UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS

Renan Augusto Starke

UMA ABORDAGEM DE ESCALONAMENTO HETEROGÊNEO PREEMPTIVO E NÃO PREEMPTIVO PARA SISTEMAS DE TEMPO REAL COM GARANTIA EM MULTIPROCESSADORES

Florianópolis

2012

Renan Augusto Starke

UMA ABORDAGEM DE ESCALONAMENTO HETEROGÊNEO PREEMPTIVO E NÃO PREEMPTIVO PARA SISTEMAS DE TEMPO REAL COM GARANTIA EM MULTIPROCESSADORES

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: Rômulo Silva de Oliveira, Dr.

Florianópolis

2012

Catalogação na fonte pela Biblioteca Universitária
da
Universidade Federal de Santa Catarina

S795a Starke, Renan Augusto
Uma abordagem de escalonamento heterogêneo preemptivo e não preemptivo para sistemas de tempo real com garantia em multiprocessadores [dissertação] / Renan Augusto Starke ; orientador, Rômulo Silva de Oliveira. - Florianópolis, SC, 2012.
200 p.: il., grafs., tabs.

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

Inclui referências

1. Engenharia de sistemas. 2. Automação. 3. Multiprocessadores. 4. Escalonamento. 5. Memória cache. I. Oliveira, Rômulo Silva de. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas. III. Título.

CDU 621.3-231.2(021)

Renan Augusto Starke

UMA ABORDAGEM DE ESCALONAMENTO HETEROGÊNEO PREEMPTIVO E NÃO PREEMPTIVO PARA SISTEMAS DE TEMPO REAL COM GARANTIA EM MULTIPROCESSADORES

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre em Engenharia de Automação e Sistemas", e aprovada em sua forma final pelo programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Florianópolis, 10 de fevereiro 2012.

José Eduardo Ribeiro Cury, Dr.
Coordenador

Banca Examinadora:

Rômulo Silva de Oliveira, Dr
Presidente

Alexandre José da Silva, MSc

Carlos Barros Montez, Dr

George Marconi de Araújo Lima, Ph.D

AGRADECIMENTOS

Agradeço ao meu orientador Rômulo Silva de Oliveira pela dedicação, incentivo, paciência e pelas ótimas sugestões quando o caminho parecia obscuro.

Agradeço à minha família pela confiança e apoio dado no decorrer do curso. Em especial agradeço aos meus pais pelo acesso à uma educação de boa qualidade e íntegra.

Agradeço também aos amigos do LTIC pela amizade, ajuda em correções, compartilhamento e aprimoramento das ideias.

Por fim agradeço a CAPES pelo apoio financeiro que tornou possível a realização deste trabalho.

Aprender é a única coisa de que a mente nunca se cansa, nunca tem medo e nunca se arrepende.

Leonardo da Vinci

RESUMO

Sistemas de tempo real são sistemas onde o correto funcionamento não depende somente da resposta lógica correta, mas também do tempo no qual ela foi dada. Igualmente do ponto de vista lógico, a viabilidade temporal da aplicação deve ser determinada através de técnicas, como por exemplo análise do tempo de reposta. Este tipo de aplicação está cada vez mais presente atualmente e a demanda de processamento é tamanha que necessita-se de processadores com múltiplos núcleos complexos. É perceptível que o desenvolvimento dos multiprocessadores está muito mais avançado em relação às técnicas de análise de tais sistemas e, portanto, é evidente a necessidade de pesquisa com objetivo de promover maior confiabilidade e redução de superdimensionamentos. O objetivo deste trabalho é promover uma solução de escalonamento que considere a escalonabilidade em conjunto com a analisabilidade do código da aplicação. Atualmente, a pesquisa de sistemas de tempo real trata o problema do escalonamento isolado do problema de obtenção do parâmetro do tempo de computação da tarefas (WCET – *Worst Case Execution Time*). Dependendo da arquitetura do processador, as premissas adotadas no cálculo do WCET são incompatíveis com as premissas de escalonamento, o que gera uma contradição fundamental entre o cálculo do WCET e os algoritmos de escalonamento. A incompatibilidade das premissas pode ser ilustrada pela preempção em arquiteturas com memória *cache*, onde o cálculo de WCET assume execução contínua da tarefa, o que não é verdade em grande parte dos algoritmos de escalonamento. Este trabalho propõe o uso de uma abordagem heterogênea em multiprocessadores onde parte dos núcleos operam em regime preemptivo e parte em regime não preemptivo para tentar lidar com as diferentes considerações sobre preempção. As análises realizadas mostram que existe vantagem em usar a abordagem heterogênea.

Palavras-chave: tempo real, *Worst Case Execution Time*, multiprocessadores, escalonamento.

ABSTRACT

Real-time systems are systems where the correct functioning depends not only on the logically correct response, but also the time when it was given. As the the logic functionality, the application response time could be analyzed to determine the viability of a real-time system. This type of application is increasingly present today and the processing demand is such that complex multi-core processors are needed. It is noticeable that the development of multiprocessor is a long way ahead compared with the techniques of analysis of such systems and is therefore necessary researches to promote more reliability and to reduce over-specified systems. The objective of this work is to promote a solution that considers scheduling in conjunction with the analyzability of the application code. Currently, the real-time research considers the scheduling problem isolated from the WCET (Worst Case Execution Time) problem. Depending on the processor architecture, the values obtained by computing WCET are incompatible with the scheduling model which creates a fundamental contradiction between the assumptions of calculation of WCET and scheduling algorithms. This work proposes the use of a heterogeneous approach where part of the multiprocessor cores operate under preemptive and part on a non-preemptive scheduling. The analysis shows that there are advantages using the heterogeneous approach.

Keywords: real-time, Worst Case Execution Time, multiprocessors, scheduling.

LISTA DE FIGURAS

Figura 1 Técnicas de balanceamento de *pipeline*. 38
Figura 2 Exemplo de unificação de recursos de um *pipeline*. 41
Figura 3 Exemplo de *pipeline* diversificado. 43
Figura 4 Exemplo de *pipeline* dinâmico. 44
Figura 5 Relação e hierarquia de memória. 45
Figura 6 Interação entre *cache* e TLB. 50
Figura 7 Modelo padrão desenvolvido para análise estática de código (CULLMANN et al., 2010). 57
Figura 8 Exemplo de grafo de controle fluxo 63
Figura 9 Exemplo de grafo de análise de pilha 63
Figura 10 Exemplo de grafo de análise de tempo 64
Figura 11 Grafo de fluxo de controle para o exemplo da tabela *hash*. . . . 66
Figura 12 Grafo de pior tempo de execução da tabela *hash*. 68
Figura 13 Grafo de utilização de pilha para exemplo da tabela *hash*. 69
Figura 14 Exemplo de anomalia temporal. 89
Figura 15 Exemplo de atrasos de *cache* direto e indireto. 96
Figura 16 Atraso de preempção da tarefa 1 sobre a tarefa 2. 98
Figura 17 Resumo das técnicas de inclusão de efeitos na cache no teste de escalonabilidade 103
Figura 18 Diagrama de blocos do processador *FreeScale* MPC5668G (CULLMANN et al., 2010) 106
Figura 19 Particionamento de um conjunto de tarefas $\mathcal{T}$ sob o mesmo regime de preempção. 112
Figura 20 Particionamento de um conjunto de tarefas $\mathcal{T}$ sob diferentes regimes de preempção. 112
Figura 21 Resultados do escalonamento não preemptivo. 128
Figura 22 Resultados do escalonamento preemptivo, $\gamma = 20\%$ de C_i 130
Figura 23 Resultados do escalonamento heterogêneo para 4 processadores, $\gamma = 20\%$ de C_i 133
Figura 24 Resultados do escalonamento heterogêneo para 16 processadores, $\gamma = 20\%$ de C_i 134
Figura 25 Comparação entre particionamento homogêneo e heterogêneo para 4 processadores com $\gamma = 20\%$ de C_i 135

Figura 26 Comparação entre particionamento homogêneo e heterogêneo para 16 processadores com $\gamma = 20\%$ de C_i 136
Figura 27 Técnicas de balanceamento de *pipeline* (Apêndice). 155
Figura 28 Exemplo de unificação de recursos de um *pipeline* (Apêndice). 159
Figura 29 *Pipeline* com inclusão de atalhos (Apêndice). 163
Figura 30 *Pipeline* do i486 (a) e Pentium (b) (Apêndice). 165
Figura 31 Exemplo de *pipeline* diversificado (Apêndice). 166
Figura 32 Exemplo de *pipeline* dinâmico (Apêndice). 167
Figura 33 Exemplo de *pipeline* superescalar (SEHN; LIPASTI, 2005) (Apêndice). 170
Figura 34 Exemplo de máquina de estados finitos para previsão de controle de fluxo (Apêndice). 172
Figura 35 Relação e hierarquia de memória (Apêndice). 173
Figura 36 Propriedades da memória *cache* (Apêndice). 179
Figura 37 Interação entre *cache* e TLB (Apêndice). 185
Figura 38 Processamento escalar e vetorial de um supercomputador (Apêndice). 192
Figura 39 Porções de trabalho vetorizado e sequencial (Apêndice). 193
Figura 40 Perfil de execução de um *pipeline* de *N* estágios (Apêndice). . 194
Figura 41 Perfil de execução de um *pipeline* com flutuação (Apêndice). 194
Figura 42 Modelo de um *pipeline* com flutuação (Apêndice). 196

LISTA DE TABELAS

Tabela 1 Especificação das instruções aritméticas/ponto flutuante..... 39
Tabela 2 Especificação das instruções de acesso a memória..... 40
Tabela 3 Especificação das instruções controle de fluxo..... 40
Tabela 4 WCET para diferentes configurações de memória..... 70
Tabela 5 Classificação dos algoritmos de escalonamento (CARPENTER et al., 2004)..... 81
Tabela 6 Especificação das instruções aritméticas/ponto flutuante (Apêndice)..... 157
Tabela 7 Especificação das instruções de acesso à memória (Apêndice). 157
Tabela 8 Especificação das instruções de controle de fluxo (Apêndice). 158
Tabela 9 Máxima penalidade envolvendo violações semânticas (Apêndice)..... 161
Tabela 10 Máxima penalidade com implementação de atalhos (Apêndice)..... 162
Tabela 11 Relação de mudança de parâmetros sobre faltas (Apêndice). . 181
Tabela 12 Ações do protocolo de coerência MESI (Apêndice)..... 187

LISTA DE ABREVIATURAS E SIGLAS

ACET	*Average Case Execution Time* – Tempo de execução de caso médio
BI	Estágio de Busca de Instrução
BIA	*Branch Instruction Address* – endereço da instrução alvo
BO	Estágio de Busca de Operandos
BTB	*Branch Target Buffer* – memória de endereço alvo
CISC	*Complex Instruction Set Computer* – computador com um conjunto complexo de instruções
CP	Contador de programa
CPI	*Cycles per Instruction* – ciclos por instrução
CPU	*Central Processing Unit* – processador
DC	*Deadline* relativo crescente
DD	*Deadline* relativo decrescente
DI	Estágio de Decodificação de Instrução
DM	*Deadline Monotonic* – prioridade das tarefas atribuídas pelo prazo (*deadline*) monotônico: menor prazo, maior prioridade
DRAM	*Dynamic random access memory* – memória dinâmica de acesso aleatório
ECU	*Electronic Control Unit* – unidade de controle eletrônica
EDF	*Earliest Deadline First* – prazo (*deadline*) mais próximo antes
EX	Estágio de Execução
FC	Folga (*laxity*) crescente
FD	Folga (*laxity*) decrescente
FIFO	*First In, First Out* – primeiro a entrar, primeiro a sair
FPGA	*Field Programmable Gate Array* – arranjo de portas programável em campo
FPU	*Floating Point Unit* – unidades de ponto flutuante
GO	Estágio de Gravação de Operandos
GPU	*Graphics Processing Unit* – unidade de processamento gráfico
IPET	*Implicit Path Enumeration Technique* – Técnica de enumeração implícita de caminhos
ISA	*Instruction Set Architecture* – arquitetura de conjunto de instruções

JOP	*Java Optimized Processor* – processador Java otimizado para tempo real
L1	Nível primário (mais próximo do processador) da memória *cache*
L2	Nível secundário da memória *cache*
LIFO	*Last In, First Out* – último a entrar, primeiro a sair
LLF	*Least Laxity First* – menor folga antes
LRU	*Least Recently Used* – menos recentemente utilizado
LT	Estágio de Leitura dos registradores
ULA	Unidade Lógica e Aritmética
MAM	*Memory Accelarator Module* – módulo acelerador de memória
MEM	Estágio de acesso à Memória
MEI	Estados e nome do protocolo de coerência: *Modified*, *Exclusive* e *Invalid*
MESI	Estados e nome do protocolo de coerência: *Modified*, *Exclusive*, *Shared* e *Invalid*
MOESI	Estados e nome do protocolo de coerência: *Modified*, *Owned*, *Exclusive* e *Invalid*
MESIF	Estados e nome do protocolo de coerência: *Modified*, *Exclusive*, *Invalid* e *Forward*
PD	Período (intervalo mínimo de chegadas) decrescente
PC	Período (intervalo mínimo de chegadas) crescente
P-LRU	Pseudo-*Least Recently Used* – pseudo menos recentemente utilizado
REG	Estágio de Escrita dos registradores
RM	*Rate Monotonic* – prioridades das tarefas atribuídas em taxa monotônica: menor período, maior prioridade
RISC	*Reduced Instruction Set Computer* – computador com um conjunto reduzido de instruções
RTA	*Response Time Analysis* – análise de tempo de resposta
SRAM	*Static Random Access Memory* – memória estática de acesso aleatório
TDMA	*Time Division Multiple Access* – acesso múltiplo por divisão de tempo
TLB	*Translation Lookaside Buffer* – memória de tradução de endereços
UC	Utilização crescente

UD	Utilização decrescente
VILW	*Very Long Instruction Word* – arquitetura de processador com palavras de instrução longas
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
γ	Custo adicional de preempção
$h(t)$	Demanda total de processador no tempo t
L	Período de ocupação do processador
m	Número total de processadores disponíveis: cardinalidade de Π ($m = \#\Pi$)
m_p	Número de processadores em regime preemptivo: cardinalidade de Π_p ($m_p = \#\Pi_p$)
m_{np}	Número de processadores em regime não preemptivo: cardinalidade de Π_{np} ($m_{np} = \#\Pi_{np}$)
n	Número de tarefas
nP	Não preemptivo
$\aleph_0$	Conjunto dos número naturais mais o número 0
P	Preemptivo
PC_i	Custo adicional de preempção determinado por problema de otimização
π_j	Processador j
PP	Pontos de preempção
Π	Conjunto total de processadores do sistema
Π_{np}	Conjunto de processadores em regime não preemptivo
Π_p	Conjunto de processadores em regime preemptivo
R	Tempo de resposta de uma tarefa
S	Conjunto discreto de tempo: $S = \{t_1, ..., t_{max}\}$
t	Tempo
T	Intervalo mínimo de chegadas ou período de uma tarefa
τ	Tarefa
$\mathcal{T}$	Conjunto total de tarefas do sistema
$\mathcal{T}_{np}$	Conjunto de tarefas em regime não preemptivo não alocadas
$\mathcal{T}_p$	Conjunto de tarefas em regime preemptivo não alocadas
$\mathcal{T}_{\pi_j}$	Conjunto de tarefas alocadas ao processador j
u	Utilização de uma tarefa
U_{π_j}	Utilização do processador j

$W_i(t)$ Demanda no processador para uma tarefa i no tempo t (escalonamento *Rate Monotonic*)

$Wo(t)$ Carga de trabalho do processador no tempo t

SUMÁRIO

1 INTRODUÇÃO 29
1.1 CONCEITOS BÁSICOS E MOTIVAÇÃO 29
1.2 OBJETIVO DO TRABALHO 33
1.3 ORGANIZAÇÃO DO TEXTO 33
2 ARQUITETURA DE PROCESSADORES CONTEMPORÂNEOS 35
2.1 PRINCÍPIOS DE *PIPELINE* 36
2.2 PROCESSADORES SUPERESCALARES 42
2.3 TÉCNICAS DE PREVISÃO DE FLUXO 44
2.4 ARQUITETURA DE MEMÓRIA 45
2.4.1 Memória *cache* 46
2.4.2 Proteção de memória 48
2.4.3 Interação *cache* e TLB 49
2.4.4 Protocolos de coerência de *cache* para multiprocessadores 50
2.5 DESEMPENHO DE PROCESSADORES 51
2.6 CONCLUSÕES E TENDÊNCIAS 52
3 ANÁLISE ESTÁTICA DO WCET – *WORST CASE EXECUTION TIME* 55
3.1 FUNDAMENTOS DE ANÁLISE ESTÁTICA TEMPORAL DE CÓDIGO 56
3.1.1 Análise de valor 57
3.1.2 Análise de fluxo de controle 58
3.1.3 Análise da microarquitetura 59
3.1.4 Obtenção do pior caminho 60
3.2 AVALIAÇÃO PRÁTICA DE FERRAMENTA WCET – AIT 61
3.2.1 Obtenção do WCET de uma tarefa 64
3.2.2 Resultados da Análise WCET e de pilha 67
3.3 CONCLUSÃO 71
4 DISCUSSÃO SOBRE A OBTENÇÃO DE GARANTIA TEMPORAL NA ANÁLISE DE SISTEMA DE TEMPO REAL CRÍTICO 73
4.1 ESCALONAMENTO DE TEMPO REAL 74
4.1.1 Escalonamento com prioridade fixa para um processador 75
4.1.2 Escalonamento com prioridade dinâmica para um processador 77
4.1.3 Escalonamento para multiprocessadores 79
4.1.3.1 Resultados fundamentais independentes de algoritmos 82
4.1.3.2 Escalonamento particionado 83

4.1.3.3 Escalonamento global 84
4.1.3.4 Escalonamento híbrido 85
4.1.4 Escalonamento cooperativo (pontos fixos de preempção) para um processador 85
4.1.5 Escalonamento não preemptivo para um processador 86
4.2 CONSIDERAÇÕES SOBRE WCET E ARQUITETURA DE PROCESSADORES 88
4.3 *CACHE* NA ANÁLISE DE ESCALONABILIDADE 93
4.3.1 Incorporação dos efeitos de *cache* como interferência direta no tempo de computação 96
4.3.2 Incorporação dos efeitos de *cache* como problema de otimização 98
4.3.3 Incorporação dos efeitos de *cache* considerando interferências entre tarefas explicitamente 102
4.3.4 Resumo das abordagens 103
4.4 MODELO DE ARQUITETURA DETERMINISTA 104
4.5 CONCLUSÃO 107
5 PROPOSTA DE UMA ABORDAGEM DE ESCALONAMENTO HETEROGÊNEO 109
5.1 MOTIVAÇÃO 110
5.2 DESCRIÇÃO DA ABORDAGEM PROPOSTA 111
5.3 FORMALIZAÇÃO DO SISTEMA 113
5.4 ALGORITMO DE ESCALONAMENTO HETEROGÊNEO 114
5.5 ALGORITMOS DE CLASSIFICAÇÃO 116
5.6 ALGORITMO PARA PARTICIONAMENTO ENTRE PROCESSADORES E ESCALONAMENTO LOCAL 118
5.7 EXEMPLO DE IMPLEMENTAÇÃO DA ABORDAGEM 118
5.7.1 Algoritmo para regime preemptivo 119
5.7.2 Algoritmo para regime não preemptivo 120
5.8 CONCLUSÃO 122
6 AVALIAÇÃO DA ABORDAGEM PROPOSTA 125
6.1 CONDIÇÕES DOS EXPERIMENTOS 125
6.2 ESCOLHA DA ORDENAÇÃO PARA OS ALGORITMOS DE PARTICIONAMENTO 126
6.2.1 Seleção para escalonamento não preemptivo 128
6.2.2 Seleção para escalonamento preemptivo 129
6.3 RESULTADOS PARA ESCALONAMENTO HETEROGÊNEO 131
6.4 COMPARAÇÃO ENTRE MODELO PREEMPTIVO, NÃO PREEMPTIVO E HETEROGÊNEO 132
6.5 CONCLUSÕES 135
7 CONSIDERAÇÕES FINAIS 139

REFERÊNCIAS 143
APÊNDICE A – Arquitetura de processadores contemporâneos – versão expandida 151

1 INTRODUÇÃO

Atualmente, sistemas de tempo real são encontrados nas mais diversas aplicações como eletrônica automotiva, aviação, telecomunicação, sistemas espaciais, medicina e eletrônicos normais de consumo. Em todas estas aplicações, há um rápido desenvolvimento tecnológico. A indústria dos sistemas de tempo real embarcados, assim como todas as outras, deseja ser competitiva e ter sucesso no mercado. Para ter sucesso, esta indústria deve atender os desejos e necessidades dos clientes promovendo sistemas mais eficazes e mais flexíveis trazendo-os ao mercado o mais breve possível. Este desejo de progresso tecnológico resulta em um aumento na complexidade do *software* e, consequentemente, esta necessidade de processamento é requisitada do *hardware*.

Para atender às necessidades de processamento, os fabricantes de processadores não se concentram mais apenas na miniaturização necessária para o aumento da velocidade dos processadores. Esta abordagem causa problemas de alto consumo energético e consequentemente alta dissipação de calor. Desta forma, a concentração para o aumento de desempenho está mais voltada para fabricação de processadores com múltiplos núcleos. Esta tendência, anteriormente focada nos processadores de propósito geral de alto desempenho, também é verificada nas aplicações de tempo real.

De forma geral as aplicações de tempo real necessitarão cada vez mais de capacidade de processamento e esta capacidade é fornecida por processadores cada vez mais complexos e com múltiplos núcleos.

1.1 CONCEITOS BÁSICOS E MOTIVAÇÃO

Sistemas computacionais de tempo real são identificados como aqueles sistemas submetidos a requisitos de natureza temporal (FARINES et al., 2000). Nestes sistemas, os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto. As falhas de natureza temporal nestes sistemas são, em alguns casos, consideradas críticas no que diz respeito às suas consequências.

O tempo é o recurso mais precioso a ser gerenciado nos sistemas de tempo real. As tarefas devem ser alocadas e escalonadas para estarem completas antes de seus prazos máximos permitidos, chamados *deadlines*. Em alguns sistemas, mensagens são enviadas entre tarefas caracterizando sua interatividade, e tais mensagens devem cumprir seus *deadlines* de envio/recebimento.

Conforme Liu (2000), os sistemas de tempo real são classificados conforme a criticalidade dos seus requisitos temporais. Nos sistemas tempo real críticos com garantia (*hard real-time*) o não atendimento de um requisito temporal pode resultar em consequências catastróficas tanto no sentido econômico quanto em vidas humanas. Quando os requisitos temporais não são críticos (*soft real-time*) eles apenas descrevem o comportamento desejado. O não atendimento de tais requisitos reduz a utilidade da aplicação mas não a elimina completamente nem resulta em consequências catastróficas.

Os sistemas de tempo real são formados por tarefas. Conforme Farines et al. (2000), tarefas ou processos são computações executadas em um processador de forma sequencial, ou seja, são abstrações de trabalhos que disputam por recursos. No caso de tarefas de tempo real, seu trabalho está sujeito a um prazo máximo de conclusão (*deadline*).

As tarefas são classificadas quanto à frequência de ativação em tarefas periódicas, esporádicas e aperiódicas (BUTTAZZO, 2005). Nas tarefas periódicas o período de ativação é completamente conhecido e a cada período há geração de uma instância da tarefa (*job*). As tarefas esporádicas são definidas como tendo um período mínimo de chegadas mas não necessariamente há geração de instâncias a cada período. As tarefas aperiódicas não possuem tempo de ativação definido sendo este aleatório. Além do período comumente define-se um tempo de computação (C_i) que é o tempo necessário para a tarefa realizar o trabalho específico. As características das tarefas e suas especificações definem um "modelo de tarefas" que varia conforme a aplicação de tempo real.

Na relação do período com o *deadline*, este pode ser implícito, limitado ou arbitrário. Quando o *deadline* é implícito, seu valor numérico é igual ao período da tarefa ($D_i = T_i$). No caso de sistemas com *deadline* limitado, seu valor nunca é maior que o período ($D_i \leq T_i$). Quando o *deadline* é arbitrário, ele pode ser maior, menor ou igual ao período das tarefas.

O conjunto de tarefas restrito ao modelo, requisitos temporais e os recursos (número processador, etc) formam um problema de escalonamento que na sua forma geral é identificado como um problema intratável (NP-completo) (FARINES et al., 2000). O escalonamento define a ordenação de execução das tarefas em uma escala respeitando todas as restrições impostas. As políticas de escalonamento definem as regras de ordenação de execução das tarefas. Uma escalona é dito ótimo quando, dados os critérios estabelecidos pelo modelo de tarefas e política de escalonamento, a escala gerada é a melhor possível no atendimento das restrições temporais (FARINES et al., 2000).

No caso de sistemas de tempo real críticos a viabilidade da escala geralmente é provada por testes de escalonabilidade. Os testes são específicos

e restritos para cada modelo de tarefas e política de escalonamento. Conforme Farines et al. (2000) os testes de escalonabilidade são classificados em testes exatos, suficientes e necessários, onde:

- exatos: identificam os conjuntos de tarefas escalonáveis e não escalonáveis. Podem ser complexos a ponto de serem inviáveis praticamente.
- suficientes: classificam corretamente os conjuntos de tarefas escalonáveis mas os conjuntos classificados negativamente podem ser escalonáveis.
- necessários: os conjuntos negativamente classificados são realmente não escalonáveis, mas se um conjunto passa no teste, não necessariamente é escalonável.

Os testes de escalonabilidade geralmente fazem uma análise utilizando os parâmetros C_i, T_i e D_i de cada tarefa e determinam se todas as tarefas do sistema podem ou não ser escalonadas cumprindo todos os prazos individualmente.

Um conjunto de tarefas é dito "viável" com respeito a um dado sistema se existir algum algoritmo de escalonamento que pode escalonar todas as possíveis sequências de instâncias geradas pelo conjunto de tarefas daquele sistema e todos os *deadlines* são cumpridos (DAVIS; BURNS, 2009b). Um algoritmo de escalonamento é dito "ótimo" com respeito a um sistema e modelo de tarefas se ele pode escalonar todos os conjuntos de tarefas que estão de acordo com o modelo de tarefas e que são viáveis no sistema. Conforme Davis e Burns (2009b), um tarefa é escalonável de acordo com dado algoritmo de escalonamento se seu pior tempo de resposta é menor ou igual ao seu *deadline*. Desta forma, um conjunto de tarefas é escalonável por dado algoritmo de escalonamento se todas as suas tarefas são escalonáveis.

A pesquisa na área de escalonamento de sistemas de tempo real teve origem no final da década de 1960 e início de 1970 com grandes avanços em 1980 e 1990. Atualmente, apesar de haver um grande espaço para pesquisas, a teoria de escalonamento de tempo real para sistemas com um único processador é considerada razoavelmente madura (DAVIS; BURNS, 2009b).

A teoria de escalonamento de tempo real para multiprocessadores também teve suas origens no final da década de 1960 e início de 1970. Em 1969, conforme (DAVIS; BURNS, 2009b), notou-se o escalonamento de tempo real para multiprocessadores é um problema intrinsecamente mais difícil que o caso para monoprocessador. Poucos dos resultados obtidos para monoprocessador são generalizados diretamente para o caso de multiprocessadores.

No final da década de 1990 a tendência do aumento de capacidade de processamento através de frequências elevadas foi abandonada e ficou claro

para a comunidade de tempo real a necessidade de desenvolver teorias de escalonamento para sistemas multiprocessados. Isto significou grande esforço e pesquisa focado no problema do escalonamento de tempo real para multiprocessadores. Muitos artigos foram publicados nesta área desde 2000 e grande progresso foi feito, porém há inúmeras questões em aberto para pesquisas.

Além da motivação incorporada pelas grandes questões em aberto no escalonamento de tempo real para multiprocessadores, tem-se também um problema relacionado com o parâmetro do pior tempo de execução (WCET) das tarefas. Qualquer análise de escalonabilidade de tempo real requer este parâmetro sendo um elemento essencial para análise e garantia, especialmente em sistemas de tempo real críticos (*safety-critical hard real-time systems*). O WCET é calculado através de ferramentas específicas de fluxo de dados e análise do *hardware* do processador derivando um limite superior para o tempo de execução das tarefas de um sistema. Todos os elementos da arquitetura do processador (*pipeline*, *cache*, etc) devem ser analisados para computar um limite superior seguro do WCET.

Considerando apenas um núcleo individual de um multiprocessador, adota-se soluções estatísticas em *hardware* para melhorar a relação de custo/-desempenho. Estas soluções fazem com que os processadores acabem atingindo um desempenho muito elevado mas incorporam elementos não previsíveis onde não se consegue estabelecer um limite superior das latências das instruções executadas pelo processador. Além disso, dependendo da complexidade da microarquitetura, a explosão de estados que devem ser analisados na computação do WCET pode-se tornar intratável computacionalmente. Sem um limite superior de tempo de execução ou com um conjunto de estados intratável, é impossível computar o WCET preciso e seguro. O pior problema de todos aqueles relacionados com o tempo de computação está ligado ao não suporte de interrupções, exceções e preempções nas ferramentas que calculam WCET. A estimação do WCET é válida somente se o código for executado fim a fim sem ser interrompido por nenhum mecanismo de *software* ou *hardware*. O problema de não suportar preempções e interrupções na análise do WCET em conjunto com a premissa de preempção nos algoritmos de escalonamento gera uma contradição fundamental na análise de escalonabilidade de sistemas de tempo real, ou seja, o WCET calculado não é diretamente aplicável na análise de escalonabilidade se o mesmo algoritmo de escalonamento adotar preempção como premissa. Assim, a grande motivação deste trabalho é como conciliar as limitações do WCET com análises de escalonabilidade e sistemas de tempo real com garantia multiprocessados.

1.2 OBJETIVO DO TRABALHO

O objetivo deste trabalho é elaborar uma solução de escalonamento de tempo real com garantia para multiprocessadores que considere as limitações das ferramentas de cálculo do WCET – *Worst Case Execution Time*.

Sabe-se que para conservar as propriedades do WCET, as tarefas devem executar sem serem interrompidas. Contudo, como será apresentado, grande parte das teorias de escalonamento tomam como premissa que as tarefas possam ser interrompidas (preemptadas) em qualquer ponto dentro da sua janela de execução por outra com requisito maior. A preempção melhora a escalonabilidade, mas não é diretamente compatível com as premissas do cálculo do WCET. Desta forma, serão investigados os algoritmos de escalonamento de tempo real para multiprocessador existentes avaliando suas características a fim de selecionar os mais compatíveis com a contradição existente entre preempção e o cálculo do WCET.

Para cumprir este objetivo, também será necessário desenvolver um estudo sobre arquitetura dos processadores contemporâneos dando ênfase às soluções estatísticas presentes no *hardware* dos processadores modernos. Esta fundamentação é importante devido aos conceitos de *pipeline*, *cache* e memória necessários quando se realiza estimativas de WCET de tarefas. As limitações e características de uma ferramenta WCET serão avaliadas experimentalmente.

1.3 ORGANIZAÇÃO DO TEXTO

Este trabalho está organizado em sete capítulos começando com o atual, que contempla conceitos básicos, motivação e os objetivos do trabalho.

No capítulo 2 serão apresentadas algumas características de arquitetura de processadores contemporâneos fornecendo os conceitos básicos para entendimento das limitações das ferramentas de cálculo de WCET.

Uma avaliação empírica de uma ferramenta de cálculo de WCET será apresentada no capítulo 3. Como será visto, a estimativa dos WCET das tarefas é um problema difícil e uma avaliação empírica de uma ferramenta existente esclarece o motivo destas dificuldades.

No capítulo 4 será apresentada a revisão da literatura contemplando resultados fundamentais sobre escalonamento de tempo real para multiprocessador, análises de escalonabilidade que consideram os efeitos dos elementos estatísticas dos processadores (*cache*, por exemplo) e um exemplo de como seria uma arquitetura de processador voltada ao determinismo.

Após todas estas considerações, no capítulo 5 será apresentada a proposta de uma abordagem de escalonamento heterogêneo contemplando a formalização matemática do problema e do sistema, algoritmos para solucionar o problema e um exemplo de implementação. A avaliação de desempenho e comparação com algoritmos citados neste trabalho será apresentada no capítulo 6.

Por fim, as conclusões e sugestões para trabalhos futuros serão apresentadas no capítulo 7.

2 ARQUITETURA DE PROCESSADORES CONTEMPORÂNEOS

Textos sobre arquitetura de computadores definem três níveis de abstração (SEHN; LIPASTI, 2005): arquitetura, implementação e realização.

Arquitetura é conhecida como arquitetura de conjunto de instruções – *Instruction Set Architecture* (ISA) – que especifica o conjunto de instruções que caracteriza o comportamento fundamental do processador. Todo o *software* deve ser codificado em um ISA correspondente a tal máquina. Alguns exemplos de ISA são: Intel IA32, AMD x86_64, PowerPC e ARM.

Uma implementação especifica o projeto da arquitetura, referenciado como microarquitetura. Exemplos de implementação são PowerC 604 e Intel P6. Atributos como projeto do *pipeline*, memórias *cache* e previsão de fluxo (*branch prediction*) fazem parte da implementação.

A realização de uma implementação consiste no encapsulamento físico do projeto. No caso dos microprocessadores, esse encapsulamento geralmente é um *chip*. As realizações podem variar em termos de frequência, capacidade de memória *cache*, barramentos, tecnologia de fabricação e empacotamento. Atributos relacionados à realização incorporam o *die size*, encapsulamento, potência, refrigeração e cofiabilidade.

O ISA é definido como sendo um contrato entre o *software* e o *hardware* que torna o desenvolvimento de programas e máquinas uma tarefa independente. Desta forma, programas escritos em dado ISA podem rodar em diversas implementações com diferentes níveis de desempenho, aumentando a longevidade do *software*. Contudo ISAs com uma grande quantidade de *softwares* já desenvolvidos tendem a ficar no mercado por um bom tempo e um exemplo disso é o ISA Intel IA32.

Servindo como base no desenvolvimento de *softwares* ou compiladores, o ISA também serve como especificação de processadores. No lançamento de uma nova microarquitetura, esta deve ser validada para que se garanta todos os requisitos funcionais, bem como que *softwares* já existentes possam ser executados. Um ISA define um conjunto de instruções chamadas instruções *assembly*, sendo que cada instrução define um operador e um ou mais operandos. ISAs têm sido diferenciados de acordo com o número de operandos que possam ser explicitamente especificados em cada instrução. Alguns ISAs usam um acumulador como operando implícito, onde este é usado sempre como fonte ou destino. Outros ISAs assumem que operandos estão contidos em uma pilha formando um estrutura LIFO – *last in, first out*, sendo que estruturas e operandos são operados no topo da pilha. Esta estrutura de pilha é utilizada, por exemplo, em unidades de ponto flutuante – *Floating Point Units* (FPU). Contudo, ISAs modernos consideram que os

operandos estão gravados em registradores especiais formando o *multientry register file* do processador. Todas as operações aritméticas e lógicas são realizadas sobre estes registradores. Instruções especiais relacionadas com a memória como *load* e *store* são responsáveis em mover os operandos dos registradores à memória principal e vice-versa.

Este capítulo é uma versão simplificada do estudo realizado sobre arquitetura de processadores. Maiores detalhes podem ser encontrados no apêndice A que contempla a versão expandida deste capítulo.

2.1 PRINCÍPIOS DE *PIPELINE*

A utilização de *pipeline* é uma técnica poderosa para aumentar o desempenho do sistema sem uma replicação massiva de *hardware*. O Intel i486 foi a primeira implementação da arquitetura IA32 com *pipeline*.

A técnica do *pipeline* particiona o sistema entre múltiplos estágios adicionando uma memória ou *buffer* entre eles. A computação original é decomposta em k estágios, sendo que uma nova tarefa pode ser iniciada assim que a anterior atravessar o primeiro estágio. Assim, ao invés de iniciar uma tarefa a cada D unidades de tempo, inicializa-se uma a cada D/k, e o processamento das k computações são sobrepostas pelo *pipeline*.

