalls* no *pipeline*) e estados das unidades de previsão de fluxo (*branch prediction*) é que minimiza o pessimismo. É possível assumir o pior caso de todos os componentes sem análise mas, provavelmente, o resultado do ponto de vista prático não será utilizável (ALVAREZ, 2013; RUFINO, 2008; STARKE, 2013).

2.2.4 Obtenção do pior caminho

Na literatura existem três classes principais de métodos que combinam tempos medidos ou determinados analiticamente, essas classes são: baseadas em estruturas (*structure-based*), baseadas em caminhos (*path-based*), e técnicas que utilizam enumeração implícita de caminhos (*Implicit Path Enumeration Technique − IPET*) (WILHELM et al., 2008).

No cálculo de limites baseados em estruturas, calcula-se um limite superior em um percurso *bottom-up* da árvore sintática da tarefa, ou seja, é feita uma representação de todo o programa em nodos que descrevem a estrutura do programa como: sequências, laços ou condições, que resultam em blocos básicos. Essa abordagem apresenta alguns problemas, como: nem todos os fluxos de controle podem ser representados através da árvore sintática, correspondência muito direta entre as estruturas do código fonte e código objeto (otimizações do código pelo compilador não são facilmente admissíveis), geralmente não é possível acrescentar informação adicional do fluxo (WILHELM et al., 2008; RUFINO, 2008).

No cálculo de limites baseados em caminhos, calculam-se os limites para os diferentes caminhos da tarefa, buscando a rota global com maior tempo de execução. As rotas possíveis de execução são representadas explicitamente. Essa abordagem é natural dentro de uma iteração de *loop* único, mas tem problema com o fluxo de informações estendendo entre níveis de *loop* aninhados. É necessária a utilização de métodos heurísticos de busca, pois o número de caminhos é exponencial em relação ao número de pontos de decisão (WILHELM et al., 2008; RUFINO, 2008).

Ferramentas mais avançadas são baseadas em *IPET*, que é capaz de lidar com diferentes tipos de informação de fluxo. Elas utilizam técnicas de programação linear inteira ou programação constante, re-

sultando em uma complexidade exponencial em relação ao tamanho da tarefa (WILHELM et al., 2008; STARKE, 2012).

No *IPET* o fluxo do programa e os tempos dos blocos básicos são combinados em uma série de restrições aritméticas. A técnica é aplicada globalmente com todas as informações de fluxo relativas a toda a tarefa.

O resultado do cálculo do *IPET* é um limite temporal superior para cada variável de contagem de execução no pior caso. Nesta técnica, cada bloco básico e arco de fluxo de código dentro da tarefa possuem (ALVAREZ, 2013):

- Um coeficiente temporal (t_{entity}) que corresponde ao limite superior da contribuição dessa entidade para o tempo total de execução a cada vez que a entidade é executada;
- Uma variável de contagem (x_{entity}) que corresponde ao número de vezes que a entidade é executada, informação obtida da análise do fluxo.

É determinado um limite superior maximizando a soma do produto das contagens e tempos de execução, Equação 2.1, onde as variáveis de contagem da execução estão sujeitas a restrições que refletem a estrutura da tarefa e os possíveis fluxos (ALVAREZ, 2013).

$$\sum_{i \in entities} x_i * t_i \tag{2.1}$$

Para obter o *WCET*, os limites dos tempos de execução são calculados formulando um problema de otimização baseado nos tempos dos blocos básicos, informação de chamadas de funções e fluxo de dados da tarefa (WILHELM et al., 2008).

2.3 ANÁLISE DE VALOR

O conjunto de valores que uma variável pode assumir em um determinado ponto do programa é uma das propriedades mais importantes para determinar o tempo de um programa. O que afeta o fluxo de execução de um programa são os condicionais (*if-then-else*, *switch*, *while*, *for*), que vão identificar onde deve continuar a execução, com base nos valores atuais que são obtidos das variáveis (BYGDE, 2010).

A análise de valor computa os estados dos registradores do processador que serão utilizados na análise da microarquitetura em conjunto com o modelo de hierarquia de memória do sistema. A análise

da microarquitetura computa o tempo de execução de cada instrução do processador. Após analisado o sistema, é obtido o *WCET* do programa executando no modelo de processador e hierarquia de memória em questão, conforme os vários estados obtidos na análise de valor, fluxo e microarquitetura (ALVAREZ, 2013).

2.3.1 Trabalhos relacionados

A análise estática ao nível do código de máquina tem mostrado bons resultados para a estimativa do *WCET*. Existem várias abordagens de análise estática para a implementação da análise de valor (WILHELM et al., 2008; THESING, 2004; STARKE, 2012; PATTERSON, 1995).

Na literatura, a modelagem para a implementação da análise de valor geralmente é realizada pela técnica de interpretação abstrata (SAINRAT et al., 2013; HARRISON, 1977; MARKOVSKIY, 2002; WILHELM et al., 2008; GUSTAFSSON et al., 2003). Essa técnica é baseada na utilização de uma semântica abstrata para representar os valores durante a execução de um programa, para extrair propriedades de um programa sem que seja necessário executá-lo. É necessário definir as semânticas concretas de forma simplificada que descrevam os aspectos interessantes para a análise de valor, também definir as semânticas abstratas que coletam os aspectos temporais em cada ponto do programa (THESING, 2004).

A interpretação abstrata é uma técnica promissora mas, para cada processador suportado, é necessário construir um modelo seguindo a teoria da interpretação abstrata. É uma solução interessante para analisar um processador comercial, quando seu modelo não está disponível ou já existe uma ferramenta que interprete as semânticas da interpretação abstrata. A implementação deste modelo é uma tarefa complexa exigindo um estudo detalhado da especificação do processador e para obter uma análise confiável são necessários vários passos de validação (THESING, 2004).

Na execução deste trabalho não foi utilizada técnica de interpretação abstrata para a modelagem da análise de valor. A evolução da especificação do processador no nosso caso acompanha a construção da ferramenta de análise *WCET*, diferente da abordagem padrão que deseja analisar um processador comercial existente.

As ferramentas de *WCET* geralmente reconstroem o fluxo de controle do programa a partir do arquivo executável, analisam as chamadas de funções e a descrição do fluxo em nível de blocos básicos

observando apenas o *assembly*. Neste trabalho o *CFG* é gerado pelo compilador, diminuindo um passo da análise de valor.

2.4 CONSIDERAÇÕES FINAIS

A análise de sistemas de tempo real é uma tarefa desafiadora e requer a união de várias técnicas, para a obtenção analítica do pior tempo de execução de tarefas, uma etapa importante no desenvolvimento e validação de sistemas de tempo real críticos.

Em processadores que possuem memórias de dados com latências diferentes, é necessária a análise de valor para o reconhecimento dos acessos à memória com o objetivo de determinar sua latência, e consequentemente gerar um limite superior do *WCET* mais apertado.

Os valores estimados ou calculados do pior tempo de execução podem ser utilizados para realizar a análise de escalonamento (ALVAREZ, 2013).

3 CONTEXTO DO TRABALHO: PROCESSADOR E A FERRAMENTA DE ANÁLISE *WCET*

3.1 CONSIDERAÇÕES INICIAIS

Segundo Patterson e Hennessy (2013), o desempenho de uma máquina é determinado por três principais fatores: contagem de instruções, tempo de ciclo de *clock* e ciclos de *clock* por instrução (*Cycles Per Instruction* − CPI). O tempo de ciclo de *clock* e o número de CPI são determinados pela implementação do processador.

O *hardware* utilizado na criação de sistemas de tempo real críticos tem grande efeito na facilidade ou dificuldade em relação à previsibilidade na execução dos programas. No caso dos processadores simples, a construção do modelo abstrato é mais fácil do ponto de vista do comportamento temporal e a problemática resume-se à correta construção do fluxo de controle. Os processadores modernos compostos por *caches*, *pipelines*, execução fora de ordem tornam extremante difícil obter o *WCET*, devido ao fato que o tempo de execução de uma instrução pode depender do histórico de execução, necessitando de ferramentas e métodos mais avançados.

Neste capítulo apresenta-se o contexto do trabalho e onde estará a contribuição deste projeto de mestrado, seguido da especificação do processador utilizado, para o entendimento da determinação do *WCET* que corresponde ao máximo tempo de execução de um programa em um processador com possíveis dados de entrada e estado inicial bem definidos. Na sequência são apresentadas as instruções de acesso à memória, o cálculo do endereço de memória e sua penalidade que foi definida pelo projetista do processador. E posteriormente apresenta-se a análise *WCET*.

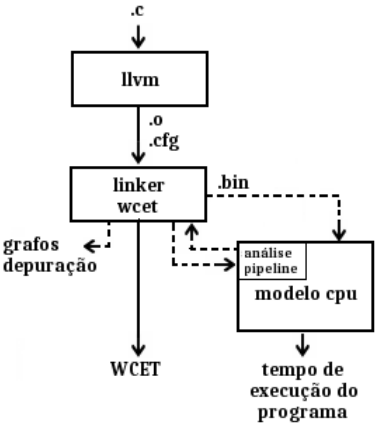
3.2 CONTEXTO DO TRABALHO E ONDE ESTARÁ A CONTRIBUIÇÃO

Conforme apresentado no Capítulo 1, o desenvolvimento deste trabalho faz parte de um projeto maior, composto pelo projeto de *hardware* e o projeto do compilador (ambos trabalhos de doutorado executados em paralelo), onde já existe uma infraestrutura de *software* capaz de compilar, analisar e executar programas compilados em linguagem C. Para a avaliação do projeto maior em relação ao *WCET* desenvol-

veram uma infraestrutura para experimentação, conforme mostrado na Figura 4.

Com o objetivo de obter desempenho e determinismo, integraram na mesma ferramenta o processador, o compilador e a análise *WCET*, possibilitando a reutilização de informações importantes (grafo de fluxo de controle e a modelagem do processador).

Figura 4: Infraestrutura de experimentação existente



Fonte: Starke (2013)

A partir de um arquivo fonte do programa escrito em linguagem C (.c), com a utilização de um compilador são produzidos dois dados importantes: o objeto (binário) do programa compilado (.o) e o grafo de fluxo de controle (.cfg). Após a compilação, o código objeto passa por uma ferramenta de ligação (*linker*). Esta é responsável pela leitura das tabelas do código objeto para fazer o mapeamento dos dados e das funções para a configuração de memória do processador. Na sequência, é possível executar o arquivo binário (.bin) na plataforma alvo. Destaca-se que não são suportados programas com recursão ou com ponto flutuante, devido ao fato que o compilador não consegue gerar o grafo de fluxo de controle (STARKE, 2013).

A análise *WCET* é integrada na mesma ferramenta que o *linker*, simplificando o fluxo de informações. A análise *WCET* necessita do grafo de fluxo de controle e das informações sobre o mapeamento de memória do processador (STARKE, 2013).

O processador (modelo cpu) basicamente é um *pipeline* de cinco estágios, busca simples, em ordem, *cache* de instruções e *ScratchPad*

Memory. As funcionalidades básicas da ferramenta de *software* "modelo cpu" são: a simulação semântica e temporal do programa, que gera o tempo de execução real do programa na plataforma alvo; e uma interface utilizada pela ferramenta de *WCET* para modelar o comportamento do *pipeline* do processador. Apresentar-se-á mais detalhes do processador na próxima seção (STARKE, 2013).

Neste contexto, a ferramenta desenvolvida simula o aspecto temporal do processador, a partir do arquivo gerado pelo *linker* possibilita escrever no componente da memória do processador, faz a simulação à nível de ciclo do processador executando o programa especificado, permite a parametrização das configurações da memória *cache* e do tamanho da memória (principal e *ScratchPad*), apresenta a ordem de execução das instrução durante a simulação, e no final da simulação, apresenta o tempo de execução do programa, o arquivo com o conteúdo de memória do processador e o número de faltas e acertos na memória *cache* (STARKE, 2013).

Dada a infraestrutura de experimentação desenvolvida, observou-se que com a arquitetura de memória utilizada, onde possui memória com latência diferentes, duas situações devem ser classificadas:

- Acessos à memória principal;
- Acessos à *ScrathPad Memory*.

A distinção estática destes acessos é importante devido a diferença no tempo de acesso da memória principal e da *ScratchPad Memory*. Verificou-se a necessidade da análise de valor com o objetivo de determinar qual região de memória as instruções acessam e consequentemente utilizar o tempo correto de execução, considerando o tempo de acesso a cada região de memória, sendo esta a principal contribuição deste trabalho.