A princípio, o ganho de desempenho é proporcional ao comprimento do *pipeline*, ou seja, quanto mais estágios, maior desempenho. Porém, há limitações físicas relacionadas à frequência que determinam a quantidade de estágios que podem ser utilizados.

O aumento de desempenho em k consiste em três idealismos básicos (SEHN; LIPASTI, 2005):

- sub-computações uniformes: a computação a ser realizada é divida em sub-computações com latências iguais;
- computações idênticas: a mesma computação é realizada repetidamente em uma grande quantidade de dados;
- computações independentes: as computações são independentes entre si.

As sub-computações uniformes mantêm o idealismo que cada estágio possui o mesmo tempo de execução, ou seja, T/k. Contudo esta suposição não é verdadeira sendo impossível particionar computações em estágios perfeitos e balanceados. Exemplificando, uma operação de $400ns$ foi dividia em três estágios de $125ns$, $150ns$ e $125ns$. Como a frequência do *pipeline* é determinada pelo estágio mais longo, esta será limitada pelos $150ns$. Assim, o

primeiro e o terceiro estágio possuem uma ineficiência de 25*ns*, e o tempo total da operação utilizando o *pipeline* será de 450*ns* ao invés de 400*ns* originalmente. Essa ineficiência é denominada fragmentação interna, sendo o primeiro desafio no desenvolvimento de processadores.

As computações idênticas assumem que todos os estágios do *pipeline* são sempre utilizados em todos os conjuntos de dados. Esta suposição não é válida quando há execução de múltiplas funções, onde nem todos os estágios são necessários para suportá-las. Como nem todos os conjuntos de dados precisarão de todos os estágios, estes ficaram esperando mesmo que não precisem executar tal subfunção devido a característica síncrona do sistema. Esta ineficiência introduz a fragmentação externa dos estágios do *pipeline*.

As computações independentes assumem que não há dependência de dados ou controle em qualquer par de computações. Esta suposição permite que o *pipeline* opere em fluxo contínuo, ou seja, nenhuma operação precisa esperar a finalização da operação anterior. Em alguns sistemas esta suposição é válida, mas para *pipelines* de instruções uma operação precisará de resultado da anterior para prosseguir. Se esta situação ocorre e as duas operações encontram-se em processamento, a operação deverá esperar o resultado da anterior, ocasionando a flutuação do *pipeline – pipeline stall*. Se há uma operação em flutuação, todos os estágios anteriores também deverão esperar, ocasionando uma série de estágios ociosos.

O desenvolvimento de processadores com *pipeline* lidam com três desafios relacionados com os três idealismos básico (citados anteriormente):

- sub-computações uniformes: requer balanceamento dos estágios;
- computações idênticas: requer unificação dos tipos de instrução;
- computações independentes: minimização das flutuações.

Um ciclo de instrução básico pode ser divido em cinco subcomputações genéricas:

1. busca de instruções (BI) – *instruction fetch*;
2. decodificação da instrução (DI) – *instruction decode*;
3. busca dos operandos (BO) – *operand's fetch*;
4. execução da instrução (EX) – *instruction execution*;
5. gravação dos operandos (GO) – *operand store*.

As subcomputações genéricas correspondem a uma partição natural de um ciclo de instrução formando um *pipeline* genérico de cinco estágios. Múltiplas subcomputações com baixa latência podem formar um único estágio ou ainda alguns dos cinco estágios genéricos também podem ser agrupados formando um melhor balanceamento do *pipeline* como um todo, conforme a figura 1.

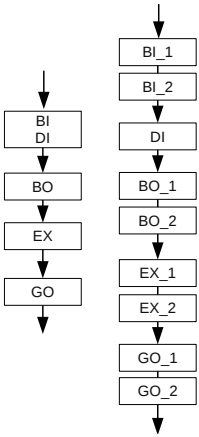


Figura 1: Técnicas de balanceamento de *pipeline*.

Supõe-se que a computação de 5 estágios genéricos leve $300ns$ e que o tempo de ciclo dos *pipelines* de 4 e 9 estágios da figura 1 é $80ns$ e $40ns$, respectivamente. Consequentemente, a latência do *pipeline* de 4-estágios é de $(80ns \times 4) = 320ns$ e o de 9-estágios é $(40ns \times 9) = 360ns$. A diferença entre a nova latência e a antiga representa a fragmentação interna, ou seja, $(320ns - 300ns) = 20ns$ para o de 4-estágios e $(360ns - 300ns) = 60ns$ para o de 9-estágios. Baseando-se neste ponto de vista, conclui-se que o *pipeline* de 4-estágios é mais eficiente que o de 9-estágios, porém em termos de *throughtput*, tem-se $300ns/40ns = 7.5$ contra $300ns/80ns = 3.75$, assim se deve considerar ambos os fatores no projeto de *pipelines*.

Para realizar uma computação, precisa-se de três tarefas genéricas:

1. operações aritméticas;
2. movimentação de dados;
3. sequenciamento de instruções.

Em uma arquitetura RISC moderna, o conjunto de instruções emprega

tipos de instruções dedicadas para cada uma das tarefas genéricas, formando assim uma classificação de três tipos:

1. Instruções da Unidade Lógica e Aritmética (ULA);
2. Instruções de memória (*load/store*): movimentação de dados;
3. Instruções de fluxo (*branch*): para o manipulação do fluxo de controle.

As instruções ULA são restritas somente aos registradores. Apenas as instruções de movimentação de dados podem acessar a memória. As instruções de fluxo e movimentação empregam endereçamento simples de memória, tipicamente apenas o modo de endereçamento indireto com deslocamento é suportado. Geralmente o endereçamento relativo ao contador de programa (CP) também é suportado para instruções de controle de fluxo. Nas tabelas 1, 2 e 3, conforme Sehn e Lipasti (2005), encontra-se o detalhamento das instruções relacionadas com as operações e as cinco subcomputações típicas de um *pipeline*, considerando uma arquitetura de *cache* de dados e instrução independentes (D-*cache*/I-*cache*).

Subcomputação	Instrução Inteira	Instrução ponto-flutuante
BI	Buscar (acessar I-*cache*)	Buscar (acessar I-*cache*)
DI	Decodificar	Decodificar
BO	Acessar registradores	Acessar registradores PF
EX	Realizar operação	Realizar operação PF
GO	Regravar nos registradores	Regravar nos registradores

Tabela 1: Especificação das instruções aritméticas/ponto flutuante.

A análise das especificações das instruções proporciona uma verificação da semântica de cada instrução. Apesar delas compartilharem subcomputações em termos de busca e decodificação, elas necessitarão de recursos diferenciados. Assim, na figura 2 demonstra-se um exemplo de unificação de recursos para um *pipeline* genérico de seis estágios.

Na coluna da esquerda da figura 2 apresenta-se as cinco tarefas básicas de busca (BI) e decodificação de instrução (DI), busca dos operandos (BO), execução (EX) e gravação de operandos (GO). A direita apresenta-se um possível *pipeline* compartilhando as unidades e na parte superior, apresenta-se as 4 instruções.

Subcomputação	Instrução *Load*	Instrução *store*
BI	Buscar (acessar I-*cache*)	Buscar (acessar I-*cache*)
DI	Decodificar	Decodificar
BO	Acessar registradores (enderço base) Gerar endereço efetivo (base + deslocamento) Acessar memória (acessar D-*cache*)	Acessar Registradores (operando e endereço base)
EX		
GO	Regravar nos registradores	Gerar endereço efetivo (base + deslocamento) Acessar memória (acessar D-*cache*)

Tabela 2: Especificação das instruções de acesso a memória.

Subcomputação	Salto incondicional	Salto condicional
BI	Buscar (acessar I-*cache*)	Buscar (acessar I-*cache*)
DI	Decodificar	Decodificar
BO	Acessar registradores (endereço base) Gerar endereço efetivo (base + deslocamento)	Acessar Registradores (endereço base) Gerar endereço efetivo (base + deslocamento)
EX		Avaliar condição
GO	Atualizar CP com endereço alvo	Se condição for verdadeira, atualizar CP com endereço alvo

Tabela 3: Especificação das instruções controle de fluxo.

Os seis diferentes estágios do *pipeline* da figura 2 implementam seis instruções. Contudo, se ocorrer qualquer falta de *cache* (*cache miss*), haverá flutuação no *pipeline*. Neste modelo, o acesso à *cache* possui uma latência de um ciclo, mas conforme a *cache* torna-se maior e a lógica do processador

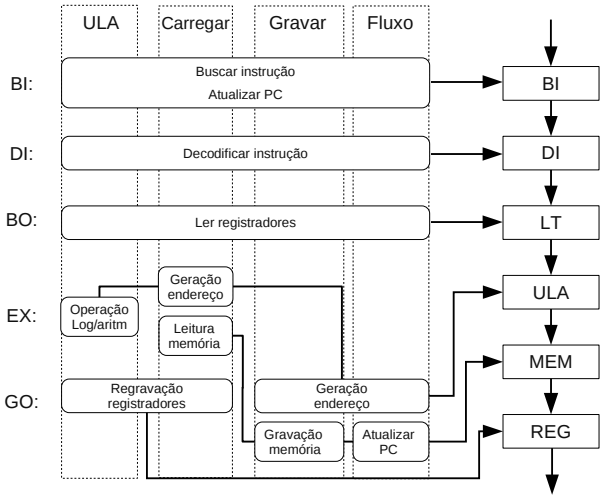


Figura 2: Exemplo de unificação de recursos de um *pipeline*.

aumenta de complexidade, manter uma latência de um ciclo torna-se muito difícil.

Um programa é uma sequência de instruções em *assembly*. Estas instruções podem não ser independentes: o resultado do cálculo de uma instrução *i* precedida de *j* pode ser necessário pela própria instrução *j*, formando uma dependência de dados. Se as instruções *i* e *j* forem próximas ao ponto de executarem simultaneamente no *pipeline*, o *hardware* deve estar preparado para que os dados necessários para operação já estejam calculados, ou seja, deve-se resolver o problema de dependência das instruções por *hardware*. Uma potencial violação das dependências é conhecida como *pipeline hazzards*

A resolução dos problemas de dependência por mecanismos de *hardware* é conhecido como *pipeline interlock*. Este mecanismo deve detectar todos os possíveis *pipeline hazzards* e garantir que todas as dependências sejam satisfeitas. Este mecanismo envolve a criação de flutuações no *pipeline* em certos estágios bem como criação de atalhos para que os dados cheguem ao estágio em espera antes do término do cálculo corrente.

Conforme visto até então, um *pipeline* de *k*-estágios pode potencialmente alcançar um desempenho de um fator *k* em relação ao projeto normal. Contudo, com a construção de *pipelines* longos as penalidades relacionadas com dependências e suas resoluções tornam-se maiores. As penalidades envolvendo instruções de busca de operandos, memória e operações aritméticas

geralmente são resolvidas por caminhos de atalho e incorporam poucos ciclos, mas penalidades relacionadas as instruções de controle de fluxo podem incorporar muitos ciclos de ociosidade. Portando, no projeto de microprocessadores, sempre busca-se um equilíbrio entre a quantidade de estágios, frequência de operação, desempenho e custo objetivando o melhor *throughput* possível nestas condições.

2.2 PROCESSADORES SUPERESCALARES

As máquinas superescalares são capazes de avançar múltiplas instruções pelos estágios do *pipeline* incorporando múltiplas unidades funcionais. Tais sistemas aumentam a capacidade de processamento concorrente à nível de instrução aumentando o *throughput*. Outro atributo fundamental é a habilidade de executar instruções em ordem diferente do programa original. A ordem sequencial das instruções no programa implica em algumas precedências desnecessárias entre as instruções. A capacidade de executar as instruções fora de ordem alivia a imposição sequencial e permite mais processamento paralelo sem modificação do programa original.

Um *pipeline* de k-estágios teoricamente pode aumentar o desempenho em um fator k. De modo alternativo, pode-se atingir o mesmo fator de velocidade empregando k cópias do mesmo *pipeline* processando as k instruções em paralelo. Estas duas maneiras de paralelismo são conhecidas como paralelismo temporal e paralelismo espacial respectivamente (SEHN; LIPASTI, 2005). *Pipelines* paralelos empregados nos processadores superescalares utilizam as duas técnicas de paralelismo.

O primeiro passo para a construção de um *pipeline* superescalar é implementar unidades diversificadas na unidade de execução, formando um *pipeline* diversificado. Tal *pipeline* pode ser visualizado na figura 3. Neste exemplo quatro unidades funcionais são implementadas e o estágio BO envia as instruções para as unidades corretas baseando-se nos tipos de instrução.

Neste projeto, cada unidade funcional pode ser customizada para uma instrução particular resultando em um *hardware* eficiente. Cada instrução possui sua própria latência e utiliza todos os estágios de sua unidade funcional. Se as dependências intra-instruções são resolvidas antes do envio às unidades funcionais, não há flutuações no *pipeline*. Este esquema permite controle independente e distribuído de cada *subpipeline* de execução.

Para evitar ciclos de flutuação desnecessários no *pipeline* diversificado, permite-se que novas instruções sejam adiantadas quando há ciclos de flutuação, por dependências ou faltas na memória *cache*. Este adiantamento pode mudar a ordem de execução das instruções da ordem sequencial do có-

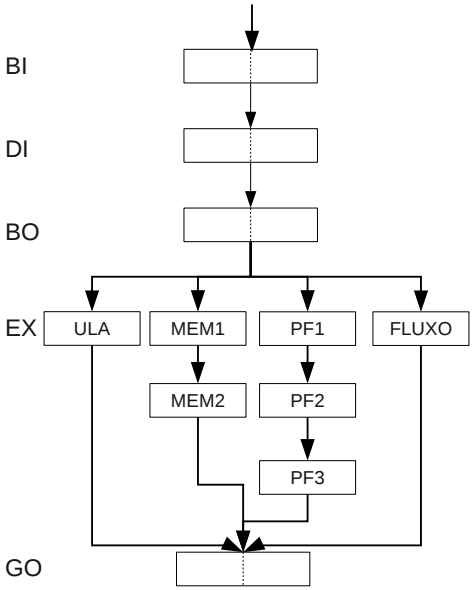


Figura 3: Exemplo de *pipeline* diversificado.

digo do programa. A execução fora de ordem permite que estas sejam executadas assim que os operandos estejam disponíveis. Um *pipeline* paralelo que permite execução fora de ordem é chamado de *pipeline* dinâmico, onde a execução dinâmica é possível utilizando-se *buffers* multientrada complexos. Tal sistema permite que as instruções entrem e saiam em ordens diferentes. Um exemplo de *pipeline* dinâmico pode ser visualizado na figura 4.

Com um *pipeline* largo, o estágio de busca acessa *s* instruções da I-*cache* em cada ciclo de máquina. Isto implica que a organização da I-*cache* deve ser larga o suficiente para que cada linha armazene *s* instruções e que toda a linha possa ser acessada em cada ciclo de instrução. Contudo, não é possível que se garanta que a banda máxima de busca seja atendida devido ao desalinhamento das instruções na I-*cache* e também a presença de instruções de controle de fluxo.

O estágio de decodificação em um ISA RISC deve identificar dependências das instruções e verificar quais as instruções que podem ser despachadas em paralelo. Além disso, deve identificar mudanças do controle de fluxo das instruções decodificadas para promover uma rápida resposta do estágio de busca. Para um ISA CISC a tarefa de decodificação pode ser ainda mais complexa precisando de múltiplos sub-estágios. Tanto o Pentium quanto o

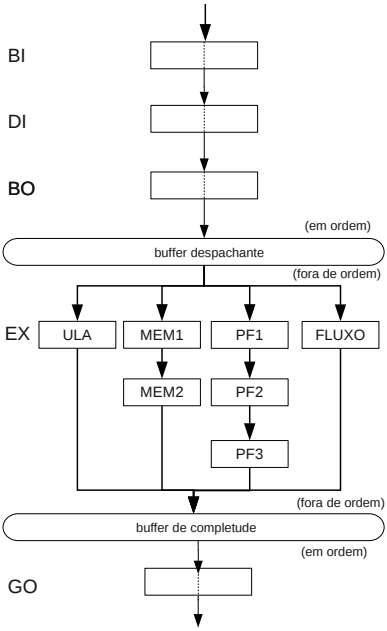


Figura 4: Exemplo de *pipeline* dinâmico.

AMD K5 empregam dois sub-estágios para decodificação da instrução IA32 e o Intel Pentium Pro precisa de um total de cinco ciclos de máquina para acessar a I-*cache* e decodificar uma instrução. A utilização de instruções de tamanho variável (ISA CISC) impõe sequenciamento na decodificação, pois uma instrução deve ser decodificada e seu tamanho conhecido para que a próxima instrução possa ser identificada.

2.3 TÉCNICAS DE PREVISÃO DE FLUXO

O *pipeline* dos processadores é construído para evitar ao máximo ciclos de ociosidade relativos a dependências de dados entre as instruções. Contudo, as dependências de controle ainda induzem uma quantidade significativa de flutuações e por isso são adotadas técnicas de previsão do controle de fluxo (*branch prediction*).

Estudos apontam que o comportamento das instrução de controle de fluxo é altamente previsível (SEHN; LIPASTI, 2005). A técnica chave para minimizar as penalidades e maximizar o desempenho é especular tanto o en-

dereço quanto a condição do salto nas instruções de controle. Obviamente que a utilização de técnicas especulativas requerem mecanismos de recuperação de erros de previsão.

A especulação do endereço envolve o uso de um *buffer – Branch Target Buffer* (BTB) – para armazenar o endereço alvo da instrução de controle anterior. O BTB é uma memória *cache* pequena acessada durante o estágio de busca de instruções. Cada entrada no BTB contém dois campos: endereço da instrução – *Branch Instruction Address* (BIA) – e o endereço alvo – *Branch Target Address* (BTA).

Se acerta-se a previsão do salto não há ciclo de penalidade. Caso contrário um erro de previsão ocorre e deve-se iniciar a recuperação. O resultado da execução não especulativa é utilizada para atualizar os conteúdos do BTB.

Pode-se notar que um erro de previsão é custoso, além dos ciclos perdidos no processamento de instruções não utilizadas, deve-se realizar toda a limpeza das instruções falsamente previstas. Porém testes de desempenho apontam que cerca de 82,5% a 96,2% das predições são corretas (SEHN; LIPASTI, 2005).

2.4 ARQUITETURA DE MEMÓRIA

Os processadores modernos utilizam uma configuração hierárquica de memória conforme a figura 5. Pode-se perceber que as memórias com maior banda e custo possuem baixa latência e consequentemente baixa capacidade. Por esta razão que os processadores geralmente utilizam memória *cache* para reduzir a latência no acesso aos dados. A memória *cache* é mais lenta que os registradores do processador, mas devido a usabilidade dos dados, as referências às memórias são geralmente atendidas pela *cache*.

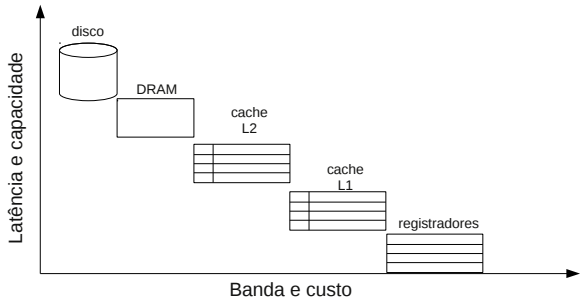


Figura 5: Relação e hierarquia de memória.

2.4.1 Memória *cache*

A memória *cache* foi criada para explorar as propriedades de localidade espacial e temporal, criando uma ilusão de memória rápida de grande capacidade. O princípio básico é colocar uma memória da baixa capacidade, rápida e de alto custo entre o processador e a memória principal, sendo esta mais barata e com grande capacidade. Mesmo a memória *cache* de nível primário (L1) consegue satisfazer 90% das referências na maioria dos casos (SEHN; LIPASTI, 2005). Referências na *cache* são considerados acertos (*hit*) e referências não satisfeitas são considerados faltas (*misses*).

Cada nível de *cache* deve ser projetado para suportar os requerimentos de banda e latência. Como os níveis mais baixos devem operar em velocidades comparáveis ao núcleo do processador, estes devem ser implementados usando *hardware* rápido e com complexidade limitada.

Ainda, cada nível deve implementar um mecanismo que permita uma procura, de baixa latência, para verificar se dado bloco está ou não contido na *cache*. O tamanho do bloco ou tamanho da linha descreve a granularidade na qual a *cache* opera. Cada bloco é uma sequência contínua de *bytes* e inicia em alinhamento comum. O menor tamanho utilizável de um bloco possui o tamanho natural de palavras do processador, ou seja, 4-*bytes* para máquinas de 32-*bits* ou 8-*bytes* para máquinas de 64-*bits*.

Em termos de localização de blocos há três possíveis organizações: mapeamento direto, totalmente associativo e associativo por conjunto.

No mapeamento direto, um endereço particular pode residir apenas em uma única localização na *cache*. Esta localização é normalmente determinada extraindo os *n bits* do endereço e os utilizando para índice direto das 2^n possíveis localizações na *cache*. Como há um mapeamento de muitos endereços para somente um lugar na *cache*, cada localização necessita de uma etiqueta (*tag*) que corresponde aos *bits* restantes do bloco gravados naquela localização. Em cada procura, o *hardware* precisa ler a etiqueta e comparar com o endereço da referência e determinar se há um acerto ou falta.

A organização totalmente associativa permite um mapeamento de múltiplos endereços a múltiplas localizações na *cache*, ou seja, qualquer endereço de memória pode residir em qualquer localização na *cache*. Como todas as localizações da *cache* devem ser verificadas para encontrar os dados, não há *bits* de índice para determinação da localização. Novamente cada localização deve possuir uma etiqueta do endereço dos dados que está armazenando, para o *hardware* comparar e detectar uma falta ou acerto.

Na associatividade por conjunto há um mapeamento de múltiplos endereços para algumas localizações na *cache*. Em cada procura, um subconjunto de *bits* de endereço são utilizados para geração do índice igualmente ao

caso de mapeamento direto. Contudo, o índice corresponde a um conjunto de entradas que são procuradas em paralelo em busca da etiqueta correta, ou seja, a associatividade por conjunto é um compromisso entre o mapeamento direto e a associatividade total.

Como cada nível de *cache* possui uma capacidade finita, deve haver uma política para despejar ocupantes atuais que possibilita espaço para blocos com referências mais recentes. Uma política de substituição determina que algoritmo utilizar para identificar um candidato para ser despejado. Em um mapeamento direto o problema é trivial já que há somente um lugar para cada endereço.

Na associatividade completa ou por conjunto, há escolhas a serem feitas: um novo bloco pode ser colocado em várias localizações, e todas as entradas são candidatas para substituição. Há basicamente três políticas implementadas no projeto de *cache*: FIFO (primeiro a entrar, primeiro a sair), menos recentemente utilizado – *Least Recently Used* (LRU) – e aleatoriamente.

A utilização de memória *cache* implica em múltiplas cópias do bloco. Enquanto há apenas leitura não ocorre nenhum problema, contudo deve haver mecanismos para atualização dos blocos nos outros níveis da memória. Existem basicamente dois mecanismos conhecidos como *write-through* e *write-back*.

O mecanismo *write-through* simplesmente propaga a atualização para os níveis inferiores. É de fácil implementação e não há ambiguidade sobre a versão dos dados. Contudo exige muita demanda de barramento e a disparidade da frequência entre o processador e a memória principal torna impossível a utilização em todos os níveis da hierarquia. Este mecanismo ainda deve especificar se vai ou não alocar espaço na *cache* quando ocorre uma falta em escrita de um bloco. A política *write-allocate* implica em alocar o bloco na *cache*, enquanto a *write-no-allocate* evitará a instalação do bloco na *cache*, realizando esta operação somente em faltas de leitura.

O mecanismo *write-back* prorroga a atualização das outras cópias até o ponto que se precisa manter a ordem de correção. Neste tipo de política, uma ordem de prioridade implícita é utilizada para encontrar a cópia mais atualizada do bloco e somente esta cópia é atualizada. Se um bloco é encontrado no nível mais elevado da *cache*, esta cópia é atualizada, enquanto as cópias dos níveis mais baixos permanecem desatualizados. Se uma cópia é apenas encontrada um nível mais baixo, esta é promovida ao topo e é ali atualizada. As cópias atualizadas são marcados com um *bit* (*dirty bit*) sinalizando que os outros níveis possuem cópias desatualizadas. Quando um bloco sinalizado é substituído, este é atualizado para o nível inferior, certificando que a cópia mais recente permaneça. A cópia deste nível é agora marcada

com *bit* sinalizador para que quando for despejado, seja atualizado nos níveis inferiores.

2.4.2 Proteção de memória

Em alguns cenários é desejável que múltiplos processos acessem a mesma página física de memória permitindo comunicação ou evitando duplicação de cópias de bibliotecas compartilhadas ou binários na memória. Contudo, o sistema operacional também é residente na memória e deve proteger suas estruturas de dados internas dos programas normais. Desta forma, o sistema de memória virtual deve promover mecanismos de proteção de páginas compartilhadas de programas maliciosos ou defeituosos além de garantir que acessos errôneos não corrompam o estado do sistema.

O mapeamento da memória virtual à física deve ser gravado em uma memória de tradução. O sistema operacional é responsável por atualizar esse mapeamento quando necessário, enquanto o processador deve acessar a memória de tradução para determinar o endereço físico de cada acesso virtual que ocorre. Cada entrada de tradução deve conter o endereço virtual, o endereço real correspondente, *bits* de permissão para leitura, escrita e execução, além de *bits* de referência e modificação.

As memórias de tradução são comumente chamadas de tabelas de páginas e podem ser organizadas em tabelas de páginas diretas (*forward page tables*) ou tabelas de páginas invertidas (*inverted page tables/hashed page tables*). A primeira organização, mais simples, possui uma entrada para cada bloco do espaço de endereço virtual do processo usando a tabela de páginas, resultando em uma enorme estrutura com muitas entradas não utilizadas.

As tabelas de páginas são mantidas em memória e gerenciadas pelo sistemas operacional mas processadores modernos utilizam uma estrutura chamada de *Translation Lookaside Buffer* (TLB) que pode ser definida como uma parte do modo de proteção do processador. O TLB contém um número pequeno de entradas de páginas (tipicamente 64) organizadas por associatividade completa. O processador provê um rápido *hardware* de pesquisa associativa para converter referências de cada instrução de busca, carregamento ou gravação. Faltas no TLB resultam em faltas de página que invocam o sistema operacional. Este usa sua própria tabela de página, podendo ser outra estrutura de dados para encontrar a tradução correta e atualizar o TLB, utilizando instruções privilegiadas.

Contudo, processadores que implementam uma arquitetura que especifica a organização da tabela de páginas incluem uma máquina de estados para acesso à memória por *hardware* para procura da tabela de páginas. Tais

processadores promovem tradução de endereços para todas as referências de memória, onde a estrutura da tabela de página é fixa, acessada tanto pelo sistema operacional como pelos mecanismos de *hardware*. Este sistema é utilizado nos ISA PowerPC e Intel IA-32, conhecido como *hardware TLB miss handler*.

2.4.3 Interação *cache* e TLB

Um dos problemas dos sistema de tradução de endereços de memória é o fato de dois acessos à memória serem necessários para cada referência à memória principal por uma instrução. Primeiramente a tabela de páginas deve ser acessada para obtenção do número da página física, e então chega-se à memória física usando o endereço de tradução. Dependendo da implementação, a tabela de páginas é tipicamente armazenada na memória principal (em endereços alocados pelo sistema operacional), assim cada referência à memória por uma instrução requer dois acessos sequenciais à memória física. Esta situação torna-se um sério gargalo de desempenho. Por essa razão, porções da tabela de página são armazenadas no TLB.

O TLB é basicamente uma memória *cache* para a tabela de páginas. Por esta razão, o TLB pode ser implementado por qualquer esquema de memória *cache*: mapeamento direto, associatividade completa e associatividade por conjunto. O TLB com mapeamento direto é simplesmente uma versão menor e rápida da tabela de páginas.

A *cache* de dados é utilizada para armazenar porções da memória principal e o TLB é utilizado para armazenar porções da tabela de páginas. A interação entre o TLB e a *cache* de dados pode ser vista na figura 6.

Conforme a figura 6, o endereço virtual consiste em um número de página (v-bits) e um deslocamento (g-bits). Se o TLB tem associatividade por conjunto, os v-bits do número de página virtual são divididos em índice (k-bits) e etiqueta ($v-k$-bits). O segundo estágio da unidade de memória corresponde ao acesso ao TLB utilizando o número de página. Assumindo que não há falta no TLB, o TLB trará o número da página física (p-bits), que é então concatenado ao deslocamento (g-bits) para produzir o endereço físico de m-bits, onde $m = p + g$ e m não é essencialmente igual a n. Durante o terceiro estágio do *pipeline* o endereço físico é utilizado para acessar a *cache* de dados. A exata interpretação do endereço físico depende da arquitetura da *cache* de dados. Se os blocos contém múltiplas palavras, os b-bits de baixa ordem são usados como deslocamento para selecionar a palavra específica do bloco selecionado. O bloco selecionado é determinado pelo bits restantes ($m-b$) bits. Se a *cache* tem associatividade por conjunto esses bits restantes

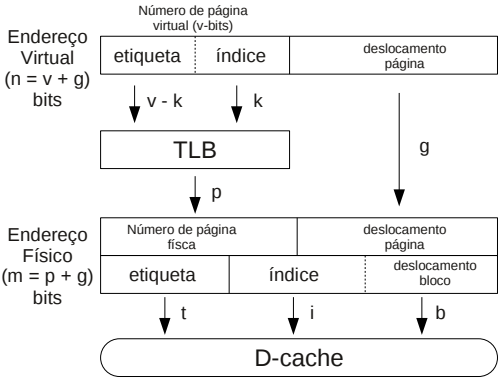


Figura 6: Interação entre *cache* e TLB.

são divididos em uma etiqueta (t-bits) e um índice (i-bits). O valor i é determinado pelo tamanho total da *cache* e sua associatividade: deverá haver i conjuntos na *cache* de associatividade por conjunto. Se não houve falta da *cache*, no final do terceiro estágio do *pipeline* os dados estarão disponíveis.

2.4.4 Protocolos de coerência de *cache* para multiprocessadores

Atualmente, protocolos de atualização de *cache* não são mais utilizados devido ao consumo excessivo de banda nas interconexões. Desta forma, para sistemas multiprocessados com memória compartilhada utiliza-se protocolos de invalidação. Neste protocolo, apenas um processador pode escrever em uma linha de *cache* em qualquer ponto. Esta política é reforçada certificando-se que a linha que será modificada será a única cópia do sistema invalidando as outras presentes nos outros processadores.

O protocolo de invalidação requer que o diretório de *cache* mantenha no mínimo dois estados para cada linha de *cache*: modificado (M) e inválido (I). No estado inválido, o endereço requisitado não está presente e deve ser buscado da memória. No estado modificado o processador sabe que a cópia em questão é exclusiva no sistema e permite-se realizar leituras e escritas na linha. Obviamente que cada linha modificada deve ser regravada na memória. Uma otimização simples incorpora um *bit* sinalizador (*dirty bit*) no estado da linha da *cache* que permite que o processador diferencie as linhas que são exclusivas do processador (conhecido como estado E) e outras que são exclusivas e estão desatualizadas devido uma escrita (estado M). Este protocolo é conhecido como MEI e foi utilizado em processadores PowerPC G3.

Pelo protocolo MEI, não se permite que linhas de *cache* existam em mais de um processador ao mesmo tempo. Para resolver este problema e permitir leituras de linhas, inclui-se o estado compartilhado (estado S (*Shared*)). Este estado indica que uma ou mais cópias remotas existem. Se deseja-se modificar uma linha no estado S, primeiramente modifica-se o estado para o estado M invalidando as cópias remotas. A inclusão do estado S nomeia o protocolo MEI para MESI.

Um aprimoramento do protocolo MESI pode ser obtido adicionando o estado *owned* (O), resultando no protocolo MOESI. O estado O é atingido quando acontece uma leitura remota do bloco sinalizado (*dirty*) no estado M. Assim, o estado O significa que múltiplas cópias válidas do bloco existem, onde o requisitor remoto recebeu uma cópia válida satisfazendo a leitura, enquanto o processador local também mantém uma cópia. Contudo ele é diferente do estado convencional S pois se evita a atualização dos dados na memória. Este estado também é conhecido como *shared-dirty*, pois o bloco é compartilhado, mas aguarda a atualização. O sistema que implementa o estado O pode colocar tanto o processador remoto quanto o local em estado S: basta uma cópia ser sinalizada para atualização.

Uma comparação dos protocolos de coerência de *cache* MOESI e MESIF pode ser encontrada no trabalho de Hackenberg et al. (2009).

2.5 DESEMPENHO DE PROCESSADORES

O objetivo principal no desenvolvimento de microarquiteturas é aumentar o desempenho, mas recentemente tem-se visado a redução de consumo de energia dos processadores. Na equação 2.1 (SEHN; LIPASTI, 2005) apresenta-se, de forma simplificada, os termos relacionados com a métrica de desempenho de um processador.

$$\begin{aligned} \frac{1}{desemp.} &= \frac{tempo}{programa} = \\ &= \frac{instrucoes}{programa} \times \frac{ciclos}{instrucao} \times \frac{tempo}{ciclo} \end{aligned} \tag{2.1}$$

Primeiramente, o desempenho é relacionado com quanto tempo o processador leva para executar dado programa ($tempo/programa$). Esta métrica pode ser formulada pela multiplicação dos três termos seguintes, onde o primeiro ($instrucoes/programa$) indica o número de instruções necessárias para executar o programa particular; o segundo representa a média de quantos ci-

clos de máquina são consumidos por cada instrução – *cycles per instruction* (CPI); e o terceiro representa o tempo de cada ciclo de máquina. Contudo aumentar o desempenho não é uma tarefa trivial, pois os três termos não são independentes havendo interações complexas entre eles: redução de qualquer um deles pode favorecer o aumento de outro.

2.6 CONCLUSÕES E TENDÊNCIAS

Devido às barreiras de aumento de frequência ocasionada pela fuga de tensão de componentes internos do *chip* e altas perdas térmicas, fabricantes têm desenvolvidos processadores com mais núcleos ou com mais *threads* de *hardware*(*HyperThreading*, etc) (SODAN et al., 2010).

Antigamente, os desenvolvedores de processadores usavam a área adicional devido a miniaturização para o desenvolvimento de processadores superescalares com replicação das unidades de execução e *pipelines* profundos para explorar o paralelismo ao nível de instrução. Contudo, a atual direção emprega o espaço disponível para adicionar mecanismos de *multithread* ou *multicore*. Esses processadores suportam multitarefa rodando programas em paralelo ou ainda diversos programas concorrentes.

Os CPUs *multithread* suportam execução concorrente de *threads* à nível de instrução, com o objetivo de melhorar a utilização dos recursos enviando instruções de várias *threads*. CPUs *multicore* possuem uma concorrência em um nível mais alto, focando em menos utilização por núcleo objetivando a escalabilidade com replicação de núcleos.

Cada *thread* de *hardware* precisa de seus próprios componentes de estado, como registradores de controle e ponteiros de instrução. Contudo, o Intel Xeon precisa apenas de um adicional de 5% de espaço para suportar uma segunda *thread*. Atualmente, processadores comerciais apenas enviam instruções de duas *threads* por core por ciclo, mas existem protótipos como o Tera/Cray MTA com 128 *threads*, apropriados para *High performance computing*.

O *multithread* por núcleo tem limitação de escalabilidade, limitada pela saturação das unidades de execução e o custo de *threads* adicionais. Por isso, os CPUs *multicore* são mais escaláveis. Estes chips mantém muito da arquitetura dos seus antecessores, replicando apenas as unidades de controle e execução e compartilhando outras, como: *cache*, controlador de memória, FPUs, componente de refrigeração, pinos, etc. No entanto, compartilhamento tem a desvantagem da necessidade de controle e da saturação desses recursos.