3.3 PROCESSADOR

O processador utilizado suporta um subconjunto de instruções *ISA MIPS R2000* de $32bits$ e possui a especificação mostrada na Tabela 1 (PATTERSON; HENNESSY, 2013):

Tabela 1: Especificação do processador

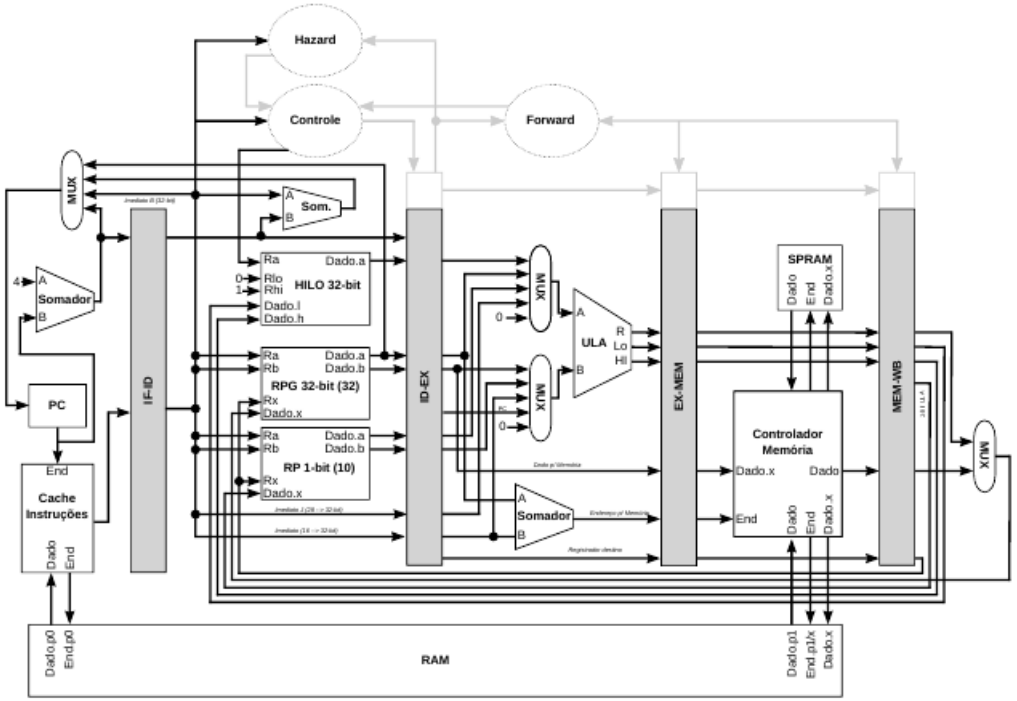
Recursos	Processador
Pipeline	– 5 estágios – Busca simples – Em ordem
Registradores	– Um banco com 32 de 32-*bits* para propósito geral – Um banco com 10 de 1-*bit* para predicados – Dois registradores de 32-*bits* para resultado de operações de produto e divisão
Unidade lógica aritmética	– Uma unidade para operações inteiras
Memória	– Cache para instruções com mapeamento direto – Uma principal unificada (dados e instruções) – Uma *ScratchPad* (*cache* de dados) – Espaço unificado de endereçamento de 32-*bits* (*ScratchPad* e memória principal no espaço de endereçamento) – Endereçamento relativo ao PC (*branch*) ou indireto (relativo à registrador) – Instruções individuais para acesso a *byte*, *half-word* (16-*bits*) ou *word* (32-*bits*)
Dependências entre instruções	– *Forward* (dependências entre operações aritméticas e lógicas resolvidas pelo *hardware*) – *Hazzard* (detecção e correção de dependências entre uma operação de acesso à memória e outra aritmética)
Saltos	– Desacoplamento das instruções de *branch* (comparação e salto)

O processador foi modelado em *SystemC* (INITIATIVE, 2013) através da metodologia *Register-transfer Level Design (RTL)* em nível de ciclo (todo comportamento temporal do *pipeline* e memórias é considerado). O diagrama do processador é representado na Figura 5. A modelagem e a descrição detalhada está fora do escopo deste trabalho. Esse mesmo modelo é utilizado para a obtenção do tempo de execução das tarefas e também para a obtenção do tempo de execução dos blocos básicos.

Um ciclo de instrução básico pode ser divido nos cinco estágios do *pipeline*:

- Busca de instruções (F) - *instruction fetch*;
- Decodificação da instrução (D) - *instruction decode*;
- Execução da instrução (E) - *instruction execution*;
- Acesso à memória (M) - *memory access*;
- Gravação dos resultados (WB) - *write-back*.

Figura 5: Diagrama do processador para tempo real deste trabalho



Fonte: Starke (2013)

Um ciclo genérico busca uma nova instrução da *cache*, decodifica-a determinando que tipo de trabalho será realizado. Geralmente um ou mais operandos devem ser buscados nos registradores. A operação da instrução é executada e o ciclo termina gravando o resultado nos registradores ou na memória (PATTERSON; HENNESSY, 2013).

A partir da Figura 5 observa-se a inexistência de uma *cache* de dados com o objetivo de melhorar o determinismo. A *cache* de dados foi substituída pela *SPRAM* (*ScratchPad Memory*), uma *cache* com conteúdo gerenciado pelo programa e não por *hardware* (PATTERSON; HENNESSY, 2013). Em função dessa característica é necessário classificar os acessos à *RAM* (memória principal) e à *SPRAM* durante a análise *WCET*.

3.4 OPERANDOS DO *HARDWARE*

As linguagens de programação possuem variáveis simples que contêm elementos de dados isolados, mas também possuem estruturas de dados complexas. Essas estruturas complexas podem conter mais elementos de dados do que a quantidade de registradores (PATTERSON; HENNESSY, 2013).

O processador mantém uma pequena quantidade de dados nos registradores, mas a memória contém muitos elementos de dados. Com isso, as estruturas de dados (*arrays* e estruturas) são mantidas na memória. É necessário instruções que transfiram dados entre a memória e os registradores (PATTERSON; HENNESSY, 2013).

A instrução de transferência de dados que copia dados da memória para um registrador é chamada de *load*. O formato desta instrução em *assembly* é o nome da operação seguido pelo registrador a ser carregado, depois uma constante e o registrador utilizado para acessar a memória. A soma da parte constante da instrução com o conteúdo do segundo registrador forma o endereço da memória (PATTERSON; HENNESSY, 2013).

A instrução de transferência de dados que copia dados de um registrador para a memória é chamada de *store*. O formato desta instrução em *assembly* é semelhante ao de uma *load*, o nome da operação, seguido pelo registrador a ser armazenado, depois o *offset* para selecionar o elemento do *array* e finalmente o registrador base (PATTERSON; HENNESSY, 2013).

O formato das instruções de acesso à memória é denominado de *tipo-I* (de imediato), ou *formato I*. É a forma de representação de uma

instrução imediata e de transferência de dados composta de campos de números binários, conforme mostrado na Tabela 2. Os seus campos são (PATTERSON; HENNESSY, 2013):

- Op: operação básica da instrução (*opcode*);
- R_s: registrador base;
- R_t: registrador destino;
- Endereço ou constante: endereço de 16 *bits*.

Tabela 2: Formato das instruções

Op	Rs	Rt	constante ou endereço
6 *bits*	5 *bits*	5 *bits*	16 *bits*

Para exemplificar, considerando as seguintes instruções:

- $lw\ R_4, R_1(R_{29})$
- $sw\ R_4, R_1(R_{29})$

A representação em linguagem de máquina dessas instruções, usando os números decimais e registradores, são mostradas respectivamente na Tabela 3.

Tabela 3: Representação das instruções

Op	Rs	Rt	constante ou endereço
35	R_1	R_4	$R_1 + R_{29}$
43	R_1	R_4	$R_1 + R_{29}$

3.5 INSTRUÇÕES DE ACESSO À MEMÓRIA

As instruções de acesso à memória suportadas nesse trabalho são mostradas na Tabela 4.

- R_t e R_s representam o endereço dos registradores codificados na instrução;

- *Im* é uma constante de 16-*bits* codificada na instrução.

Tabela 4: Instruções de acesso à memória

Mnemônico	Função	Descrição
lw	Rt = MEM[Rs + Im]	Carrega 4-*bytes* da memória (*word*)
lb	Rt = MEM[Rs + Im]	Carrega *byte* da memória
lbu	Rt = MEM[Rs + Im]	Carrega *byte* da memória sem sinal
lh	Rt = MEM[Rs + Im]	Carrega 2-*bytes* da memória (*half-word*)
lhu	Rt = MEM[Rs + Im]	Carrega 2-*bytes* da memória sem sinal (*half-word*)
sb	MEM[Rs + Im] = Rt	Grava *byte* na memória
sh	MEM[Rs + Im] = Rt	Grava 2-*bytes* na memória (*half-word*)
sw	MEM[Rs + Im] = Rt	Grava 4-*bytes* na memória (*word*)

A especificação das instruções de memória (*load*/*store*) é mostrada na Tabela 5. Considerando uma arquitetura de memória de dados e instruções independentes, relacionando as instruções com as operações e as cinco subcomputações típicas de um *pipeline*.

Tabela 5: Especificação das instruções de memória

	Load	*Store*
F	Acessar memória de instrução	
D	Decodificar e acessar registradores	
E	Gerar endereço efetivo (base + *offset*)	
M	Acessar memória de dados	
WB	Gravar dado nos registradores	

3.6 ENDEREÇAMENTO DE MEMÓRIA E SUA PENALIDADE

O endereço de acesso é calculado somando a base com o *offset* (Rs + Im). Conforme o valor obtido a instrução é classificada, de acordo com a Tabela 6.

Tabela 6: Endereçamento de memória e sua penalidade

Memória	Endereçamento (*byte*)	Penalidade (ciclo)
Principal	0x0000 - 0x3FFF	5
ScratchPad	0x4000 - 0x4FFF	0

Observa-se na Tabela 6 que a penalidade definida pelo projetista do processador, o acesso à memória principal é cinco vezes superior a penalidade de acesso à *ScratchPad Memory*.

Quando não há dependência de dados ou controle em qualquer par de computações, o *pipeline* opera em fluxo contínuo, ou seja, nenhuma operação precisa esperar a completude da operação anterior. Para *pipelines* de instruções uma operação depende do resultado da operação anterior para prosseguir. Se ambas as operações encontram-se em processamento, a operação deverá esperar o resultado da anterior, ocasionando *stall* no *pipeline* e todos os estágios anteriores também deverão esperar, ocasionando uma série de estágios ociosos. Portanto, quando uma instrução é classificada como um acesso à memória principal e há dependência de dados ocasiona um grande tempo de *stall*.

Na Subseção 3.7.2 será apresentado um exemplo de dependência de instruções que incluem ciclos de *stall* no *pipeline*.

3.7 ANÁLISE *WCET*

Como o cálculo do *WCET* é complexo, geralmente utilizam-se métodos tradicionais que assumem monotonicidade e composição básica, possibilitando que as análises de valor, *cache*, *pipeline* e a obtenção do pior caminho de execução possam ser construídas isoladamente (WENZEL et al., 2005). Para isso, não são permitidas anomalias temporais, então o processador deve conter apenas recursos ordenados (*in-order resources*).

Monotonicidade: Quando uma informação incerta é processada por uma abordagem de análise *WCET*, um aumento da latência para

um instrução impõe necessariamente um tempo de execução igual ou maior sobre as sequências de instruções sob análise. O pior caso local não diminui o pior caso global (WENZEL et al., 2005; STARKE, 2013).

Composição básica: Denota o fato de que os valores *WCET* de subcaminhos podem ser seguramente compostos durante o cálculo do *WCET* global (WENZEL et al., 2005; STARKE, 2013).

Anomalias Temporais: Desvio não esperado do *hardware* real em contraste com o modelo utilizado na análise. Anomalias temporais dificultam a análise *WCET*, especialmente quando é necessário um limite superior seguro do *WCET*. Os métodos de execução serial utilizados geralmente são pessimistas, levando a superestimações ou abordagens extremante complexas em *hardware* que possuem anomalias temporais (WENZEL et al., 2005; LUNDQVIST, 2002).

O objetivo das análises de valor, *cache* e *pipeline* é a determinação do tempo de execução dos blocos básicos que, considerando uma arquitetura composicional, é calculado pela Equação 3.1 (STARKE, 2013):

$$t_{bb} = t_p + n_c * t_c + n_m * t_m \tag{3.1}$$

t_{bb}: tempo de execução de um bloco básico.
t_p: tempo do *pipeline* para execução das instruções do bloco básico.
n_c: número de faltas de *cache*.
t_c: tempo de penalidade da *cache*.
n_m: número de acessos à memória.
t_m: tempo de acesso à memória.

Os valores estimados ou calculados para o *WCET* dependem das análises para serem mais precisos. Neste trabalho o objetivo é contribuir com a identificação do número de acessos à memória com a utilização da técnica de análise de valor.

Nas próximas seções é apresentado brevemente como foi obtido o tempo de cada bloco básico nas análises de *cache* e *pipeline* e a resolução do problema de otimização para obter o *WCET*. A implementação desses está fora do escopo deste trabalho.

3.7.1 Análise da memória cache de instruções

A memória cache é utilizada para minimizar o intervalo entre o processador e a memória. Neste trabalho, utilizou-se uma memória *cache* de instruções mapeada diretamente, onde um bloco de memória pode estar somente em uma linha de *cache*. O objetivo é determinar quais instruções de um bloco básico ocasionarão falta na *cache*.

A memória *cache* é caracterizada pela sua **capacidade** que é o número total de *bytes* disponibilizados pela *cache* de instrução; **tamanho da linha** que é a quantidade de *bytes* transferidos da memória para a *cache* quando ocorre uma falta (*cache miss*); e **associatividade** que determina em quais linhas um bloco de memória pode residir.

Sobre se uma instrução está na *cache* ou não, temos que (STARKE, 2013):

Instrução está potencialmente na *cache* se: Existir uma sequência de transição no *CFG* onde o bloco de memória correspondente a instrução já foi referenciado em blocos básicos anteriores; este bloco de memória é referenciado em instruções anteriores no mesmo bloco básico.

Estado abstrato: É o subconjunto de todos os blocos de memória que podem estar na *cache* antes da execução do bloco básico.

Estado abstrato alcançável: É o subconjunto de todos os blocos de memória que podem ser alcançáveis pelas transições do *CFG*.

Estado abstrato efetivo: É o subconjunto de todos os blocos de memória que podem ser alcançáveis considerando todos os caminhos do *CFG* que levam ao bloco básico em análise.

As instruções do programa são classificadas em:

- *A_MISS*: poderá haver uma falta na *cache* (*always miss*) toda vez que a instrução é executada;
- *A_HIT*: a instrução será encontrada na *cache* (*always hit*) toda vez que for executada;
- *F_MISS*: haverá uma falta na *cache* somente na primeira iteração do laço (*first miss*);
- *CONFLICT*: existem vários caminhos que alcançam o bloco básico da instrução em análise e cada um destes caminhos possui

um estado efetivo diferente, dependendo do fluxo efetivo podem ocorrer acertos ou faltas.

O número de faltas influencia o tempo do bloco básico, pode ser contabilizado com a classificação das instruções. O tempo de execução de um bloco básico varia em função do comportamento do fluxo e de outros blocos básicos.

3.7.2 Análise do pipeline

A análise do *pipeline* determina o tempo de execução de instruções e blocos básicos quando executado no *pipeline* do processador. Os demais elementos de *hardware* como memória principal e *cache* de instruções não são considerados.

Em processadores com *pipeline*, o tempo de execução de uma instrução em ciclos será equivalente ao número de ciclos do comprimento do *pipeline*, se não houver nenhuma dependência.

Dado um processador com busca simples e *pipeline* de 5 estágios, a latência de uma instrução é de 5 ciclos. Se existem duas instruções sem dependência, o tempo total será de 6 ciclos, e assim por diante. Se existe dependência de dados, esta deve ser modelada para que se tenha uma análise *WCET* correta. Considere um exemplos simples de dependência de instruções:

$$
\begin{array}{ll}
1 : addiu\ R_1, 2 & \#R_1 = 2 \\
2 : lw\ R_2, R_0, 0 & \#R_2 = MEM[R_0 + 0] \\
3 : add\ R_3, R_1, R_2 & \#R_3 = R_1 + R_2
\end{array}
$$

Os parâmetros da instrução 3 dependem do resultado das instruções 1 e 2. Quando a instrução 3 encontra-se no estágio de execução (E), a instrução 2 encontra-se em execução no estágio de acesso à memória (M). Para garantir o resultado da instrução 3 é necessário parar o *pipeline* até que a instrução 2 chegue no estágio de gravar o resultado (WB). Conforme descrito, o conjunto de instruções provocam ciclos de ociosidade, incluindo ciclos de *stall* no *pipeline*.

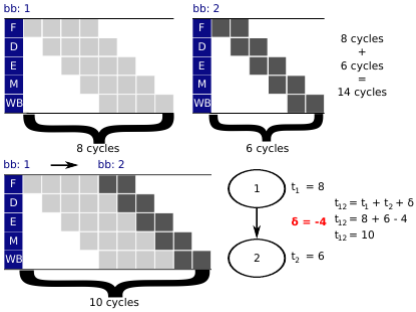
A interpretação dos blocos básicos ignora aspectos da *cache* e do acesso à memória, uma vez que estes são considerados em análises separadas. Os tempos de bloco básico são obtidos individualmente utilizando o modelo em *SystemC* (STARKE, 2013).

Para modelar o fluxo de execução deve ser aplicada uma correção. Considere a Figura 6, onde o objetivo é obter o tempo de

execução do fluxo entre os blocos básicos 1 e 2, com tempos de 8 e 6 ciclos, respectivamente. A soma dos tempos de execução de ambos os blocos é de 14 ciclos e este não representa o tempo de execução real do fluxo (STARKE, 2013; ENGBLOM, 2002).

O tempo de execução correto é de 10 ciclos, devido a uma sobreposição no *pipeline* entre os dois blocos básicos. Durante a análise de fluxo e a obtenção do *WCET*, deve-se subtrair 4 ciclos para cada transição limite entre blocos básicos (STARKE, 2013; ENGBLOM, 2002).

Figura 6: Exemplo de composição de tempo de dois blocos básicos sucessivos



Fonte: Starke (2013)

Esta correção é aplicada diretamente na formulação do problema usando programação linear inteira para a obtenção do *WCET* de todo o programa, como será mostrado na subseção seguinte.

3.7.3 Obtenção do pior caminho

Para obter o pior caminho de execução foram utilizadas técnicas de enumeração implícita de caminhos (*IPET*). É uma técnica que apresenta estimativas precisas e é independente dos dados de entrada do programa analisado. O fluxo do programa e os tempos dos blocos básicos são combinados em uma série de restrições aritméticas.

Geralmente o resultado do cálculo do *IPET* é um limite temporal superior para cada variável de contagem de execução no pior caso. Nesta técnica, cada bloco básico e aresta de fluxo de código dentro da tarefa possuem um **coeficiente temporal**, correspondendo ao limite superior da contribuição dessa entidade para o tempo total de execução cada vez que a entidade é executada, e uma **variável de contagem**,

correspondendo ao número de vezes que a entidade é executada, informação obtida da análise do fluxo (ALVAREZ, 2013).

O limite superior é determinado maximizando a soma do produto das contagens e tempos de execução. As variáveis de contagem da execução estão sujeitas a restrições que refletem a estrutura da tarefa e os possíveis fluxos, conforme mostrado no Capítulo 2.

Com a abordagem utilizada para obtenção dos tempos dos blocos básicos, esta estimativa torna-se pessimista. A análise considera o tempo a partir da primeira instrução do bloco básico entrar no *pipeline* até a última sair. A execução de blocos básicos sucessivos é aglutinada no *pipeline*, como mostrado da seção anterior. No *Integer Linear Programming (ILP)* isto é modelado como um desconto de δ cada vez que um bloco básico é executado e o limite superior é determinado maximizando a soma do produto das contagens e tempos de execução menos o produto das contagens e o desconto, conforme a Equação 3.2.

$$\sum_{\forall bb_j \rightarrow bb_i} x_i * t_i - x_i * \delta \tag{3.2}$$

Para simplificar o problema, diminuindo a quantidade de variáveis, as entidades foram modeladas considerando somente as arestas do *CFG*. A contagem x_i para um bloco básico i pode ser reescrita conforme Equação 3.3.

$$x_i = \sum_{\forall bb_j \rightarrow bb_i} dj\_i \tag{3.3}$$

A Equação 3.2 pode ser reescrita como na Equação 3.4.

$$\sum_{\forall bb_i} \left(\sum_{\forall bb_j \rightarrow bb_i} dj\_i * t_i - dj\_i * \delta \right) \tag{3.4}$$

Para garantir a convergência da função do limite superior são necessárias algumas restrições lineares. As modelagens das restrições lineares feitas no contexto do analisador implementado seguem uma abordagem semelhante à apresentada em (LI; MALIK, 1995). As seguintes restrições devem ser aplicadas:

Restrição de início e fim de execução: toda execução do programa deve ter um início e um fim. O fluxo deve passar exatamente uma vez pela entrada e saída do *CFG*, representados como *dstart* e *dend* no *CFG*, respectivamente. A restrição é descrita pela Equação 3.5:

$$dstart = 1 \qquad \& \qquad dend = 1 \tag{3.5}$$

Restrições de conservação de fluxo: todo fluxo que entra em um bloco básico a partir de um predecessor, deve sair a partir de um sucessor. O fluxo deve ser conservado durante a execução do programa. Esta restrição é modelada pela Equação 3.6:

$$\sum_{\forall bb_j \to bb_i} dj\_i - \sum_{\forall bb_i \to bb_k} di\_k = 0 \tag{3.6}$$

Restrições de limitação de laços: blocos básicos devem executar de acordo com os limites dos laços aos quais pertencem. Um bloco básico pode pertencer a vários laços, desde que estes estejam aninhados. No caso de um bloco básico ser o cabeçalho do laço, este poderá executar uma vez a mais ao final, para avaliar a condição de saída. Para limitação de laços, a restrição é descrita pela Equação 3.7, para cada bloco básico bb_i, onde lb_i é o limite superior de iterações do laço ao qual bb_i pertence.

$$\sum_{\forall bb_j \to bb_i} dj\_i \leq lb_i \tag{3.7}$$

Restrição de componentes conservativos/conexos: um bloco básico dentro de um laço só poderá ter execução contabilizada se efetivamente o fluxo entrar no laço. Um ciclo é executado quando o fluxo entra no seu cabeçalho. Esta restrição é representada pela Equação 3.8:

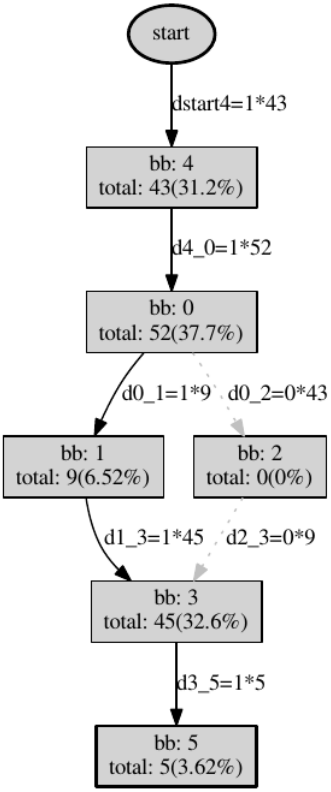
$$\sum_{\forall bb_j \to bb_i \wedge bb_j \notin loop(bb_i)} dj\_i * ilb_i - \sum_{\forall bb_i \to bb_k \wedge bb_k \in loop(bb_i)} dj\_i \tag{3.8}$$

Para exemplificar, considere o programa C abaixo, contendo uma sentença *if-then-else*.

```
int main(int argc, char** argv){
        int i = 1; int j = 0;

        if(i < 2){
                i++;
        } else {
                i--;
        }
}
```

A representação da solução obtida graficamente do resultado do *IPET* desse programa é apresentada na Figura 7, o qual é composto de um conjunto de restrições que consideram principalmente o fluxo e limitações de laço. As arestas em preto representam o pior caminho, e as arestas tracejadas, em cinza, não pertencem ao pior caminho, ou seja, no pior caso o fluxo não passará pelo bloco básico 2. O fluxo do bloco básico 4 ao 0 é representado pela aresta $d4\_0 = 1 * 52$, o $BB : 4$ executa 1 vez e leva 52 ciclos.

Figura 7: Resultado do IPET



3.8 CONSIDERAÇÕES FINAIS

Neste capítulo apresentou-se o processador e as análises de *cache* e *pipeline* existentes, com o objetivo de determinar o tempo de execução

dos blocos básicos, considerando uma arquitetura composicional. E por fim, apresentou-se simplificadamente a resolução do problema de otimização para obter o *WCET*.

O processador utilizado neste trabalho possui memória de dados com latências diferentes. A classificação correta das instruções de acesso à memória é de grande importância para obter um limite superior para o *WCET* menos pessimista.

4 EMPREGO DA ANÁLISE DE VALOR NO CONTEXTO DO TRABALHO

4.1 CONSIDERAÇÕES INICIAIS

Neste capítulo será apresentada a metodologia empregada na implementação da análise de valor, como são classificadas as instruções de acesso à memória e dois *benchmarks* para exemplificar a classificação das instruções.

O objetivo da análise de valor neste trabalho é determinar qual região de memória as instruções acessam e consequentemente utilizar o tempo correto de execução, considerando o tempo de acesso a cada região de memória.

4.2 IMPLEMENTAÇÃO

A partir de um arquivo fonte do programa escrito em linguagem C, com a utilização de um compilador (nesse caso o *Clang+LLVM* estendido para suportar o processador apresentado no Capítulo 3) são produzidos dois dados importantes:

- Objeto (binário) do programa compilado;
- Grafo de fluxo de controle.

Após a compilação, o código objeto passa por uma ferramenta de ligação (*linker*). Esta é responsável pela leitura das tabelas do código objeto para fazer o mapeamento dos dados e das funções para a configuração de memória do processador. Destaca-se que não são suportados programas com recursão ou com ponto flutuante.

A análise *WCET* é integrada na mesma ferramenta que o *linker*, simplificando o fluxo de informações. A análise *WCET* necessita do grafo de fluxo de controle e das informações sobre o mapeamento de memória do processador.

No grafo de fluxo do programa, cada nó representa um bloco básico que é composto por uma sequência de instruções executadas linearmente sem possibilidade de salto e o limite dos nós são instruções de chamadas, testes ou pulos (*calls*, *branches*, *jumps*).

A nomenclatura dos vértices é formada pelo índice do bloco básico e entre chaves o seu endereço inicial e final. As arestas do grafo indicam o fluxo do programa nos blocos básicos.

É necessário realizar uma análise de fluxo de dados para a propagação do valor de um bloco básico para o outro sucessivamente.

Utilizou-se a técnica *forward dataflow analysis* para a análise de fluxo de dados. O estado de saída dos registradores é copiado dos blocos antecessores e é alterado o estado dos registradores que são acessados internamente (SETHI et al., 2008). O algoritmo é mostrado na sequência.