Tendências de desenvolvimento indicam a replicação de componentes adicionais em CPUs *multicore*, como controladores de memória e *caches*.

A arquitetura de memória varia, havendo *caches* privadas nos níveis próximos ao processador e compartilhada no nível sucessor à memória principal. Colocando o controlador de memória no chip aumenta a banda e diminui a latência, o que justifica a atual tendência. Outra recente pesquisa, condiz com a integração de GPUs ao CPUs, formando processadores híbridos.

Devido a grande diferença de desempenho das memórias e dos CPUs, estes atualmente contém grande quantidade de memória *cache* no chip. Outras arquiteturas não utilizam *cache*, aliviando o problema da memória com altos níveis de *multithreading* (Tera/Cray MTA), ou utilizando memórias de alta velocidade diretamente acessadas (IBM Cell SPE).

O benefício de *caches* privadas ou compartilhadas não depende somente do espaço disponível no *chip*, mas também das características da aplicação. *Caches* compartilhadas são importantes se as *threads* da mesma aplicação executando em múltiplos núcleos compartilham uma quantidade significativa de dados. Contudo, *caches* compartilhadas impõe alta demanda de interconexão. Aplicações que não compartilham muitos dados vão competir por *cache*. Esta situação dificulta a previsão de cada *thread* pois depende de detalhes como padrões de acesso, localidade de memória como também a carga do sistema. Novas pesquisas propõem uma abordagem híbrida de *cache* que permite particionamento dinâmico.

A interconexão dos componentes impacta diretamente no desempenho do CPU. Inicialmente, utilizava-se um barramento para essa conexão. Contudo, a tendência é utilizar mecanismos avançados como *crossbar* para reduzir a latência e contenção. A AMD emprega o *crossbar* enquanto a Tilera implementa um *fast nonblocking multilink mesh*. O *crossbar* faz a conexão de múltiplos dispositivos de maneira matricial e o *fast nonblocking multilink mesh* faz a conexão em três dimensões. Esses dois tipos de interconexões são relativamente caros. Por exemplo, um *crossbar* de 8 x 8 consome tanta área de chip quanto cinco núcleos e consome energia equivalente a dois (SODAN et al., 2010).

A comunicação para componentes exteriores evolui de barramento (*Front Side Bus* – FSB) para comunicação de pacotes, com conexão ponto a ponto. A AMD implementa esse conceito com o *HyperTransport* seguido pela Intel como o *QuickPath Interconnect*. Estas conexões podem interligar vários processadores em diferentes *chips* como também todo o sistema de entrada/saída.

Processadores que focam no desempenho por *thread* possuem frequências mais elevadas dos que focam no *multithread*. Contudo, a utilização do espaço do chip para aumento por *thread* resulta em ganhos não lineares, onde a experiência sugere que o desempenho apenas dobra com o quádruplo da complexidade dos componentes adicionais para suporte de *threads* extras.

Grandes quantidades de núcleos simples são preferíveis desde que a parte sequencial da aplicação seja pequena, caso contrário, CPUs complexos têm provado ser mais efetivos. O desempenho não domina mais o objetivo do desenvolvimento: custos de fabricação, tolerância a falhas, eficiência energética e dissipação de calor são também considerações críticas. Como os núcleos são simplificados, o consumo energético diminui linearmente, o que é uma grande vantagem de processadores *multicore*.

3 ANÁLISE ESTÁTICA DO WCET – *WORST CASE EXECUTION TIME*

O *Worst-Case Execution Time* (WCET) corresponde ao máximo tempo de execução de dado programa em um processador específico. O WCET é um elemento essencial para a análise de escalonabilidade e garantia de tempo de sistemas de tempo real, especialmente em sistemas de tempo real críticos (*safety-critical hard real-time systems*) como aviônicos e outros sistemas de controle. Neste trabalho, os termos tempo real crítico e tempo real com garantia são utilizados de forma intercambiável. Geralmente o WCET é derivado da predição de um limite superior do tempo de execução de tarefas de um programa. A redução da sobrestimação melhora a precisão e é um elemento indicativo da qualidade da ferramenta que obtém o WCET.

O analisador WCET computa os limites do WCET de tarefas. Geralmente, a ferramenta analisa o binário executável diretamente levando em conta o comportamento intrínseco do processador (*cache* e *pipeline*). A estimação do WCET é uma tarefa desafiadora e as técnicas baseadas em medição repetitiva, além de serem tediosas, tipicamente não são consideradas seguras. Além disso, é geralmente impossível provar que todas as condições que determinam o pior tempo de execução são consideradas na medição. Os processadores modernos compostos por *caches* e *pipelines* complicam consideravelmente a tarefa de determinação do WCET, pois o tempo de execução de uma única instrução pode depender do histórico de execução.

Para considerar todos os aspectos e fornecer um limite seguro e preciso, as ferramentas analisam o comportamento da *cache* e do *pipeline* baseadas em modelos formais. Este tipo de análise é conhecida como interpretação abstrata. As ferramentas de cálculo WCET podem suportar interfaces que fornecem a visualização gráfica dos piores caminhos de execução e também do estado do processador nos vários blocos e contextos de execução.

Este capítulo tem como objetivo apresentar as características fundamentais presentes atualmente na obtenção do WCET para aplicações de tempo real críticas. Além disso, ilustrar-se-á a utilização de uma ferramenta de análise estática de código (*AbsInt AiT*) amplamente utilizada em aplicações automotivas e aviônicas. Este capítulo está organizado em: seção 3.1 apresenta os fundamentos para análise estática temporal de código, seção 3.2 ilustra a utilização da ferramenta *AbsInt AiT* e, por fim, na seção 3.3, são apresentadas as conclusões.

3.1 FUNDAMENTOS DE ANÁLISE ESTÁTICA TEMPORAL DE CÓDIGO

Os tempos do pior caso de execução exatos são impossíveis ou muito difíceis de determinar, mesmo para classes de aplicações bem restritivas e determinadas. Os problemas dos métodos atuais e um descrição completa das ferramentas contemporâneas podem ser consultados em (WILHELM et al., 2008). De qualquer forma, o objetivo da análise consiste em garantir limite seguro e o mais preciso dos tempos de execução das tarefas do sistema.

Conforme apontado por (CULLMANN et al., 2010), os requerimentos da análise temporal podem ser resumidos em:

- robustez: para assegurar a confiança das garantias;
- eficiência: para ser utilizada em aplicações práticas (indústria, por exemplo);
- resultados precisos: para melhorar as chances do sistema ser seguramente viável.

Qualquer programa, quando executado em um processador moderno de alto desempenho, sofre variações do tempo de execução que dependem dos dados de entrada, estado do *hardware* e interferências do ambiente. De modo geral, o espaço de estados, resultado da combinação dos dados de entrada e possíveis estados iniciais, é muito grande para uma busca exaustiva e completa de todas as possibilidades de execução. Desta forma, busca-se a obtenção de limites superiores de tempo de execução de blocos básicos e constrói-se o limite superior do sistema como um todo.

Para tornar a análise viável, são realizadas abstrações da plataforma (*hardware* real). Em tais abstrações, há perda de informação e esta é uma das características que dificultam a obtenção de limites exatos dos tempos de execução. A quantidade de informação perdida depende dos métodos de análise e das propriedades dos sistemas, como arquitetura de *hardware* e analisabilidade do código.

Conforme apresentado no capítulo 2, os processadores modernos contemplam *caches*, *pipelines* e especulação para melhorar o desempenho no caso médio. *Caches* amenizam a diferença de velocidade entre memória e processador e *pipelines* aumentam a velocidade de execução das instruções. Estes elementos fazem com que a execução de instruções individuais, e consequentemente do programa, varie muito. O tempo de execução de uma instrução está presente em um intervalo formado por dois casos: execução completamente "suave" e uma execução completamente "conturbada". Na execução suave, a instrução transcorre o *pipeline* sem formação de flutuações: todas

as referências de memória são acertos na *cache*, todos os operandos estão prontos e não há conflito com outras instruções nas unidade de execução. Na execução conturbada, considera-se o pior cenário possível no *pipeline*: faltas na *cache*, dependências entre operandos e recursos ocupados nas unidades de execução.

Baseando-se na pesquisa dos últimos anos, desenvolveu-se um modelo padrão para análise temporal estática de código (CULLMANN et al., 2010). Tal modelo pode ser visualizado na figura 7.

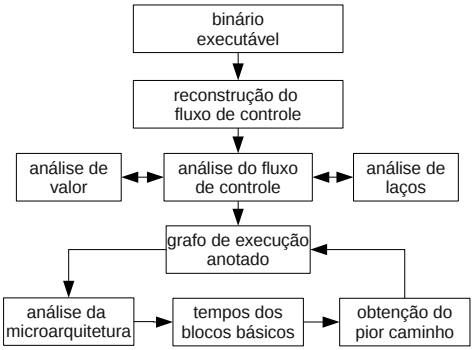


Figura 7: Modelo padrão desenvolvido para análise estática de código (CULLMANN et al., 2010).

Dentro deste modelo pode-se distinguir quatro blocos principais:

- análise de valor;
- reconstrução do fluxo de controle e análise estática para fluxo de controle e dados;
- análise da microarquitetura que computa os limites mínimos e máximos dos blocos de execução básicos;
- obtenção do pior caminho de execução dentre todos possíveis caminhos dentro do programa.

3.1.1 Análise de valor

A análise de valor é utilizada para determinar os valores dos registradores e dos endereços efetivos de memória acessados. Esta análise é utilizada

para determinar o comportamento dos dados e da memória *cache* do processador. O endereço efetivo dos acessos à memória está somente disponível na execução do programa, contudo 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 (WILHELM et al., 2008). A obtenção dos endereços é realizada computando limites dos valores nos registradores do processador e das variáveis locais a cada ponto do programa. Esta análise também é útil para determinar limites de iterações de laços e caminhos de execução inviáveis.

3.1.2 Análise de fluxo de controle

A análise de fluxo de controle tem como objetivo levantar informações sobre os possíveis caminhos de execução. Este conjunto é sempre finito, pois a terminação do programa deve ser garantida. Novamente o conjunto exato dos caminhos geralmente não pode ser determinado, mas a determinação de um superconjunto é um método seguro. O tempo de execução de um caminho seguramente eliminado pode ser completamente ignorado.

A entrada para a análise de fluxo consiste em um grafo de chamadas e um grafo de controle de fluxo da tarefa. Informações adicionais como limite dos dados de entrada e limites superiores dos laços também podem ser considerados quando esta informação não pôde ser detectada pela análise de valor. O resultado da análise pode ser vista como limitações sobre o comportamento dinâmico da tarefa, contendo dados como quais funções podem ser chamadas, dependências entre condições e caminhos viáveis e inviáveis.

A análise de fluxo de controle é considerada mais fácil no código fonte ao invés do binário. Contudo é difícil mapear os resultados que o compilador incorpora à estrutura do fluxo de controle devido a otimização e ao ligamento (*linking*) de código. Por isso, geralmente as ferramentas de análise estática temporal utilizam o código binário já compilado.

Existem várias técnicas utilizadas para realizar esse tipo de análise. As mais comuns são a busca de padrões e técnicas de interpretação abstrata. Maiores detalhes sobre estas técnicas podem ser consultados nas referências de (WILHELM et al., 2008).

Após a conclusão da análise de fluxo, os resultados são traduzidos para um sistema de restrições que pode ser resolvido por técnicas de enumeração implícita de caminhos (IPET – *Implicit Path Enumeration*).

3.1.3 Análise da microarquitetura

Conforme apresentado no capítulo 2, um processador contém vários componentes que fazem o tempo de execução dependente do contexto, como memórias, *caches*, *pipelines* e previsores especulativos. O tempo de execução de instruções individuais e até acessos à memória depende de estados anteriores.

Para estimar limites de tempos de execução precisos para uma dada tarefa, é necessário analisar os estados atuais e anteriores dos componentes do processador que levam a instrução corrente. O analisador do processador determina invariantes sobre os estados de ocupação dos componentes da tarefa e a princípio não há ferramenta tão completa capaz de considerar todos os periféricos na análise (WILHELM et al., 2008) (CULLMANN et al., 2010) (SCHOEBERL, 2009b).

A análise da microarquitetura é novamente realizada sobre o código binário da tarefa pois somente este contém todas as informações necessárias. Há um modelo abstrato do processador, dos subsistemas de memória, barramentos e periféricos que são conservativos em respeito ao comportamento do *hardware* real, ou seja, nunca fornecem tempos menores do que o *hardware* concreto. A complexidade em derivar um modelo abstrato depende fortemente da classe de processador utilizada (WILHELM et al., 2008):

- processadores simples (geralmente *8-bit/16bit*): a construção do modelo é simples. Os fatores que complicam a análise do comportamento do processador incluem instruções com tempo de execução variável devido a contextos com tempos de acessos a memória diferenciados.

- processadores de *16-bit/32bit*: processadores mais avançados com *pipeline* escalar e *caches* simples onde se pode analisar elementos do processador separadamente. Os fatores que complicam a análise estão ligados aos tempos de acesso variados devido a faltas na *cache* e a interação das instruções no *pipeline*.

- processadores superescalares avançados: nestes processadores a construção do modelo é extremamente complexa e os elementos do processador não podem ser analisados separadamente, tornando a análise menos modular e muito mais complexa.

Após a construção do modelo abstrato do processador, utiliza-se a abordagem de Análise de Fluxo de Dados baseado na teoria de Interpretação Abstrata para tornar a análise viável (WILHELM et al., 2008). O método de interpretação abstrata é usado para computar invariantes para cada contexto

de fluxo sobre os estados do processador a cada ponto do programa. Se há um invariante de estado para cada ponto do programa, este mantém-se para todos os caminhos que levam a tal ponto. Os invariantes e o conjunto dos caminhos que levam até tal ponto formam os contextos de execução. Estes contextos expressam o conhecimento estático sobre os conteúdos da *cache*, ocupação das unidade funcionais do processador e estados das unidades de previsão de fluxo (*branch prediction*). O conhecimento da *cache* pode ser realizada por análise conforme Ferdinand e Wilhelm (1998) e é utilizada para classificar referências à memória em acerto e faltas. O conhecimento das unidades funcionais é utilizada para incluir ou excluir flutuações no *pipeline*. É possível concluir que o caminho de maior tempo de execução é o pior estado de todas as unidades funcionais assumindo que "tudo deu errado". Processadores modernos incorporam anomalias temporais, portanto assumir que a pior situação local gera a pior situação global não é realmente seguro (WILHELM et al., 2008) (CULLMANN et al., 2010).

3.1.4 Obtenção do pior caminho

Na literatura a obtenção analítica do pior tempo de execução após as análises de valor, reconstrução do fluxo de controle e análise da microarquitetura é realizada através de métodos baseados em estrutura (*structure-based*), baseados em caminhos (*path-based*) e técnicas que utilizam enumeração implícita de caminhos (IPET – *Implicit Path Enumeration*) (WILHELM et al., 2008).

As ferramentas mais avançadas são baseadas em IPET. Neste método o fluxo do programa e os tempos dos blocos básicos são combinados em uma série de restrições aritméticas. A técnica é tradicionalmente aplicada globalmente em toda a tarefa com todas as informações de fluxo. O IPET utiliza técnicas de programação linear inteira ou programação constante derivando uma complexidade potencialmente exponencial em relação ao tamanho da tarefa.

Em resumo, para derivar 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.

3.2 AVALIAÇÃO PRÁTICA DE FERRAMENTA WCET – AIT

O *AiT WCET Analyzers* é uma ferramenta comercial implementada pela empresa alemã *AbsInt Angewandte Informatik GmbH* em conjunto com Universidade de Saarland. As licenças acadêmicas são livres e as comerciais variam entre €17.970 a €38.970 dependentes do processador e compilador. Há disponibilidade para uma série de compiladores/processadores entre eles ARM7, PowerPC (5xx, 603e e 55xx) e NEC V850E. As avaliações contidas neste capítulo foram realizadas com o processador ARM7-TDMI-S e compilador GCC – *GNU C Compiler* mas o princípio básico de operação é idêntico para as outras versões. Segundo (ABSINT, 2011), de forma geral, o pessimismo introduzido pela análise do AiT é menor que 10% em alguns processadores.

O processador ARM7 possui um *pipeline* simples de três estágios e não sofre com dependências inter-estágios conforme citado no capítulo 2. Há um estágio de busca, um de decodificação e outro de execução. O *hardware* dos estágios são independentes onde a busca, decodificação e execução são operações totalmente paralelas. O processador é mais eficiente na execução de código linear pois assim que uma condição de salto é encontrada e avaliada, todas as instruções em execução são descartadas, ou seja, não há elementos especulativos para suposição de endereços de saltos. A arquitetura baseia-se no método de "carregar-gravar" (*load-and-store architecture*) onde qualquer computação somente é realizada pelo conjunto central de 16 registradores e o resultado final é regravado na memória. Maiores detalhes sobre a microarquitetura ARM7 podem ser encontrados em (ARM, 2010) ou (MARTIN, 2007).

Nesta ferramenta, os resultados da análise de valor são utilizados para a análise da memória *cache* para prever o comportamento da *cache* de dados. A saída da análise de *cache* é utilizada na análise do *pipeline* que permite a predição de flutuações relacionadas às faltas de *cache*. Os resultados de *cache* e *pipeline* são usado para computar o tempo de execução dos caminhos do programa. No caso do *AiT WCET Analyzers*, a análise de valor, análise de *cache* e de *pipeline* são feitas por interpretação abstrata (ABSINT, 2011) para análise estática de programas. A análise de caminhos é computada por Programação Linear Inteira (IPET – *Implicit Path Enumeration Technique* – Técnica de enumeração implícita de caminhos).

Para executar uma análise no AiT é necessário fornecer o arquivo executável, neste caso formato binário *arm-elf*, o endereço inicial do código a ser analisado (tipicamente o nome da função, como *main*), uma descrição simples das memórias, modelo e frequência do processador e anotações opcionais (*AIS file*). Estas anotações podem ser necessárias para chamadas de

funções indiretas, limites superiores de laços, profundidade de recursões, códigos não viáveis, além de outros detalhes particulares quando a informação não é detectada pela ferramenta. Contudo, a quantidade de anotações é muito reduzida pelas técnicas de análises de laços e reconhecimento de chamadas de funções tabeladas.

Apesar da análise ser realizada sobre o arquivo binário, o *software* ainda permite a entrada de aquivos de código fonte. Os fontes são utilizados para algumas visualizações, anotações no *disassembly* e tarjas em blocos na visualização gráfica do programa. Além disso, é permitido a construção das anotações opcionais no próprio código fonte.

A ferramenta fornece como saída um limite superior para o tempo de execução do código analisado. As visualizações resultantes da análise mostram informações detalhadas sobre vários aspectos do programa, como por exemplo, o pior caminho de execução ou o estado do processador em dado ponto do programa. O código que será analisado para a estimativa do limite superior do tempo de execução é estritamente sequencial, ou seja, não deve haver *threads*, paralelismo ou preempção. O AiT não assume nenhum evento externo, como exceções, interrupções, atualizações de memórias (*DRAM refresh*), entrada/saída, temporizadores ou co-processadores. Estruturas dinâmicas baseadas na família de funções *alloc (malloc, realloc, etc)* não devem ocorrer no código analisado como também construções da linguagem C envolvendo *setjmp* e *longjmp*. Finalmente, a estrutura padrão de chamadas de funções deve ser obedecida bem como endereços de retorno não podem ser modificados.

A versão avaliada do analisador AiT provê a construção do grafo de controle de fluxo, analisador de valores, analisador de pilha e analisador WCET.

O grafo de controle de fluxo (*control flow graph*) cria uma visualização combinada do grafo de chamadas e controle de fluxo da aplicação. Na figura 8 encontra-se um grafo aplicado à função “main” de um programa demonstrativo da ferramenta.

Através do analisador de valores (*Value Analyzer*) pode-se explorar os efeitos de instruções individuais nos registradores e memória.

O analisador de pilha (*Stack Analyzer*) computa o uso máximo da pilha. Códigos produzidos em C para ARM7 tipicamente utilizam a pilha de programa para gravar endereços de retorno, parâmetros e variáveis locais. O analisador de pilha calcula esses níveis de utilização sendo útil em aplicações embarcadas com definições estáticas do tamanho da pilha. Na figura 9 é mostrado o resultado da análise de pilha a partir da função “main”.

A análise temporal implica na análise de memórias e de pipeline. Na figura 10 é apresentado o resultado de uma análise onde os arcos segmentados

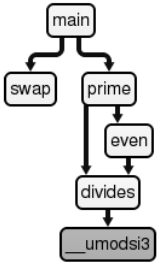


Figura 8: Exemplo de grafo de controle fluxo

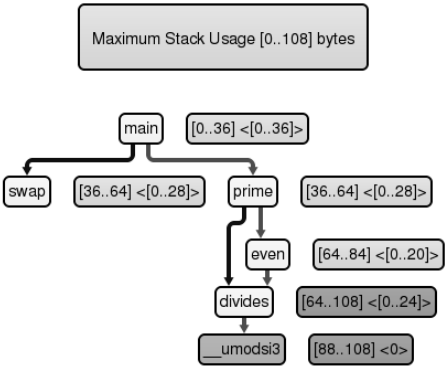


Figura 9: Exemplo de grafo de análise de pilha

demonstram o caminho tomado para computar o WCET de execução e as barras abaixo de cada rótulo de função representam a porcentagem de tempo de cada função em relação ao tempo total computado.

Além disso, apesar do processador ARM7 utilizado não possuir *cache* propriamente dita, a ferramenta permite fazer análises com diferentes parâmetros de *cache*. No caso de alguns microcontroladores que utilizam o núcleo ARM7 como o microcontrolador NXP LPC2368, há uma espécie de *cache* para a memória *flash* conhecido como módulo acelerador de memória (MAM – *Memory Accelarator Module*) (MARTIN, 2007). Esta *cache* básica de instruções deve e pode ser analisada pela ferramenta de WCET quando o processador busca o código executável da memória *flash* bastando configurar o AiT com esses parâmetros.

Outras ferramentas que fazem estimação do WCET de tarefas exis-

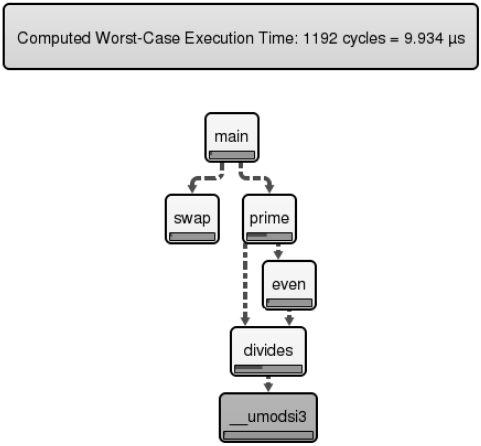


Figura 10: Exemplo de grafo de análise de tempo

tem, como por exemplo *Bound-T*, *Chronos*, *Heptane*, *SWEET*, *SymTA/P* e *RapiTime*, além de outros protótipos como apresentados em Wilhelm et al. (2008). O *Bound-T* possui o funcionamento semelhante ao AiT mas com menor acesso à gráficos e detalhes do processador. A ferramenta *Chronos* possui código fonte livre mas necessita de ferramentas externas como por exemplo o simulador de arquitetura *SimpleScalar*. O *Heptane* é outra ferramenta de código fonte livre que faz análise de WCET através do binário e do código fonte do programa. O *SWEET* é desenvolvido de forma modular (vários pequenas análises isoladas) e a análise de fluxo é realizada em conjunto com um compilador distribuído com a ferramenta. As ferramentas *SymTA/P* e *RapiTime* são híbridas, realizando análises de pior caminho pelas técnicas de otimização mas a obtenção dos tempos de execução é realizado através de medições no *hardware* real com código instrumentado incorporado à tarefa em análise.

3.2.1 Obtenção do WCET de uma tarefa

Neste programa exemplo, foi proposta uma tabela *hash* para realizar a análise temporal e de pilha. Cada item da tabela é formado por uma *string*, que é utilizada pela função *hash*, um *array* de dados (*payload*) e um ponteiro para uma função arbitrária, conforme o quadro 3.1. Foram definidas cinco entradas estáticas que são inseridas na tabela durante a execução do código. Na inserção dos dados, se o ponteiro da função for válido, a função é cha-

mada. Estas funções arbitrárias podem conter chamadas tabeladas, recursão ou laços.

```
/* Estrutura de dados de uma entrada na tabela HASH */
struct hash_data {
        char *str;
        unsigned char data[BUFF_SIZE];
        unsigned int (*pre_function) (unsigned int);
};

/* A tabela HASH */
struct hash_entry hash_table[HASH_TABLE_SIZE];
```

Quadro 3.1: Estrutura dos dados da tabela *hash*.

Primeiramente, para uma análise estática de código é necessário que a ferramenta construa corretamente o grafo de controle de fluxo. Nesta etapa, a ferramenta deve mapear todas as chamadas de função diretas e indiretas. Caso ocorra perda do controle do fluxo, como por exemplo chamadas indiretas por ponteiros ou tabelas de funções, erros são reportados e o usuário deve fornecer todos os possíveis caminhos de continuação do fluxo explicitamente no arquivo de anotação. Esta situação é ilustrada no quadro 3.2 onde o fluxo de controle é perdido na execução da função *hash_insert* e na função *data_4_ pre_function*. A anotação 1 explicitamente informa que *data_2_pre_function*, *data_3_ pre_function*, *data_4_pre_function* ou *data_5_pre_function* podem ser chamados durante *hash_insert*. Na anotação 2 informa-se que o fluxo de chamadas em *data_4_pre_function* ocorre pelas funções da tabela *function_array[3]*.

```
# anotação 1: perda de fluxo na inserção dos dados na tabela hash
instruction routine "hash_insert" + 1 computed is call
  and calls data_2_pre_function, data_3_pre_function,
  data_4_pre_function, data_5_pre_function;

# anotação 2: chamada por tabela -> function_array = {function_1,
# function_2, function_3}
instruction routine "data_4_pre_function" + 1 computed is a call
        and calls via function_array[3];
```

Quadro 3.2: Anotações para chamadas de funções por ponteiros e por tabelas.

Após as anotações de chamadas, o grafo de fluxo de controle do código teste é construído conforme a figura 11. Na função *main* há um laço (*main.L1(loop)*) e várias chamadas de funções. A partir de *hash_insert*, há as chamadas do conjunto de funções *data_pre_function* bem como as chamadas de *function_1* a *function_3* pelo *function_array [3]*.

Normalmente o número de iterações de laços são dependentes dos dados de entrada da aplicação, como é o caso do laço de geração do *hash* da tabela. Neste caso deve haver uma anotação para fornecer à ferramenta o limite de pior caso das iterações. Baseando-se nisto, através do quadro 3.3

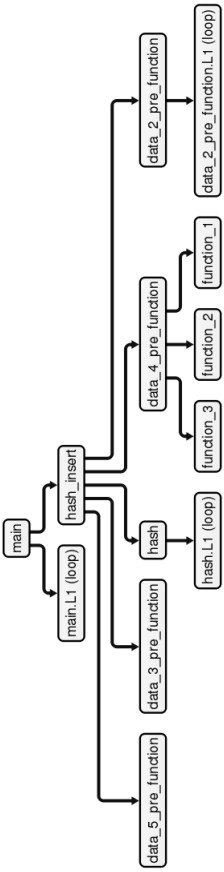


Figura 11: Grafo de fluxo de controle para o exemplo da tabela *hash*.

ilustra-se a anotação para o limite do laço considerando a pior dado de entrada e também o código problemático.

```
# anotação: o tamanho máximo da string dos itens da tabela é:
# struct hashdatadata5 =
# {.str = "hackers_never_die"} ---> 17 chars + NULL = 18

loop "hash" + 1 loop max 18 begin;

# código de geração do hash
# unsigned int hash(char *str) {
#
#  //limite do laço não detectado, depente do contexto
#  while ((c = (unsigned char ) *str++)) {
#    hash = c + (hash << 6) + (hash << 16) - hash; } (..) }
```

Quadro 3.3: Anotações de limite do laço da função *hash*.

Após estas anotações, a ferramenta foi capaz de computar um limite superior para o WCET do código. Este exemplo ilustrou as anotações necessárias para laços com limites dinâmicos, chamadas por ponteiros de funções e tabelas. O analisador de utilização de pilha não necessitou de nenhuma anotação. As anotações são dicas e instruções necessárias quando certas condições não podem ser detectadas (chamadas por ponteiros) ou ainda condições que possam variar conforme os possíveis dados de entrada da tarefa (limite de laços como o campo *string* dos dados da tabela). As anotações devem sempre considerar o pior caso das condições.

3.2.2 Resultados da Análise WCET e de pilha

Através da figura 12 apresenta-se o resultado da análise WCET do programa. Abaixo de cada rótulo há uma barra que indica a porcentagem de tempo de cada função em relação ao valor total 45.661 ciclos de execução (laço de *data_2_pre_function* é o mais custoso). As arestas segmentadas indicam o fluxo de execução que gerou o pior tempo de execução, formado por $main \rightarrow main.L1 \rightarrow hash\_insert \rightarrow hash \rightarrow hash.L1 \rightarrow data\_2\_pre\_function \rightarrow data\_2\_pre\_function.L1$.

É interessante notar que o código implementado chama as funções *data_2_pre_function*, *data_3_pre_function*, *data_4_pre_function* e *data_5_pre_function* diferentemente a cada item inserido na tabela. Contudo, visualizando-se a figura 12, somente a chamada para *data_ 4_pre_function* é considerada no tempo de execução. Esta situação acontece porque na anotação de chamadas, *hash_insert* pode chamar qualquer uma destas funções, portanto, o WCET é composto somente por chamadas de *data_2_pre_function* (execução mais custosa) toda a vez que houver uma inserção na tabela.

O gráfico de utilização da pilha pode ser visualizado na figura 13.

Como é possível especificar a configuração de memória do processador, construiu-se uma configuração com cache (ilustrando a possibilidade de análise de uma *cache* para a *flash* como no microcontrolador LPC2368) e outra configuração mais básica considerando apenas tempos de acessos entre memórias RAM e *flash* diferenciados. A configuração de memória pode ser implementada conforme o quadro 3.4 que deve constar também no arquivo de anotações. O parâmetro *time* explicita a quantidade de ciclos necessários para acessar a região em questão e *clock-ratio* especifica a relação de frequência de operação da memória em relação ao processador. As configurações de cache

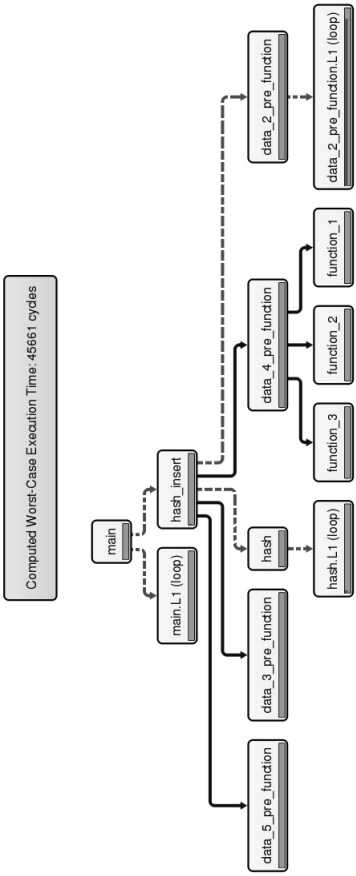


Figura 12: Grafo de pior tempo de execução da tabela *hash*.

para dados e instrução são: conjuntos de 1024, tamanho de 4 bytes, mapeamento direto, política de despejo LRU (*Last Recently Used*) e penalidade de 4 ciclos na gravação da memória principal. Na tabela 4 apresenta-se os resultados com os tempos de execução com as três configurações de memória.

```
# Configuração para tempos de acesso diferenciados
# Regiões conforme o linker do compilador
//código
area 0x00000000..0x00070000 access clock-ratio = 1:1;
//flash
area 0x00070000..0x00080000 access clock-ratio = 1:1, time 4;
//SRAM
area 0x40000000..0x40080000 access clock-ratio = 1:1;
```

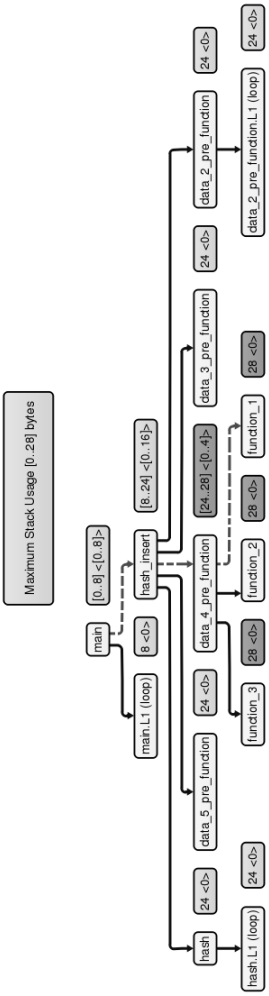


Figura 13: Grafo de utilização de pilha para exemplo da tabela *hash*.

```
# Configuração para simulação com cache
# Toda a memória é considerada com cache
area 0x0 .. 0xffffffff
   access code time=4, cached and  # code cache miss = 4 cycles
   access data time=4, cached ;    # data cache miss = 4 cycles

# Descrição da cache
cache instruction
```

```
    set-count=1024,
    assoc=1,
    line-size=4,
    policy=lru,
    wb-penalty=4
and
    data
    set-count = 1024,
    assoc=1,
    line-size=4,
    policy=lru,
    wb-penalty=4;
```

Quadro 3.4: Anotações para configurações diferenciadas de memória.

Configuração	**Ciclos**
Padrão (1:1)	34.759
Flash (1:4)	42.760
Cache LRU	45.661

Tabela 4: WCET para diferentes configurações de memória

O AiT fornece os estados do *pipeline* para diferentes blocos do programa sendo possível visualizar o comportamento do *pipeline* e dos estados da análise estática da cache (interpretação abstrata *must/may* (FERDINAND; WILHELM, 1998)). A análise realizada com *cache* é mais complexa, onde a ferramenta deve prever todo o comportamento da *cache* com respeito aos acertos/faltas e refleti-los no *pipeline*. Nos gráficos da ferramenta é possível visualizar as diretivas *must* que representam os dados que estarão com certeza na *cache* e *may* que podem estar. Com base nesta classificação, é composto o pior estado de *fetch* de instruções e dados onde se tem o conhecimento para computar corretamente os tempos de execução de cada instrução. Maiores detalhes sobre a simulação de cache podem ser consultados no manual da ferramenta (ABSINT, 2010).

A apresentação dos passos e construção das anotações da análise desta tarefa pôde dar uma visão global da ferramenta WCET AiT. Em termos de anotações foram abordados a maioria dos aspectos contemplando chamadas de funções indiretas, chamadas tabeladas (*arrays*), limites de laços e configuração de memória. A quantidade de anotações necessárias é dependente do código e das otimizações do compilador. A possibilidade de análise com diferentes tipos de acesso a memórias representa uma boa alternativa para escolha de processadores e análise do impacto das configurações sobre o tempo de execução.

3.3 CONCLUSÃO

A importância de obter um valor real mais preciso para o WCET é óbvia. Tal dado é usado em toda e qualquer análise de escalonabilidade e seu valor em última análise vai determinar a viabilidade ou não do sistema. Medições não são capazes de substituir o cálculo do WCET com a mesma eficácia. Ou a medição é otimista e os *deadlines* do sistema não estarão garantidos, ou a medição foi artificialmente aumentada por algum fator de segurança e é exageradamente pessimista.

Este capítulo apresentou os fundamentos para obtenção de WCET para aplicações de tempo real crítico e a aplicação do uso da ferramenta AiT para cálculo do WCET. Foram abordadas as características da ferramenta, os dados de entrada necessários, alguns gráficos de resultados e principalmente os passos e anotações necessários para análise de um programa proposto.