Algoritmo de análise de fluxo

```
1: vector<basic_block*>* bb_list = graph->get_bbs();
2: bool change = true;
3: int32_t constants = 2;
4: int32_t count = 0;
5: bool propagation = false;
6:
7: while (change) {
8:    change = false;
9:
10:    // Para todos os blocos básicos
11:    for (unsigned int i = 0;
12:      i < bb_list->size(); i++) {
13:        basic_block *bb = (*bb_list)[i];
14:
15:        register_state *pred_state;
16:        register_state old_state;
17:        old_state.copy_state(&bb->dfa_rstate);
18:
19:        if (bb->predecessors.size() == 0) {
20:            bb->dfa_rstate.copy_state(&bb->my_rstate);
21:        }
22:
23:        // Para todos os predecessores
24:        for (unsigned int j = 0;
25:       j < bb->predecessors.size(); j++) {
26:            basic_block *pred = bb->predecessors[j];
27:
28:            pred_state = new register_state;
29:            pred_state->last_value(0, 0);
30:
31:            // Método das instruções
32:            if (count >= constants) propagation = true;
33:            dfa_instruction(bb, pred, pred_state, propagation);
34:
35:            delete(pred_state);
36:        }
37:
38:        // Iterar até convergir
39:        change = change |
40:       !(old_state.is_egual(&bb->dfa_rstate));
41:
```

```
42:         bb->dfa_rstate.print();
43:     }
44:     count++;
45:}
```

O método *dfa_instruction* (linha 33) é responsável pela interpretação estática das instruções, respeitando as seguintes condições:

- Desconsiderar o registrador destino da instrução de leitura de memória (*lw*, *lb*, *lh*, *lhu*), pois o conteúdo da memória não é considerado nesta análise.

- Instruções que não acessam a memória são interpretadas apenas quando seus parâmetros (Rs, Rt) são conhecidos.

Primeiramente é necessário que as constantes do programa sejam propagadas, ou seja, são calculadas as instruções que não dependem de informações cujo valor é desconhecido. As instruções com parâmetros conhecidos são computadas e o registrador apontador de pilha (*Stack Pointer*) é inicializado. Devido à presença de condicionais são necessárias até duas iterações do algoritmo, dependendo do código que está sendo analisado, para que o valor do apontador de pilha seja propagado por todo o fluxo do programa. Posteriormente as instruções de memória podem ser classificadas.

No algoritmo temos uma condição que vai garantir que a classificação só acontecerá após a propagação das constantes (linha 32):

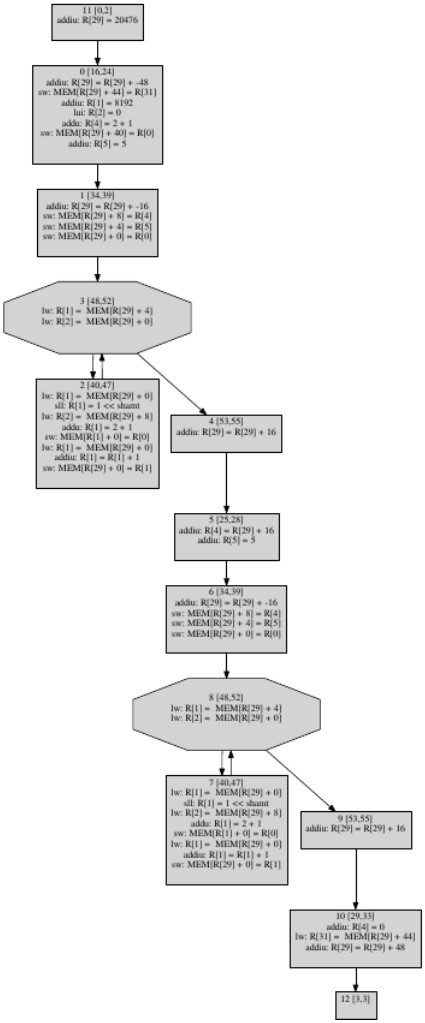
```
32:    if (count >= constants) propagation = true;
```

Considere o seguinte código C e seu respectivo *CFG* em conjunto com as instruções de máquina (Figura 8):

```
void init (int *p, int x);

int data[5];

int main ()
{
  int i;
  int sp_data[5];

  init(&data, 5);

  init(&sp_data, 5);

  return 0;
}
```

```
void init (int *p, int x)
{
  for (int i = 0; i < x; i++)
  {
    p[i] = 0;
  }
}
```

Figura 8: CFG com as instruções de máquina



Observa-se que na função *main* do código exemplo existem duas chamadas da mesma função (*init*). A função é composta por uma estrutura condicional (*for*) que varre as posições de um vetor de inteiros, atribuindo 0 em cada um das posições. Em ambas as chamadas da função *init* a passagem de um dos parâmetros é por referência. É enviada para a função uma referência da variável utilizada. Porém, em uma das chamadas o parâmetro é uma variável local, ou seja, apenas a função onde a variável está definida pode usá-la. E na outra função é uma variável global, ou seja, a variável é acessível em todo o escopo do programa, por qualquer função. E qualquer função pode alterar o valor, utilizá-la em um processo ou até mesmo atribuir um valor específico.

Esse exemplo é um caso típico onde são necessárias duas iterações do algoritmo. No bloco básico 7 (Figura 8) existe a instrução *sw:* MEM[R[1] + 0x0000] = R[0] que, com apenas uma iteração do algoritmo, o valor de R[1] é desconhecido. Verificou-se que, para todos os casos testados, duas iterações do algoritmo são suficientes.

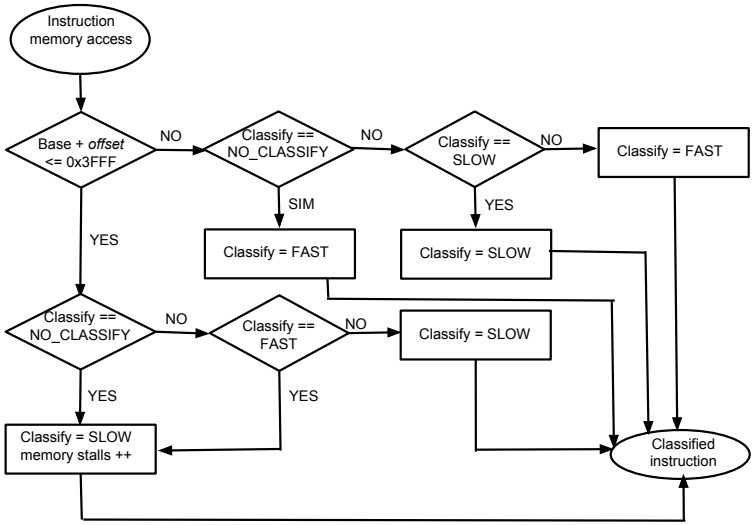
Para a classificação das instruções de memória o endereço de acesso é calculado. A instrução é classificada em *SLOW* ou *FAST* conforme a Tabela 7:

Tabela 7: Classificação da memória

Classificação	Memória	Endereçamento (*byte*)
SLOW	Principal	0x0000 - 0x3FFF
FAST	*ScratchPad*	0x4000 - 0x4FFF

O fluxograma da classificação das instruções de acesso à memória é mostrado na Figura 9. O objetivo é representar através de figuras geométricas a sequência lógica utilizada para classificar uma instrução de acesso à memória. A elipse é utilizada para representar o estado inicial e final do programa, o losango para representar uma tomada de decisão, e o retângulo para representar um processamento de dados.

Figura 9: Classificação das instruções de acesso à memória



Uma instrução de acesso à memória encontra-se em um dos três estados existentes, sendo eles:

- $NO\_CLASSIFY$: a instrução ainda não foi classificada;
- $SLOW$: a instrução foi classificada como acesso à memória principal;
- $FAST$: a instrução foi classificada como acesso à *ScratchPad Memory*.

Quando uma instrução é identificada como acesso à memória, o endereço de acesso é calculado somando a base com o *offset* (Rs + Im). Se a soma for menor ou igual a $0x3FFF$ e:

1. A instrução ainda não foi classificada ($classify == NO\_CLASSIFY$), então ela passa para o estado de $SLOW$ e é incrementado a variável *memory stall*;
2. A instrução foi classificada como $FAST$ ($classify == FAST$), então ela passa para o estado de $SLOW$ e é incrementado a variável *memory stall*;
3. A instrução foi classificada como $SLOW$ ($classify == SLOW$), então ela continua no estado de $SLOW$.

Caso o endereço de acesso seja maior que $0x3FFF$ e:

1. A instrução ainda não foi classificada ($classify == NO\_CLASSIFY$), então ela passa para o estado de $FAST$;
2. A instrução foi classificada como $SLOW$ ($classify == SLOW$), então ela continua no estado de $SLOW$;
3. A instrução foi classificada como $FAST$ ($classify == FAST$), então ela continua no estado de FAST.

Caso a instrução de acesso à memória não se encaixe em nenhuma das opções e ainda não foi classificada, por exemplo, quando seus parâmetros são desconhecidos, esta será classificada no seu pior caso ($SLOW$) e é incrementada a variável *memory stall*.

Como não há penalidade para o acesso à *ScratchPad Memory*, a variável *memory stall* é incrementada apenas quando é identificado um acesso à memória principal. O tempo de execução dos blocos básicos considera o número de acessos à memória multiplicado pelo tempo de acesso.

Um bloco básico pode ser executado várias vezes, recebendo diferentes valores. Em algumas execuções uma determinada instrução pode ser classificada como $SLOW$ e em outras execuções ser classificadas como $FAST$. Conforme descrito anteriormente, esta instrução assumirá a classificação de pior caso ($SLOW$).

Ressalta-se a necessidade da condição para propagar as constantes e posteriormente classificar as instruções de acesso à memória. Caso não tivesse esta condição, o valor do apontador de pilha poderia ser desconhecido a priori e a instrução ser classificada como $SLOW$, e após o valor do apontador de pilha ser conhecido a instrução ser classificada como $FAST$. Devido às condição assumidas para a classificação, a instrução assumiria o pior caso ($SLOW$), resultando em uma classificação errônea.

4.3 EXEMPLOS

Será considerado o processador apresentado no Capítulo 3, que tem 32 registradores de 32 *bits*, onde 30 são de uso geral. O endereço de registro:

- 31 (R_{31}) é utilizado como endereço de retorno (RA);
- 29 (R_{29}) é utilizado como ponteiro de pilha;

- 0 (R_0) tem sempre valor zero.

Para exemplificar a classificação das instruções de acesso à memória, serão mostrados dois *benchmarks* (*swap.c* e *array-pointer.c*).

4.3.1 Classificação das instruções de memória do programa *swap*

Considere o seguinte código C (*swap.c*):

swap.c

```
typedef  unsigned char  bool;
typedef  unsigned int   uint;

void inc (uint* a) {
        (*a)++;
}

int main () {
  uint x = 513239;
  inc(&x);
  return 0;
}
```

Observa-se que na função *main* do programa *swap.c* existe a chamada de uma função e sua passagem de parâmetro é por referência. É enviada para a função uma referência da variável utilizada.

O *assembly* desse programa é mostrado em sequência (*swap.s*).

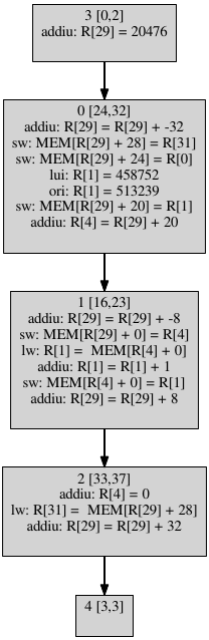
swap.s

```
inc:                                    # @inc
# BB#1:
        addiu   $sp, $sp, -8
        sw      $4, 0($sp)
        lw      $1, 0($4)
        addiu   $1, $1, 1
        sw      $1, 0($4)
        addiu   $sp, $sp, 8
        jr      $ra
        nop

main:                                   # @main
# BB#0:
        addiu   $sp, $sp, -32
```

```
        sw      $ra, 28($sp)            # 4-byte Folded Spill
        sw      $zero, 24($sp)
        lui     $1, 7
        ori     $1, $1, 54487
        sw      $1, 20($sp)
        addiu   $4, $sp, 20
        jal     inc
        nop
# BB#2:
        addiu   $4, $zero, 0
        lw      $ra, 28($sp)            # 4-byte Folded Reload
        addiu   $sp, $sp, 32
        jr      $ra
        nop
$tmp4:
        .size   main, ($tmp4)-main
```

O *CFG* em conjunto com as instruções de máquina desse programa é mostrado na Figura 10.

Figura 10: CFG com as instruções de máquina do programa *swap*



É possível verificar que no BB 3 o registrador $R[29]$ é inicializado, sendo um ponteiro para a pilha. No BB 0 o $R[29]$ tem um deslocamento

de menos 32 e em sequência tem três instruções de acesso à memória que utiliza como base o $R[29]$. É necessário realizar uma análise de fluxo de dados para a propagação do valor calculado no BB 3 para o seu sucessor (BB 0) e assim sucessivamente.

Primeiramente são calculados os valores dos registradores em cada iteração do programa *swap* propagando apenas as constantes do programa, ou seja, sem considerar as instruções de acesso à memória. O resultado é apresentado na Tabela 8.

Tabela 8: Propagação das constantes do programa *swap*

Iteração	BB: 0	BB: 1	BB: 2
1	R[0]: 0x0000 R[1]: 0x7D4D7 R[31]: 0x0021	R[0]: 0x0000 R[1]: 0x7D4D8 R[31]: 0x0021	R[0]: 0x0000 R[1]: 0x7D4D8 R[4]: 0x0000 R[31]: 0x0021
2	R[0]: 0x0000 R[1]: 0x7D4D7 R[4]: 0x4FF0 R[29]: 0x4FDC R[31]: 0x0021	R[0]: 0x0000 R[1]: 0x7D4D8 R[4]: 0x4FF0 R[29]: 0x4FDC R[31]: 0x0021	R[0]: 0x0000 R[1]: 0x7D4D8 R[4]: 0x0000 R[29]: 0x4FFC R[31]: 0x0021

Iteração	BB: 3	BB: 4
1	R[0]: 0x0000 R[29]: 0x4FFC	R[0]: 0x0000 R[1]: 0x7D4D8 R[4]: 0x0000 R[31]: 0x0021
2	R[0]: 0x0000 R[29]: 0x4FFC	R[0]: 0x0000 R[1]: 0x7D4D8 R[4]: 0x0000 R[29]: 0x4FFC R[31]: 0x0021

Após realizar a propagação das constantes, consideram-se todas as instruções. Os valores resultantes dos registradores calculados do programa *swap* são apresentados na Tabela 9.

Tabela 9: Todas as instruções do programa *swap*

BB: 0	BB: 1	BB: 2
R[0]: 0x0000 R[1]: 0x7D4D7 R[4]: 0x4FF0 R[29]: 0x4FDC R[31]: 0x0021	R[0]: 0x0000 R[4]: 0x4FF0 R[29]: 0x4FDC R[31]: 0x0021	R[0]: 0x0000 R[4]: 0x0000 R[29]: 0x4FFC

BB: 3	BB: 4
R[0]: 0x0000 R[29]: 0x4FFC	R[0]: 0x0000 R[4]: 0x0000 R[29]: 0x4FFC

Observando o *CFG* com as instruções de máquina do programa *swap*, Figura 10, e os valores resultantes dos registradores calculados, Tabelas 8 e 9, é possível verificar que o registrador destino de uma instrução de acesso à memória não é computado, devido às condições assumidas. Por exemplo, no BB 1 da Tabela 8 tem que $R[1] = 0x7D4D7$ e na Tabela 9 o R[1] do BB 1 não possui valor, pois tem a instrução *lw*: $R[1] = MEM[R[4] + 0x0000]$, desta forma, o $R[1]$ desse bloco básico passa a ser desconhecido e não é possível computar a instrução *addiu*: $R[1] = R[1] + 0x0001$. O que nesse caso não é um problema, pois o $R[1]$ resultante não é base para nenhuma instrução de memória.

Para exemplificar, a Tabela 10 ilustra a classificação das instruções de acesso à memória do programa *swap*.

Tabela 10: Classificação das instruções do programa *swap*

BB: 0	
Instrução	Classificação
sw: MEM[0x4FF8] = R[31]	FAST
sw: MEM[0x4FF4] = R[0]	FAST
sw: MEM[0x4FF0] = R[1]	FAST
BB: 1	
Instrução	Classificação
sw: MEM[0x4FD4] = R[4]	FAST
lw: R[1] = MEM[0x4FF0]	FAST
sw: MEM[0x4FF0] = R[1]	FAST
BB: 2	
Instrução	Classificação
lw: R[31]: MEM[0x4FF8]	FAST

Todas as instruções de acesso à memória do programa *swap* foram classificadas como *FAST*, então possuem latência determinada pela *ScratchPad Memory*. Sem a análise de valor, todas as instruções de acesso à memória seriam classificadas como *SLOW*. Nesse caso, o valor do *WCET* dos programas seria superestimado consideravelmente.

Conforme apresentado, com a análise de valor implementada é possível garantir que a classificação de todas as instruções de memória do programa *swap* é *FAST*, obtendo um limite superior do *WCET* menos pessimista.

4.3.2 Classificação das instruções de memória do programa *array-pointer*

Considere o seguinte código C (*array-pointer.c*):

array-pointer.c

```
int main ()
{
  int i;
  int sp_data[10];

  for (i = 0; i < 10; i++)
  {
      sp_data[i] = 0;
  }
  return 0;
}
```

Observa-se que na função *main* do programa *array-pointer.c* existe uma estrutura condicional (*for*) que varre as posições de um vetor de inteiros de 0 a 9, atribuindo 0 em cada uma dessas posições.

O *assembly* desse programa é mostrado em sequência (*array-pointer.s*).

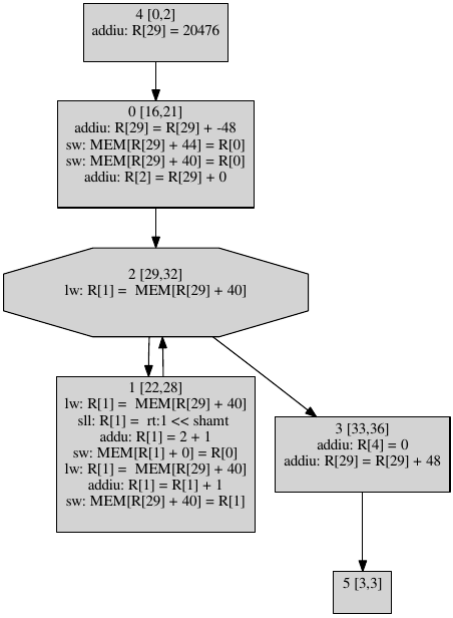
array-pointer.s

```
main:                                   # @main

# BB#0:
        addiu   $sp, $sp, -48
        sw      $zero, 44($sp)
        sw      $zero, 40($sp)
        addiu   $2, $sp, 0
        j       $BB0_2
        nop
$BB0_1:                                 #   in Loop: Header=BB0_2 Depth=1
        lw      $1, 40($sp)
        sll     $1, $1, 2
        addu    $1, $2, $1
        sw      $zero, 0($1)
        lw      $1, 40($sp)
        addiu   $1, $1, 1
        sw      $1, 40($sp)
$BB0_2:                                 # =>This Inner Loop Header: Depth=1
        lw      $1, 40($sp)
        cmpgti  $p1, $1, 9
        brf     $p1, $BB0_1
```

```
        nop
$BB0_3:
        addiu   $4, $zero, 0
        addiu   $sp, $sp, 48
        jr      $ra
        nop
$tmp1:
        .size   main, ($tmp1)-main

        .type   data,@object    # @data
        .comm   data,20,16
```

O *CFG* em conjunto com as instruções de máquina desse programa é mostrado na Figura 11.

Figura 11: CFG com as instruções de máquina do programa *array-pointer*



É possível verificar que no BB 4 o registrador $R[29]$ é inicializado, sendo um ponteiro para a pilha. No BB 0 o $R[29]$ tem um deslocamento de menos 48 e em sequência tem duas instruções de acesso à memória que utiliza como base o $R[29]$. É necessário realizar uma análise de fluxo de dados para a propagação do valor calculado no BB 4 para o seu sucessor (BB 0) e assim sucessivamente.

Primeiramente são calculados os valores dos registradores em cada iteração do programa *array-pointer* propagando apenas as constantes do programa, ou seja, sem considerar as instruções de acesso à memória. O resultado da ultima iteração é apresentado na Tabela 11.

Tabela 11: Propagação das constantes do programa *array-pointer*

BB: 0	BB: 1	BB: 2
R[0]: 0x0000	R[0]: 0x0000	R[0]: 0x0000
R[2]: 0x4FCC	R[2]: 0x4FCC	R[2]: 0x4FCC
R[29]: 0x4FCC	R[29]: 0x4FCC	R[29]: 0x4FCC

BB: 3	BB: 4	BB: 5
R[0]: 0x0000	R[0]: 0x0000	R[0]: 0x0000
R[2]: 0x4FCC	R[29]: 0x4FFC	R[2]: 0x4FCC
R[4]: 0x0000		R[4]: 0x0000
R[29]: 0x4FFC		R[29]: 0x4FFC

Após realizar a propagação das constantes, consideram-se todas as instruções. Os valores resultantes dos registradores calculados do programa *array-pointer* são apresentados na Tabela 12.

Tabela 12: Todas as instruções do programa *array-pointer*

BB: 0	BB: 1	BB: 2
R[0]: 0x0000	R[0]: 0x0000	R[0]: 0x0000
R[2]: 0x4FCC	R[2]: 0x4FCC	R[2]: 0x4FCC
R[29]: 0x4FCC	R[29]: 0x4FCC	R[29]: 0x4FCC

BB: 3	BB: 4	BB: 5
R[0]: 0x0000	R[0]: 0x0000	R[0]: 0x0000
R[2]: 0x4FCC	R[29]: 0x4FFC	R[2]: 0x4FCC
R[4]: 0x0000		R[4]: 0x0000
R[29]: 0x4FFC		R[29]: 0x4FFC

Observa-se que para esse exemplo, os valores resultantes dos registradores calculados, Tabelas 11 e 12, são equivalentes. Observando o *CFG*, verifica-se que no BB 1 existe a instrução *addu*: $R[1] = R[2] + R[1]$ que não é possível de ser computada, devido as condições assumidas, sendo nesse caso um problema, pois $R[1]$ é base para uma instrução de memória.

Para exemplificar, a Tabela 13 ilustra a classificação das instruções de acesso à memória do programa *array-pointer*.

Tabela 13: Classificação das instruções do *array-pointer*

BB: 0	
Instrução	Classificação
sw: MEM[0x4FF8] = R[0]	FAST
sw: MEM[0x4FF4] = R[1]	FAST
BB: 1	
Instrução	Classificação
lw: R[1] = MEM[0x4FF4]	FAST
sw: MEM[R[1]+0x0000] = R[0]	SLOW
lw: R[1] = MEM[0x4FF4]	FAST
sw: MEM[0x4FF4] = R[1]	FAST
BB: 2	
Instrução	Classificação
lw: R[1]: MEM[0x4FF4]	FAST

Sem a análise de valor, todas as instruções de acesso à memória seriam classificadas como *SLOW*. Nesse caso, o valor do *WCET* dos programas seria superestimado consideravelmente.

Conforme apresentado, com a análise de valor implementada é possível garantir que seis instruções de acesso à memória do programa *array-pointer* são classificadas como *FAST*, e uma como *SLOW* porque seu registrador base não é conhecido a priori. Como a análise não possuía informações suficientes para classificar corretamente a instrução, esta assume o pior caso. Isto indica a necessidade de considerar o conteúdo da memória na análise.

4.3.3 Inclusão do conteúdo da memória na análise

Considerando o conteúdo da memória na análise, o registrador destino da instrução de leitura de memória (*lw*, *lb*, *lh*, *lhu*) passou a ser considerado quando este é conhecido.

Para cada instrução de escrita na memória calcula-se o seu endereço (soma da base com o *offset*) e guarda-se nesse endereço o valor do registrador de origem.

Quando existe a ocorrência de uma instrução de leitura da memória, calcula-se a posição (soma da base com o *offset*) e obtém-se na estrutura de memória o valor a ser lido.

Um problema encontrado com a implementação do conteúdo da memória na análise foi a convergência do algoritmo de fluxo de dados para frente. Em cada iteração calcula-se a posição de memória das

instruções de acesso à memória e esses valores são atualizados, assim, é necessário um critério de parada para o algoritmo. Para resolver esse problema, utilizou-se o *bound* do *loop* do bloco básico em análise, ou seja, os blocos básicos devem executar de acordo com os limites dos laços aos quais pertencem, informação obtida através da anotação de código, e somamos este com o número de iterações necessárias para a propagação das constantes (duas iterações para a propagação das constantes). Com isto, é possível saber quantas vezes o algoritmo precisa iterar, e consequentemente, convergir. As linhas 38, 39, 40 e 41 do algoritmo apresentado na Seção 4.2 passam a ser respectivamente:

```
38:   // Iterar até convergir
39:   change = (change && (count <= (bound + constants))) |
40:   (!(old_state.is_egual(&bb->rmb_rstate)) &&
41:   (count <= (bound + constants)));
```

Será usado como exemplo o programa *swap*. As instruções de escrita na memória desse programa são mostradas na Tabela 14, contendo informação do endereço de memória e para onde aponta. Por exemplo, na terceira instrução a posição de memória é $0x4FF0$ e aponta como origem o registrador R[1], e esse registrador tem valor $0x7D4D7$.

Tabela 14: Instrução de escrita na memória do programa *swap*

BB: 0		
MEM[R[29] + 0x001C] = R[31]	MEM[0x4FF8] = R[31]	R[31] = 0x0021
MEM[R[29] + 0x0018] = R[0]	MEM[0x4FF4] = R[0]	R[0] = 0x0000
MEM[R[29] + 0x0014] = R[1]	MEM[0x4FF0] = R[1]	R[1] = 0x7D4D7
BB: 1		
MEM[R[29] + 0x0000] = R[4]	MEM[0x4FD4] = R[4]	R[4] = 0x4FF0
MEM[R[4] + 0x0000] = R[1]	MEM[0x4FF0] = R[1]	R[1] = 0x7D4D8

As instruções de leitura na memória do programa *swap* são mostradas na Tabela 15. Calcula-se a posição de memória (soma da base com o *offset*). No caso da primeira instrução dessa tabela a posição da memória é $0x4FF0$ e o registrador destino é $R[1]$. Conforme apresentado na Tabela 14, o conteúdo desta posição é $0x7D4D7$, assim, o registrador destino da instrução de leitura é $R[1] = 0x7D4D7$.

Tabela 15: Instrução de leitura na memória do programa *swap*

BB: 1		
R[1] = MEM[R[4] + 0x0000]	R[1] = MEM[0x4FF0]	R[1] = 0x7D4D7
BB: 2		
R[31] = MEM[R[29] + 0x001C]	R[31] = MEM[0x4FF8]	R[31] = 0x0021

Nesse exemplo, o resultado da classificação das instruções de acesso à memória não foi influenciado, pois o registrador cujo o conteúdo passou a ser conhecido não é base para outra instrução de memória.

A Tabela 16 e a Tabela 17 exemplificam o resultado da terceira iteração do programa *array-pointer*, quando é realização a análise das instruções que acessam à memória.