Visto que o método de obtenção do WCET já possui uma certa padronização e certas deficiências da análise de valor e de fluxo podem ser contornadas por anotações, a grande dificuldade da obtenção do WCET está relacionada a análise da microarquitetura. O *hardware* utilizado na criação de sistemas de tempo real crítico tem grande efeito na facilidade em relação à previsibilidade na execução dos programas. No caso dos processadores simples (por exemplo de *8-bit* e *16-bit*), a construção do modelo abstrato é mais fácil no ponto de vista do comportamento temporal e a problemática resume-se à correta construção do fluxo de controle. No caso de aplicações que necessitem de um desempenho maior utilizando-se de processadores como o ARM7 e os recentes Cortex R, os modelos abstratos são relativamente simples pois estas microarquiteturas incorporaram *pipelines* e estruturas de *cache* simples. No espectro de aplicações de alto desempenho como aviônica e automotiva, é necessário utilizar processadores complexos com *caches* e execução fora de ordem. Neste contexto encontra-se processadores como PorwerPC 750 e ARM11 que necessitam de ferramentas e métodos mais avançados.

Conforme a literatura, o AiT é uma das melhores ferramentas, sendo capaz de analisar muitos códigos de teste propostos em (WILHELM et al., 2008), bem como no exemplo de tarefa com tabela *hash* apresentado. A interface de fácil utilização fornece dicas de anotações em conjunto com atalhos para o código fonte. Isto permite que o usuário concentre-se no WCET e evite invocações de outros programas e interfaces adicionais. As visualizações do AiT ajudam os usuários a entender o comportamento temporal dos programas. As chamadas anotadas e os grafos de controle de fluxo podem ser dinamicamente explorados pela ferramenta de visualização. A ferramenta também detecta limites de laços, faz verificação de erros das anotações em conjunto com os resultados das análises e reporta aos usuários melhorias das

anotações.

No caso de desenvolvimento de aplicações que utilizem ferramentas WCET é necessário ficar atento ao tipo de código que irá analisar, principalmente com relação às otimizações do compilador (transformação de laços), alocação dinâmica de memória e preempção do código. Com respeito à preempção, as ferramentas consideram somente código sequencial não interrompido por nenhum recurso de *hardware* ou *software*. Se o processador não possui nenhum elemento que irá incorporar latências extras durante uma interrupção do fluxo de execução (*cache*, por exemplo), a estimação de WCET sequencial pode ser incorporada diretamente na análise de escalonabilidade em algoritmos preemptivos conforme será abordado nos próximos capítulos. Caso contrário, o WCET estimado com *cache* poderá não estar seguro pela quantidade de faltas de cache durante uma preempção que poderá ocorrer a qualquer ponto de execução da rotina.

4 DISCUSSÃO SOBRE A OBTENÇÃO DE GARANTIA TEMPORAL NA ANÁLISE DE SISTEMA DE TEMPO REAL CRÍTICO

Os fabricantes de processadores atuais adotam soluções estatísticas para melhorar a relação de custo/desempenho de seus produtos e esta tendência faz com que os processadores acabem atingindo um bom desempenho. Como as aplicações de tempo real cada vez mais requerem capacidade de processamento, estes processadores acabam sendo utilizados para implementação de aplicações de tempo real baratas e poderosas, mas sofrem com a falta de previsibilidade. Características como *pipelines* com dependências entre instruções, *caches*, elementos especulativos e execução fora de ordem, apesar de melhorar o desempenho, complicam enormemente a análise de tempo real de tais sistemas.

Como o acesso rápido à memória é essencial para a garantia de desempenho, os processadores utilizam uma hierarquia de memória na qual há uma pequena quantidade de memória SRAM rápida, seguida pela memória principal DRAM e finalmente o disco. A SRAM é utilizada para manter cópias temporárias formando a memória *cache*. As memórias *cache* são fundamentais para entregar com velocidade os dados ao processador. Contudo a organização padrão da *cache* é projetada para desempenho de caso médio, sendo difícil de modelar e estimar na análise WCET.

A análise de escalonabilidade de tempo real requer o parâmetro de pior tempo de execução (WCET – *Worst-Case Execution Time*) da tarefa. O WCET é um elemento essencial para a análise de escalonabilidade e garantia de sistemas de tempo real, especialmente em sistemas de tempo real críticos (*safety-critical hard real-time systems*) como aviônicos e outros sistemas de controle. O WCET é calculado através de ferramentas específicas como apresentado no capítulo 3. O ganho de desempenho atingido com as soluções de *pipeline*, *cache*, etc, deve ser considerado no cálculo do WCET da tarefa e estes elementos de *hardware* complicam os modelos abstratos da ferramenta WCET.

Além dos problemas relacionados com a arquitetura, as ferramentas WCET assumem que o código executado é contínuo, isto é, preempções não são permitidas. Esta limitação restringe a aplicação das ferramentas para sistemas não preemptivos como executivo cíclico ou escalonamento cooperativo. Para resolver esta limitação, pode-se considerar a *cache* mais previsível (com técnicas de particionamento de *cache*) ou modelar o comportamento dos efeitos da *cache* e incorporá-los na análise de escalonabilidade.

Este capítulo apresenta uma discussão sobre a obtenção de garantia

temporal na análise de sistema de tempo real crítico considerando elementos como *cache* e sistemas multiprocessados. Na seção 4.1 são apresentados conceitos fundamentais sobre o escalonamento de tempo real e análises de escalonabilidade clássicas. Na seção 4.2 são apresentados os elementos de *hardware* arquiteturais que complicam a análise de tempo real. Como a memória *cache* tem um comportamento indesejável em sistemas preemptivos, na seção 4.3 são apresentadas algumas técnicas para incorporação dos efeitos da *cache* na análise de escalonabilidade. Na seção 4.4 são apresentados alguns fundamentos para implementação de uma arquitetura voltada a sistemas de tempo real e algumas soluções utilizadas em arquiteturas convencionais.

4.1 ESCALONAMENTO DE TEMPO REAL

Antigamente, os sistemas de tempo real eram escalonados por "tabelas fixas" formando os executivos cíclicos. Durante 1970 e 1980, chegou-se a conclusão que estas abordagens de escalonamento estáticas produzem sistemas inflexíveis e difíceis de manter (SHA et al., 2004). Neste período, organizações e pesquisadores cooperaram para desenvolver uma nova infraestrutura de tempo real consistindo na teoria de escalonamento com prioridade fixa, padrões de *hardware* e *software* e ferramentas de análise de escalonabilidade.

Grande parte das teorias de escalonamento de tempo real utilizam o conceito de prioridade para as tarefas. No escalonamento com prioridades, é atribuída uma prioridade às instâncias da tarefa seguindo alguma política. Se para todas as instâncias da tarefa é sempre atribuída a mesma prioridade (no projeto por exemplo), o escalonamento é dito como prioridade fixa por tarefa (DAVIS; BURNS, 2009b). Quando as prioridades das instâncias variam, podendo esta variação ocorrer a qualquer instante de tempo, o escalonamento é conhecido como prioridade dinâmica (DAVIS; BURNS, 2009b). Se diferentes instâncias de uma tarefa podem ter prioridades diferentes, mas cada instâncias possui uma única prioridade fixa individual, o escalonamento é conhecido como prioridade fixa por instância (DAVIS; BURNS, 2009b). Esta classificação considera sempre a variação de prioridade em relação a tarefa.

Quanto aos algoritmos de escalonamento, estes podem ser classificados quanto à preempção em preemptivos, não preemptivos ou cooperativos (BUTTAZZO, 2005). Nos preemptivos, uma tarefa em execução pode ser interrompida permitindo que outra execute a qualquer momento. Nos não preemptivos, a partir do momento que uma tarefa entra em execução, ela é executada até ser completada. Quanto aos cooperativos, as tarefas são interrompidas em pontos definidos de sua execução, onde a execução corresponde a uma série de pontos com execução contínua.

Com base nestas classificações, nas próximas seções são apresentadas as teorias de escalonamento com prioridade fixa e dinâmica para sistemas monoprocessados, escalonamento para sistemas multiprocessados, escalonamento cooperativo e escalonamento não preemptivo.

4.1.1 Escalonamento com prioridade fixa para um processador

Um dos trabalhos pioneiros sobre escalonamento de tempo real com prioridade fixa foi elaborado por Liu e Layland em 1973 (SHA et al., 2004). Neste trabalho, as seguintes premissas são assumidas:

1. todas as tarefas são periódicas;
2. todas as tarefas estão prontas no início do período e o *deadline* é igual ao período;
3. todas as tarefas são independentes;
4. limite máximo do tempo de computação fixo e menor que o período;
5. não há auto suspensão de tarefas;
6. sistema preemptivo;
7. custos adicionais de sistema operacional, *cache*, etc (*overheads*) são ignorados;
8. sistema com somente um processador.

Baseando-se nestas premissas, constrói-se um modelo periódico de tarefas onde para cada tarefa τ_i tem-se um período (T_i), um *deadline* (D_i), o pior tempo de computação (C_i) e uma prioridade. O pior tempo de computação é o WCET – *Worst Case Execution Time* que teoricamente é calculado pelas ferramentas de WCET como apresentado no capítulo 3.

A formalização deste modelo possibilitou o desenvolvimento da "Teoria do Instante Crítico"(SHA et al., 2004). O instante crítico para uma tarefa é o tempo na qual ela está pronta para execução e seu tempo de resposta é o mais longo possível. Para o modelo de tarefas descrito anteriormente, o instante crítico acontece quando uma tarefa está pronta em conjunto com todas as tarefas de prioridade mais alta, ou seja, todas as tarefas do sistema estão prontas simultaneamente.

Baseado na teoria do instante crítico, Liu e Layland construíram e provaram um teste de escalonabilidade suficiente baseado em utilização. Um

teste baseado em utilização considera o somatório de utilização das tarefas, onde a utilização é dada pela razão do tempo de computação pelo período ($u_i = \frac{C_i}{T_i}$). Este teste de Liu e Layland (equação 4.1) é válido quando as prioridades são distribuídas em taxa monotônica (tarefas de menor período possuem prioridade mais alta).

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{2}} - 1) \tag{4.1}$$

Se n tender ao infinito, o escalonamento em taxa monotônica chega ao limite de utilização máximo de $69,3\%$, considerando os piores casos. Quando os períodos das tarefas são harmônicos entre si (período de cada tarefa é um múltiplo inteiro exato do menor período das tarefas), a taxa de utilização deste escalonamento pode alcançar 100%.

O teste de utilização é bastante simples mas possui algumas limitações: é somente suficiente (pessimista), limita o modelo para *deadlines* implícitos ($D_i = T_i$) e as prioridades devem ser somente atribuídas à taxa monotônica.

Conforme Sha et al. (2004), em 1987 Lehoczky, Sha e Ding propuseram a equação da demanda de tempo, conforme a equação 4.2. Esta análise considera que se todas as tarefas estão prontas no mesmo instante, a tarefa de mais alta prioridade irá cumprir seu prazo se existir um tempo entre $0 < t \leq T_i$ que a demanda no processador ($W_i(t)$) é menor do que t.

$$W_i(t) = \sum_{j=1}^{i} \left\lceil \frac{t}{T_i} \right\rceil C_j \leq t \tag{4.2}$$

Em paralelo à da demanda de tempo, foi desenvolvido o teste de tempo de resposta proposto Audsley et al. (1991). Este teste calcula o pior tempo de resposta de uma tarefa iterativamente, conforme a equação 4.3.

$$R_i^{k+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j \tag{4.3}$$

Considerando o modelo de tarefas proposto por Liu e Layland em 1973, o teste da equação 4.3 é exato. Se $\forall \tau_i : (1 \leq i \leq n)\ R_i$ convergir para um valor menor ou igual que D_i o sistema é escalonável. Como as tarefas estão ordenas por prioridade não decrescente, j representa as tarefas com prioridade mais alta que i.

O teste da equação 4.3 é independente da política de prioridades. No caso de sistemas onde o *deadline* é menor que o período ($D_i < T_i$), o teste também pode ser utilizado. Quando $D_i < T_i$ a ordenação ótima de prioridades é dada pelo *deadline* onde as tarefas com *deadline* menor possuem prioridade mais alta formando a política conhecida como *Deadline Monotonic*.

O teste de tempo de resposta é exato e mais flexível. Pode-se facilmente incorporar outras interferências como bloqueios por recursos (B_i) ou efeitos como variações de início de instâncias das tarefas (*release jitter*) (j_i) conforme a equação 4.4. A incorporação de outros efeitos no teste pode ser consultado em (BUTTAZZO, 2005). Pode-se também flexibilizar o modelo de tarefas e os efeitos sobre o teste podem ser consultados em (SHA et al., 2004) e (LIU, 2000). No caso de sistemas com tarefas esporádicas, o teste é o mesmo pois tarefas esporádicas possuem um período mínimo entre chegadas.

$$R_i^{k+1} = C_i + B_i + j_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k + j_j}{T_j} \right\rceil C_j \tag{4.4}$$

Os algoritmos que utilizam atribuição de prioridade com taxa monotônica ao período (*Rate monotonic*) e ao *deadline* (*Deadline monotonic*) são considerados ótimos dentro da sua classe para sistema com um único processador.

4.1.2 Escalonamento com prioridade dinâmica para um processador

No escalonamento com prioridade dinâmica, a prioridade das tarefas é atribuída às instâncias (*jobs*) ao contrário do esquema de prioridade fixa por tarefa (SHA et al., 2004). Neste sentido a prioridade da tarefa muda durante a execução conforme quais instâncias estão ativas. Um dos algoritmos mais utilizados desta classe é o EDF – *Earliest Deadline First*. Nesta política, a prioridade das tarefas é inversamente proporcional ao *deadline* absoluto das instâncias ativas, ou seja, quanto mais próximo do *deadline*, mais alta a prioridade da tarefa.

A análise inicial de escalonabilidade foi também proposta por Liu e Layland em 1973 com as mesmas premissas do escalonamento com prioridade fixa:

1. todas as tarefas são periódicas;

2. todas as tarefas estão prontas no início do período e o *deadline* é igual ao período;

3. todas as tarefas são independentes;
4. limite máximo do tempo de computação fixo e menor que o período;
5. não há auto suspensão de tarefas;
6. sistema preemptivo;
7. custos adicionais de sistema operacional, *cache*, etc (*overheads*) são ignorados;
8. sistema com somente um processador.

Nestas condições, um conjunto de n tarefas escalonadas por EDF é escalonável se e somente se a utilização total das tarefas for menor que 1, conforme a equação 4.5.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{4.5}$$

O EDF é ótimo em relação a todos os algoritmos preemptivos para um processador, ou seja, se há um conjunto de tarefas viável pelo regime preemptivo escalonado por qualquer algoritmo, este conjunto também será escalonado pelo EDF (LIU, 2000). Em termos de algoritmos ótimos, há também o LLF – *Least Laxity First* que escalona a tarefa com menor "folga" em relação ao seu *deadline*. O LLF possui alto custo de processamento (*overhead*) na sua execução devido o constante cálculo de folga das tarefas e alta quantidade de mudanças de contextos.

Quando o modelo de tarefas permite *deadlines* menores que o período, a equação 4.5 torna-se apenas um teste necessário. Portanto, para melhorar a precisão do teste, é possível utilizar o método de "demanda de processador" proposto por Baruah et al. (1990). A demanda de processamento em um intervalo $[t_1, t_2]$ é a quantidade tempo de processador $g(t_1, t_2)$ necessário para executar as instâncias ativas em $[t_1, t_2]$ e também que devem ser completadas neste intervalo. Dentro destas condições, o sistema é viável se, e somente se, a demanda de processamento for menor que o tempo disponível, conforme a equação 4.6.

$$\forall t_1, t_2 \;\; g(t_1, t_2) \leq (t_2 - t_1) \tag{4.6}$$

O trabalho de Baruah et tal mostrou que um conjunto de tarefas periódicas com *deadline* menor que o período e simultaneamente ativadas é

escalonável por EDF se e somente se $U < 1$ e a inequação da equação 4.7 for verdadeira dentro de um intervalo de tempo X.

$$\forall X > 0, \sum_{i=1}^{n} \left\lfloor \frac{X + T_i - D_i}{T_i} \right\rfloor C_i \leq X \tag{4.7}$$

Apesar do EDF ter um limite de escalonabilidade de pior caso mais elevado que as técnicas de prioridade fixa, a flexibilização do modelo de tarefas e incorporação de tempos de bloqueio é mais problemática. Além disso, os sistemas de prioridade fixa são mais deterministas no ponto de vista de preempção das tarefas. Outro ponto negativo do EDF está no fato de que se uma tarefa perde o *deadline* todo o sistema é afetado (se não houver mecanismos de proteção específicos), diferentemente do esquema de prioridade fixa, onde somente as tarefas de prioridade mais baixa que a tarefa problemática sofrem com o tempo de resposta estendido.

4.1.3 Escalonamento para multiprocessadores

Devido aos problemas de consumo energético e dissipação de calor, os fabricantes de processadores investiram a capacidade de miniaturização em processadores com vários núcleos ao invés do aumento da frequência (DAVIS; BURNS, 2009b). Dado que certas aplicações de tempo real possuem alta demanda de processamento, a utilização destes processadores é quase certa. Além disso a utilização de processadores com vários núcleos pode, por exemplo, reduzir a quantidade de unidades eletrônicas de controle em aplicações aviônicas e automotivas.

A teoria de escalonamento de tempo real para sistemas monoprocessados é razoavelmente madura havendo vários livros publicados como Buttazzo (2005), Farines et al. (2000) e Liu (2000). No caso de sistemas multiprocessados, poucos dos resultados da teoria de escalonamento monoprocessado são generalizados para multiprocessadores. Conforme apontado por Davis e Burns (2009b), o simples fato de uma tarefa utilizar somente um processador enquanto há vários disponíveis no mesmo instante complica enormemente o escalonamento.

Os sistemas multiprocessados geralmente são classificados em heterogêneos, homogêneos ou uniformes. Nos heterogêneos, os processadores são diferentes e as tarefas não podem ser atribuídas a qualquer processador. Quando os processadores são idênticos bem como a taxa de execução das tarefas é a mesma em todos os núcleos, o sistema é considerado homogêneo. Os uniformes são sistemas onde a taxa de execução das tarefas depende da

velocidade do processador: um processador com velocidade 2 executa com o dobro de velocidade do que um de velocidade 1.

As mesmas definições do modelo de tarefas dos sistemas monoprocessados também são utilizadas em multiprocessadores contemplando tarefas periódicas, esporádicas e aperiódicas. Os *deadline* são classificados igualmente em implícitos, limitados e arbitrários. Há também as mesmas classificações sobre preempção: sistema preemptivo, não preemptivo ou cooperativo.

No escalonamento para multiprocessadores, os processadores não ficam ociosos quando há tarefas prontas para execução, formando a propriedade conhecida como *work-conserving*.

Conforme Davis e Burns (2009b), o escalonamento para multiprocessador tenta resolver dois problemas: alocação e prioridade. A alocação define em qual processador a tarefa deve executar e a prioridade define quando e em que ordem as tarefas devem executar. No caso da alocação, pode-se considerar:

- sem migração: cada tarefa é alocada a um processador;
- migração por tarefa (*task-level*): tarefas podem executar em diferentes processadores, mas suas instâncias não migram durante a execução;
- migração por instâncias (*job-level*): instâncias podem migrar, mas sem execução paralela.

Os algoritmos sem migração são conhecidos como particionados enquanto os com migração são conhecidos como globais (migração por instâncias). A classificação quanto as prioridades é idêntica ao sistema monoprocessado:

- prioridade fixa de tarefas: prioridade atribuída a uma tarefa é igual para todas as suas instâncias;
- prioridade fixa por instância (dinâmica em relação à tarefa): as instâncias de uma tarefa podem ter prioridades diferentes, mas cada uma tem uma prioridade estática (algoritmo EDF – *Earliest Deadline First*);
- prioridade totalmente dinâmica: uma instância pode ter prioridade diferente em tempos diferentes (algoritmo LLF – *Least Laxity First*).

A combinação dos parâmetros de alocação e prioridade forma uma ampla gama de classes de algoritmos conforme a tabela 5. A classe $(3,3)$ é a mais geral mas que incorpora mais *overheads* enquanto a classe $(1,1)$, o problema é reduzido ao escalonamento para um processador possuindo menos

overheads de migração e chaveamentos de contexto. A principal desvantagem dos algoritmos particionados é o particionamento que entra na classe de algoritmos de *bin-packing* sendo estes NP-difícil (DAVIS; BURNS, 2009b).

prioridade / migração	**1: estática**	**2: dinâm. por instância**	**3: total. dinâmica**
1: particionado	(1,1)	(1,2)	(1,3)
2: por tarefa	(2,1)	(2,2)	(2,3)
3: total	(3,1)	(3,2)	(3,3)

Tabela 5: Classificação dos algoritmos de escalonamento (CARPENTER et al., 2004).

Conforme apresentado por Davis e Burns (2009b) estas classes geralmente são comparadas pelas seguintes métricas considerando apenas a escalonabilidade:

- limites de utilização (*Utilizations Bounds*): o pior caso de limite de utilização (U_A) de um algoritmo "A" é definido como a mínima utilização de qualquer conjunto de tarefas com *deadlines* implícitos ($D_i = T_i$) que é apenas escalonável por tal algoritmo;

- taxa de aproximação (*Approximation Ratio*): comparação do desempenho de um algoritmo "A" com um ótimo (relação de qualquer métrica de desempenho de "A" pelo desempenho do ótimo);

- acréscimo de recursos (*Resource Augmentation*): método alternativo de comparação de um algoritmo "A" com um ótimo. Esta métrica ao invés de considerar o aumento de processadores, considera o aumento de velocidade dos processadores que seria necessário (aumento linear);

- medições empíricas: porcentagem de conjuntos de tarefas que são escalonáveis como os trabalhos de (BRANDENBURG et al., 2008), (CALANDRINO et al., 2006) e (BASTONI, 2010). Em comparações empíricas é importante utilizar um algoritmo de geração de tarefas imparcial (*unbiased*).

Nas investigações empíricas há geração de conjuntos de tarefas aleatórias que são classificados em escalonáveis e não escalonáveis. Neste tipo de estudo, parâmetros como o número de tarefas, número de processadores, utilização, períodos, distribuição dos *deadlines* e das utilizações são variados para medir o desempenho dos algoritmos e dos testes em diferentes cenários. Procura-se relacionar o desempenho teórico em contextos práticos quando

ocorre custos relacionados a decisões de escalonamento, mudanças de contexto e migrações.

4.1.3.1 Resultados fundamentais independentes de algoritmos

Em termos de algoritmos ótimos, em 1974 Horn elaborou um algoritmo ($O(n^3)$) que é capaz de determinar uma escala ótima para qualquer conjunto arbitrário de n tarefas com instâncias com tempos de chegada e execução conhecidos a priori. Em 2007 Fisher mostra que não há algoritmo *online* ótimo para conjuntos de tarefas esporádicas com *deadlines* arbitrários. Algoritmos ótimos são conhecidos para tarefas periódicas com *deadlines* implícitos.

Nem todas as classes de algoritmos formadas pela combinação dos parâmetros de alocação e prioridade são comparáveis. O escalonamento global com prioridade totalmente dinâmica domina todas as outras classes. As três classes com prioridade fixa (particionada, migração por tarefa e migração por instâncias) são incomparáveis. As três classes particionadas (com prioridade fixa por tarefa, por instâncias ou prioridade totalmente dinâmica) também são incomparáveis com respeito as classes com migração de tarefas (CARPENTER et al., 2004). Diferentemente do escalonamento monoprocessado onde existem algoritmos ótimos para tarefas periódicas e esporádicas com prioridade fixa (instâncias ou tarefas), nos multiprocessadores a prioridade totalmente dinâmica é essencial para um algoritmo ser ótimo (DAVIS; BURNS, 2009b). No caso do escalonamento global (com migração, preempção e com prioridade totalmente dinâmica) o limite de utilização dos processadores é m, onde m é o número de processadores. Todas as outras classes são limitadas, no pior caso, pelo limite de utilização da equação 4.8. Isto significa, que considerando apenas o escalonamento, todas as classes de algoritmos da tabela 5, com exceção do global, possuem utilização máxima de 50% dos processadores.

$$U = \frac{m+1}{2} \tag{4.8}$$

Davis e Burns (2009b) também apresentam o conceito de previsibilidade (*predictability*): um algoritmo é previsível se o tempo de resposta das tarefas não aumenta com a diminuição do tempo de execução, mantendo-se todos os outros parâmetros constantes. É provado que os algoritmos com prioridade fixa por tarefa e por instâncias são previsíveis enquanto os dinâmicos é necessário considerar e provar a previsibilidade para serem considerados

úteis (DAVIS; BURNS, 2009b) . A previsibilidade é importante por que os limites de utilização e tempos de resposta calculados consideram sempre o pior tempo de execução o que pode não acontecer durante a execução do sistema.

A sustentabilidade de um algoritmo com respeito à um modelo de tarefas acontece se, e somente se, dado um conjunto compatível com o modelo implica na escalonabilidade do mesmo conjunto quando este é modificado por: diminuição do tempo de execução, aumento do períodos de chegada ou aumento dos *deadlines* (DAVIS; BURNS, 2009b). A sustentabilidade está ligada no fato de que, se houver melhora das condições de escalonabilidade (diminuição da utilização, etc), o conjunto de tarefas continua escalonável pelo mesmo algoritmo e é importante no escalonamento multiprocessado devido a presença de anomalias.

Uma anomalia acontece quando uma modificação nos parâmetros do conjunto de tarefas causa um efeito contra intuitivo. Por exemplo, diminuição da utilização das tarefas individualmente reduz a utilização global do sistema. Anomalias são presentes nos algoritmos de alocação de tarefas no esquema particionado devido a alocação. Quando um parâmetro (tempo de execução ou período) varia, pode gerar outra alocação que pode tornar o conjunto não escalonável. O escalonamento global também está sujeito a anomalias (DAVIS; BURNS, 2009b).

Quanto ao instante crítico, no escalonamento global com prioridade fixa, uma tarefa não necessariamente tem o seu pior tempo de resposta quando todas as tarefas chegam ao mesmo momento. Em sistemas multiprocessados, o tempo de resposta é pior para uma tarefa de menor prioridade quando ela executa e nenhuma ou apenas algumas tarefas de alta prioridade executam em outros processadores, mas quando as outras tarefas de prioridade mais alta ficam prontas, todas elas monopolizam os processadores e a execução da tarefa em questão é postergada. O instante crítico do escalonamento global e do particionado é diferente. No esquema particionado, todas as tarefas prontas de maneira síncrona continua sendo o instante crítico. O instante crítico é um problema na análise de escalonamento global com prioridade fixa por instância pois não foram identificados conjuntos de chegada de pior caso para o escalonamento global de tarefas esporádicas (DAVIS; BURNS, 2009b).

4.1.3.2 Escalonamento particionado

O escalonamento particionado tem algumas vantagens em relação ao global. Uma delas está no fato de, se uma tarefa excede o seu tempo de computação, somente as tarefas de um processador são afetadas. Não há penalidade relacionada com a migração de tarefas devido a faltas adicionais de

cache, nem de protocolos de coerências e não há interferências inter-*cache* das tarefas. Além destas, o esquema particionado utiliza uma fila de tarefas para cada processador, diferentemente do global que incorpora *overheads* consideráveis de manipulação de filas em sistemas grandes.

Conforme apontado por Davis e Burns (2009b), na prática a vantagem principal de se utilizar o escalonamento particionado está no fato que depois de alocadas as tarefas aos processadores, todas as técnicas e análises do escalonamento para um processador podem ser utilizadas. Os algoritmos ótimos como *Rate monotonic*, *Deadline Monotonic* e EDF têm grande influência nas pesquisas dos esquemas particionados.

A principal desvantagem da abordagem particionada está na alocação das tarefas devido ao problema ser análogo ao *bin-packing*, ou seja, NP-difícil. Para o particionamento geralmente se utiliza heurísticas polinomiais como *First-Fit*, *Next-Fit*, *Best-Fit*, *Worst-Fit* e ordenação de tarefas por utilização decremental (*Decreasing Utilization*). No trabalho de (DAVIS; BURNS, 2009b) são apresentadas as comparações das técnicas de particionamento nos vários algoritmos para um processador em termos taxa de aproximação e limites de utilização. Lembrando que qualquer algoritmo que não seja global (classe (3,3)) dentro daqueles apresentados na tabela 5, a utilização máxima no pior caso é dada por $\frac{m+1}{2}$, com m igual ao número de processadores.

4.1.3.3 Escalonamento global

Em relação ao particionado, o escalonamento global tem algumas vantagens. Conforme Davis e Burns (2009b), há menos mudanças de contexto pois só há preempção quando não há processador ocioso e a capacidade sobressalente em algum processador pode ser utilizada por outras tarefas. Se ocorrer tempo de execução maior do que o previsto, a probabilidade de uma falta geral do sistema é menor. Geralmente o escalonamento global é mais apropriado para sistemas abertos, onde não há necessidade de rodar algoritmos de alocação ou balanceamento de carga quando o conjunto de tarefas muda.

No trabalho de (DAVIS; BURNS, 2009b) são apresentados uma ampla gama de algoritmos de escalonamento global bem como comparações entre eles. Nos trabalhos de Brandenburg et al. (2008), Calandrino et al. (2006) e Bastoni (2010) são apresentadas investigações empíricas dos algoritmos globais e penalidade associadas a migração e memória *cache*.

Uma das grandes questões em aberto é que a análise de escalonabilidade considera apenas limites de utilização e propriedades dos algoritmos e

é ignorado completamente os *overheads* de manipulação de filas, migração e memória *cache*. Como é apresentado nos capítulos 3 e 4 sobre o cálculo do tempo de computação das tarefas, este cálculo considera apenas execução sem preempção e ignora completamente interferências entre tarefas, bem como migração entre núcleos. Estes efeitos podem ter custos temporais pequenos, mas se não forem incorporados à análise, a segurança dos sistemas críticos não é garantida.

4.1.3.4 Escalonamento híbrido

No escalonamento global há o problema das penalidades extras associadas à migração, memória *cache* e manipulação de filas. O escalonamento particionado, apesar de ter inúmeras vantagens sofre com o limite de utilização de 50% e com a alocação das tarefas. O escalonamento híbrido ou semiparticionado tenta compensar estas duas limitações permitindo que algumas tarefas migrem, aumentando os limites de escalonabilidade.

Dificuldades e abordagens sobre o escalonamento semiparticionado podem ser encontrados nos trabalhos de Bastoni et al. (2011) e Kato e Yamasaki (2009). Normalmente o foco dos trabalhos sobre escalonamento particionado é identificar quando, como e quais tarefas vão migrar.

4.1.4 Escalonamento cooperativo (pontos fixos de preempção) para um processador

Conforme apresentado na seções anteriores, as teorias de escalonamento para um processador e para multiprocessadores geralmente ignoram completamente as penalidades intrínsecas do escalonador, mudanças de contextos e como calcular o tempo de execução das tarefas. Obviamente que quando estes custos são ignorados, o modelo preemptivo é mais eficiente em termos de utilização e permite melhor escalonabilidade.

No capítulo 2 são apresentadas as características dos processadores modernos bem como o esclarecimento que tais processadores são dimensionados para alto desempenho de caso médio, mas determinismo não é considerado. Devido a essa característica a migração e preempção descontrolada torna a análise com garantia extremamente pessimista ou até mesmo inviável na prática.

Por outro lado, um modelo não preemptivo é muito restrito para algumas aplicações dificultando a escalonabilidade das tarefas. Para contornar estas dificuldades, uma das propostas da literatura é evitar preempções ar-

bitrárias e limitar o comprimento das regiões não preemptáveis com pontos fixos de preempção. De acordo com este modelo, cada tarefa é divida em regiões não preemtáveis inserindo pontos predefinidos de preempção no código da tarefa. Quando uma tarefa de alta prioridade fica pronta entre os pontos não preemptáveis, a preempção é adiada até o próximo ponto.

O escalonamento de prioridade fixa com pontos fixos de preempção foi proposto por Burns (1994), que aplicou o teste de tempo de resposta (equação 4.3) para verificar a escalonabilidade. Este trabalho inicial considerou sistema com um processador, tarefas com prioridade fixa e *deadlines* limitados, mas não foi considerado como calcular o tempo máximo destas regiões. O trabalho recente de Yao et al. (2009) trata deste problema e apresenta um método para o escalonamento com prioridade fixa que calcula o tempo máximo das regiões não preemptáveis das tarefas sem afetar o escalonamento com relação ao modelo totalmente preemptivo. No trabalho de Bril et al. (2007) foi identificado uma situação em que, dada a execução da última porção não preemptável da tarefa τ_i, esta pode postergar a execução das outras tarefas de mais alta prioridade que podem afetar o tempo de resposta da próxima instância de τ_i. Nestas condições τ_i afeta seu próprio tempo de resposta e este efeito é conhecido como *self-pushing phenomenon*. Este fenômeno não é considerado no teste de tempo de resposta, onde se considera apenas o tempo de resposta de uma instância por tarefa. Bril et al. (2007) identificou o problema bem como propôs um teste de escalonabilidade que lida com este fenômeno.

Contudo, conforme o trabalho de Yao et al. (2010), se o modelo de tarefas contemplar *deadlines* limitados ($D_i \leq T_i$) e o conjunto de tarefas for escalonável pelo modelo preemptivo com prioridade fixa monoprocessado, o teste de escalonabilidade para escalonamento cooperativo pode-se limitar à primeira instância de cada tarefa. Assim, Yao et al. (2010) prova formalmente estas considerações e apresenta um algoritmo para o teste de escalonabilidade. Em termos gerais é provado que o *self-pushing phenomenon* não acontece quando os *deadlines* são limitados e o conjunto de tarefas é viável no regime preemptivo sob o sistema de prioridade fixa monoprocessado.

4.1.5 Escalonamento não preemptivo para um processador

O escalonamento não preemptivo recebeu menos atenção do que os preemptivos por ter menos sucesso na escalonabilidade dos sistemas. Contudo, para sistemas com limitações severas de recursos, o escalonamento preemptivo pode não ser viável. Uma alternativa não preemptiva é justificada por (SHORT, 2010):

- o escalonamento não preemptivo é mais fácil de implementar e possui menor custo de execução;
- o escalonamento não preemptivo garante naturalmente acesso exclusivo à recursos compartilhados;
- sistemas preemptivos necessitam pilhas individuais para as tarefas enquanto os não preemptivos podem compartilhar uma única pilha economizando memória;
- os valores do tempo de computação (WCET) podem ser utilizados diretamente na análise de escalonabilidade do sistema.

Apesar destas vantagens, o escalonamento não preemptivo incorpora alguns problemas como tempos de resposta geralmente maiores. Há um bloqueio inerente das tarefas pela inversão de prioridade (tarefa de prioridade mais baixa bloqueia de mais alta) e o problema torna-se NP-completo devido a característica combinatória.

O equacionamento para o teste de viabilidade com algoritmos de prioridade fixa não preemptivos podem ser consultado em George et al. (1996). Para este caso, os conjuntos de tarefas escalonáveis são muito restritivos e portanto, neste trabalho será apenas considerado o de prioridade dinâmica EDFnp – *Earliest Deadline First* não preemptivo. No EDFnp as tarefas com *deadline* mais próximas são escolhidas para execução, mas somente quando a instância atualmente em execução terminar sua execução.