Tabela 16: Instrução de escrita na memória do programa *array-pointer*

BB: 0		
MEM[R[29] + 0x002C] = R[0]	MEM[0x4FF8] = R[0]	R[0] = 0x0000
MEM[R[29] + 0x0028] = R[0]	MEM[0x4FF4] = R[0]	R[0] = 0x0000
BB: 1		
MEM[R[1] + 0x0000] = R[0]	MEM[0x4FCC] = R[0]	R[0] = 0x0000
MEM[R[29] + 0x0028] = R[1]	MEM[0x4FF4] = R[1]	R[1] = 0x0001

Tabela 17: Instrução de leitura na memória do programa *array-pointer*

BB: 1		
R[1] = MEM[R[29] + 0x0028]	R[1] = MEM[0x4FF4]	R[1] = 0x0000
R[1] = MEM[R[29] + 0x0028]	R[1] = MEM[0x4FF4]	R[1] = 0x0000
BB: 2		
R[1] = MEM[R[29] + 0x0028]	R[1] = MEM[0x4FF4]	R[1] = 0x0001

Sem a implementação da análise do conteúdo da memória o conteúdo do registrador destino da instrução de leitura seria desconhecido. Caso este fosse base para outra instrução de memória a mesma poderia não ser classificada corretamente, como aconteceu para o programa *array-pointer*.

Conforme apresentado anteriormente, analisando o *CFG* desse exemplo, o BB 1 tem a instrução *sw*: $MEM[R[1]+0x0000] = R[0]$ que sem a implementação da análise do conteúdo da memória é classificada como *SLOW*, pois $R[1]$ torna-se desconhecido pela instrução *lw*: $R[1] = MEM[R[29]+0x0028]$. Com a implementação da análise do conteúdo da memória, esse problema foi resolvido, os valores são conhecidos e a instrução *sw* foi classificada corretamente como *FAST*.

Os valores resultantes dos registradores com a implementação da memória para o programa *array-pointer* são mostrados na Tabela 18.

Tabela 18: Considerando a implementação da memória para o *array-pointer*

BB: 0	BB: 1	BB: 2
R[0]: 0x0000	R[0]: 0x0000	R[0]: 0x0000
R[2]: 0x4FCC	R[1]: 0x000C	R[1]: 0x000C
R[29]: 0x4FCC	R[2]: 0x4FCC	R[2]: 0x4FCC
	R[29]: 0x4FCC	R[29]: 0x4FCC

BB: 3	BB: 4	BB: 5
R[0]: 0x0000	R[0]: 0x0000	R[0]: 0x0000
R[1]: 0x000C	R[29]: 0x4FFC	R[1]: 0x000C
R[2]: 0x4FCC		R[2]: 0x4FCC
R[4]: 0x0000		R[4]: 0x0000
R[29]: 0x4FFC		R[29]: 0x4FFC

A implementação da análise do conteúdo da memória possibilitou uma aproximação mais precisa do limite superior do *WCET*.

4.4 CONSIDERAÇÕES FINAIS

A partir da reconstrução do fluxo do controle do programa são formados os blocos básicos de execução e é realizado um mapeamento dos acessos aos segmentos de memória, possibilitando classificar as instruções (acesso à memória principal ou *ScratchPad Memory*). Como a penalidade de acesso à memória principal é cinco vezes superior a da *ScratchPad Memory*, a classificação correta das instruções é importante.

Considerou-se primeiramente um cenário onde na interpretação estática das instruções não é considerado o conteúdo da memória, o registrador destino da instrução de leitura de memória passa a ser desconhecido. As instruções que não acessam a memória são interpretadas

apenas quando seus parâmetros (R_s, R_t) são conhecidos. Posteriormente, considerou-se o conteúdo da memória, melhorando os resultados da interpretação estática das instruções e gerando um limite superior mais apertado para o *WCET*.

5 VALIDAÇÃO E AVALIAÇÃO DOS RESULTADOS DA ANÁLISE DE VALOR

5.1 CONSIDERAÇÕES INICIAIS

Neste capítulo é apresentado o método usado para a validação da ferramenta de análise de valor, verificando se os resultados obtidos são corretos. Também são apresentados os resultados da análise *WCET* para alguns *benchmarks* que são geralmente utilizados na verificação de ferramentas *WCET*. Considera-se a análise de valor sem e com a implementação da análise do conteúdo da memória.

5.2 VALIDAÇÃO DOS RESULTADOS DA ANÁLISE DE VALOR

A mesma ferramenta que implementa o modelo do processador em *SystemC*, disponibiliza uma interface (através de uma biblioteca compartilhada) para fornecer o tempo de execução de blocos básicos individualmente.

Para a validação da ferramenta de análise de valor é realizada uma comparação do valor dos registradores por bloco básico do *trace* de execução do programa com os obtidos pelo algoritmo proposto neste trabalho, verificando se os valores resultantes são equivalentes. E, em cada instrução de acesso à memória, verifica-se sua classificação, ou seja, se o valor do endereço estiver entre $0x0000$ a $0x3FFF$ a classificação deve ser *SLOW* e se estiver entre $0x4000$ a $0x4FFF$ a classificação deve ser *FAST*.

A validação do resultado do *benchmark swap* pode ser visualizada nas Figuras 12, 13 e 14. É apresentado o resultado do simulador para a execução das instruções do bloco básico a ser analisado (destacado em vermelho), o *assembly* (destacado em verde), o *CFG* com as instruções de máquina (destacado em alaranjado) e o resultado do algoritmo (destacado em azul).

Figura 12: Validação do BB: 0 do *benchmark swap*

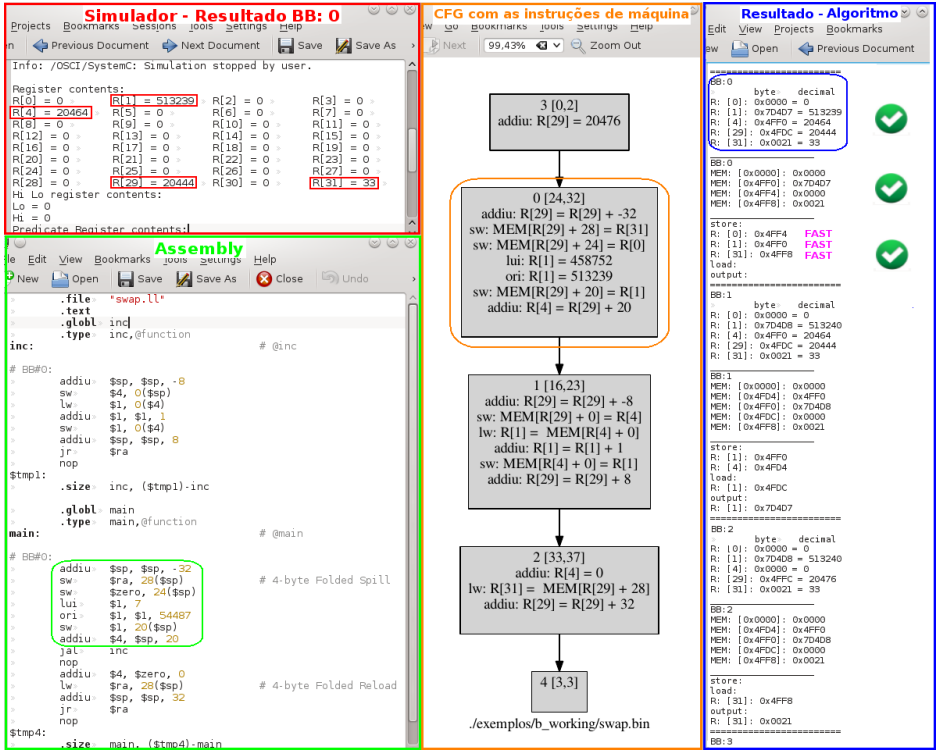


Figura 13: Validação do BB: 1 do *benchmark swap*

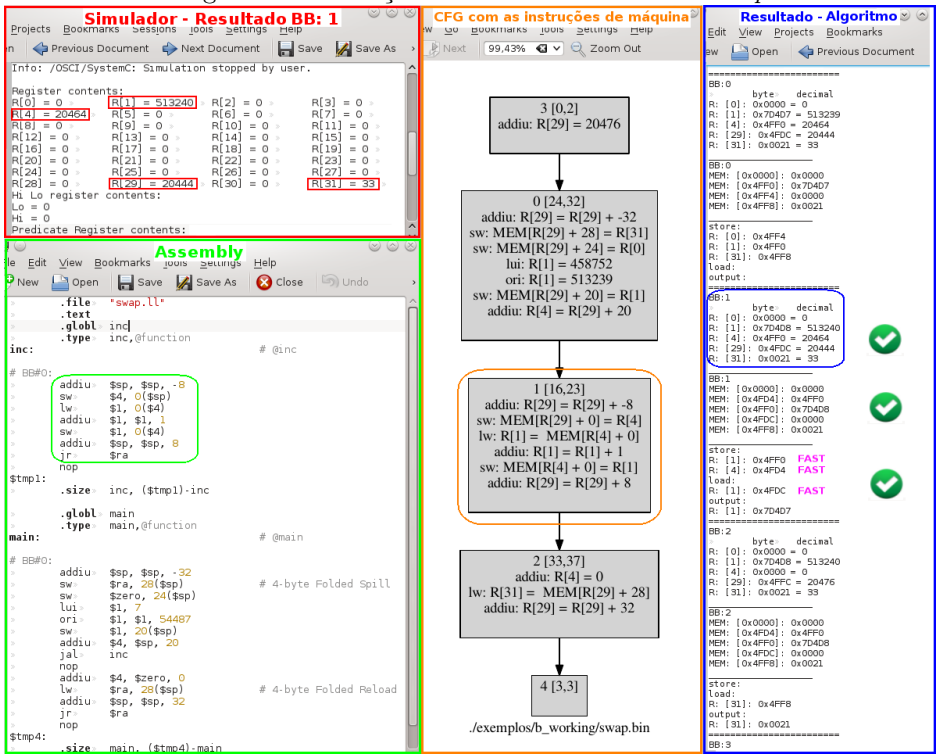
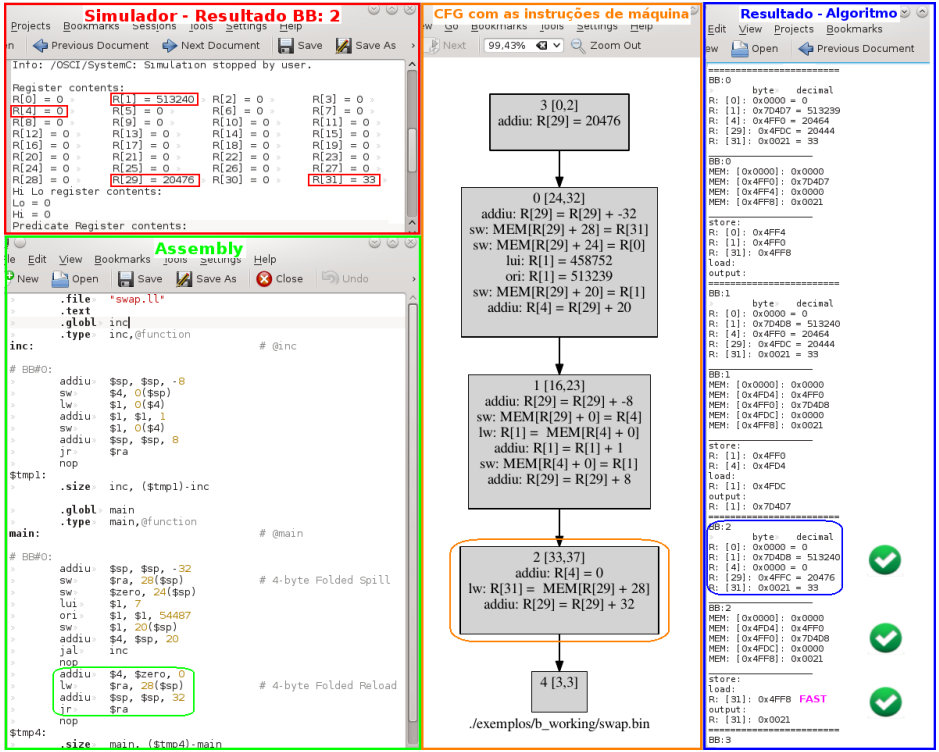


Figura 14: Validação do BB: 2 do *benchmark swap*



Observa-se no resultado do simulador para a execução das instruções de cada bloco básico que os registradores R[1], R[4], R[29] e R[31] tiveram seus valores alterados. Comparando com o resultado do algoritmo, verifica-se que os valores resultantes são idênticos. Podemos concluir que os valores dos registradores estão corretos.

Conforme apresentado no Capítulo 4, todas as instruções de acesso à memória do *benchmark swap* foram classificadas como *FAST*, então o valor calculado para o endereço dessas instruções devem estar entre $0x4000$ a $0x4FFF$. Podemos observar nas Figuras 12, 13 e 14 que está correta a classificação.

A validação da ferramenta de análise de valor pela comparação com o simulador foi realizada apenas para alguns *benchmark*. Em programas mais complexos é difícil a validação, devido a grande quantidade de blocos básicos e caminhos possíveis de execução, além do fato que a análise de valor considera todos os caminhos possíveis e a execução do simulador pode ser diferente, tornando impossível a comparação dos valores dos registradores. Contudo, observou-se a classificação das instruções de memória em outros exemplos.

5.3 AVALIAÇÃO DOS RESULTADOS DA ANÁLISE DE VALOR

Para avaliar a solução, verificando se o resultado da análise do *WCET* considerando a análise de valor não é otimista, comparou-se o resultado do *WCET* com o simulador. Na melhor condição o resultado foi equivalente, ou seja, foi possível classificar todas as instruções corretamente, e na pior condição o resultado da análise do *WCET* foi pessimista, ou seja, a informação não era suficiente para classificar a instrução, então assumiu-se o pior caso.