A viabilidade do conjunto de tarefas sob o regime não preemptivo com *deadlines* implícitos ($D_i = T_i$) é verificada pelas condições impostas pelas equações 4.9 e 4.10 propostas por Jeffay et al. (1991). Neste teste, seja $\mathcal{T}_{np} = \{\tau_1, \tau_2, ..., \tau_n\}$ um conjunto de tarefas esporádicas em ordem não decrescente em relação ao período (para um par τ_i e τ_j, se $i < j$ então $T_i \geq T_j$). Se as condições das equações 4.9 e 4.10 forem verdadeiras, $\mathcal{T}_{np}$ é escalonável. Detalhes sobre a utilização prática das equações 4.9 e 4.10 podem ser consultados em (JEFFAY et al., 1991)

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{4.9}$$

$$\begin{gathered} \forall i, 1 < i \leq n; \forall t, T_1 < t < T_i \\ t \geq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t-1}{T_j} \right\rfloor C_j \end{gathered} \tag{4.10}$$

Para o caso de *deadlines* limitados ($D_i \leq T_i$), a viabilidade do conjunto de tarefas sob o regime não preemptivo é verificada pelas condições impostas pela equação 4.11 propostas por George et al. (1996). Neste teste, seja $\mathcal{T}_{np} = \{\tau_1, \tau_2, ..., \tau_n\}$ um conjunto de tarefas esporádicas com $U \leq 1$ é escalonável por EDFnp se e somente se:

$$\forall t, t \geq h(t) + max_{D_i > t}\{C_i - 1\} \tag{4.11}$$

Com $max_{D_i>t}\{C_i - 1\} = 0$ se $\nexists t : D_i > t$ e $h(t)$ é a demanda de processador no tempo t definida pela equação 4.12.

$$h(t) = \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right) C_i \tag{4.12}$$

4.2 CONSIDERAÇÕES SOBRE WCET E ARQUITETURA DE PROCESSADORES

Conforme Cullmann et al. (2010), a experiência com o uso de análise temporal estática para sistemas críticos automotivos e aviônicos tem se mostrado positiva ao longo do tempo. Contudo, tanto a precisão dos resultados e a eficiência dos métodos de análise são altamente dependentes da previsibilidade da plataforma na qual o sistema será executado. O tipo de arquitetura determina se a análise estática é possível na prática e também se os resultados mais precisos são precisos suficientemente.

A arquitetura dos processadores geralmente é projetada para melhorar o caso médio de execução e a previsibilidade destas características não é um elemento considerado. Normalmente projeta-se o processador para "fazer o caso comum rápido e o não comum correto" (SCHOEBERL, 2009b). O problema já é difícil para sistemas monoprocessados e o nível de severidade do problema aumentou ainda mais na utilização de multiprocessadores em sistemas críticos com garantia.

Em relação à microarquitetura, o analisador estático deve contar com um modelo abstrato do *hardware* considerando elementos como por exemplo o *pipeline* e *caches*. Em uma arquitetura com *pipeline* as instruções devem ser consideradas coletivamente para obtenção do tempo de execução. Quando mais partes do *pipeline* influenciam no tempo de execução das instruções, mais o modelo abstrato deve implementar o *hardware* concreto. Quanto maior o tamanho dos *buffers* (filas de pré busca por exemplo) ainda há influência de eventos passados no tempo de execução da instrução corrente, ou seja,

o tempo de execução é dependente do histórico de execução. Um longo histórico de execução leva a uma explosão de estados para o cálculo do WCET e geralmente ocorrem simplificações que levam a estimativas conservadoras.

Os estados abstratos do modelo podem não ter informação do estado de alguns dos elementos do processador devido a alguma simplificação. As transições no *pipeline* podem depender destas informações faltantes causando com que o *pipeline* do modelo seja não determinista ao contrário do *pipeline* real que é determinista. Nessas situações pode-se considerar que o caminho de maior tempo deve ser tomado, contudo, dado a presença de anomalias temporais esta consideração não é segura e todos os possíveis estados sucessores devem ser analisados pela ferramenta de WCET.

Uma anomalia temporal, no contexto de WCET, é uma situação onde o pior caso local não contribui para o pior caso global (CULLMANN et al., 2010). Como por exemplo, uma falta da *cache* local pode resultar em um tempo de execução global menor como ilustrado na figura 14. Neste exemplo, na linha de execução "acerto" há um acerto na *cache* para instrução "A". A próxima instrução depende da avaliação de uma condição de salto (*branch*) que ainda não é determinada no final da execução de "A" e por isso, a busca da instrução "B" é iniciada ocasionando uma falta na *cache*. No tempo de busca à memória a condição de salto é resolvida e percebe-se que "C" é o fluxo correto de execução. Considerando agora a linha de execução "falta", há uma falta na *cache* na execução de "A" que permite que a condição de salto seja resolvida antes do término de "A" e portanto "C" é executado. Neste contexto, o tempo de execução do conjunto de instruções "A" e "C" é menor quando ocorre uma falta na *cache* do que se houvesse um acerto, o que é contra intuitivo.

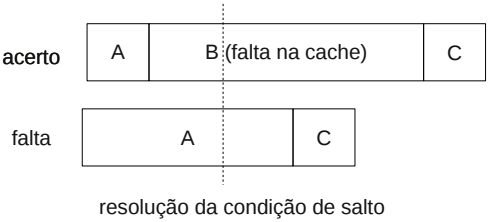


Figura 14: Exemplo de anomalia temporal.

Além das anomalias temporais há sistemas que apresentam um efeito conhecido como efeito dominó. Conforme (CULLMANN et al., 2010), um sistema exibe efeito dominó se há dois estados de *hardware* s e t nos quais a diferença do tempo de execução (sobre o mesmo ponto do programa pas-

sando pelos estados s e t) pode ser arbitrariamente alto, ou seja, o tempo dos diferentes estados não é limitado por uma constante. Por exemplo, dado um laço do programa, as execuções nunca convergem para o mesmo estado de *hardware* e a diferença nos tempos de execução aumenta a cada iteração. Se não há presença deste efeito, há formação de invariantes nos blocos básicos do laço: se este repete 50 vezes e o pior tempo de execução do corpo é b e as outras iterações executam 2 vezes mais rápido, pode-se aproximar o tempo de execução global do laço como $49 \times 0.5 \times b$. Assim, quando há efeito dominó, não é possível descartar estados durante a análise limitando-os por uma constante e todos os estados e iterações devem ser considerados para obter-se o pior caso. O efeito dominó aparece em sistemas concretos como por exemplo no *pipeline* do processador PowerPc 755 e políticas de despejo de *cache* como *Pseudo Least Recent Used* (PLRU) (CULLMANN et al., 2010), *Firs-in-First-out* (FIFO) e *Pseudo Round-Robin* (HECKMANN et al., 2003).

Conforme a presença das anomalias temporais e o efeito dominó as arquiteturas podem ser classificadas em:

- arquitetura temporalmente composicional (*fully timing compositional arquitecture*): O modelo abstrato da arquitetura não apresenta anomalias temporais e a análise pode seguramente considerar somente os caminhos de pior caso. Quando há um problema no fluxo de execução, todos os elementos do *pipeline* são parados até que se resolva o problema. Desta forma as diferentes análises (*cache*, barramento, etc) podem ser realizadas separadamente e simplesmente os tempos de penalidade podem ser somados ao pior caso de execução. Um exemplo dessa classe são os processadores baseados no núcleo ARM7.

- arquitetura composicional com efeitos limitados (*compositional arquitecture with constant-bounded effects*): exibem anomalias temporais mas não há efeito dominó. Geralmente a análise deve considerar todos os caminhos, mas para haver um compromisso entre precisão e eficiência, é possível descartar caminhos de melhor caso locais adicionando número constante de ciclos. O processador *Infineon TriCore* é assumido como pertencente à essa classe mas não é formalmente provado (CULLMANN et al., 2010).

- arquitetura não composicional: essa arquitetura apresenta efeito dominó e anomalias temporais como por exemplo o processador PowerPC 755. Nesse tipo de sistema o analisador deve considerar todos os caminhos locais pois um efeito local pode influenciar o tempo de execução futuro arbitrariamente.

Em geral, *pipelines* simples são fáceis de modelar na análise de WCET.

Se não há formação de flutuações, a latência de execução é igual ao comprimento do *pipeline* e a execução efetiva das instruções é de apenas um ciclo. Contudo, a formação de flutuações é natural sendo induzida pelas dependências de dados e controle entre as instruções. Os *pipelines* complexos com dependências podem fazer com que instruções antigas (com tempo maior que o comprimento do próprio *pipeline*) influenciem na execução da instrução atual. Conforme visto no capítulo 3, esse tipo de *pipeline* é analisado com interpretação abstrata e o tempo de análise pode tornar-se impraticável. Em Schoeberl (2009b) é citado que são necessários cerca de 1000 estados por instrução para modelar o processador PowerPC 755. Este processador foi introduzido em 1998, e portanto, para os processadores mais modernos a situação pode ser ainda mais crítica.

A busca de instruções (*fetch*) é normalmente desacoplada da memória pela unidade de pré-busca (*prefetch*). Esta unidade carrega a fila de instruções independente do *pipeline* principal. A pré-busca é ainda mais importante para arquitetura com comprimento variável de instruções (como x86), onde o estado de carga da fila depende do histórico do fluxo de instruções. O possível comprimento do histórico pode ser não limitado e neste caso assume-se um tamanho máximo e considera-se a fila vazia em tal ponto de corte. Em termos de complexidade, em Schoeberl (2009b) é citado que até para modelo de um processador Intel 80188 com fila de tamanho de 4 bytes teve de ser simplificado para lidar com a complexidade resultante das interações entre a execução e pré-carga de instruções.

As unidades de previsão de salto do fluxo de controle (*branch predictors*) são extremamente importantes para manter o fluxo contínuo de instruções em *pipelines* longos. Um erro da previsão acarreta em uma penalidade relacionada diretamente com o comprimento do *pipeline*. As unidades previsoras modernas baseiam-se no histórico dos caminhos anteriores que acarreta em uma explosão de estados na análise estática. O problema dos previsores é que a partir do momento que um único índice da tabela do histórico é desconhecido, toda a informação de previsor é perdida na análise daquele ponto de execução. Além disso, os previsores avançados de dois níveis (como Pentium III e Pentium 4) ainda contam com anomalias temporais, onde a diminuição do número de iterações de um laço pode aumentar o tempo de execução. Os previsores também interferem nos conteúdos da *cache*. Quando o analisador não pode antecipar o valor do previsor, ambas as direções devem ser consideras na análise da *cache*.

O aumento do nível de paralelismo das instruções (ILP – *Instruction Level Paralelism*) é uma característica que gera maior desempenho. Esta característica pode ser realizada estaticamente pelos processadores VLIW (*Very Long Instruction Word*), arquitetura Itanium por exemplo, ou por técnicas que

formam os processadores super escalares. Para os processadores VLIW o escalonamento das instruções é construído pelo compilador sendo uma vantagem para o analisador WCET que necessita desta informação. Porém, para cada geração de processador VLIW é necessário um novo compilador e novas otimizações. Nos super escalares, o nível de paralelismo das instruções é aumentado por combinação de várias unidades de execução paralela com execução fora de ordem (para o Pentium 4, pode haver 128 instruções em execução no *pipeline*). A análise estática de um programa com um modelo preciso do processador superescalar pode ser impossível na prática devido a explosão de estados decorrentes da complexidade do *pipeline*.

O paralelismo por instrução geralmente é limitado a duas instruções por ciclo. Assim, alguns processadores implementam o paralelismo por *thread* (TLP – *Thread Level Paralelism*) formando, por exemplo, a tecnologia *HyperThreading*. O problema do paralelismo por *thread* para sistemas de tempo real está no fato que o tempo de execução de uma *thread* depende do estado de outra. Estas interações estão ligadas principalmente à *cache* e memória compartilhada. No pior caso, toda a melhoria de desempenho dos sistemas *HyperThreading* deve ser ignorada e o tempo de execução das tarefas é considerado como execuções seriais. Como as *threads* compartilham a *cache* de nível primário, uma *thread* invalida o estado abstrato da *cache* de outra. Se combinado o ILP e o TLP forma-se o sistema SMT – *Simultaneous Multithreading*. Esta técnica permite que várias *threads* estejam ativas nos mesmos estágios de *pipeline*. A modelagem das diferentes *threads* em um sistema SMT é considerado um problema intratável por Schoeberl (2009b).

No caso das análises de *cache* de um único nível, o analisador tenta classificar os acessos à memória entre acertos e faltas. Para os acessos que não podem ser classificados ambas as possibilidades devem ser consideradas se a arquitetura não é temporalmente composicional. Além disso, tanto a precisão e a eficiência da análise dependem fortemente da política de despejo. A política LRU (*Last Recently Used* possui melhor previsibilidade enquanto as abordagens FIFO (*First-In-First-Out*) e Pseudo-LRU atingem limites WCET menos precisos devido a menor quantidade de referências que podem ser precisamente classificadas (CULLMANN et al., 2010). Em termos de eficiência, quanto menor a precisão da classificação, maior a quantidade de estados e caminhos que devem ser analisados para se determinar qual o pior tempo de execução.

No caso de processadores com vários núcleos (CMP – *Chip Multiprocessors*), um dos problemas de aplicações de tempo real está relacionado com as interações das memórias *cache* nível 2, nível 3 e a principal que são compartilhadas. O modelo de memória compartilhada promove um modelo de programação simples com o custo de tempo acesso aos dados imprevisí-

vel. A *cache* nível 2 é praticamente não analisável devido as interferências *inter-threads*. Os protocolos de coerência permitem uma visão coerente e consistente da memória para todos os núcleos. Contudo, estes protocolos trocam informações entre os núcleos a cada acesso à memória e introduzem variabilidade no tempo de acesso à *cache* mesmo em um acerto (*cache hit*).

Além das interações das *caches* compartilhadas, um dos grandes desafios para a análise estática de sistemas CMP é causada pela interferência na utilização de recursos compartilhados. Os recursos são compartilhados por razões relacionadas com custo, energia e comunicação. Quando um recurso de *hardware* é compartilhado por diversos núcleos, o acesso pode ser interlaçado por uma infinidade de maneiras, em particular se os utilizadores não são sincronizados. Estas diferentes sequências de acesso resultam em diferentes estados no recurso. Enquanto acessos interlaçados já podem incorporar diferentes tempos de execução, o estado resultante do recurso ainda pode aumentar a diferença dos tempos de execução nos comportamentos futuros. Assim, os autores de (CULLMANN et al., 2010) citam dois tipos de interferências relacionadas à recursos compartilhados:

- interferências inerentes: é o comportamento que pode ser observado na execução do sistema. Estas interferências aumentam o tempo de execução das tarefas e então, consequentemente, os limites para a estimação do WCET.

- interferências virtuais: são introduzidas pela abstração do sistema devido à falta de alguma informação. Esta interferência pode nunca acontecer no sistema real, mas a análise não pode provar. Por exemplo, se a análise da tarefa τ abstrai completamente a execução de outras tarefas, ela deve assumir uma interferência de uma outra tarefa τ' toda a vez que acessar um recurso. Esta falta de informação é causada pela abstração total do conjunto de tarefas e o escalonamento do sistema, que pode restringir o que realmente ocorre.

4.3 *CACHE* NA ANÁLISE DE ESCALONABILIDADE

A analisabilidade dos sistemas descritos até então considera apenas tarefas executadas sem interrupção, ou seja, a ferramenta de WCET não considera preempção na análise. Contudo, há conjuntos de tarefas, principalmente aqueles que possuem tarefas de alta prioridade e *deadlines* curtos, que são somente escalonáveis por regime preemptivo. Nos *hardwares* modernos a preempção não é livre de efeitos colaterais e as interferências relacionadas com a memória *cache* são um grande problema que não é resolvido pela fer-

ramenta WCET. Até mesmo as interrupções de *hardware* podem ser vistas como tarefas preemptivas e, como foi descrito no capítulo 3, também não são consideradas pelo analisador WCET.

Quando incorporado no teste de escalonabilidade, os efeitos da *cache* levam em consideração o comportamento inter-tarefas (extrínseco) da *cache*. O comportamento intrínseco da *cache* é considerado como resolvido pela ferramenta de WCET.

Conforme citado por (CULLMANN et al., 2010), a interferência da *cache* pode ser resolvida basicamente por três métodos:

1. evita-se a interferência com particionamento de *cache*: para cada tarefa é atribuída uma parte da *cache* onde não há interferência já pelas premissa do projeto conforme (BUI et al., 2008), (KIRK, 1988), (WOLFE, 1994) e (MUELLER, 1995). O analisador estático pode ser utilizado diretamente como se o sistema fosse não preemptivo, considerando as partições da *cache* diretamente na análise.

2. incorporar a interferência da *cache* durante a análise WCET: tipicamente essas abordagens consideram os piores pontos e números de preempções para a garantia de segurança. Se a quantidade de código ou dados é maior que a *cache* e há algum reúso de memória, a análise faz uma grande sobrestimação do WCET da tarefa.

3. análise dos custos da interferência da *cache* separadamente: um limite superior do atraso da preempção é derivado com utilização de análises estáticas e de fluxo de dados. É computado um limite superior das faltas de *cache* e assume-se uma penalidade constante das faltas. Esta análise é válida se for limitada a arquiteturas temporalmente composicionais. Com esta limitação, esta análise não é diretamente aplicada a sistemas com efeito dominó (*cache* com políticas P-LRU ou FIFO). Durante a execução uma tarefa preemptada pode modificar os estados da *cache* e produzir o efeito dominó.

Quando um sistema é preemptivo, além da problemática de estabelecer valores para o WCET das tarefas, é necessário verificar a viabilidade do sistema em relação ao tempo máximo de resposta permitido para cada tarefa (*deadline*). Esta verificação é tipicamente realizada por testes de escalonabilidade. Conforme (FARINES et al., 2000), o objetivo do teste é verificar se existe uma escala realizável para dado conjunto de tarefas e os testes variam conforme o modelo de tarefas e políticas adotadas. O modelo de tarefas e as políticas definem um problema de escalonabilidade.

Assim como os testes, as abordagens para incorporação dos efeitos da *cache* também dependem do modelo de tarefas e das políticas de escalona-

mento. Em sistemas monoprocessados, considerando políticas que utilizam prioridade, há sistemas que possuem atribuição de prioridade fixa e prioridade dinâmica. Nas de prioridade fixa geralmente utilizam-se algoritmos de escalonamento como o *Rate-Monotonic* ou *Deadline-Monotonic* (LIU, 2000) e prioridade das tarefas não varia durante a execução do sistema. Na prioridade dinâmica, as prioridades das tarefas variam durante a execução como por exemplo no algoritmo EDF – *Earliest Deadline First*. Os detalhes sobre os algoritmos de escalonamento citados podem ser consultados em (FARINES et al., 2000) ou (LIU, 2000) e o modelo de tarefas bem como a política deste trabalho serão formalizados no capítulo 5. A incorporação dos efeitos *cache* em sistemas com prioridade dinâmica pode ser consultada no trabalho de Ju et al. (2007).

As técnicas que serão descritas consideram sempre que a análise de escalonabilidade é realizada pelo teste de tempo de resposta (RTA – *Response Time Analysis*). Para tal teste, considera-se conjunto de tarefas esporádicas com prioridade fixa. Como a atribuição de prioridade é independente para o teste RTA, ele pode ser utilizado em qualquer algoritmo de escalonamento com prioridade fixa. Além disso, para os métodos que serão descritos, conjunto de tarefas é sempre estático, não se considera interferências do escalonador e interrupções de *hardware*.

No teste RTA considera-se que:

- a demanda mais alta de processamento ocorre quando todas as tarefas estão prontas simultaneamente (instante crítico);
- para cada tarefa τ_i, o somatório do seu tempo de computação e da interferência imposta por tarefas de mais alta prioridade deve ser menor ou igual ao seu *deadline*.

O equacionamento do tempo de resposta pode ser consultado em (AUDSLEY et al., 1991), conforme a equação 4.13.

$$R_i^{k+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j \tag{4.13}$$

O atraso da recarga de *cache* devido à preempção afeta a tarefa de interesse de modo direto e indireto. Diretamente a tarefa sofre pela carga de *cache* adicional devido a preempção quando retorna a sua execução. Indiretamente as tarefas com prioridade mais alta, que também podem sofrer preempções, pagam a sua recarga de *cache*, há aumento do seu tempo de execução e há aumento do tempo de resposta da tarefa de interesse. Consequentemente a ta-

refa sob análise também é afetada pelo tempo de execução maior das tarefas com prioridade mais alta que a sua.

Para ilustrar, considere a escala de tarefas conforme a figura 15. A tarefa τ_2 sofre preempção de τ_1 no ponto p_2. O tempo t_{c2} é um atraso direto devido a recarga de memória *cache*. Quando τ_2 retorna a executar, a *cache* do processador não contém mais os dados de τ_2 pois possivelmente as linhas foram utilizadas por τ_1 ou pelo próprio escalonador. Em relação a tarefa τ_3, sem considerar o atraso t_{c2}, o início de sua execução seria em t_{i3} obtendo um tempo de resposta t_{ra3}. Contudo devido ao atraso de preempção de τ_2 o tempo de resposta da tarefa τ_3 é maior. Este é um atraso indireto devido a recarga adicional de *cache* sofrida por τ_2 na preempção. Estes tempos adicionais não são considerados na análise da WCET na ferramenta, pois a análise abstrata da memória *cache* considerada o código como sendo totalmente sequencial.

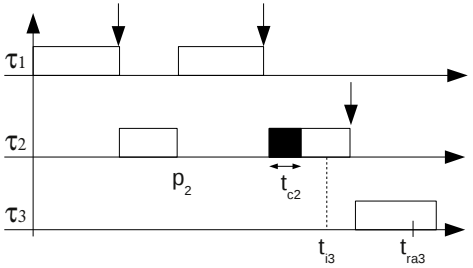


Figura 15: Exemplo de atrasos de *cache* direto e indireto.

4.3.1 Incorporação dos efeitos de *cache* como interferência direta no tempo de computação

No trabalho de Busquets Mataix et al. (1996), a interferência extrínseca sofrida por cada tarefa é considerada como sendo simplesmente outra interferência a ser adicionada na fórmula do tempo de resposta. O problema pode ser divido em quantas vezes a interferência é contabilizada e quanto é o tempo de penalização. O subproblema relacionado com o tempo de penalização pode ser medido através de cinco maneiras diferentes:

1. considerar o tempo para recarregar toda a memória *cache*;

2. considerar o tempo para recarregar as linhas retiradas pela tarefa que interrompeu a execução;

3. considerar o tempo para recarregar as linhas utilizadas pela tarefa interrompida;

4. considerar o tempo para recarregar as linhas úteis (que serão novamente referenciadas) que a tarefa interrompida pode conter na *cache* considerando o pior ponto de preempção;

5. considerar o tempo para recarregar a intersecção das linhas entre a tarefa interrompida e interruptora.

Abordagens 2, 3, 4 e 5 requerem o conhecimento do número de linhas referenciadas por cada tarefa antes da análise. Adicionalmente, a abordagem 5 requer o conhecimento de quais linhas exatamente foram referenciadas por cada tarefa, não somente o número delas. Abordagens 3, 4 e 5 requerem uma caracterização exata da preempção direta sofrida por cada tarefa porque a penalidade depende da tarefa que foi interrompida. Estas abordagens (3, 4 e 5) são consideradas incompatíveis com este modelo de análise de tempo de reposta (equação 4.14) por que não é possível fazer uma distinção entre uma preempção adicional sofrida por penalidade indireta ou direta, simplesmente é considerado o limite de preempções (Busquets Mataix et al., 1996). Além disso, como nas abordagens 3, 4 e 5 deve haver o conhecimento das linhas tocadas por cada tarefas, o tempo de penalização da *cache* também depende do instante da preempção. A abordagem 1 é totalmente independente das linhas usadas pelas tarefas. Desta forma, na solução de Busquets Mataix et al. (1996) considerou-se apenas as abordagens 1 e 2. Portanto, a equação 4.14 apresenta o termo adicional γ que representa o tempo de recarga de *cache* adicional devido a uma preempção. γ pode representar os tempos calculados pelas abordagens 1 ou 2.

$$R_i^{k+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil \times (C_j + \gamma_j) \tag{4.14}$$

Em análise da equação 4.14, o termo γ foi somado ao tempo de execução das tarefas de mais alta prioridade. Essa abordagem é interessante e é explicada considerando a figura 16. As tarefas τ_1 e τ_2 estão prontas simultaneamente. τ_2 começa sua execução é interrompida por τ_1. Quando τ_2 retornar para executar τ_2 sofre um retardo que é representado pela área escura devido à utilização de memória *cache* por τ_1. Em termos práticos, essa interferência é facilmente modelada como se τ_1 executasse pelo tempo γ, ou seja, a soma de $(C_j + \gamma_j)$ representa a parcela de interferência da execução normal de τ_1 mais o tempo de recarga da memória *cache* necessário para τ_2

continuar executando, ou seja, o comportamento extrínseco da *cache*. Conforme comentado anteriormente, o comportamento intrínseco da *cache* (faltas durante a execução de uma mesma tarefas devido a conflitos por capacidade ou associatividade) são resolvidos pela ferramente que calcula o *C* (WCET) de cada tarefa. Além disso, como o tempo de computação representa o pior caso, no início da execução de cada tarefa, o estado inicial da *cache* deve ser o pior possível (*cache* totalmente fria ou totalmente desorganizada, neste caso contabilizando o tempo de despejo dos dados na memória principal).

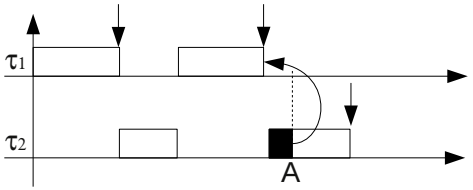


Figura 16: Atraso de preempção da tarefa 1 sobre a tarefa 2.

Além da introdução do parâmetro da *cache* no teste, o trabalho de Busquets Mataix et al. (1996) comparou o teste de tempo de resposta com a abordagem de *cache* particionada e foi concluído que o tamanho da *cache* é um parâmetro importante. A técnica de particionamento possui um desempenho melhor com o aumento do tamanho, enquanto a abordagem que considera a recarga completa dos blocos é pior. Em todos os casos há um compromisso entre o tempo de leitura de memória com o tamanho da *cache*.

4.3.2 Incorporação dos efeitos de *cache* como problema de otimização

Em Lee et al. (1998) também é proposta uma técnica para analisar o atraso do custo de preempção relacionado com a memória *cache* no contexto do escalonamento com prioridade fixa. Num primeiro momento há a análise por tarefa para estimar o custo de preempção relacionado com a memória *cache* para cada ponto de execução da tarefa em análise. Após isto, é computado o pior tempo de resposta de cada tarefa incluindo os atrasos de preempção utilizando a equação da análise de tempo de resposta (RTA) (equação 4.13) e técnicas de programação linear.

Conforme Lee et al. (1998), os trabalhos anteriores (como o de Busquets Mataix et al. (1996)) assumem que cada bloco de memória utilizado pela tarefa interruptora irá substituir todos os blocos da tarefas interrompida (preemptada). Esta abordagem é considerada muito pessimista pois é possí-

vel que alguns dos blocos substituídos não sejam mais necessários pela tarefa preemptada.

Novamente há uma modificação da equação do tempo de resposta adicionando uma parcela relacionada com os atrasos de preempção da memória *cache*, conforme a equação 4.15.

$$R_i^{k+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil \times C_j + PC_i(R_i^k) \tag{4.15}$$

O termo $PC_i(R_i^k)$ representa o atraso total de preempção relacionado com *cache* da tarefa τ_i durante R_i, ou seja, o tempo total de recarga de *cache* de $\tau_1, \tau_2, ...\tau_i$ durante R_i. Baseando-se no conceito de Busquets Mataix et al. (1996), $PC_i(R_i^k)$ incorpora os atrasos diretos (preempção direta de uma tarefa de alta prioridade) e indiretos (τ_2 interrompe τ_3, mas depois τ_1 interrompe τ_2). Por exemplo $PC_3(R_3)$ é o atraso de preempção relacionado com *cache* da tarefa τ_3 durante R_3, que soma os tempos de recarga de τ_1, τ_2 e τ_3 durante R_3.

$PC_i(R_i^k)$ é computado da seguinte forma:

- análise por tarefa: cada tarefa é analisada estaticamente para determinar o custo de preempção a cada ponto de execução. Este é o custo pago pela tarefa em uma preempção e é limitada pelo número de blocos reutilizáveis nos pontos de execução. Baseando-se nesta informação, propõe-se uma tabela do número de preempções (1 a n) e o seu custo (f) $(1, 2, 3, ...n \times f_1, f_2, f_3, ...n)$. O custo f_k é a k-ésimo custo marginal de preempção, ou seja, o custo que a tarefa paga no pior caso pela sua k-ésima preempção sobre a $(k-1)$-ésima preempção. Maiores detalhes da construção da tabela serão apresentados abaixo.

- análise do atraso de preempção: é utilizado uma técnica de programação linear para computar $PC_i(R_i^k)$ utilizando as tabelas de custo de preempção (da análise anterior) e um conjunto de restrições sobre o número de preempções da tarefa devido às tarefas de prioridade mais alta.

A análise por tarefa deve estimar quais são os blocos utilizáveis de memória *cache* para construir a tabela de custos. A técnica baseia-se na "análise de fluxo de dados" sobre o código da tarefa expressado em grafo de fluxo. Neste grafo cada nó representa um bloco básico como uma sequência de instruções consecutivas onde o fluxo de controle entra e sai sem possibilidade de interrupção ou decisão. Através da análise de fluxo de dados classifica-se

os blocos em "blocos alcançáveis" e "bloco vivos". Um bloco de *cache* é considerado utilizável em um ponto p se ele está presente nos conjuntos vivo e alcançável, ou seja, já foi referenciado em alguma vez (presente na *cache*) e será novamente referenciado após p.

A tabela de preempção é construída com as informações do custo de preempção de cada ponto e a sua pior quantidade de visitas, o qual pode ser diretamente derivado do grafo de fluxo do programa e dos limites dos laços. A construção assume o pior caso de preempção, pois não se sabe, a priori, quando as preempções vão ocorrer. O pior cenário de preempção ocorre quando a primeira preempção ocorre no ponto de maior custo (ponto com maior quantidade de blocos utilizáveis) e, então, a segunda ocorre no ponto com segundo maior custo e assim por diante. Este cenário é abordado para considerar a análise segura. Com base nesta situação de pior caso, o primeiro item da tabela (p_1) é o ponto com o maior custo de preempção (maior número de blocos utilizáveis). Supondo que para p_1 há, no pior caso, 5 visitas, 5 itens da tabela são preenchidos com o custo de preempção $p1$, ou seja, como há 5 passagens do fluxo de controle pelo ponto p_1, é possível que haja 5 preempções e todas ocorram em p_1. Os próximos pontos da tabela são preenchidos com o mesmo raciocínio considerando o segundo ponto mais longo de preempção e seu número de visitas, o terceiro ponto e assim por diante. Este processo é repetido até que todas as entradas sejam exauridas para cada tarefa do conjunto.

Após a construção da tabela de custo de preempção para cada tarefa, os autores propõem a equação 4.16 para estimar o pior caso de $PC_i(R_i)$.

$$PC_i(R_i) = \sum_{j=1}^{i} \sum_{l=1}^{\infty} g_{j,l}(R_i) f_{j,l} \tag{4.16}$$

O termo $g_{j,l}(R_i)$ representa o número de invocações de τ_j que são preemptadas pelo menos l vezes durante o tempo de reposta R_i. Para a tarefa de mais alta prioridade, $g_{1,l}(R_i) = 0, \forall l$ (tarefa de mais alta prioridade nunca é preemptada). O termo $f_{j,l}$ representa o valor derivado da tabela do custo de preempção da tarefa τ_j com número de preempções l. Se assumir que se conhece os valores de $g_{j,l}(R_i)$ que causam o pior cenário de preempção entre as tarefas, o pior custo de preempção é calculado pela equação 4.16.

Em geral, não se pode determinar exatamente qual a combinação de $g_{j,l}(R_i)$ que irá retornar o pior caso de atraso de preempção de uma tarefa. Portanto, para tornar o valor de preempção seguro, resolve-se a equação 4.16 por programação linear inteira, maximizando o valor de $PC_i(R_i)$ sujeito a duas restrições. A primeira restringe o valor de $g_{j,l}(R_i)$ para dado intervalo

R_i não podendo ser maior que o número de invocações de τ_j durante este intervalo. A segunda restringe que o número de preempções da tarefa τ_j durante o intervalo R_i não pode ser maior que o número de invocações das tarefas de mais alta prioridade durante este intervalo de tempo.

Assim para cada iteração da equação de tempo de resposta (equação 4.15) deve-se resolver o problema de programação linear inteira para computar os atrasos de preempção.

Para verificar a precisão dos resultados, Lee et al. (1998) comparam quatro métodos para prever o tempo de resposta considerando o atraso de preempção da memória *cache*. O método "A" considera que o custo de preempção é a recarga completa da memória *cache*, "C" é o método explicado na seção anterior (proposto por Busquets Mataix et al. (1996)), "U" é o método que considera o custo de recarga completa do código utilizado pela tarefa preemptada e P é o método proposto que considera somente a recarga dos blocos utilizáveis. Em termos de pessimismo as análises são ordenadas como "A", "C", "U" e "P", sendo "A" a com maior pessimismo. Contudo os valores que comparam as três abordagens em uma plataforma real consideram um sistema com penalidade de apenas 4 ciclos à memória e portanto, a penalidade relacionada com a memória *cache* é muito menor que 1% do tempo de resposta para os conjuntos de tarefas analisados.

Para realizar uma melhor análise foi proposto simulações que variam o tempo de acesso a memória de 0 a 300 ciclos e consideram a relação custo de preempção (PC) sobre o tempo de resposta (R) – fração PC/R. O método "A" é extremamente pessimista com crescimento quase exponencial: com cerca de 20 ciclos de tempo de acesso à memória o tempo de preempção já é 50% do tempo de resposta. Os métodos "C" e "U" possuem um crescimento linear da relação PC/R até pontos onde ocorrem invocações adicionais das tarefas de alta prioridade dentro da janela R_i. "P" é o método com menor pessimismo e com crescimento linear dentro da faixa da simulação.

Em todos os casos, nas simulações de Lee et al. (1998), até para o método menos pessimista, o atraso de preempção relacionado com a memória *cache* pode chegar até a 10% do tempo de resposta de uma tarefa se considerado um acesso de 100 ciclos à memória principal, ou seja, efeitos da *cache* não deveriam ser ignorados.

Em termos gerais, a abordagem de Lee et al. (1998) é mais precisa do que a de Busquets Mataix et al. (1996). Contudo é necessário uma quantidade razoável de análises para calcular o atraso de preempção, que torna a análise de escalonabilidade computacionalmente muito mais complexa.

4.3.3 Incorporação dos efeitos de *cache* considerando interferências entre tarefas explicitamente

Em Lee et al. (2001) é proposto um método aprimorado para analisar e estimar um limite do atraso relacionado à memória *cache* durante preempções no escalonamento de prioridade fixa preemptivo. A técnica proposta é uma melhoria do trabalho descrito em Lee et al. (1998) que leva em conta a relação entre a tarefa preemptada e o conjunto de tarefas que é executado durante a preempção e também leva em consideração as fases de execução das tarefas desconsiderando interações impossíveis. Novamente utiliza-se a técnica de fluxo de dados para estimar as interações de blocos de memória entre as tarefas e programação linear inteira para computar os atrasos de preempção.

Contudo, para melhorar a precisão da análise de Lee et al. (1998) são incorporadas duas novas características. Primeiro, leva-se em conta a relação da tarefa preemptada e o conjunto das tarefas que executam durante esta preempção no cálculo de blocos reutilizáveis de *cache*. Segundo, são consideradas as fases de início de execução das tarefas e são eliminadas interações inter-tarefas impossíveis. Estas duas características são modeladas como restrições do problema de programação linear inteira.

A tabela de preempção proposta por Lee et al. (1998) é estendida e incorpora a relação entre a tarefa preemptada e o conjunto de tarefas que executa durante esta preempção. Para tal, é considerado um número de grupos disjuntos calculado por $2^k - 1$, onde k é o número de tarefas com prioridade mais alta. Esses grupos formam a combinação de preempções. Por exemplo, se há 4 tarefas, há $2^3 - 1 = 7$ cenários de preempção para a tarefa τ_4 de prioridade mais baixa: $\{\tau_1\}, \{\tau_2\}, \{\tau_3\}, \{\tau_1, \tau_2\}, \{\tau_2, \tau_3\}, \{\tau_1, \tau_3\}, \{\tau_1, \tau_2, \tau_3\}$. Estes grupos representam todas as possíveis combinações de interferência que τ_4 pode sofrer durante sua janela de execução. Baseando-se nestes grupos, forma-se uma nova tabela de preempção.