5.3.1 WCET com a análise de valor

Para a verificação da ferramenta desenvolvida, utilizando alguns *benchmarks* de análise do *WCET* (GUSTAFSSON et al., 2010), considerou-se para cada *benchmark* o *WCET* com análise de valor obtido (*WCET* com AV), o tempo de execução, e a relação (R_1), entre o tempo de execução (SIM) e o *WCET com AV*. A relação (R_1) é dada pela Equação 5.1 e pode ser visualizada na Tabela 19.

$$R_1 = \left(\frac{SIM}{WCET\ com\ AV} \right) * 100 \tag{5.1}$$

O valor de R_1 fica entre 0 (quando *WCET* é expressivamente maior que SIM) e 100 (quando o SIM é igual a *WCET*). Caso R_1 seja maior que 100, isto significa que *WCET* é menor que SIM, o que é claramente um erro da análise (análise é otimista).

Tabela 19: Análise do *WCET*

Benchmark	*WCET* com AV	SIM	R_1
break2	150	150	100
bs	232	226	97,4
bsort100	924669	460830	49,8
cnt	8963	8963	100
cover	5834	5834	100
crc	149437	74082	49,6
edn	338775	337286	99,6
fdct	7222	7222	100
fibcall	184	184	100
insertsort	6029	1781	29,5
janne complex	3537	544	15,4
lcdnum	716	486	67,9
matmult	582729	582729	100
ns	25815	25809	100
prime	13514	13496	99,9
swap	31	31	100

Observa-se que em alguns *benchmarks* (*break2*, *cnt*, *cover*, *fdct*, *fibcall*, *matmult*, *ns*, *swap*) o valor obtido para o *WCET* com a análise de valor foi igual ao tempo de execução (SIM) ($R_1 = 100$).

Existem casos onde há uma grande diferença entre o *WCET* e a execução real do programa. Conforme os resultados apresentados da análise dos *benchmarks*, essa diferença ocorre normalmente quando o número máximo de iterações dos laços é superior à execução real.

Para facilitar a visualização dos resultados da ferramenta *WCET* com a análise de valor (WCET com AV) comparada com os resultados do simulador (SIM), os *benchmarks* foram divididos em dois gráficos, um com número de ciclos inferior a 10000 e outro com número de ciclos superior a este, mostrados respectivamente nas Figuras 15 e 16.

Figura 15: Comparativo do simulador com a análise *WCET*

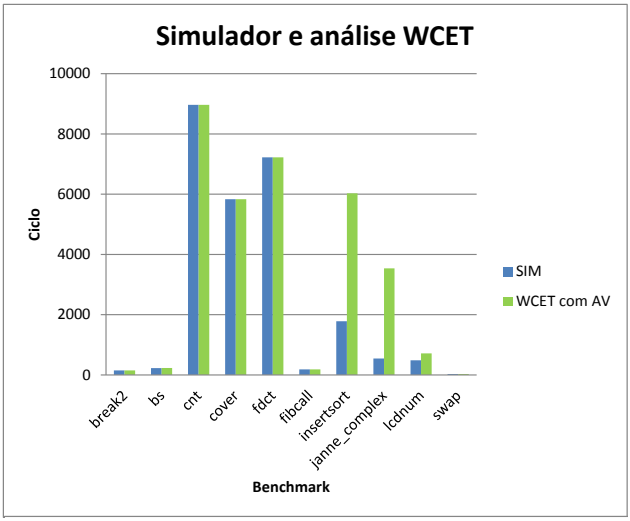
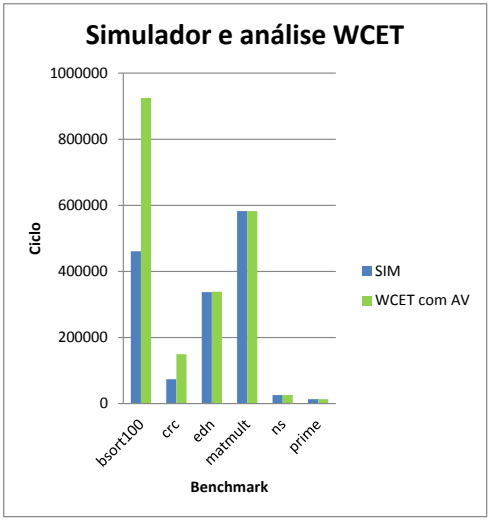


Figura 16: Comparativo do simulador com a análise *WCET*



Na Tabela 20, mostra-se para cada *benchmark* o *WCET* obtido sem considerar a análise de valor (qualquer instrução de acesso à memória possui latência determinada pela memória principal), e a análise de *WCET* com a análise de valor implementada. A relação (R_2) dessas análises é dada pela Equação 5.2 e pode ser visualizada na Tabela 20.

$$R_2 = \left(\frac{(WCET\ sem\ AV) - (WCET\ com\ AV)}{(WCET\ sem\ AV)} \right) * 100 \qquad (5.2)$$

AV: análise de valor.

Tabela 20: Análise do *WCET* com e sem análise de valor

Benchmark	*WCET* com AV	*WCET* sem AV	R_2
break2	150	396	62,1
bs	232	526	55,9
bsort100	924669	1891953	51,1
cnt	8963	21113	57,5
cover	5834	12452	53,1
crc	149437	370231	59,6
edn	338775	831993	59,3
fdct	7222	18940	61,9
fibcall	184	688	73,3
insertsort	6029	10325	41,6
janne complex	3537	9417	62,4
lcdnum	716	1574	54,5
matmult	582729	1064715	45,3
ns	25815	60501	57,3
prime	13514	37010	63,5
swap	31	73	57,5

Observa-se que a análise de *WCET* com a análise de valor diminuiu significativamente o número de ciclos. Por exemplo, o *WCET* do *benchmark break2* sem análise de valor foi 396 e com a análise de valor foi 150, diminuindo 246 ciclos. As Figuras 17 e 18 ilustram o resultado da ferramenta *WCET* com a análise de valor (WCET com AV) comparada com a ferramenta sem a análise de valor (WCET sem AV). Para facilitar a visualização dos resultados a primeira figura é composta por *benchmark* com número de ciclos inferior a 25000, e a segunda por *benchmark* com número de ciclos superior a este.

Figura 17: Comparativo da análise *WCET* com e sem a análise de valor

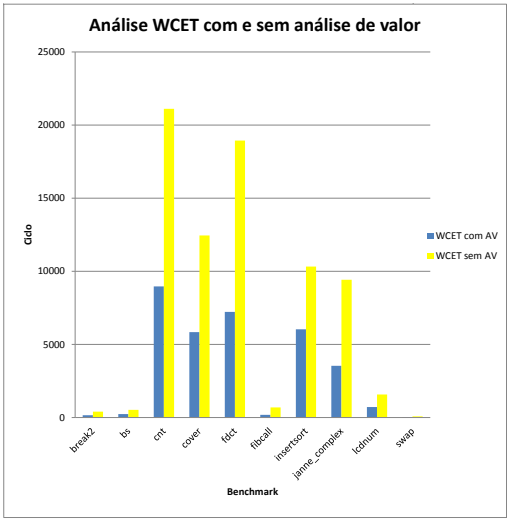
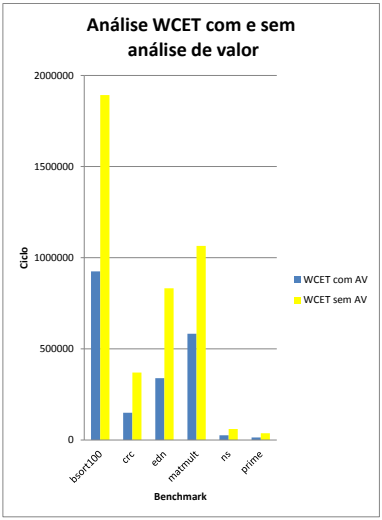


Figura 18: Comparativo da análise *WCET* com e sem a análise de valor



5.3.2 WCET com a análise de valor considerando o conteúdo da memória

Na Tabela 21, mostra-se para cada *benchmark* o *WCET* com a análise de valor considerando a implementação da memória, o tempo de execução, e a relação (R_3), entre o tempo de execução (SIM) e o *WCET*.

A relação (R_3) é dada pela Equação 5.3.

$$R_3 = \left(\frac{SIM}{WCET} \right) * 100 \tag{5.3}$$

Tabela 21: Análise do *WCET* considerando o conteúdo da memória

Benchmark	*WCET*	SIM	R_3
break2	150	150	100
bs	232	226	97,4
bsort100	924669	460830	49,8
cnt	8963	8963	100
cover	5834	5834	100
crc	149425	74082	49,6
edn	338775	337286	99,6
fdct	7222	7222	100
fibcall	184	184	100
insertsort	4517	1781	39,4
janne complex	3537	544	15,4
lcdnum	716	486	67,9
matmult	582729	582729	100
ns	25815	25809	100
prime	13508	13496	99,9
swap	31	31	100

O valor de R_3 fica entre 0 (quando *WCET* é expressivamente maior que SIM) e 100 (quando o SIM é igual a *WCET*). Caso R_3 seja maior que 100, isto significa que *WCET* é menor que SIM, o que é claramente um erro da análise (análise é otimista).

As Figuras 19 e 20 ilustram o resultado da ferramenta *WCET* com a análise de valor considerando o conteúdo da memória (*WCET* com AV+MEM) comparada com o resultado do simulador. Para facilitar a visualização, a primeira figura é composta por *benchmark* que varia o número de ciclos de 0 a 10000, e a segunda por *benchmark* com número de ciclos superior a este.

Figura 19: Comparativo do simulador com a análise *WCET*

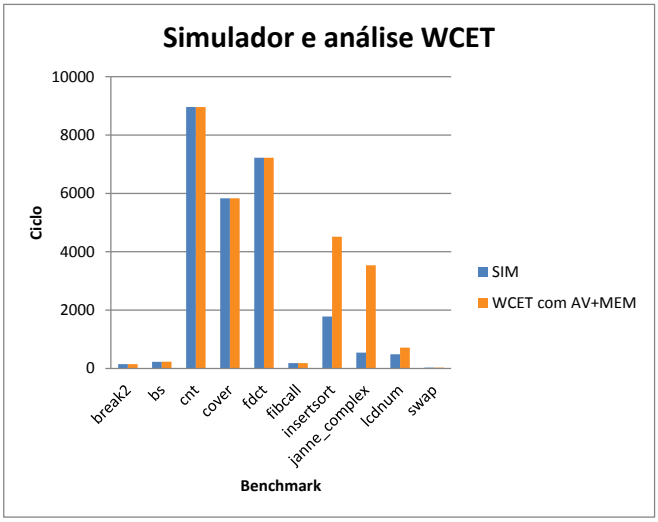
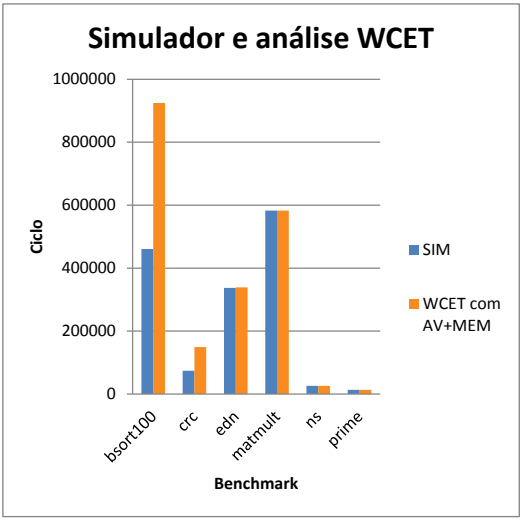


Figura 20: Comparativo do simulador com a análise *WCET*



Observa-se que a análise do *WCET* com a análise de valor considerando a implementação da memória diminuiu o número de ciclos em alguns casos. O conteúdo do registrador destino de uma instrução de leitura da memória passa a ser conhecido, gerando um limite superior mais apertado para o *WCET*.

5.3.3 Análise *WCET*

As Figuras 21 e 22 ilustram o tempo de execução obtido do simulador, da ferramenta *WCET* com a análise de valor implementada considerando o conteúdo da memória (*WCET* com AV+MEM), sem considerar o conteúdo de memória (*WCET* com AV), e uma análise pessimista que considera todo acesso à memória no seu pior caso (*WCET* sem AV). Para facilitar a visualização, a primeira figura é composta por *benchmark* com número de ciclos inferior a 10000, e a segunda por *benchmarks* com número de ciclos superior a este.