Utilizando a nova tabela de preempção e estendendo as restrições de Lee et al. (1998) para que englobe as fases impossíveis, maximiza-se o atraso de preempção $PC_i(Ri)$ da equação interativa de tempo de resposta, conforme a equação 4.16.

O problema mais sério da técnica proposta é o crescimento exponencial com o aumento das tarefas. A complexidade da análise de fluxo de dados é igual à técnica proposta por Lee et al. (1998). Contudo, devido a consideração dos cenários de preempção que formam um problema combinatório, esta etapa tem complexidade $O(2^n)$ onde n representa o número de tarefas. O segundo passo que engloba a maximização do custo de preempção por programação linear inteira tem complexidade proporcional a quantidade de variáveis e restrições. Como o número de variáveis é o número de cená-

rios de preempção chega-se novamente a $O(2^n)$. Há algumas simplificações que reduzem a complexidade mas, ou reduzem a precisão da análise, ou fazem considerações sobre menor número de cenários de preempção (blocos de *cache* utilizáveis com menor sobreposição $\cong$ *cache* particionada entre as tarefas).

Considerando conjuntos de tarefas gerados aleatoriamente (conjuntos com 4 a 8 tarefas), a técnica proposta incorpora uma melhoria de 5% a 18% sobre o trabalho de Lee et al. (1998). Contudo, apesar de ser uma solução elegante, elevou-se a complexidade linear de (LEE et al., 1998) para exponencial obtendo um ganho médio máximo de 18%. Além disso, é apenas considerado os efeitos das tarefas, em nenhum dos trabalhos considera-se linhas de *cache* sujas pelo escalonador ou por interrupções de *hardware*.

4.3.4 Resumo das abordagens

A figura 17 apresenta um resumo das técnicas apresentadas para a análise de escalonabilidade com garantia considerando a memória *cache*. As técnicas citadas consideram apenas um nível de *cache*.

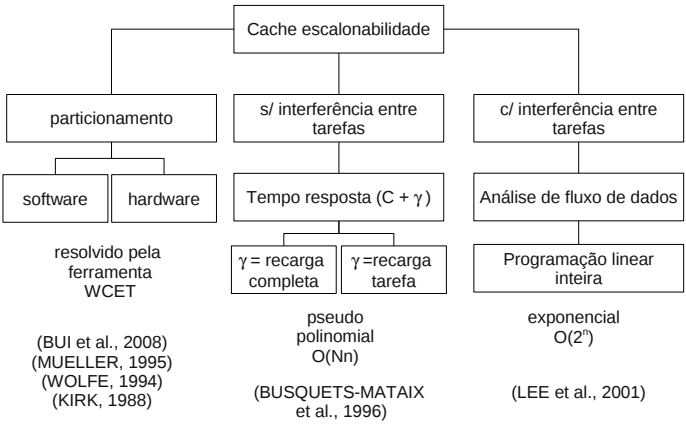


Figura 17: Resumo das técnicas de inclusão de efeitos na cache no teste de escalonabilidade

A técnica sem consideração explícita da interferência das tarefas é apresentada em Busquets Mataix et al. (1996). É mais simples e segura mas incorpora pessimismo decorrente a consideração de que todas as preempções

despejarão todo o conteúdo da *cache*. O problema desta técnica está relacionada com arquiteturas com efeito dominó pois é difícil encontrar os estados da *cache* que devido a interação com as outras tarefas ocasionará no pior tempo de recarga.

Em Lee et al. (2001) e Lee et al. (1998) são incorporadas as interferências das tarefas na análise de escalonabilidade através de uma análise de fluxo de dados e formulação de problemas de otimização. O resultado da análise é mais preciso e é aplicado somente a modelo de tarefas totalmente conhecido e estático. Na utilização de servidores de aperiódicas onde não se sabe a priori que o processador executará, o conhecimento sobre a *cache* realizado pelas técnicas de fluxo de dados que incorpora a interatividade das tarefas já não é mais válido.

Com base no estudo da literatura, para análise de escalonabilidade com garantia, considera-se apenas *caches* individuais (dados ou instrução). Arquiteturas com mais de um nível de *cache* ou unificada (dados e instruções compartilhados) nem são citados. Conforme (SCHOEBERL, 2009a) as interações das tarefas nesse tipo de arquitetura destrói todos os estados abstratos da *cache* e consequentemente torna a análise pessimista ou não segura.

4.4 MODELO DE ARQUITETURA DETERMINISTA

Uma das alternativas também apresentadas na literatura é descartar os elementos de *hardware* dos processadores que não são deterministas. Assim, estes trabalhos consideram que os processadores de propósito geral atuais são otimizados para alto desempenho (*throughput*) de caso médio e os sistemas de tempo real necessitam de processadores com tempo de execução (WCET) razoável e conhecido.

Como descrito nas seções anteriores, características como *pipelines* com dependências entre instruções, *caches*, elementos especulativos e execução fora de ordem, apesar de melhorar o desempenho, complicam a análise WCET, levam para estimativas muito conservadoras e influenciam gravemente a análise de escalonabilidade. Para sistemas de tempo real, a ideia principal é “fazer o WCET rápido e o sistema como um todo fácil de analisar”.

O objetivo de trabalhos que sugerem implementação de novas arquiteturas (como Schoeberl (2009b)) é endereçar os problemas relacionados com a estimativa WCET dos processadores atuais e propor melhorias para tornar o processador mais previsível tornando a análise estática mais simples. De modo geral, é propor elementos da arquitetura que diminuam o pessimismo do WCET e que tornem os pontos de menor tempo de execução (BCET –

Best Case Execution Time), tempo de execução médio (ACET – *Average Case Execuion Time*) e WCET mais próximos.

Para o *pipeline*, propõe-se um sistema simples. Se houver dependências entre as instruções, estas devem ser minimizadas para evitar os efeitos de atrasos sem limite.

O conjunto de instruções deve conter instruções com tamanho fixo para evitar a fila de pré-busca. As *caches* são segmentadas conforme explicado no trabalho de Schoeberl (2009a), onde se propõe *caches* para áreas diferentes de dados: uma *cache* de instrução, uma de pilha, uma para dados estáticos, uma pequena totalmente associativa para acessos de alocação dinâmica (*heap*) e uma para constantes.

As unidades de previsão de salto do fluxo de controle devem ser simples. Evita-se os previsores baseados em histórico e sugere-se a utilização de tomadas de caminhos estáticas (polarização para *taken/not-taken* dependendo do endereço) ou predições geradas pelo compilador.

Em termos de paralelização por instrução, sugere-se utilizar uma arquitetura VLIW pois escalonamento das instruções é estático sendo realizado pelo compilador. Arquiteturas dinâmicas super-escalares são totalmente descartadas.

Nos sistemas com paralelismo por *thread*, sugere-se utilizar um escalonamento conhecido e não dependente dos estados das *threads*, como por exemplo, sugere-se utilizar *Round-robin*. Para evitar os problemas de *cache*, além de duplicação dos registradores, sugere-se duplicar as *caches* para cada *thread* simultânea permitida. Esquemas SMT são completamente descartados devido à complexidade das interações entre as *threads*.

Para os processadores com vários núcleos deve-se construir um acesso a memória principal e periféricos previsível. Desta forma, sugere-se acessar a um modelo TDMA (*Time Division Multiple Access*) onde cada núcleo tem a sua fatia de tempo para acessar a memória.

A maioria das características propostas para uma arquitetura previsível foram implementadas no processador JOP (*Java Optimized Processor*) (SCHOEBERL, 2011) e está disponível publicamente sob licença livre GPL. O processador JOP foi implementado em um FGPA Altera Cyclone rodando em 100 MHz. Foi configurado com uma *cache* de instruções de 4kB e 1KB para pilha. Após a implementação, o processador foi comparado com vários sistemas embarcados Java comerciais e mostrou-se que a arquitetura temporalmente previsível não necessariamente é lenta. Além disso, a arquitetura proposta possui um *footprint* pequeno e pode-se explorar desempenho com uma arquitetura com vários núcleos.

Em resumo, um dos princípios importantes no projeto de sistemas temporalmente previsíveis é alcançar um bom compromisso entre custo, desem-

penho, previsibilidade, preocupando-se com o compartilhamento ou duplicação de recursos. Os aspectos acima formam uma arquitetura temporalmente composicional, *cache* de instrução e dados não unificada, *caches* com política de despejo LRU, barramento de acesso à recurso com limite de tempo de acesso, *caches* privadas e memórias privadas ou com acesso determinista.

Como tal arquitetura previsível ainda não se encontra disponível no mercado, há alguns métodos apontados por Cullmann et al. (2010) ilustrando a utilização do processador *FreeScale* MPC5668G em sistemas automotivos embarcados. O diagrama de blocos do processador pode ser visualizado na figura 18.

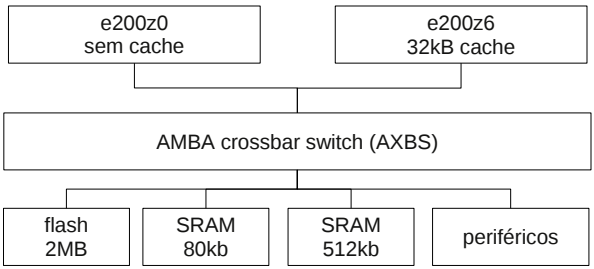


Figura 18: Diagrama de blocos do processador *FreeScale* MPC5668G (CULLMANN et al., 2010)

Conforme o diagrama de blocos percebe-se dois núcleos diferentes o e200z6 e o e200z0, um com *cache* e outro sem. O núcleo e200z6 é relativamente complexo possuindo um *pipeline* de sete estágios com única busca de instrução (*single issue*) e execução em ordem das instruções (núcleo escalar). Há um previsor dinâmico de saltos (*dynamic branch prediction*) com *buffer* de 8 estágios com política de despejo FIFO e uma *cache* unificada para dados e instrução (associatividade de 8 níveis e política de despejo pseudo-*round robin*). O núcleo z0 é uma versão simplificada do z6. Assim, para uma análise estática mais precisa, recomenda-se (CULLMANN et al., 2010):

- configurar a *cache* unificada (dados e instrução) para comportar-se como *cache* disjunta (não há conflito das linha da *cache* entre dados e instruções);
- devido a política de despejo ser pseudo-*round robin* que gera efeito dominó, sugere-se diminuir a associatividade para 2 níveis;
- desativar o previsor dinâmico de saltos;

- cada núcleo acessa sua porção de memória: z6 acessa o banco de 512kb e o z0 acessa o banco de 80kb, por exemplo, para diminuir a interferência;
- o acesso à memória *cache* utiliza um mecanismo de pré-busca. Somente um núcleo deve utilizá-lo e o outro deve ser configurado para executar o código em memória RAM, para diminuir as possíveis interferências.

4.5 CONCLUSÃO

Cada vez mais há necessidade de processadores de alto desempenho em sistemas de tempo real críticos. A quantidade de sistemas de controle em automóveis, aviões e aplicações industriais é muito grande e a complexidade e a inteligência desses sistemas tende a aumentar ainda mais. Exemplificando no campo da indústria automotiva, a utilização de processadores com vários núcleos pode reduzir a quantidade de unidades eletrônicas de controle (ECU – *electronic control unit*) diminuindo gasto energético bem como a necessidade de camadas de comunicação. Alguns veículos modernos podem ter até 80 ECUs (controle do *airbag*, controle de freios, trava de portas, janelas elétricas, luzes, controle do motor, direção elétrica, etc).

Para suprir esta demanda de alto desempenho e complexidade, os processadores simples de fácil análise já não são mais suficientes. Assim, as aplicações de tempo real críticas tendem a adotar processadores modernos que sacrificam o determinismo em prol do desempenho. Como apresentado, os *pipelines* com dependências entre instruções, *caches*, elementos especulativos, execução fora de ordem e múltiplos núcleos complicam enormemente a análise de tempo real desses processadores.

Em termos de arquitetura, cada vez mais o analisador estático deve contar com um modelo abstrato do *hardware* parecido com o real. A construção de tal modelo abstrato é desafiante pois além de ser seguro, deve fornecer precisão suficiente para não descartar sistemas possivelmente viáveis. Pode-se notar que o compromisso entre precisão e viabilidade de análise reduz enormemente a classe de processadores utilizáveis em sistemas de tempo real com garantias. A dificuldade de análise não está somente presente nos elementos internos do processador (*pipeline*, *cache*) mas também no sistema de memória, acesso aos periféricos e barramentos. A presença de anomalias temporais e efeito dominó aumenta o espaço de estados da análise enormemente podendo comprometer a viabilidade prática da análise estática. Toda esta complexidade já está presente na análise de tarefas de forma sequencial e o problema aumenta significativamente quando o sistema é preemptivo.

A preempção no processador não é livre de efeitos colaterais e há um custo envolvido, principalmente em sistemas com memória *cache*. O custo de preempção pode não ser fator determinante mas a destruição dos conteúdos abstratos da *cache* durante uma preempção inutiliza a estimativa do WCET computado sequencialmente. Dado a presença de anomalias temporais e efeito dominó, não se pode considerar o pior caso global no retorno de execução das tarefas. Portanto, preempções em pontos arbitrários, o que acontece na execução do sistema, são um problema quando o conteúdo da *cache* não é explicitamente controlado (ex: particionamento da *cache*).

Sabe-se que a viabilidade de um sistema de tempo real não só depende dos valores estimados do WCET de cada tarefa, mas também se durante o escalonamento todas as tarefas cumprem seus *deadlines*. Esta caracterização forma um problema de escalonamento e a incorporação dos esfeitos da memória *cache* já é estudado por um longo período. No contexto de sistemas preemptivos com prioridade fixa, geralmente estende-se a equação do tempo de resposta (RTA) para incorporar os efeitos da *cache* na análise de escalonabilidade. Como apresentado, existem técnicas simples com baixa complexidade como a de Busquets Mataix et al. (1996) que considera o atraso de preempção relativo à *cache* estático para cada tarefa e para cada preempção. Esta técnica sofre com problemas de precisão, pois na estimação do custo de preempção é sempre considerado o pior caso, e não é adequado em sistemas com efeito dominó se não há controle explícito do conteúdo da *cache* ou dos pontos de preempção. No caso de soluções mais elegantes resolvidas por programação linear (Lee et al. (2001)) a complexidade computacional é muito grande e também não é adequada para sistemas com efeito dominó.

Dados todos estes problemas, há autores que propõe arquiteturas especificamente projetadas para tempo real tendo em mente a previsibilidade e facilidade de análise. Como tais processadores ainda não existem e dado que custo dos processadores genéricos é baixo devido ao grande volume de mercado, arquiteturas totalmente previsíveis serão restritas para algumas aplicações ou implementadas FPGA.

Em termos práticos, quando o sistema de tempo real for crítico, possivelmente utilizar-se-á processadores dimensionados para desempenho de caso médio desativando recursos de *hardware*. Na implementação do sistema ocorrerá otimizações como particionamento de *cache*, escalonamento cooperativo ou preempções controladas para tornar viável a análise de escalonabilidade em conjunto com a obtenção do WCET seguros e mais precisos.

5 PROPOSTA DE UMA ABORDAGEM DE ESCALONAMENTO HETEROGÊNEO

Para suprir a necessidade de garantia, os sistemas de tempo real contam com uma série de testes e análises. Conforme apresentado nos capítulos anteriores, a obtenção da garantia é dividida basicamente em duas análises: obtenção do pior tempo de execução das tarefas (WCET – *Worst Case Execution Time*) e análise de escalonabilidade.

A obtenção do WCET é um grande problema atualmente devido à complexidade dos processadores e a severidade da dificuldade aumentou significativamente com a utilização de processadores com vários núcleos. Neste contexto, para haver um compromisso entre precisão e eficiência, os analisadores geralmente consideram código estritamente sequencial: não são analisadas preempções, exceções, migrações nem interrupções. Enquanto isso, a preempção é indiscutivelmente necessária para maior escalonabilidade e a migração para geração de escalas ótimas.

Apesar de geralmente o problema de escalonabilidade e o problema do WCET serem tratados isoladamente, eles estão intensamente ligados. Um limite WCET apertado e preciso aumenta os níveis de escalonabilidade bem como considerações dos custos de preempção na análise de escalonabilidade melhora a garantia do sistema. É importante que a análise de viabilidade do sistema como um todo considere ambos os aspectos.

O objetivo deste capítulo é apresentar uma solução de escalonamento heterogêneo que considere os problemas de precisão do WCET bem como a viabilidade em termos de escalonamento. Dadas as características problemáticas de obtenção de custos da migração e a implementação mais custosa, o sistema será particionado. O particionamento permite a utilização de testes de escalonabilidade em monoprocessadores, embora exista o problema do particionamento em si. A ideia é particionar considerando a necessidade de preempção dividindo o sistema em processadores sob regime preemptivo e não preemptivo.

O capítulo está organizado em uma seção sobre a motivação, uma sobre a descrição da abordagem proposta, formalização do problema, algoritmo de classificação, algoritmos para particionamento entre os processadores e escalonamento local, exemplo de implementação da abordagem e por fim as conclusões.

5.1 MOTIVAÇÃO

Uma proposta de escalonamento heterogêneo considerando processadores sob regimes de preempção diferenciados é motivada pela contradição fundamental que existe na análise de escalonabilidade dos sistemas de tempo real. Por um lado há as ferramentas WCET que não suportam preempção e por outro há a escalonabilidade que necessita de preempção para melhores resultados.

As ferramentas de WCET fazem a análise da microarquitetura e calculam os limites mínimos e máximos dos blocos de execução básicos. Esta análise considera aspectos do processador através de um modelo abstrato considerando o subsistema de memória e o fluxo de execução interno do *pipeline* do processador. Para cada bloco de execução forma-se invariantes determinando estados do processador e da memória. A complexidade do modelo abstrato depende fortemente da classe do processador. Dada a forma como os estados evoluem durante a execução da tarefa, é impossível determinar seguramente o estado de um processador complexo real no retorno de uma mudança de contexto devido a uma preempção. Desta forma, as ferramentas que calculam limites do WCET consideram execução contínua da tarefa para conservar estados do processador e da memória *cache* devido ao grande impacto desses elementos na segurança e precisão do WCET.

Em contrapartida, sabe-se que sistemas não preemptivos possuem problemas de limitação de escalonabilidade. As tarefas muito grandes de baixa prioridade fazem com que outras sejam prejudicadas bem como tarefas com *deadlines* curtos geralmente não são escalonáveis no regime não preemptivo. As próprias interrupções de *hardware* do processador podem ser consideradas como tarefas curtas com *deadlines* curtos que interrompem a execução das tarefas do sistema. Considerando apenas o escalonamento, a preempção sempre é melhor.

A problemática básica é como conciliar estas duas questões na análise de viabilidade de sistemas de tempo real em multiprocessadores.

Duas das possíveis soluções da literatura é considerar a preempção durante a obtenção do WCET ou na obtenção do tempo de resposta da tarefa. As arquiteturas modernas são cada vez mais complexas e a preempção gera pessimismo cada vez maior devido ao impacto da mudança de contexto sobre o *pipeline* e sobre a *cache*. Preempções não são livres de efeitos colaterais e grande parte dos custos adicionais de uma preempção são causados por interferências sobre a memória *cache*. Embora poucas preempções possam realmente ocorrer, não se conhece os momentos exatos durante a análise e se pressupõe os piores casos de interferência incorporando excesso de pessimismo. Além disso, dependendo da arquitetura, a perda ou suposição dos

estados da *cache* ou do *pipeline* durante o retorno da mudança de contexto pode acarretar uma situação melhor devido ao efeito dominó e anomalias temporais dos processadores. Dadas as circunstâncias pode haver otimismo na consideração dos estados iniciais relacionados, por exemplo, com políticas de despejo de *cache* e elementos especulativos do processador.

Outra solução é fazer o particionamento da *cache* entre as tarefas. O problema desta abordagem é que o particionamento é difícil de se resolver e coloca-se mais um problema para gerenciar. Nem todos os processadores são capazes de realizar particionamento facilmente (*caches* com associatividade, por exemplo), ou seja, coloca-se mais uma restrição ao *hardware* da aplicação. Outra questão é como adequar as tarefas com conjunto de trabalho de dados (WSS –*Working Set Size*) variável ao particionamento da *cache* sendo que o desempenho da tarefa tende a diminuir com espaço de *cache* limitado. Se o particionamento fosse dinâmico seria possível continuamente adequá-lo às necessidades das tarefas com respeito à quantidade de *cache* ocupada, contudo neste cenário seria necessário sincronizar a execução das diferentes tarefas para melhor particionar a *cache*, o que é impossível para sistemas com tarefas esporádicas.

Assim, ao invés de particionar a memória *cache* ou incorporar pessimismo adicional destes efeitos na análise de escalonabilidade, sugere-se particionar os processadores disponíveis considerando regimes de preempção diferenciados.

5.2 DESCRIÇÃO DA ABORDAGEM PROPOSTA

A abordagem de escalonamento de sistemas multiprocessados geralmente considera o sistema homogêneo como apresentado pela figura 19. Nesta abordagem, um conjunto de tarefas $\mathcal{T}$ é particionado em m processadores (CPUs) sob mesmo regime de escalonamento: não preemptivo (nP), pontos de preempção (PP) ou preemptivo (P).

Neste trabalho, com o objetivo de conciliar a contradição fundamental que existe na análise de viabilidade dos sistemas de tempo real com garantia, considera-se um escalonamento heterogêneo com respeito à preempção conforme a figura 20. Nesta nova proposta, um conjunto de tarefas $\mathcal{T}$ é particionado em m processadores (CPUs) regidos sob diferentes regimes de preempção: preemptivo (P), pontos de preempção (PP) e não preemptivo (nP). As tarefas são classificadas e particionadas conforme suas características considerando os aspectos temporais (período, *deadline*, utilização) e também custos associados à preempção. Através desta abordagem evita-se colocar maiores requisitos sobre o *hardware* da aplicação e não há camadas adici-

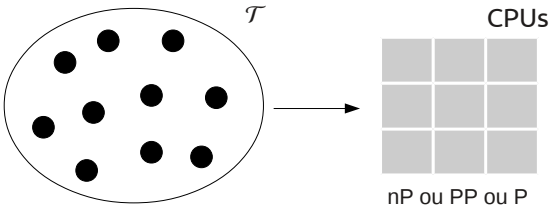


Figura 19: Particionamento de um conjunto de tarefas $\mathcal{T}$ sob o mesmo regime de preempção.

onais de gerência no sistema operacional. Tenta-se aproximar as questões ligadas ao problema de obtenção de WCET e a escalonabilidade. O escalonamento heterogêneo pode ser visto como uma forma de clusterização, porém o conceito deste último implica em grupos de processadores selecionados para escalonamento global. No caso da solução heterogênea, forma-se grupos de processador em regimes de preempção diferenciados.

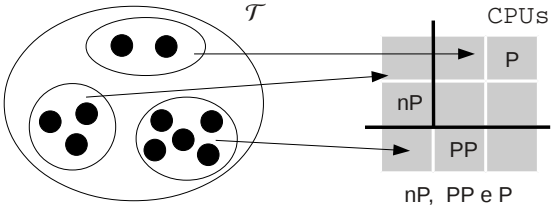


Figura 20: Particionamento de um conjunto de tarefas $\mathcal{T}$ sob diferentes regimes de preempção.

A abordagem proposta baseia-se nos seguintes fatos:

1. existem tarefas com *deadline* tão apertado que precisam executar em regime preemptivo (interrupções urgentes);

2. existem tarefas com WCET grande e com conjunto de trabalho de dados (WSS) tal, que é benéfica a não preempção (preservar estados de hardware do processador) com objetivo de se obter WCET menor e mais preciso;

3. uma tarefa com WCET grande pode, hipoteticamente, ser particionada pelo programador em alguns segmentos não preemptáveis. Ex.: todo

ponto de sincronização com técnicas de exclusão mútua é automaticamente um possível ponto de preempção.

Assim é considerada uma abordagem heterogênea com muitos processadores divididos em conjuntos não preemptivo, preemptivo ou com pontos de preempção. A alocação das tarefas no particionamento considera essas características e a viabilidade do sistema é verificada por testes de escalonabilidade para cada regime. O número de processadores de cada conjunto é definido no próprio processo de alocação. O objetivo do método é escalonar as tarefas sob o regime não preemptivo quando possível para conservar as propriedades do WCET (precisão e segurança) e aumentar a escalonabilidade executando as demais tarefas no regime preemptivo.

5.3 FORMALIZAÇÃO DO SISTEMA

Considera-se o modelo de tarefas esporádicas onde uma tarefa esporádica $\tau_i = (C_i, D_i, T_i, \gamma_i)$ é caracterizada pelo seu pior tempo de execução (WCET) C_i, seu *deadline* relativo D_i, o seu intervalo mínimo entre chegadas ou período T_i e uma penalização γ_i relacionada à preempção. O termo γ está associado à estratégia de escalonamento no regime preemptivo podendo corresponder, por exemplo, a um tempo de execução C_i' com $C_i' > C_i$, um fator (%) de aumento de C_i quando a tarefa executa em regime preemptivo ou tempo para recarga da memória *cache* e salvamento/restauração de contextos. O valor C_i corresponde ao WCET no regime sem preempção, ao passo que o WCET aumenta da penalização γ no caso do regime preemptivo. Para cada tarefa esporádica, $C_i \leq T_i$ bem como $C_i \leq D_i$. A utilização de uma tarefa (u_i) representa a quantidade de processamento requisitado por τ_i em um processador sendo denotada pela equação 5.1.

$$u_i = \frac{C_i}{T_i} \tag{5.1}$$

Cada tarefa τ_i gera uma sequência infinita de instâncias (*jobs*) que possuem tempo de execução máximo C_i. Uma instância de τ_i pronta para execução no instante t possui um *deadline* absoluto no tempo $t + D_i$ e a geração de instâncias sucessivas são separadas de no mínimo T_i unidades de tempo.

Assume-se que o sistema é formado por um conjunto Π de m processadores idênticos onde $\Pi = \{\pi_1, \pi_2, ...\pi_m\}$, e um conjunto de tarefas $\mathcal{T}$. Seja $\mathcal{T}$ formado por n tarefas esporádicas independentes onde $\mathcal{T} = \{\tau_1, \tau_2, \tau_i, ..., \tau_n\}$ e $\tau_i = (C_i, D_i, T_i, \gamma_i)$ para todo i, com $1 \leq i \leq n$. Uma tarefa $\tau_i \in \mathcal{T}_{\pi_j}$ representa uma tarefa τ_i executando no processador π_j, ou seja, $\mathcal{T}_{\pi_j}$ corresponde ao con-

junto de tarefas que executam no processador j, com $1 \leq j \leq m$. Além disso considera-se *deadlines* limitados onde para cada tarefa esporádica $\tau_i \in \mathcal{T}$ é satisfeita a condição $D_i \leq T_i$. A utilização de um processador (U_j) é quantidade de processador requisitado por todas as tarefas escalonadas por π_j, conforme a equação 5.2.

$$U_j = \sum_{i=1}^{n} u_i, \forall \tau_i \in \mathcal{T}_{\pi_j} \tag{5.2}$$

O sistema multiprocessado adotado não permite migração das tarefas entrando na classe de algoritmo de escalonamento particionado. Neste contexto, o conjunto de tarefas $\mathcal{T}$ é particionado em vários subconjuntos disjuntos e cada subconjunto é associado a um único processador ($\pi_1, \pi_2, ...\pi_m$). Todas as instâncias geradas pelas tarefas dentro do subconjunto devem executar em seu processador correspondente.

O conjunto Π de processadores também será dividido em regimes de preempção diferenciados. Um subconjunto disjunto de processadores Π_p executará tarefas no regime preemptivo considerando alguma penalização γ adicional das tarefas devido a preempção. O outro subconjunto disjunto de processadores Π_{np} irá escalonar as tarefas sob o regime não preemptivo. Ao contrário do subconjunto Π_p, quando uma tarefa τ_i iniciar sua execução em um processador pertencente a Π_{np}, não será possível interromper (preemptar) sua execução.

O objetivo do escalonamento é, dado um conjunto $\mathcal{T}$ de tarefas e um conjunto de Π de m processadores, escalonar o máximo de tarefas sob regime não preemptivo com no máximo m processadores.

5.4 ALGORITMO DE ESCALONAMENTO HETEROGÊNEO

No contexto deste trabalho, serão empregados apenas os regimes preemptivo e não preemptivo, apesar da abordagem também permitir o emprego do regime cooperativo (pontos de preempção).

O algoritmo da abordagem será dividido em duas partes:

1. particionar de forma escalonável todo o conjunto de tarefas sob regime não preemptivo obtendo um número m_{np} de processadores;

2. se $m_{np} > m$, processadores com regime não preemptivo serão transformados em preemptivos.

Apresentando um exemplo simples, supõe-se um sistema com $m = 16$

processadores. Após o particionamento não preemptivo (parte 1), chega-se a $m_{np} = 18$ processadores. Como $m_{np} > m$, é iniciada a parte 2 do algoritmo pois precisa-se de 2 processadores adicionais para escalonar o sistema de forma completamente não preemptiva. Para tentar aumentar a escalonabilidade, transforma-se as tarefas dos $m_{np} - m + i = 18 - 16 + i = 2 + i$ processadores em regime preemptivo, onde $i \leq m$ corresponde ao número de processadores que serão transformados em regime preemptivo. Na primeira iteração ($i = 1$), os últimos processadores até π_m são transformados em preemptivos, neste exemplo $2 + i = 3$ processadores são transformados: π_m, π_{m+1} e π_{m+2}. Todas as tarefas pertencentes aos processadores em transformação são desalocadas e reparticionadas em regime preemptivo. Se $i = 1$ não for suficiente, incrementa-se i até o sistema ser escalonado. Na pior das hipóteses, todo o sistema será escalonado sob regime preemptivo. Se mesmo assim não for possível atender todos os *deadlines* das tarefas, o sistema é considerado não escalonável em m processadores.

Considerando a definição do modelo, tem-se:

- Π: conjunto total de processadores do sistema;
- Π_p: conjunto de processadores em regime preemptivo;
- Π_{np}: conjunto de processadores em regime não preemptivo;
- $\mathcal{T}$: conjunto total de tarefas do sistema;
- $\mathcal{T}_p$: conjunto de tarefas em regime preemptivo não alocadas;
- $\mathcal{T}_{np}$: conjunto de tarefas em regime não preemptivo não alocadas;
- m: número total de processadores disponíveis: cardinalidade de Π ($m = \#\Pi$);
- m_p: número de processadores em regime preemptivo: cardinalidade de Π_p ($m_p = \#\Pi_p$);
- m_{np}: número de processadores em regime não preemptivo: cardinalidade de Π_{np} ($m_{np} = \#\Pi_{np}$);
- $\mathcal{T}_{\pi_j}$: conjunto de tarefas alocadas ao processador j;

A abordagem é formalizada pelo algoritmo 1. Este algoritmo de particionamento escalonamento heterogêneo faz o particionamento do conjunto de tarefas $\mathcal{T}$ do sistema em processadores não preemptivos (Π_{np}) e preemptivos (Π_p).

Algoritmo 1 Escalonamento heterogêneo; Entrada: $\mathcal{T}$, m e Π; Saída: Π_p e Π_{np}

$\Pi_{np} \leftarrow \emptyset$
$\Pi_p \leftarrow \emptyset$

{Particiona $\mathcal{T}$ pelo regime não preemptivo}
$\Pi_{np} \leftarrow$ PARTICIONA_NP ($\mathcal{T}$)

{Se conjunto não é escalonável nos m processadores, faz transformações para preemptivo}
while $(m_{np} + m_p > m) \wedge (m_{np} < m)$ **do**

{Constrói conjunto $\mathcal{T}_p$ das tarefas ou conjunto Π_p de processadores: algoritmo que classifica processadores e tarefas que serão regidos pelo regime preemptivo}
$\mathcal{T}_p \leftarrow$ CLASSIFICA (Π_p, Π_{np})
$\mathcal{T}_{np} \leftarrow \mathcal{T} - \mathcal{T}_p$

{Particiona $\mathcal{T}_p$ pelo regime preemptivo}
$\Pi_p \leftarrow$ PARTICIONA_P ($\mathcal{T}_p$)
end while

A heurística apresentada na proposta de escalonamento heterogêneo do algoritmo 1 é independente do escalonamento local dos processadores. Quaisquer algoritmos preemptivo e não preemptivo podem ser utilizados nesta abordagem desde que se tenha testes de escalonabilidade em monoprocessador para os algoritmos locais escolhidos. Além disso é necessário um algoritmo auxiliar que classifica quais processadores e quais tarefas que serão transformadas para o regime preemptivo. O critério desta escolha também é independente da proposta de escalonamento heterogêneo sugerido. Nas seções 5.5, 5.6 e 5.7 serão abordados algoritmos para classificação, particionamento e escalonamento local respectivamente.

5.5 ALGORITMOS DE CLASSIFICAÇÃO

O algoritmo de classificação tem como objetivo definir quais processadores e quais tarefas serão escalonadas sob regime preemptivo.

Dentro da abordagem proposta o critério de classificação é livre podendo considerar elementos como processadores com baixa utilização, pro-

cessadores com alta utilização, utilização individual das tarefas, folga ($T_i - D_i$), penalização de preempção (γ), além de outros fatores ou combinação destes.

Contudo, uma possível e simples solução, inicialmente apresentada no exemplo da seção 5.4, classificaria no regime preemptivo todas as tarefas alocadas ao último processador permitido (processador m) e as tarefas dos processadores adicionais que foram necessários para escalonar o sistema em regime não preemptivo. Caso esta iteração inicial não seja suficiente, as tarefas de outro processador não preemptivo, em ordem reversa, são transformadas para regime preemptivo. Esta classificação simples transforma para preemptivo as tarefas particionadas em regime não preemptivo em ordem reversa à da alocação inicial. Este algoritmo de classificação é formalizado pelo algoritmo 2

Algoritmo 2 Classificação em relação ao regime de preempção das tarefas e dos processadores; Entrada: Π_p e Π_{np}; Saída: $\mathcal{T}_p$

{Variável i conta número de invocação do algoritmo: inicialmente $i = 0$}
if $i = 0$ **then**
  {Constrói conjunto $\mathcal{T}_p$ das tarefas: união das tarefas dos processadores π_{np} que excedem m e do último processador permitido (π_{np_m})}
  $\forall \tau_k \in \mathcal{T}_{\pi_{np_j}} : m \leq j \leq m_{np}; \mathcal{T}_p \leftarrow \mathcal{T}_p \cup \tau_k; \mathcal{T}_{\pi_{np_j}} \leftarrow \emptyset$

  {Atualiza conjunto de processadores preemptivos e não preemptivos}
  $\forall \pi_{np_j} \in \Pi_{np} : m \leq j \leq m_{np}; \Pi_{np} \leftarrow \Pi_{np} - \pi_{np_j}$
  $m_{np} \leftarrow m_{np} - (m_{np} - m + 1)$

  $\Pi_p \leftarrow \emptyset$
else
  {Constrói conjunto $\mathcal{T}_p$ das tarefas}
  $\forall \tau_k \in \mathcal{T}_{\pi_{np_j}} : j = m - i; \mathcal{T}_p \leftarrow \mathcal{T}_p \cup \tau_k; \mathcal{T}_{\pi_{np_j}} \leftarrow \emptyset$

  {Atualiza conjunto de processadores preemptivos e não preemptivos}
  $\forall \pi_{np_j} \in \Pi_{np} : j = m - i; \Pi_{np} \leftarrow \Pi_{np} - \pi_{np_j}$
  $m_{np} \leftarrow m_{np} - 1$

  $\Pi_p \leftarrow \emptyset$
end if

$i \leftarrow i + 1$

5.6 ALGORITMO PARA PARTICIONAMENTO ENTRE PROCESSADORES E ESCALONAMENTO LOCAL

O problema do particionamento das tarefas é NP-difícil devido a sua equivalência com o *bin-packing problem*. Desta forma pode-se resolver o particionamento como um problema de otimização ou meta heurísticas de otimização (*simulated annealing* por exemplo).

Contudo, o que geralmente se utiliza são soluções sub-ótimas que combinam algoritmos de escalonamento local em monoprocessador com heurísticas polinomiais do *bin-packing problem* (*First-Fit*, *Next-Fit*, *Best-Fit*, entre outras) que possuem o seguinte padrão:

- as tarefas são ordenadas de acordo com algum critério;
- as tarefas são atribuídas (em ordem) a um processador até a sua capacidade de acordo com alguma condição (teste necessário, suficiente ou exato).

Uma tarefa τ_i é atribuída a um processador π_k se ela pode ser atribuída a este processador e escalonada por um dado algoritmo de escalonamento monoprocessado de modo que τ_i e todas as outras tarefas anteriormente atribuídas à π_k sempre cumpram seus *deadlines*. O critério que determina se uma tarefa pode ser atribuída a algum processador depende do algoritmo utilizado relacionado ao nível monoprocessado.

A proposta do algoritmo heterogêneo necessita de dois algoritmos para o escalonamento local: preemptivo e não preemptivo. Quaisquer algoritmos preemptivos e não preemptivos podem ser utilizados, desde que se tenha teste de escalonabilidade que garantam a escalonabilidade para cada processador individualmente. O teste do escalonamento preemptivo deve considerar a penalização de preempção γ descrita anteriormente na formalização do sistema. O algoritmo preemptivo pode ser intercambiável com um algoritmo de escalonamento cooperativo (com pontos fixos de preempção) desde que se considere também algum tipo de penalização γ.

5.7 EXEMPLO DE IMPLEMENTAÇÃO DA ABORDAGEM

Neste trabalho considerou-se o algoritmo de prioridade fixa DM – *Deadline Monotonic* para o regime preemptivo e o EDFnp – *Earliest Deadline First non-preemptive* para regime não preemptivo. Ambos algoritmos serão descritos nas próximas subseções considerando as heurísticas de particionamento.

5.7.1 Algoritmo para regime preemptivo

Para o algoritmo preemptivo considera-se a utilização do algoritmo de prioridade fixa *Deadline Monotonic* (DM). No DM a prioridade de cada tarefa é atribuída em ordem não crescente em relação ao seu *deadline*, ou seja, uma prioridade de τ_i mais alta que τ_j implica que $D_i \leq D_j$. Neste regime de escalonamento, uma prioridade única é associada para cada tarefa e todas as instâncias geradas possuem a mesma prioridade associada a sua tarefa. Empates de prioridade são resolvidos arbitrariamente pelo escalonador. Neste sentido, se uma tarefa τ_1 possui prioridade mais alta que uma tarefa τ_2 e ambas tarefas possuem instâncias ativas, a instância de τ_1 terá prioridade de execução sobre a instância de τ_2.

O regime de prioridade fixa preemptivo não é o melhor em termos de escalonabilidade mas representa um compromisso entre preempção e previsibilidade das tarefas, principalmente em relação a incorporação de custos adicionais de preempção γ. Por mais que haja preempções em tempos arbitrários sabe-se quais as possíveis tarefas que podem interromper a execução de uma outra.

A viabilidade do conjunto de tarefas sob o regime preemptivo em DM é verificada pelo teste de tempo de resposta conforme a equação 5.3 proposto Audsley et al. (1991) e a viabilidade com o termo γ foi inicialmente proposta por Busquets Mataix et al. (1996). O termo γ na equação 5.3 representa uma interferência adicional imposta pela execução das tarefas de prioridade mais alta dentro da janela de execução da tarefa τ_i que pode considerar tempo de recarga da memória *cache* ou outro atraso adicional a ser considerado na preempção. Neste teste, seja $\mathcal{T}_p = \{\tau_1, \tau_2, ..., \tau_n\}$ um conjunto de tarefas esporádicas em ordem não decrescente em relação à prioridade (para um par τ_i e τ_j, se $i < j$ então prioridade de τ_i é mais alta que a prioridade de τ_j). Se $\forall \tau_i \in \mathcal{T}_p : (1 \leq i \leq n)\ R_i \leq D_i$, o sistema é escalonável.

$$R_i^{k+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{T_j} \right\rceil (C_j + \gamma_j) \tag{5.3}$$

O particionamento no regime preemptivo em *Deadline Monotonic* será realizado por DM-FFDU – *Deadline Monotonic - First Fit Decreasing Utilization* sendo apresentado pelo algoritmo 3. O DM-FFDU apresentado é uma modificação do RM-FFDU – *Rate Monotonic - First Fit Decreasing Utilization* que foi proposto originalmente por Oh e Son (1995) e possui uma taxa de aproximação de 1.66 em relação a um particionamento ótimo.

Algoritmo 3 Particionamento em regime preemptivo; Entrada: $\mathcal{T}_p$; Saída: m_p e Π_p

Ordenar $\mathcal{T}_p$ em ordem não crescente de utilização (u_i);
$m_p \leftarrow 1; i \leftarrow 1$;
while $\mathcal{T}_p \neq \emptyset$ **do**
  {Atribui primeira tarefa ao próximo processador preemptivo}
  $\mathcal{T}_{\pi_{m_p}} \leftarrow \mathcal{T}_{\pi_{m_p}} \cup \tau_i$
  {Remove tarefa de $\mathcal{T}_p$}
  $\mathcal{T}_p \leftarrow \mathcal{T}_p - \tau_i$
  {Enquanto for escalonável pela equação 5.3}
  while $\forall \tau_j \in \mathcal{T}_{\pi_{m_p}} : (1 \leq j \leq n)\ R_j \leq D_j$ **do**
    $\mathcal{T}_{\pi_{m_p}} \leftarrow \mathcal{T}_{\pi_{m_p}} \cup \tau_i$
    $\mathcal{T}_p \leftarrow \mathcal{T}_p - \tau_i$
    $i \leftarrow i + 1$
  end while
  $m_p \leftarrow m_p + 1$
end while

5.7.2 Algoritmo para regime não preemptivo

Para o algoritmo não preemptivo considera-se a utilização do EDF – *Earliest Deadline First* não preemptivo (EDF-np). No EDF-np a tarefa com *deadline* absoluto mais próximo será escolhida para executar, mas somente quando a instância atualmente em execução terminar sua execução.

A viabilidade do conjunto de tarefas sob o regime não preemptivo com $D_i \leq T_i$ é verificada pelas condições impostas pela equação 5.4 propostas por George et al. (1996), maiores detalhes e exemplos podem ser encontrados neste trabalho. Neste teste, seja $\mathcal{T}_{np} = \{\tau_1, \tau_2, ..., \tau_n\}$ um conjunto de tarefas esporádicas com $U \leq 1$ é escalonável por EDFnp se e somente se:

$$\forall t \in S, t \geq h(t) + max_{D_i > t}\{C_i - 1\} \tag{5.4}$$

Com $max_{D_i>t}\{C_i - 1\} = 0$ se $\nexists t : D_i > t$ e $h(t)$ é a demanda de processador no tempo t definida pela equação 5.5.

$$h(t) = \sum_{D_i \leq t} \left(1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right) C_i \tag{5.5}$$

O conjunto de S é definido pela equação 5.6.

$$S = \left(\bigcup_{i=1}^{n} \{kT_i + D_i, k \in \aleph_0\} \right) \cap [0, min\{L, B_1, B_2\}] \tag{5.6}$$

Onde:

$$B_1 = \frac{\sum_{D_i \leq T_i} (1 - \frac{D_i}{T_i}) C_i + max_i\{C_i - 1\}}{1 - U} \tag{5.7}$$

$$B_2 = max \left(max_{i=1\ldots n}\{D_i\}, \frac{\sum_{i=1}^{n} (1 - \frac{D_i}{T_i}) C_i}{1 - U} \right) \tag{5.8}$$

L é definido como sendo o período de ocupação do processador, ou seja, é um intervalo de tempo que o processador é mantido continuamente ocupado executando instâncias pendentes. Formalmente L é o ponto fixo da computação iterativa definida pela equação 5.9 George et al. (1996). A iteração termina quando $L^m = L^{m+1}$.

$$L = \begin{cases} L^0 = \sum_{i=1}^{n} C_i \\ L^{m+1} = Wo(L^m) \end{cases} \tag{5.9}$$

Onde $Wo(t)$, formalizado pela equação 5.10, é a carga de trabalho do processador. Representa a quantidade de tempo de processamento requisitado por todas as instâncias cujos tempos de liberação estão no intervalo $[0, t)$ George et al. (1996).

$$Wo(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil C_i \tag{5.10}$$

O particionamento no regime não preemptivo em *EDF-np* utiliza o teste da equação 5.4 e é apresentado pelo algoritmo 4.

Algoritmo 4 Particionamento em regime não preemptivo; Entada: $\mathcal{T}_{np}$; Saída: m_{np} e Π_{np}

Ordenar $\mathcal{T}_{np}$ em ordem não decrescente de período (T_i);
$m_{np} \leftarrow 1; i \leftarrow 1$;
while $\mathcal{T}_{np} \neq \emptyset$ **do**
  {Atribui primeira tarefa ao próximo processador não preemptivo}
  $\pi_{m_{np}} \leftarrow \pi_{m_{np}} \cup \tau_i$
  {Remove tarefa de $\mathcal{T}_{np}$}
  $\mathcal{T}_{np} \leftarrow \mathcal{T}_{np} - \tau_i$
  {Enquanto for escalonável pela equação 5.4}
  while $\forall \tau_j \in \mathcal{T}_{\pi_{mnp}} : (U \leq 1) \wedge (\forall t \in S, t \geq h(t) + max_{D_i>t}\{C_i - 1\})$ **do**
    $\pi_{m_p} \leftarrow \pi_{m_p} \cup \tau_i$
    $\mathcal{T}_p \leftarrow \mathcal{T}_p - \tau_i$
    $i \leftarrow i + 1$
  end while
  $m_{np} \leftarrow m_{np} + 1$
end while

5.8 CONCLUSÃO

Para lidar com a contradição fundamental relacionada com a análise de viabilidade dos sistemas de tempo real com garantia e os limites de escalonabilidade, foi proposta uma abordagem de escalonamento heterogêneo. Nesta abordagem, tenta-se deixar o máximo de processadores em regime não preemptivo e, se necessário considerando os processadores disponíveis, transforma-se processadores para regime preemptivo. Como a preempção não é livre de efeitos colaterais, para as tarefas escalonadas em regime preemptivo deve ser considerado custos adicionais.

O algoritmo proposto é independente do particionamento e do escalonamento local dos processadores. O particionamento das tarefas é um problema NP-difícil e geralmente se utiliza de heurísticas que consideram ordenação e atribuição das tarefas em ordem aos processadores até o limite de escalonabilidade definido por um teste local para monoprocessador. Da mesma forma que os algoritmos locais são independentes, o método de incorporação do custo adicional de preempção também o é. No método proposto, por exemplo, é possível modelar os efeitos da preempção através de um novo cálculo do pior tempo de execução (maior) desativando ou limitando a agressividade dos elementos do processador (por exemplo: *cache*, execução especulativa). Tais elementos, se ativados no regime preemptivo e dependendo do processador, possivelmente tornariam a análise do sistema preemptivo inviável. De

qualquer forma, a solução heterogênea tenta conservar a precisão e segurança do pior tempo de computação das tarefas e ao mesmo tempo permite certo aumento da escalonabilidade.

Foi também apresentado um exemplo da abordagem proposta. Para o escalonamento preemptivo, utilizou-se o algoritmo de prioridade fixa DM-FFDU (*Deadline Monotonic – First Fit Decreasing Utilization*) que particiona as tarefas em ordem decrescente de utilização. O teste de escalonabilidade para o *Deadline Monotonic* considera um custo de preempção adicionando uma interferência extra somada ao tempo de execução das tarefas de prioridade mais alta. Apesar deste teste ser pessimista desprezando a possível usabilidade dos dados (se for considerado *cache*), é perfeitamente aceitável considerar toda recarga de *cache* se tal processador executa rotinas do sistema operacional e interrupções. Além disso, esta estratégia é facilmente adaptável se o algoritmo preemptivo fosse estendido para o modelo cooperativo (com pontos fixos de preempção). Já para o escalonamento não preemptivo optou-se pelo EDF não preemptivo sendo analisado pela estratégia de demanda de processador. EDF não preemptivo foi utilizado devido a baixa escalonabilidade dos algoritmos de prioridade fixa não preemptivos. Tarefas executadas em regime não preemptivo podem executar com todos os recursos (analisáveis pela ferramenta WCET) contemplando um melhor desempenho e conservação dos estados do processador. A escalonabilidade dos algoritmos não preemptivos é afetada pelas inversões de prioridade: tarefas com *deadline* mais próximo não podem executar até o término da execução da tarefa atual. Em todos os casos, a utilização da abordagem heterogênea requer apenas testes de escalonabilidade para os algoritmos preemptivos e não preemptivos.

Em termos de complexidade, a solução heterogênea é limitada pela complexidade dos algoritmos de particionamento não preemptivo, preemptivo e de classificação. Considerando que a complexidade dos algoritmos é x, y e z respectivamente ao particionamento não preemptivo, preemptivo e de classificação, a solução heterogênea possui complexidade $O[x+m(y+z)]$, com m igual ao número de processadores. No caso dos exemplos apresentados, os algoritmos de particionamento e classificação possuem complexidade pseudo polinomial ao número de tarefas e, portanto a solução heterogênea possui a mesma complexidade em relação às tarefas e linear em relação ao número de processadores.

6 AVALIAÇÃO DA ABORDAGEM PROPOSTA

No capítulo anterior foi descrita uma abordagem de escalonamento heterogêneo considerando regimes de preempção diferenciados. Nesta abordagem, em um mesmo sistema multiprocessado, há processadores que escalonam as tarefas em regime não preemptivo e outros que escalonam em regime preemptivo. O regime não preemptivo visa conservar as propriedades de precisão e segurança do cálculo do WCET enquanto o regime preemptivo tenta melhorar a escalonabilidade.

Neste capítulo é apresentada uma avaliação experimental da solução de escalonamento heterogêneo comparando-a com os algoritmos de escalonamento preemptivo e não preemptivo homogêneos. Para tal avaliação, considerou-se uma geração aleatória do conjunto das tarefas variando a utilização do sistema em pequenos passos até o máximo em relação a um número definido de processadores. Para cada sistema gerado, verifica-se o número de sistemas escalonáveis para os três algoritmos: preemptivo, não preemptivo e heterogêneo.

O objetivo desta avaliação é demonstrar os limites de utilização da solução heterogênea e comprovar que existem casos onde esta estratégia pode escalonar sistemas adicionais comparada com algoritmos homogêneos mesmo com uma heurística simples de classificação de tarefas.

O capítulo está organizado em uma seção sobre a condição dos experimentos e geração das tarefas, outra sobre seleção do critério de ordenação para o particionamento, resultados individuais do escalonamento heterogêneo, comparação entre os algoritmos homogêneos e o heterogêneo, e finalmente as conclusões.

6.1 CONDIÇÕES DOS EXPERIMENTOS

Para a avaliação e comparação da solução heterogênea, foram criados dois cenários: 4 processadores e 20 tarefas e 16 processadores e 80 tarefas. Estes cenários estão baseados nos trabalhos de Davis e Burns (2011) e Oliveira et al. (2012) e são exemplos que representam as condições geralmente utilizadas em avaliações empíricas de algoritmos de escalonamento. Varia-se a utilização no intervalo de $[0,1;4]$ para o cenário 4 processadores e $[0,4;16]$ para 16 processadores. Em ambos os casos utiliza-se 40 pontos de verificação de utilização, ou seja, incremento de $0,1$ e $0,4$ para os cenários de 4 e 16 processadores respectivamente.

O conjunto de tarefas com utilização pré-determinada é gerado utili-

zando o algoritmo de distribuição *UUnifast-Discard* Davis e Burns (2009a), gerando uma distribuição imparcial. Este algoritmo é baseado no trabalho de Bini e Buttazzo (2005) com adaptações para geração de conjuntos de tarefas com utilização maior que 1 para ambientes multiprocessados. Segundo Emberson et al. (2010), o *UUnifast-Discard* pode ser ineficiente dependendo do número de tarefas e utilização. Entretanto, para o contexto desta dissertação de mestrado, o método descrito por Davis e Burns (2009a) mostrou-se satisfatório. O intervalo mínimo de chegadas (T_i) foi gerado de acordo com uma distribuição uniforme logarítmica com um fator de 1.000 entre os valores máximos e mínimos, de $1ms$ a 1 segundo. A distribuição uniforme logarítmica foi utilizada pois gera um número igual de tarefas em cada intervalo logarítmico de tempo (1-10ms, 10-100ms, etc). Os *deadlines* das tarefas foram atribuídos de acordo com uma distribuição uniforme aleatória dentro do intervalo $[C_i; T_i]$ resultando em conjuntos de tarefas com *deadlines* limitados. O tempo de execução das tarefas é calculado individualmente através da utilização e do intervalo mínimo de chegadas: $C_i = U_i.T_i$.

Desta forma para cada ponto de verificação dentro do intervalo de teste de cada cenário, gera-se 1.000 sistemas (um sistema é um conjunto de 20 ou 80 tarefas) que devem ser escalonados em, no máximo, 4 ou 16 processadores. Assim, para a utilização de $0,1$ gera-se 1.000 conjuntos de tarefas, para $0,2$ outros 1.000 conjuntos e assim por diante. No total, as avaliações dos algoritmos possuem 40 pontos de utilização com 1.000 conjuntos de tarefas por ponto de avaliação, gerando um total de 40.000 sistemas diferentes para cada cenário.

Os conjuntos de tarefas foram gerados apenas uma vez para todas as avaliações e comparações dos algoritmos que serão apresentadas nas próximas seções. Estes testes foram implementadas em linguagem C formando um *framework* de tempo real. O *framework* apresenta implementação dos testes de escalonabilidade utilizados conforme a seção 5.7 do capítulo 5, bem como a heurística de particionamento e os algoritmos da solução heterogênea. A verificação do desempenho dos algoritmos para os 40.000 sistemas leva em torno de 5 minutos na implementação em C do *framework* em um máquina *desktop* recente.

6.2 ESCOLHA DA ORDENAÇÃO PARA OS ALGORITMOS DE PARTICIONAMENTO

Conforme apresentado no capítulo anterior, o problema do particionamento é resolvido pela heurística *Next-Fit*, onde há uma ordenação inicial das tarefas e estas são atribuídas aos processadores em ordem de acordo com

a capacidade seguindo os testes de escalonabilidade para monoprocessador (preemptivo ou não preemptivo).

Como o particionamento das tarefas é um problema combinatório (semelhança com o *bin-packing*), a ordenação inicial das tarefas tem grande impacto sobre a taxa de escalonabilidade do algoritmo. Define-se a taxa de escalonabilidade como a relação entre os sistemas considerados escalonáveis em m processadores ($m = 4$ e $m = 16$) e a quantidade total de sistemas (1.000) dentro do ponto de utilização em teste.

O objetivo desta seção é verificar empiricamente, dentro dos conjuntos de tarefas gerados pelas condições dos experimentos, qual é a melhor classificação inicial dos algoritmos preemptivos e não preemptivos abordados neste trabalho. Este levantamento é importante devido à influência da ordenação no particionamento e as comparações das soluções homogêneas (somente preemptivo e somente não preemptivo) e a heterogênea deve ser realizada considerando a melhor taxa de escalonabilidade do algoritmo, ou seja, tenta-se evitar a influência direta da ordenação.

Portanto, neste estudo, avaliou-se a escalonabilidade ordenando inicialmente as tarefas nos seguintes critérios:

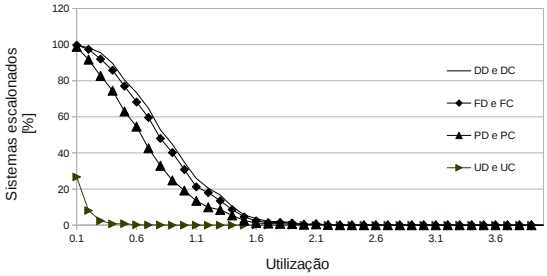
- *deadline* relativo não decrescente (sigla DC – *deadline* relativo crescente);
- *deadline* relativo não crescente (sigla DD – *deadline* relativo decrescente);
- folga (*laxity*: $D_i - C_i$) não decrescente (sigla FC – folga crescente);
- folga (*laxity*: $D_i - C_i$) não crescente (sigla FD – folga decrescente);
- período (intervalo mínimo de chegadas) não decrescente (sigla PC – período crescente);
- período (intervalo mínimo de chegadas) não crescente (sigla PD – período decrescente);
- utilização não decrescente (sigla UC – utilização crescente).
- utilização não crescente (sigla UD – utilização decrescente);

Por convenção, as ordenações são sempre "não decrescentes" e "não crescentes" por admitirem valores iguais. Com objetivo de facilitar a leitura dos gráficos, "não decrescente" é representada pela letra "C" (crescente) e "não crescente" é representada pela letra "D" (decrescente). Vale ressaltar também que o limite teórico de pior caso do escalonamento particionado é a

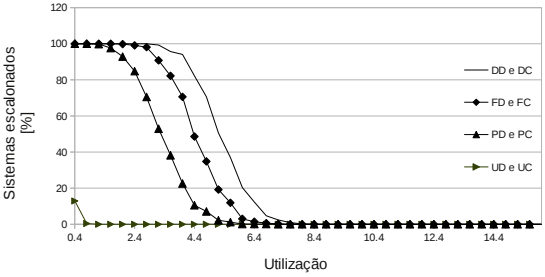
utilização $\frac{m+1}{2}$, ou seja, as curvas de taxa de escalonabilidade das próximas seções tenderão a 0 após uma utilização de 2,5 e 8,5 dos sistemas de 4 e 16 processadores respectivamente.

6.2.1 Seleção para escalonamento não preemptivo

Nos gráficos da figura 21 é apresentada a taxa de escalonabilidade em função da utilização para o particionamento homogêneo utilizando o algoritmo EDFnp – *Earliest Deadline First* não preemptivo no escalonamento local. A figura 21(a) representa o cenário com 4 processadores (20 tarefas) e a figura 21(b) com 16 processadores (80 tarefas).



(a) EDFnp, 4 cpus.



(b) EDFnp, 16 cpus.

Figura 21: Resultados do escalonamento não preemptivo.

A primeira conclusão em relação a este particionamento é que não há diferença, em termos de sistemas escalonados, relacionada à ordenação “não

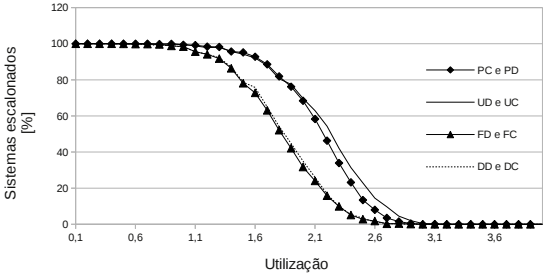
decrescente" e "não crescente" para um mesmo critério de ordenação nestes ensaios. Verificou-se que a alocação das tarefas nos processadores é completamente diferente dependendo se há ordenação crescente ou decrescente para o mesmo critério, mas a taxa de escalonabilidade dos 1.000 sistemas em um mesmo ponto de utilização é sempre muito parecida para todos os conjuntos de tarefas gerados neste trabalho. As curvas crescente e decrescente também são muito parecidos devido a utilização do particionamento utilizando a heurística *Next-Fit* que não permite outros processadores anteriores sejam utilizados a partir do momento que a um única tarefa não é escalonada durante o particionamento. Por estas razões crescente e decrescente são representadas por uma única curva nos gráficos da figura 21.

Com relação aos critérios de ordenação, percebe-se pelos gráficos da figura 21 que ordenar inicialmente o conjunto das tarefas por *deadline* relativo chega-se em uma melhor taxa de escalonabilidade para o particionamento em EDFnp.

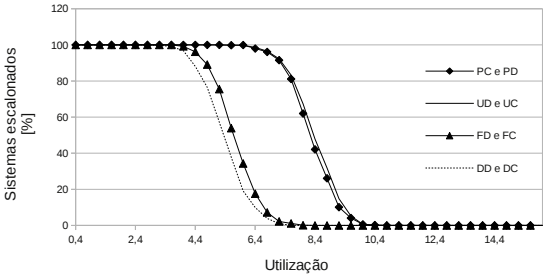
6.2.2 Seleção para escalonamento preemptivo

Nos gráficos da figura 22 é apresentada a taxa de escalonabilidade em função da utilização para o particionamento homogêneo utilizando o algoritmo *Deadline Monotonic* preemptivo no escalonamento local. A figura 22(a) representa o cenário com 4 processadores (20 tarefas) e a figura 22(b) com 16 processadores (80 tarefas).

Neste ensaio do escalonamento preemptivo, utilizou-se um custo adicional de preempção γ igual a 20% do tempo de computação das tarefas, ou seja, $\gamma = 0,2 \times C_i$. O valor escolhido de γ é exemplo hipotético e arbitrário de custo adicional de preempção proporcional ao tempo de computação. Tenta-se refletir nos 20% adicionais possíveis penalidades da preempção (*cache*, salvamento de contextos, etc) como uma interferência adicional no cálculo do tempo de resposta tendo estes custos uma certa relação com o tempo de computação da tarefa. Em um sistema real, o valor estimado de γ pode ser obtido, por exemplo, durante a análise de fluxo de dados das tarefas considerando o pior ponto de preempção em relação à usabilidade futura dos dados em *cache* ou o ponto com maior conjunto de trabalho de dados (WSS – *Working set size*). O problema dos custos de preempção é dependente da arquitetura do processador e das tarefas do sistema tendo uma relação com a obtenção do pior tempo de computação das tarefas (WCET) o que não faz parte do escopo deste trabalho. O valor de 20% é razoável considerando relatos de medições e análise feitas em sistemas reais (LEE et al., 1998), (LEE et al., 1998), (SEBEK, 2002).



(a) DM, 4 cpus.



(b) DM, 16 cpus.

Figura 22: Resultados do escalonamento preemptivo, $\gamma = 20\%$ de C_i

Da mesma forma que o particionamento não preemptivo, não há diferença significativa relacionada à ordenação "não decrescente" e "não crescente" para um mesmo critério e por isso estas ordenações são representas por uma mesma função no gráfico. Novamente, "não decrescente" e "não crescente' impactam na alocação mas possuem pouca influência na taxa de escalonabilidade. As curvas crescente e decrescente também são muito parecidos devido a utilização do particionamento utilizando a heurística *Next-Fit*.

Com relação ao critério de ordenação, percebe-se pelos gráficos da figura 22 que ordenar inicialmente o conjunto de tarefas por utilização para fins de alocação chega-se em uma melhor taxa de escalonabilidade para o particionamento quando *Deadline Monotonic* é usado como algoritmo de escalonamento local.

6.3 RESULTADOS PARA ESCALONAMENTO HETEROGÊNEO

O escalonamento heterogêneo deste trabalho utiliza os algoritmos *Deadline Monotonic* para o escalonamento local preemptivo, *Earliest Deadline First* para o escalonamento local não preemptivo e uma heurística de classificação simples para definir quais tarefas serão executadas em cada regime. A heurística de classificação adotada transforma os últimos processadores não preemptivos para o regime preemptivo após o particionamento inicial de todo o conjunto de tarefas em regime não preemptivo. A transformação somente é realizada se o sistema não for escalonado no limite dos processadores de cada cenário, ou seja, 4 e 16 processadores. O objetivo do escalonamento heterogêneo, além de tentar escalonar tarefas em regime não preemptivo, é tentar escalonar todo conjunto das tarefas no mesmo limite de processadores do particionamento homogêneo.

Conforme apresentado na seção anterior, no particionamento preemptivo a melhor taxa de escalonabilidade é obtida pela ordenação por utilização. Portanto, para o escalonamento heterogêneo, quando processadores são transformados em regime preemptivo, todas as suas tarefas são reordenadas em utilização "não crescente" ou "não decrescente". Como vários ensaios realizados comprovaram que não há diferença quanto a este critério, utilizar-se-á somente a utilização "não crescente" (UD) no particionamento preemptivo. Além disso, quando uma tarefa é escalonada pelo regime preemptivo, é considerado o custo adicional de preempção $\gamma = 20\%$ de C_i igualmente à seção anterior sobre os resultados do particionamento homogêneo preemptivo.

No caso do particionamento não preemptivo que é realizado inicialmente na estratégia heterogênea, a ordenação "não crescente" e "não decrescente" pode comprometer a escalonabilidade do sistema. A estratégia de classificação adotada migra os últimos processadores para regime preemptivo, portanto é muito conveniente classificar as tarefas em ordem na qual beneficie o escalonamento não preemptivo dos primeiros processadores deixando tarefas problemáticas por último para o escalonamento preemptivo. Após inúmeros ensaios realizando permutações da ordenação inicial das tarefas, comprovou-se que a ordenação "não crescente" por *deadline* (DD) é a mais adequada para o escalonamento heterogêneo para alocação em regime não preemptivo. Nesta ordenação as tarefas com *deadlines* curtos que são problemáticas para o escalonamento não preemptivo devido ao bloqueio inerente à não preempção, serão escalonadas pelo regime preemptivo quando necessário. Desta forma, na solução heterogênea tem-se:

- ordenação em *deadline* "não crescente" para a alocação em regime não preemptivo;

- ordenação em utilização "não crescente" para a alocação em regime preemptivo;

- interferência adicional de preempção $\gamma = 20\%$ de C_i quando as tarefas executam em regime preemptivo.

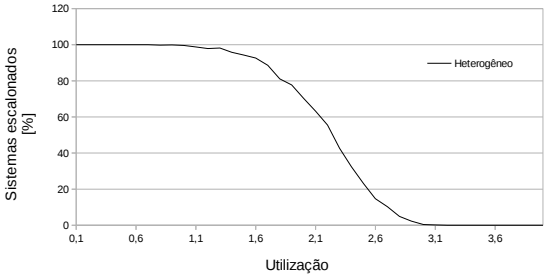
Na figura 23 são apresentados os resultados para o escalonamento heterogêneo considerando o cenário de 4 processadores e 20 tarefas. Na figura 23(a) é apresentada a taxa de escalonabilidade em função da utilização e na figura 23(b) é apresentado a porcentagem de processadores de cada regime. Com utilização baixa, todo o sistema é considerado viável utilizando apenas o regime não preemptivo. Conforme a utilização vai aumentando, não é possível garantir a viabilidade somente no regime não preemptivo e os sistemas começam a apresentar processadores executando tarefas em regime preemptivo. A queda brusca do número de processadores em regime preemptivo acontece quando o sistema não é mais escalonável em 16 processadores, ou seja, não é mais viável nem com 100% dos processadores em regime preemptivo.

Quanto ao cenário de 16 processadores e 80 tarefas, seus resultados estão apresentados na figura 24. Na figura 24(a) é apresentada a taxa de escalonabilidade em função da utilização e na figura 24(b) é apresentado a porcentagem de processadores de cada regime. Da mesma forma que o cenário de 4 processadores, conforme a utilização vai aumentando há necessidade de escalonar mais tarefas em regime preemptivo para garantir a escalonabilidade do sistema. A queda brusca dos processadores em *Deadline Monotonic* (DM) acontece quando o sistema não é mais escalonável em 16 processadores nem mesmo somente em regime preemptivo.

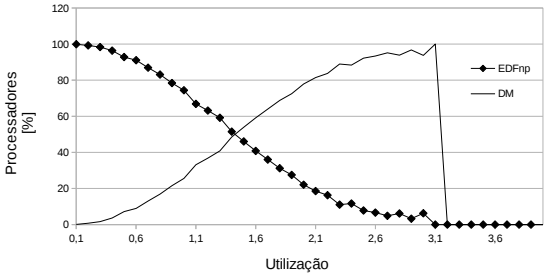
6.4 COMPARAÇÃO ENTRE MODELO PREEMPTIVO, NÃO PREEMPTIVO E HETEROGÊNEO

Esta seção apresenta uma comparação direta em termos de escalonabilidade entre os particionamentos homogêneos considerados e a solução heterogênea.

Na figura 25 apresenta-se uma comparação entre os particionamentos homogêneos (somente EDFnp e DM) e a solução heterogênea para 4 processadores e 20 tarefas. Considera-se um custo adicional de preempção $\gamma = 20\%$ de C_i para todas as tarefas que executarem em regime preemptivo (heterogêneo e homogêneo). Na figura 25(a) há as curvas de escalonabilidade em função da utilização para os três algoritmos, destacando que a solução heterogênea apresenta um desempenho levemente melhor na taxa de escalona-



(a) Escalonabilidade.



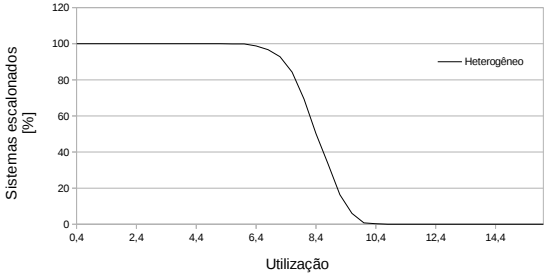
(b) Processadores de cada regime.

Figura 23: Resultados do escalonamento heterogêneo para 4 processadores, $\gamma = 20\%$ de C_i

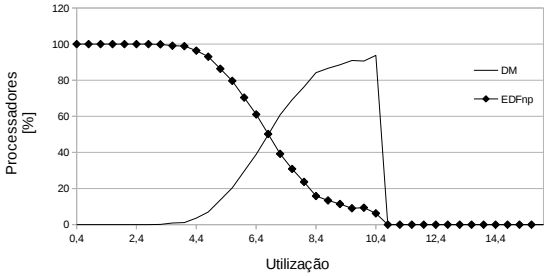
bilidade em relação ao particionamento homogêneo totalmente preemptivo. Esta melhora de desempenho é mais visível na figura 25(b) onde a curva representada é a diferença das curvas de escalonabilidade do escalonamento heterogêneo e escalonamento em *Deadline Monotonic* homogêneo que estão quase sobrepostos na figura 25(a).

Na figura 26 é apresentada a mesma comparação dos algoritmos homogêneos e heterogêneo para o cenário de 16 processadores e 80 tarefas. Novamente a solução heterogênea proposta apresenta um desempenho levemente superior comparado com a solução homogênea totalmente preemptiva (DM). Na figura 26(b) é mostrado a diferença entre as curvas do escalonamento heterogêneo e o escalonamento preemptivo comprovando o melhor desempenho da solução heterogênea em termos de escalonabilidade.

É possível concluir que o escalonamento heterogêneo apresenta um



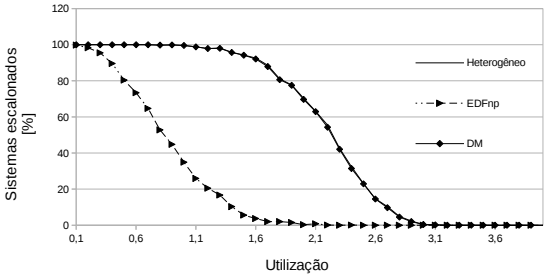
(a) Escalonabilidade.



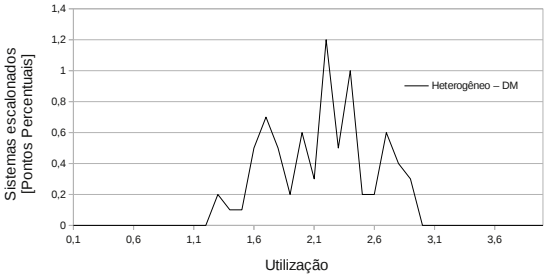
(b) Processadores de cada regime.