Figura 21: Comparativo do simulador com as análises *WCET*

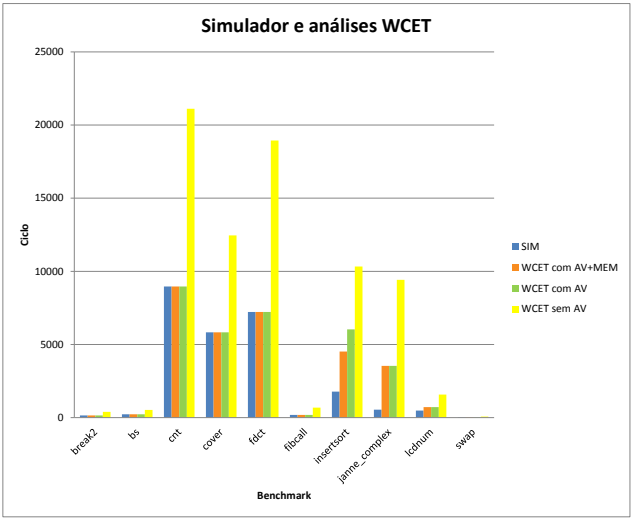
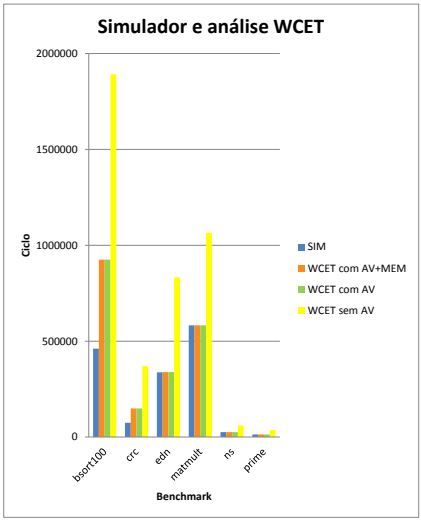


Figura 22: Comparativo do simulador com as análises *WCET*



Observa-se que, com a implementação da análise de valor, o número de ciclos diminuiu significativamente, conforme já foi exposto anteriormente. Em alguns *benchmark* o resultado da análise de valor foi equivalente ao simulador, e em outros ainda existe uma diferença.

Comparando a análise *WCET* onde é considerado que qualquer instrução de acesso à memória possui latência determinada pela memória principal, com a análise de valor desenvolvida neste trabalho, é notório o ganho na análise *WCET* com a ferramenta desenvolvida, como mostrado nas Figuras 21 e 22.

Na Tabela 22 é mostrado para cada *benchmark* que foi analisado, o número de blocos básicos e o número de instruções. Observa-se que tem *benchmark* composto por vários blocos básicos mas com poucas instruções, e o inverso também, poucos blocos básicos mas com várias instruções. O maior caso testado foi o *benchmark edn* que é composto por 58 blocos básicos e 1295 instruções.

O número de instruções é o somatório das instruções de todos os blocos básicos sem levar em consideração quantas vezes a instrução é executada, um bloco básico pode ser executado várias vezes (*loop*). Considera-se na contagem todas as instruções, não apenas as instruções

de acesso à memória.

Tabela 22: Número de blocos básicos e instruções

Benchmark	número de BB	número de instruções
break2	9	49
bs	12	85
bsort100	20	140
cnt	27	182
cover	30	151
crc	59	605
edn	58	1295
fdct	11	530
fibcall	9	57
insertsort	10	83
janne complex	17	93
lcdnum	27	185
matmult	41	262
ns	20	134
prime	42	238
swap	5	23

5.4 CONSIDERAÇÕES FINAIS

A classificação correta das instruções de acesso à memória tem grande impacto no *WCET*. Conforme apresentado no Capítulo 3, a penalidade de acesso à memória principal é cinco vezes superior ao acesso a *ScratchPad Memory*. Sem a análise de valor, o *WCET* dos programas é superestimado consideravelmente.

Existem casos onde a diferença entre o pior tempo de execução e a execução real do programa é grande. Geralmente esta diferença acontece em *benchmarks* onde o número máximo de iterações dos laços é inferior à execução real.

Em *benchmarks* onde o fluxo e número de iterações são mais comportados, a análise *WCET* apresentou resultados precisos devido ao comportamento determinista do processador e ao fluxo comportado do programa.

Através dos experimentos descritos neste capítulo, podemos con-

cluir que a ferramenta produz resultados realistas, e não é otimista.

6 CONCLUSÃO E TRABALHOS FUTUROS

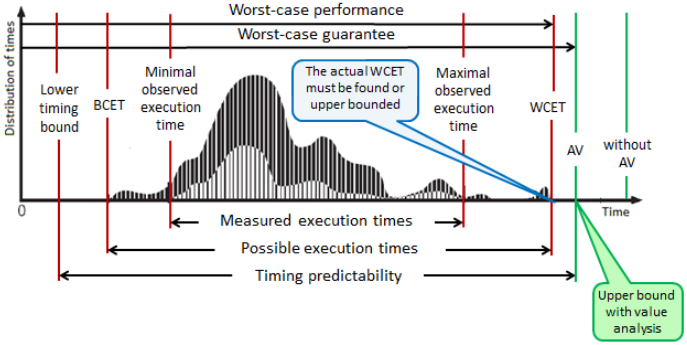
A análise de sistemas de tempo real é uma tarefa desafiadora e requer a união de inúmeras técnicas, conforme apresentado neste trabalho. A obtenção analítica do pior tempo de execução de tarefas é uma etapa importante no desenvolvimento e validação de sistemas de tempo real críticos.

Os valores estimados do pior tempo de execução podem ser utilizados para verificar: a análise de escalonamento, se as tarefas periódicas satisfazem seus objetivos de desempenho, se as interrupções possuem tempos de reação suficientemente curtos, gargalos de desempenho, entre outros (ALVAREZ, 2013).

O limite *WCET* é a base para a garantia de pior caso que a aplicação, tarefa, sistema, pode oferecer. A precisão desta informação temporal depende dos métodos utilizados na análise e das propriedades do sistema.

O processador utilizado neste trabalho possui memória de dados com latências diferentes. É necessária a análise de valor para identificar a região de memória que cada instrução acessa. Conforme mostrado na Figura 23, sem a análise de valor, o *WCET* dos programas é superestimado consideravelmente. O objetivo é fornecer um tempo de execução seguro, mas não extremamente pessimista o que dificultaria sua utilização prática.

Figura 23: Analise temporal de sistema



Fonte: Adaptado de Wilhelm et al. (2008)

A partir da reconstrução do fluxo do controle do programa são

formados os blocos básicos de execução e é realizado um mapeamento dos acessos aos segmentos de memória, possibilitando classificar as instruções (acesso à memória principal ou *ScratchPad Memory*). Como a penalidade de acesso à memória principal é cinco vezes superior, a classificação correta das instruções tem grande impacto no *WCET*.

Considerou-se primeiramente um cenário onde na interpretação estática das instruções não era considerado o conteúdo da memória, o registrador destino da instrução de leitura de memória recebia um valor desconhecido e as instruções são interpretadas apenas quando seus parâmetros são conhecidos. Posteriormente, considerou-se o conteúdo da memória na análise, melhorando os resultados da interpretação estática das instruções e imediatamente gerando um limite superior mais apertado para o *WCET*.

O ganho com a inclusão do conteúdo da memória na análise não foi tão expressivo nos *benchmarks* analisados que foram apresentados neste trabalho. Para exemplos que tem uso de ponteiro, como o programa *array-pointer*, conforme foi mostrado no Capítulo 4 a análise é interessante. A necessidade da análise do conteúdo da memória vai depender da aplicação, devido ao fato que tem um custo maior em questão de implementação.

Devido ao comportamento determinista do processador e ao fluxo comportado do programa, a análise *WCET* apresentou resultados precisos para alguns *benchmarks* nos quais o fluxo e número de iterações são comportados.

A principal contribuição deste trabalho é mostrar que, com a análise de valor, é possível determinar qual região de memória cada instrução acessa, e consequentemente utilizar o tempo correto de execução. Como o tempo de acesso às regiões de memória são diferentes, a identificação correta gera uma limite superior do *WCET* mais preciso.

Durante o desenvolvimento deste trabalho foi publicado um artigo com os resultados preliminares obtidos:

Karila Palma Silva; Renan Augusto Starke; Rômulo Silva de Oliveira. *Análise de Valor para a Determinação do WCET de Tarefas em Sistemas de Tempo Real.* IV Simpósio Brasileiro de Engenharia e Sistemas Computacionais, Manaus, 2014.

Como trabalhos futuros, pode-se citar:

Análise para determinar limites de *loop*: Método para determinar os valores mínimos e máximos de iterações de laços de um programa.

Análise para detecção de caminhos inviáveis: Método para determinar caminhos inviáveis de um programa.

Implementação da análise de valor por interpretação abstrata: Essa técnica é baseada na utilização de uma semântica abstrata para representar os valores durante a execução de um programa, para extrair propriedades de um programa sem que seja necessário executá-lo. É necessário definir as semânticas concretas de forma simplificada que descrevam os aspectos interessantes para a análise de valor, também definir as semânticas abstratas que coletam os aspectos temporais em cada ponto do programa. Para cada processador suportado é necessário construir um modelo seguindo a teoria da interpretação abstrata. (OYAMADA, 2004; THESING, 2004).

Implementação da análise de valor por análise de intervalo simbólico: Evita a computação de ponto fixo (a faixa de números que podem representar um valor é fixa e os resultados têm que estar dentro da faixa, caso contrário não produzirá resultados precisos). Nessa análise, transforma o problema do valor da variável de intervalo em uma otimização de programação linear, permitindo obter limites das variáveis precisas em tempo polinomial (MARKOVSKIY, 2002; RUGINA; RINARD, 2005).

REFERÊNCIAS

ALVAREZ, G. A. P. *Caracterização Analítica de Carga de Trabalho Baseada em Cenários de Aplicações Multimídia*. Tese (Doutorado) — Universidade de São Paulo, 2013.

BUTTAZZO, G. C. *Hard real-time computing systems: predictable scheduling algorithms and applications*. [S.l.]: Springer, 2011.

BYGDE, S. Data-flow analysis for worst-case execution time. *Sci. Comput. Program*, v. 75, n. 9, p. 796–807, 2010.

COPELAND, L. *A Practitioners Guide to Software Test Design. Norwood, MA, USA: Artech House*. [S.l.]: Inc, 2003.

CULLMANN, C. et al. Predictability considerations in the design of multi-core embedded systems. *Proceedings of Embedded Real Time Software and Systems*, p. 36–42, 2010.

ENGBLOM, J. Processor pipelines and static worst-case execution time analysis. Acta Universitatis Upsaliensis, 2002.

ENGBLOM, J.; ERMEDAHL, A.; SJöDIN, M. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer*, v. 4, p. 437–455, 2003.

FARINES, J.-M.; da Silva Fraga, J.; OLIVEIRA, R. S. de. *Sistemas de Tempo Real*. [S.l.]: Universidade Federal de Santa Catarina (UFSC), 2000.

GUSTAFSSON, J. et al. The Mälardalen WCET benchmarks – past, present and future. In: LISPER, B. (Ed.). Brussels, Belgium: OCG, 2010. p. 137–147.

GUSTAFSSON, J. et al. A tool for automatic flow analysis of c-programs for wcet calculation. In: IEEE. *Object-Oriented Real-Time Dependable Systems, 2003.(WORDS 2003). Proceedings of the Eighth International Workshop on*. [S.l.], 2003. p. 106–112.

HARRISON, H. H. Compiler analysis of the value ranges for variables. *Software Engineering, IEEE Transactions on*, n. 3, p. 243–250, 1977.

HECKMANN, R.; FERDINAND, C. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. *Automation and Test in Europe-Volume 1*, n. IEEE Computer Society, p. 618—-619, 2005.

INITIATIVE, A. S. Systemc. In: . <http://www.accellera.org/home/>: [s.n.], 2013.

LI, Y.-T. S.; MALIK, S. Performance analysis of embedded software using implicit path enumeration. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1995. v. 30, n. 11, p. 88–98.

LUNDQVIST, T. *A WCET analysis method for pipelined microprocessors with cache memories*. Tese (Doutorado) — Department of Computer Engineering. Chahalmers University of Technology. Goteborg, 2002.

MARKOVSKIY, Y. Range Analysis with Abstract Interpretation. *Semester Project, CS*, n. 13099223, p. 1–8, 2002.

OYAMADA, M. S. *Métodos de estimativa de desempenho em software embarcado*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, 2004.

PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: the hardware/software interface*. [S.l.]: Newnes, 2013.

PATTERSON, J. R. Accurate static branch prediction by value range propagation. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1995. v. 30, n. 6, p. 67–78.

RUFINO, L. M. Análise de Tempo de Execução Utilizando LLVM. Monografia (Bacharelado em Ciência da Computação). Universidade Federal de Santa Catarina. Departamento de Informática e estatística, 2008.

RUGINA, R.; RINARD, M. C. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Transactions on Programming Languages and Systems*, v. 27, n. 2, p. 185–235, mar. 2005. ISSN 01640925.

SAINRAT, P.; CASSé, H.; BIRéE, F. Multi-architecture Value Analysis for Machine Code. In: MAIZA, C. (Ed.). *13th International Workshop on Worst-Case Execution Time Analysis*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.

(OpenAccess Series in Informatics (OASIcs), Wcet), p. 42–52. ISBN 978-3-939897-54-5. ISSN 2190-6807.

SETHI, R.; ULLMAN, J. D.; LAM, M. S. [S.l.]: Pearson Addison Wesley, 2008.

STARKE, R. A. *Uma Abordagem de escalonamento heterogêneo preemptivo e não preemptivo para sistemas de tempo real com garantia em multiprocessadores*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2012.

STARKE, R. A. *Processador Determinista para Aplicações de Tempo Real*. Qualificação (Doutorado) − Universidade Federal de Santa Catarina. Acessível sob pedido. 2013.

THESING, S. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models Dissertation*. Tese (Doutorado) — Universität des Saarlandes, 2004.

WENZEL, I. et al. Principles of timing anomalies in superscalar processors. In: IEEE. *Quality Software, 2005.(QSIC 2005). Fifth International Conference on*. [S.l.], 2005. p. 295–303.

WILHELM, R. et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, v. 7, n. 3, p. 1–53, abr. 2008. ISSN 15399087.