Figura 24: Resultados do escalonamento heterogêneo para 16 processadores, $\gamma = 20\%$ de C_i

desempenho sucintamente melhor principalmente no limite de escalonabilidade do particionamento ($\frac{m+1}{2}$), ou seja, a solução heterogênea consegue escalonar sistemas adicionais do que o *Deadline Monotonic* particionado. Se os picos de melhor desempenho das curvas de diferença (figuras 25(b) e 26(b)) forem comparados com a porcentagem de processadores de cada regime da solução heterogênea (figuras 23(b) e 24(b)) percebe-se que este aumento de sistemas escalonados existe devido a uma certa quantidade de processadores em regime não preemptivo. Ou seja, o pico de melhor desempenho da figura 26(b), utilização de 8.4, possui cerca 16% dos processadores em regime não preemptivo (figura 24(b)) dos 1.000 sistemas analisados quando estes são escalonáveis. Além disso, percebe-se que o algoritmo tenta conservar as propriedades do WCET mantendo certo número de processadores em regime não preemptivo.



(a) Escalonabilidade em função da utilização.



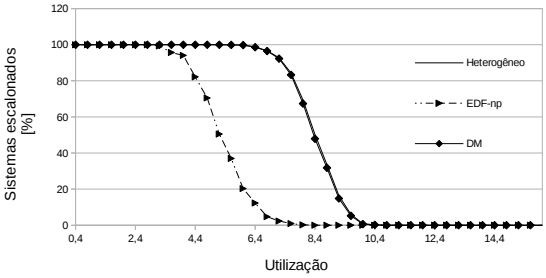
(b) Diferença entre as curvas de escalonabilidade entre heterogêneo e DM.

Figura 25: Comparação entre particionamento homogêneo e heterogêneo para 4 processadores com $\gamma = 20\%$ de C_i

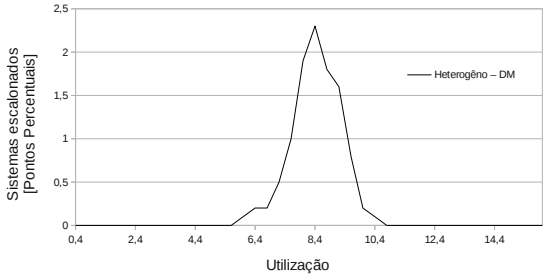
6.5 CONCLUSÕES

Este capítulo apresentou a avaliação experimental da solução de escalonamento heterogêneo proposta no capítulo anterior. Para tal avaliação foram criados dois cenários de teste com 4 e 16 processadores e com 20 e 80 tarefas respectivamente através do algoritmo UUnifast. O UUnifast, como apontado pela literatura, fornece conjuntos de tarefas imparciais, o que é extramente importante para a avaliação empírica dos algoritmos de escalonamento.

O escalonamento particionado é um problema combinatório que neste trabalho foi resolvido através da heurística *First Fit* resultando em uma solução sub-ótima mas com baixa complexidade computacional. Esta solução de



(a) Escalonabilidade em função da utilização.



(b) Diferença entre as curvas de escalonabilidade entre heterogêneo e DM.

Figura 26: Comparação entre particionamento homogêneo e heterogêneo para 16 processadores com $\gamma = 20\%$ de C_i

particionamento requer uma ordenação inicial das tarefas e pode-se dizer que tal ordenação tem grande impacto na taxa de escalonabilidade dos algoritmos. Por isso, o estudo desenvolvido para ordenações por *deadline*, folga, período e utilização (todos "não decrescente" e "não crescente") foram importantes para determinar-se a melhor escalonabilidade dos algoritmos homogêneos e também no desenvolvimento do heterogêneo. Além disso, o impacto da ordenação para alocação pode mascarar o desempenho de um dado algoritmo fornecendo resultados tendenciosos, o que foi evitado neste trabalho utilizando sempre a configuração com melhor desempenho nas comparações e conclusões. O mesmo fator de custo de preempção também foi conservado em todas as avaliações.

A heurística adotada que classifica quais tarefas e processadores são

regidos pelo regime preemptivo e não preemptivo é realmente bem simples. Simplesmente há uma conversão dos últimos processadores, e consequentemente suas tarefas, para o regime preemptivo até que o sistema seja ou não escalonado. Pode-se concluir que, mesmo utilizando-se de uma heurística simples de classificação entre regimes, consegue-se atingir o objetivo de escalonamento e até escalonar alguns sistemas adicionais. Pode-se dizer que, dentro dos algoritmos avaliados, existem sistemas que somente são viáveis se o escalonamento for heterogêneo.

Outra conclusão importante é que a solução heterogênea tenta conservar processadores em regime não preemptivo. Ocorre conversão entre regimes somente quando o sistema não é viável dentro do limite de processadores disponíveis (4 e 16). Este é um fato importante, pois escalonar tarefas em regime não preemptivo garante as propriedades do WCET. A obtenção de WCET preciso e seguro é uma tarefa complicada. Um escalonamento que tenta conservar estas características é importante para os sistemas de tempo real principalmente enquanto houver a contradição entre obtenção do WCET e o escalonamento o que é muito aparente na arquitetura dos processadores atuais.

7 CONSIDERAÇÕES FINAIS

Cada vez mais as aplicações de tempo real necessitam de capacidade de processamento. A demanda de funcionalidade destas aplicações é crescente e, consequentemente, há aumento da complexidade do *software* que necessita de *hardware* poderoso. Os processadores para estas aplicações sofisticadas não são mais simples, incorpora-se múltiplos núcleos em um único *chip*, elementos especulativos e arquiteturas de memória complexas. Pode-se perceber que o desenvolvimento de processadores multiprocessados está muito à frente em relação aos esforços de pesquisa que determinam os melhores mecanismos, políticas e análises para tais sistemas. No melhor dos casos, gera-se sistemas superdimensionados e caros, e no pior dos casos têm-se sistemas com falhas temporais que comprometem sua confiabilidade.

A diferença existente entre o desenvolvimento dos processadores e as técnicas de análise é muito visível na contradição entre o cálculo do WCET e os algoritmos de escalonamento. Conforme a literatura, o WCET obtido através de técnicas de interpretação abstrata é robusto e apresenta resultados precisos. Infelizmente, estas técnicas são limitadas ao número pequeno de plataformas e as que são suportadas apresentam dificuldades em relação a análise, especialmente quando se utiliza de processadores mais avançados com memória *cache* e elementos especulativos. E ainda, grande parte das pesquisas realizadas no campo de escalonamento de tempo real consideram a preempção e migração fundamentais para o desenvolvimento de escalas ótimas, mas estas considerações são incompatíveis com as premissas do cálculo do WCET.

Para tentar amenizar esta contradição, foi proposto neste trabalho uma solução de escalonamento heterogêneo. Neste modelo de escalonamento as tarefas são divididas em processadores com escalonamento em regime não preemptivo e preemptivo. O regime preemptivo escalona tarefas problemáticas para o não preemptivo e o objetivo deste último é tentar conservar as propriedades de segurança e precisão do cálculo do WCET. O modelo de tarefas adotado contempla tarefas esporádicas e é incorporado ao modelo um custo adicional quando há execução em regime preemptivo. O custo de preempção é um parâmetro flexível que depende do algoritmo de escalonamento bem como a arquitetura do sistema. Se a arquitetura permitir, pode-se considerar um tempo adicional de recarga de *cache* (como apresentado no teste de tempo de reposta do algoritmo *Deadline Monotonic* no capítulo 4) ou, como sugestão, pode-se desativar todos os elementos especulativos do processador e recalcular um novo WCET mais determinista. Não é do conhecimento do autor nenhum trabalho da literatura de tempo real em que uma aborda-

gem heterogênea foi utilizada no escalonamento de multiprocessadores como apresentado aqui.

Os algoritmos de escalonamento local também são independentes na solução apresentada. Qualquer algoritmo pode ser utilizado desde que se tenha um teste de escalonamento para monoprocessador devido a solução utilizar o escalonamento particionado. O particionamento é um problema difícil devido a sua semelhança com o *bin packing* e não há soluções ótimas. Contudo, após o particionamento das tarefas nos processadores, qualquer algoritmo e teste de escalonamento para monoprocessador pode ser utilizado o que incorpora duas vantagens básicas. A primeira é não ter custo adicionais de migração e nem manipulação de filas globais o que dificultaria ainda mais a utilização do parâmetro WCET. Outra vantagem, está relacionada ao fato dos algoritmos para monoprocessador estarem bem desenvolvidos com testes exatos e técnicas abrangentes.

As avaliações e comparações realizadas entre as abordagens homogêneas (preemptivo e não preemptivo) e a heterogênea mostraram que a solução heterogênea é promissora. Neste trabalho foi utilizada uma heurística simples de classificação entre regimes e, mesmo assim, pode-se escalonar sistemas adicionais em relação às soluções homogêneas. Pode-se concluir que, dentro dos algoritmos avaliados, há sistemas que somente são escalonados se for adotada a estratégia heterogênea e este melhor desempenho acontece quando há processadores executando tarefas no regime não preemptivo. É interessante notar que se a utilização do sistema for menor que o limite teórico do escalonamento ($\frac{m+1}{2}$), mais processadores permanecem no regime não preemptivo conservando as propriedades do WCET e cumprindo os objetivos do trabalho.

Como trabalhos futuros existem inúmeros aspectos que podem ser melhorados. Primeiramente, pode-se investigar outros métodos para incorporar os custos de preempção na escalonabilidade desenvolvendo ou melhorando ferramentas que utilizem técnicas de fluxo de dados em conjunto com a análise WCET. Com relação à solução heterogênea, pode-se investigar algoritmos mais avançados para classificar quais tarefas executam nos diferentes regimes como, por exemplo, utilizar técnicas ou meta heurísticas de otimização. Outros algoritmos de escalonamento para monoprocessador podem ser avaliados e ainda verificar o impacto na escalonabilidade se for utilizado escalonamento semi-preemptivo (com pontos fixos de preempção) ao invés do preemptivo. Pode-se ainda, ao invés de utilizar dois regimes de preempção (preemptivo e não preemptivo), utilizar três regimes como preemptivo, semi-preemptivo e não preemptivo.

Dos frutos deste trabalho, houve duas publicações nacionais referenciadas em (STARKE; OLIVEIRA, 2011a) e (STARKE; OLIVEIRA, 2011b).

A primeira trata do estudo realizado sobre a ferramenta WCET AiT apresentada no capítulo 3 e a segunda trata da da influência de um modo especial de operação dos processadores x86 nos sistemas de tempo real. Um novo artigo será submetido com o resultado principal deste trabalho, o qual ainda não foi publicado.

REFERÊNCIAS

ABSINT. *AbsInt Advanced Analyzerfor ARM7 – User Documentation, a3 Version: 10.08 Build 145451*. [S.l.]: AbsInt, 2010.

ABSINT. *aiT Worst-Case Execution Time Analyzers*. 2011. <http://www.absint.com/ait/>.

ARM. *ARM7 CPUs*. ARM, 2010. <http://www.arm.com/products/processors/classic/arm7/index.php>.

AUDSLEY, N. et al. Hard real-time scheduling: The deadline-monotonic approach. In: *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*. Citeseer, 1991. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.4438>.

BARUAH, S. K.; ROSIER, L. E.; HOWELL, R. R. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, v. 2, n. 4, p. 301–324, nov. 1990. ISSN 0922-6443. <http://www.springerlink.com/index/10.1007/BF01995675>.

BASTONI, A. Cache-Related Preemption and Migration Delays : Empirical Approximation and Impact on Schedulability. In: *Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. [s.n.], 2010. p. 33–34. <http://www.cs.unc.edu/ anderson/papers/ospert10.pdf>.

BASTONI, A.; BRANDENBURG, B. B.; ANDERSON, J. H. Is Semi-Partitioned Scheduling Practical? In: *2011 23rd Euromicro Conference on Real-Time Systems*. IEEE, 2011. p. 125–135. ISBN 978-1-4577-0643-1. <http://wwwx.cs.unc.edu/ bbb/papers/ecrts11.pdf http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6001775>.

BINI, E.; BUTTAZZO, G. Measuring the performance of schedulability tests. *Real-Time Systems*, Springer, v. 30, n. 1, p. 129–154, 2005. <http://www.springerlink.com/index/V4772171850TL379.pdf>.

BRANDENBURG, B. B.; CALANDRINO, J. M.; ANDERSON, J. H. On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. *2008 Real-Time Systems Symposium*, Ieee, p. 157–169, nov. 2008.

BRIL, R. J.; LUKKIEN, J. J.; VERHAEGH, W. F. Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with Deferred Preemption Revisited. *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, Ieee, p. 269–279, jul. 2007. ISSN 1068-3070.

BRYANT, R. E.; HALLARON, D. R. O. *Computer Systems: A Programmer's Perspective*. New Jersey: Prentice Hall, 2003. ISBN 0-13-034074-X.

BUI, B. D. et al. Impact of Cache Partitioning on Multi-tasking Real Time Embedded Systems. *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Ieee, p. 101–110, ago. 2008.

BURNS, A. Preemptive priority based scheduling: An appropriate engineering approach. In: *Advances in Real-Time Systems*. [S.l.]: Prentice-Hall, Inc., 1994. v. 225, p. 248. ISBN 0-13-083348-7.

Busquets Mataix, J. et al. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. *Proceedings Real-Time Technology and Applications*, IEEE Comput. Soc. Press, p. 204–212, 1996.

BUTTAZZO, G. C. *Hard Real-Time Computing Systems: Predictable Scheduling Algotithms and Applications*. 2nd. ed. [S.l.]: Springer, 2005. ISBN 0-387-23137-4.

CALANDRINO, J. et al. LITMUSˆRT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, Ieee, p. 111–126, dez. 2006. ISSN 1052-8725.

CARPENTER, J. et al. A categorization of real-time multiprocessor scheduling problems and algorithms. In: *Handbook on Scheduling Algorithms, Methods, and Models*. [S.l.]: Chapman Hall/CRC, Boca, 2004. cap. 31.

CULLMANN, C. et al. Predictability considerations in the design of multi-core embedded systems. In: *Proceedings of Embedded Real Time Software and Systems*. [S.l.: s.n.], 2010.

DAVIS, R.; BURNS, A. On Optimal Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems. In: TECHNICAL REPORT YCS-2010-451, DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF YORK. *2009 30th IEEE Real-Time*

Systems Symposium. IEEE, 2009. p. 398–409. ISBN 978-0-7695-3875-4. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5368139>.

DAVIS, R. I.; BURNS, A. A Survey of Hard Real-Time Scheduling Algorithms and Schedulability Analysis Techniques for Multiprocessor Systems. *Systems Research*, 2009.

DAVIS, R. I.; BURNS, A. FPZL Schedulability Analysis. *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, Ieee, p. 245–256, abr. 2011. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5767115>.

EMBERSON, P.; STAFFORD, R.; DAVIS, R. I. Techniques For The Synthesis Of Multiprocessor Tasksets. In: *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*. Brussels, Belgium: [s.n.], 2010. p. 6–11.

FARINES, J.-M.; FRAGA, J. d. S.; OLIVEIRA, R. S. de. *Sistemas de Tempo Real*. São Paulo: IME-USP, 2000.

FERDINAND, C.; WILHELM, R. Fast and Efficient Cache Behavior Prediction. In: *Proceedings of the Estonian Academy of Sciences, Engineering*. [S.l.]: Estonian Academy Publishers, 1998. v. 4, n. 2, p. 69–88.

GEORGE, L.; RIVIERRE, N.; SPURI, M. *Preemptive and Non-Preemptive Real-Time Uni- Processor Scheduling*. [S.l.], 1996.

HACKENBERG, D.; MOLKA, D.; NAGEL, W. E. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42*, ACM Press, New York, New York, USA, p. 413, 2009.

HECKMANN, R. et al. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, v. 91, n. 7, p. 1038–1054, jul. 2003. ISSN 0018-9219. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1215685>.

Intel Corporation. *An Introduction to the Intel QuickPath Interconnect*. [S.l.], 2009. 1–22 p.

JEFFAY, K.; STANAT, D.; MARTEL, C. On non-preemptive scheduling of period and sporadic tasks. *Proceedings Twelfth Real-Time Systems Symposium*, IEEE Comput. Soc. Press, p. 129–139, 1991. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=160366>.

JU, L.; CHAKRABORTY, S.; ROYCHOUDHURY, A. Accounting for Cache-Related Preemption Delay in Dynamic Priority Schedulability Analysis. *2007 Design, Automation & Test in Europe Conference & Exhibition*, Ieee, p. 1–6, abr. 2007.

KATO, S.; YAMASAKI, N. Semi-partitioned Fixed-Priority Scheduling on Multiprocessors. *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, Ieee, p. 23–32, abr. 2009.

KIRK, D. Process dependent static cache partitioning for real-time systems. *Proceedings. Real-Time Systems Symposium*, IEEE Comput. Soc. Press, p. 181–190, 1988.

LEE, C.-g. et al. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, v. 27, n. 9, p. 805–826, 2001. ISSN 00985589.

LEE, C.-g. et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, v. 47, n. 6, p. 700–713, jun. 1998. ISSN 00189340.

LIU, J. W. S. *Real-Time Systems*. 1. ed. [S.l.]: Prentice Hall, 2000. ISBN 0-13-099651-3.

MARR, D. et al. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, v. 6, n. 1, p. 4–15, 2002. ISSN 1535-766X.

MARTIN, T. *The Insider's Guide To The NXP LPC2000 Based Microcontrollers*. 1. ed. [S.l.]: Hitex (UK), 2007. 3000 p.

MUELLER, F. Compiler Support for Software-Based Cache Partitioning 1 Introduction 2 Software-Based Cache Parti- tioning. *Workshop on Languages, compilers, and tools for real-time systems*, v. 30, n. 11, p. 125–133, 1995.

OH, Y.; SON, S. Fixed-priority scheduling of periodic tasks on multiprocessor systems. *Department of Computer Science, University of Virginia, Tech. Rep. CS-95-16*, Citeseer, p. 1–37, 1995.

OLIVEIRA, R. S. de; CARMINATI, A.; STARKE, R. A. On Using Adversary Simulators to Obtain Tight Lower Bounds for Response Times. *SAC'2012 - The ACM 27th Symposium On Applied Computing*, 2012.

SCHOEBERL, M. Time-predictable cache organization. In: *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*. [S.l.: s.n.], 2009.

SCHOEBERL, M. Time-Predictable Computer Architecture. *EURASIP Journal on Embedded Systems*, v. 2009, p. 1–17, 2009. ISSN 1687-3955.

SCHOEBERL, M. *JOP - Java Optimized Processor*. 2011. <http://www.jopdesign.com/>.

SEBEK, F. *The real cost of task pre-emptions - measuring real-time-related cache performance with a HW / SW hybrid technique*. [S.l.], 2002. 12 p.

SEHN, J. P.; LIPASTI, M. H. *Modern Processor Design: Fundamentals of Superscalar Processors*. New York: McGraw-Hill, 2005. ISBN 0-07-059033-8.

SHA, L. et al. Real time scheduling theory: A historical perspective. *Real-time systems*, Springer, v. 28, n. 2, p. 101–155, 2004.

SHORT, M. The Case For Non-preemptive , Deadline-driven Scheduling In Real-time Embedded Systems. *Lecture Notes in Engineering and Computer Science*, v. 2183, n. 1, p. 399–404, 2010.

SODAN, A. C. et al. Parallelism via Multitrehaded and multicore cpus. *Computer*, 2010.

STARKE, R. A.; OLIVEIRA, R. S. de. Empirical Study about Using aiT Tool in WCET Estimation. In: *2011 Brazilian Symposium on Computing System Engineering*. IEEE, 2011. p. 86–89. ISBN 978-0-7695-4641-4. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6114835>.

STARKE, R. A.; OLIVEIRA, R. S. de. Impact of the x86 System Management Mode in Real-Time Systems. In: *2011 Brazilian Symposium on Computing System Engineering*. IEEE, 2011. p. 151–157. ISBN 978-0-7695-4641-4. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6114850>.

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.

WOLFE, A. Software-based cache partitioning for real-time applications. *Journal of Computer and Software Engineering*, v. 2, n. 3, p. 315–327, 1994.

YAO, G.; BUTTAZZO, G.; BERTOGNA, M. Bounding the Maximum Length of Non-preemptive Regions under Fixed Priority Scheduling. *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Ieee, n. 216008, p. 351–360, ago. 2009.

YAO, G. et al. Feasibility Analysis under Fixed Priority Scheduling with Fixed Preemption Points. *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, p. 71–80, 2010.

APÊNDICE A – Arquitetura de processadores contemporâneos – versão expandida

O primeiro microprocessador, Intel 4004, foi introduzido em 1971, consistindo em uma máquina de 4-*bits* com aproximadamente 2300 transistores. Sua aplicação inicial era a construção de calculadoras. Em termos de evolução, entre 1970 a 1980 os processadores continham de 2 mil a 100 mil transistores, uma frequência entre 0.1 a 3Mhz e 0,1 instruções a cada ciclo de máquina. Atualmente os microprocessadores contém mais de 100 milhões de transistores, frequências superiores a 2Ghz e 0.9 a 1.9 instruções a cada ciclo de máquina.

O projeto e a especificação de um microprocessador envolve a arquitetura do conjunto de instruções, que especifica quais instruções o processador é capaz de executar. Atualmente a implementação do *hardware* é descrita através de uma linguagem de descrição de *hardware – Hardware Description Language* (HDL). O HDL incorpora lógica booleana simples (*logic gates*) e *flip-flops*, como também estruturas mais complexas como decodificadores e multiplexadores. Com essas estruturas primitivas pode-se elaborar módulos funcionais inteiros como circuitos somadores e multiplicadores. Exemplos de uma implementação de arquitetura de microprocessador utilizando HDL pode ser encontrada em (BRYANT; HALLARON, 2003).

Textos sobre arquitetura de computadores definem três níveis de abstração (SEHN; LIPASTI, 2005): arquitetura, implementação e realização.

Arquitetura é conhecida como arquitetura de conjunto de instruções – *Instruction Set Architecture* (ISA) – que especifica o conjunto de instruções que caracteriza o comportamento fundamental do processador. Todo o *software* deve ser codificado em um ISA correspondente a tal máquina. Alguns exemplos de ISA são: Intel IA32, AMD x86_64, PowerPC e ARM.

Uma implementação especifica o projeto da arquitetura, referenciado como microarquitetura. Exemplos de implementação são PowerC 604 e Intel P6. Atributos como projeto do *pipeline*, memórias *cache* e previsão de fluxo (*branch prediction*) fazem parte da implementação.

A realização de uma implementação consiste no encapsulamento físico do projeto. No caso dos microprocessadores, esse encapsulamento geralmente é um *chip*. As realizações podem variar em termos de frequência, capacidade de memória *cache*, barramentos, tecnologia de fabricação e empacotamento. Atributos relacionados à realização incorporam o *die size*, encapsulamento, potência, refrigeração e cofiabilidade.

O ISA é definido como sendo um contrato entre o *software* e o *hardware* que torna o desenvolvimento de programas e máquinas uma tarefa independente. Desta forma, programas escritos em dado ISA podem rodar em diversas implementações com diferentes níveis de desempenho, aumentando a longevidade do *software*. Contudo ISAs com uma grande quantidade de *softwares* já desenvolvidos tendem a ficar no mercado por um bom tempo e

um exemplo disso é o ISA Intel IA32.

Servindo como base no desenvolvimento de *softwares* ou compiladores, o ISA também serve como especificação de processadores. No lançamento de uma nova microarquitetura, esta deve ser validada para que se garanta todos os requisitos funcionais, bem como que *softwares* já existentes possam ser executados. Um ISA define um conjunto de instruções chamadas instruções *assembly*, sendo que cada instrução define um operador e um ou mais operandos. ISAs têm sido diferenciados de acordo com o número de operandos que possam ser explicitamente especificados em cada instrução. Alguns ISAs usam um acumulador como operando implícito, onde este é usado sempre como fonte ou destino. Outros ISAs assumem que operandos estão contidos em uma pilha formando um estrutura LIFO – *last in, first out*, sendo que estruturas e operandos são operados no topo da pilha. Esta estrutura de pilha é utilizada, por exemplo, em unidades de ponto flutuante – *Floating Point Units* (FPU). Contudo, ISAs modernos consideram que os operandos estão gravados em registradores especiais formando o *multientry register file* do processador. Todas as operações aritméticas e lógicas são realizadas sobre estes registradores. Instruções especiais relacionadas com a memória como *load* e *store* são responsáveis em mover os operandos dos registradores à memória principal e vice-versa.

A.1 PRINCÍPIOS DE *PIPELINE*

A utilização de *pipeline* é uma técnica poderosa para aumentar o desempenho do sistema sem uma replicação massiva de *hardware*. O Intel i486 foi a primeira q implementação da p arquitetura IA32 com *pipeline*.

A primeira motivação é aumentar o *throughput* do sistema com baixo incremento de *hardware*, sendo que o *throughput* é caracterizado pelo número de tarefas realizadas por unidade de tempo. Para um sistema que opera uma tarefa por unidade de tempo, o *throughput* é igual a $1/D$, sendo D a latência associada ao cumprimento da tarefa.

A técnica do *pipeline* particiona o sistema entre múltiplos estágios adicionando uma memória ou *buffer* entre eles. A computação original é decomposta em k estágios, sendo que uma nova tarefa pode ser inicializada assim que a anterior atravessar o primeiro estágio. Assim, ao invés de iniciar uma tarefa a cada D unidades de tempo, inicializa-se uma a cada D/k, e o processamento das k computações são sobrepostas pelo *pipeline*.

A princípio, o ganho de desempenho é proporcional ao comprimento do *pipeline*, ou seja, quanto mais estágios, maior desempenho. Porém, há limitações físicas relacionadas à frequência que determinam a quantidade de

estágios que podem ser utilizados. De forma geral, o limite de quanto um sistema síncrono pode ser divido em estágios está associado ao tempo mínimo necessário para o funcionamento dos circuitos de *latch* e a incerteza associada ao atrasos da frequência na distribuição dos circuitos. Além das limitações físicas, há uma relação de custo/desempenho que determina a ótima quantidade de estágios de um *pipeline* (SEHN; LIPASTI, 2005), já que uma quantidade de *hardware* é adicionado ao sistema em cada estágio.

O aumento de desempenho em k consiste em três idealismos básicos:

- sub-computações uniformes: a computação a ser realizada é divida em sub-computações com latências iguais;
- computações idênticas: a mesma computação é realizada repetidamente em uma grande quantidade de dados;
- computações independentes: as computações são independentes entre si.

As sub-computações uniformes mantêm o idealismo que cada estágio possui o mesmo tempo de execução, ou seja, T/k. Contudo, esta suposição não é verdadeira sendo impossível particionar computações em estágios perfeitos e balanceados. Exemplificando, uma operação de 400ns foi dividia em três estágios de 125ns, 150ns e 125ns. Como a frequência do *pipeline* é determinada pelo estágio mais longo, esta será limitada pelos 150ns. Assim, o primeiro e o terceiro estágio possuem uma ineficiência de 25ns, e o tempo total da operação utilizando o *pipeline* será de 450ns ao invés de 400ns originalmente. Essa ineficiência é denominada fragmentação interna, sendo o primeiro desafio no desenvolvimento de processadores.

As computações idênticas assumem que todos os estágios do *pipeline* são sempre utilizados em todos os conjuntos de dados. Esta suposição não é válida quando há execução de múltiplas funções, onde nem todos os estágios são necessários para suportá-las. Como nem todos os conjuntos de dados precisarão de todos os estágios, estes ficaram esperando mesmo que não precisem executar tal subfunção devido a característica síncrona do sistema. Esta ineficiência introduz a fragmentação externa dos estágios do *pipeline*.

As computações independentes assumem que não há dependência de dados ou controle em qualquer par de computações. Esta suposição permite que o *pipeline* opere em fluxo contínuo, ou seja, nenhuma operação precisa esperar a completude da operação anterior. Em alguns sistemas esta suposição é válida, mas para *pipelines* de instruções uma operação precisará do resultado da anterior para prosseguir. Se esta situação ocorre e as duas operações encontram-se em processamento, a operação deverá esperar o resultado da anterior, ocasionando a flutuação do *pipeline – pipeline stall*. Se há uma

operação em flutuação, todos os estágios anteriores também deverão esperar, ocasionando uma série de estágios ociosos.

O desenvolvimento de processadores com *pipeline* lidam com três desafios relacionados com o idealismo básico:

- sub-computações uniformes: requer balanceamento dos estágios;
- computações idênticas: requer unificação dos tipos de instrução;
- computações independentes: minimização das flutuações.

Um ciclo de instrução básico pode ser divido em cinco subcomputações genéricas:

1. busca de instruções (BI) – *instruction fetch*;
2. decodificação da instrução (DI) – *instruction decode*;
3. busca dos operandos (BO) – *operand's fetch*;
4. execução da instrução (EX) – *instruction execution*;
5. gravação dos operandos (GO) – *operand store*.

Um ciclo genérico começa com a busca da nova instrução, seguido pela decodificação que determina que tipo de trabalho que será realizado. Normalmente um ou mais operandos devem ser buscados, onde estes podem estar contidos nos registradores ou na memória dependendo do tipo de endereçamento utilizado. Assim que os operandos estão disponíveis, a operação especificada pela instrução é executada e o ciclo termina com a gravação do resultado nos registradores ou na memória. Durante estas cinco subcomputações genéricas, alguns efeitos colaterais podem acontecer mudando o estado da máquina e são considerados efeitos colaterais por não serem explicitamente especificados na instrução. A complexidade e a latência de cada estágio varia significativamente dependendo do ISA.

As subcomputações genéricas correspondem a uma partição natural de um ciclo de instrução formando um *pipeline* genérico de cinco estágios. Esta quantificação de estágios tem como objetivo particionar o ciclo de instruções em estágios balanceados diminuindo a fragmentação interna. Assim, múltiplas subcomputações com baixa latência podem formar um único estágio ou ainda alguns dos cinco estágios genéricos também podem ser agrupados formando um melhor balanceamento do *pipeline* como um todo, conforme a figura 27.

Assim, os dois métodos de quantização são a junção de múltiplas computações em apenas uma e subdivisão de subcomputações em vários estágios.

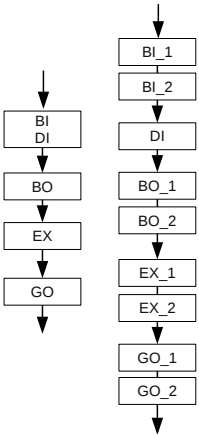


Figura 27: Técnicas de balanceamento de *pipeline* (Apêndice).

Em ambos os casos, o objetivo principal da quantização é minimizar a fragmentação interna. Tomando como exemplo, supõe-se que a computação de 5 estágios genéricos leve $300ns$ e que o tempo de ciclo dos *pipelines* de 4 e 9 estágios da figura 27 é $80ns$ e $40ns$ respectivamente. Consequentemente a latência do *pipeline* de 4-estágios é de $(80ns \times 4) = 320ns$ e o de 9-estágios é $(40ns \times 9) = 360ns$. A diferença entre a nova latência e a antiga representa a fragmentação interna, ou seja, $(320ns - 300ns) = 20ns$ para o de 4-estágios e $(360ns - 300ns) = 60ns$ para o de 9-estágios. Baseando-se neste ponto de vista, conclui-se que o *pipeline* de 4-estágios é mais eficiente que o de 9-estágios, porém em termos de *throughtput*, tem-se $300ns/40ns = 7.5$ contra $300ns/80ns = 3.75$, assim se deve considerar ambos os fatores no projeto de *pipelines*.

Conforme o comprimento do *pipeline* cresce, a quantidade de recursos de *hardware* para dar suporte cresce significativamente. O custo mais significativo está relacionado ao incremento de portas para dar acesso simultâneo aos registradores e à memória, já que os diferentes estágios precisam acessar os operandos bem como ler/escrever na memória.

Como o idealismo de computações idênticas não se mantém, requer-se o máximo da unificação dos tipos de instrução. Como o sistema deve suportar um superconjunto de instruções da arquitetura, instruções que não utilizam todos os estágios geram a fragmentação interna e o objetivo da unificação das instruções é reduzir este problema.

Para realizar uma computação, precisa-se de três tarefas genéricas:

1.operações aritméticas;

2.movimentação de dados;

3.sequenciamento de instruções.

As operações aritméticas envolvem o cumprimento de operações lógicas e aritméticas sobre os operandos e o processador pode suportar uma ampla gama dessas operações. A movimentação de dados é responsável por mover os operandos e os resultados entre diferentes locais de armazenamento. O sequenciamento de instruções está relacionado com a manipulação explícita do fluxo de controle das instruções.

Em uma arquitetura RISC moderna, o conjunto de instruções emprega tipos de instruções dedicadas para cada uma das tarefas genéricas, formando assim uma classificação de três tipos:

1.Instruções da Unidade Lógica e Aritmética (ULA);

2.Instruções de memória (*load/store*): movimentação de dados;

3.Instruções de fluxo (*branch*): para o manipulação do fluxo de controle.

As instruções ULA são restritas somente aos registradores. Apenas as instruções de movimentação de dados podem acessar a memória. As instruções de fluxo e movimentação empregam endereçamento simples de memória, tipicamente apenas modo endereçamento indireto com deslocamento é suportado. Geralmente o endereçamento relativo ao contador de programa (CP) também é suportado para instruções de controle de fluxo. Nas tabelas 6, 7 e 8, conforme (SEHN; LIPASTI, 2005), encontra-se o detalhamento das instruções relacionadas com as operações e as cinco subcomputações típicas de um *pipeline*, considerando uma arquitetura de *cache* de dados e instrução independentes (D-*cache*/I-*cache*).

A análise das especificações das instruções proporciona uma verificação da semântica de cada instrução. Apesar delas compartilharem subcomputações em termos de busca e decodificação, elas necessitarão de recursos diferenciados. Assim, na figura 28 demonstra-se um exemplo de unificação de recursos para um *pipeline* genérico de seis estágios.

Na coluna da esquerda da figura 28 apresenta-se as cinco tarefas básicas de busca (BI) e decodificação de instrução (DI), busca dos operandos (BO), execução (EX) e gravação de operandos (GO). A direita apresenta-se um possível *pipeline* compartilhando as unidades e na parte superior, apresenta-se as 4 instruções. As quatro categorias de instrução (ULA, carregar, gravar e fluxo) compartilham os três primeiros estágios: BI, DI e LT. A unidade ULA (4^o estágio) pode ser utilizada para a geração de endereços para as instruções relacionadas com a memória. O estágio MEM é responsável pelo acesso à *cache* e estágio o REG faz a gravação dos resultados das operações novamente

Subcomputação	Instrução Inteira	Instrução ponto-flutuante
BI	Buscar (acessar I-*cache*)	Buscar (acessar I-*cache*)
DI	Decodificar	Decodificar
BO	Acessar registradores	Acessar registradores PF
EX	Realizar operação	Realizar operação PF
GO	Regravar nos registradores	Regravar nos registradores

Tabela 6: Especificação das instruções aritméticas/ponto flutuante (Apêndice).

Subcomputação	Instrução *Load*	Instrução *store*
BI	Buscar (acessar I-*cache*)	Buscar (acessar I-*cache*)
DI	Decodificar	Decodificar
BO	Acessar registradores (endereço base) Gerar endereço efetivo (base + deslocamento) Acessar memória (acessar D-*cache*)	Acessar registradores (operando e endereço base)
EX		
GO	Regravar nos registradores	Gerar endereço efetivo (base + deslocamento) Acessar memória (acessar D-*cache*)

Tabela 7: Especificação das instruções de acesso à memória (Apêndice).

nos registradores. O estágio MEM após o estágio ULA inclui um ciclo de ociosidade nas operações aritméticas gerando fragmentação externa no modelo. No *pipeline* de seis estágios modelado, as instruções "carregar" utilizam todas as unidades, enquanto as outras apenas cinco.

A unificação dos tipos de instrução em um *pipeline* conduz à minimização dos recursos necessários para suportar todas as instruções. De certo

Subcomputação	Salto incondicional	Salto condicional
BI	Buscar (acessar I-*cache*)	Buscar (acessar I-*cache*)
DI	Decodificar	Decodificar
BO	Acessar registradores (endereço base) Gerar endereço efetivo (base + deslocamento)	Acessar registradores (endereço base) Gerar endereço efetivo (base + deslocamento)
EX		Avaliar condição
GO	Atualizar CP com endereço alvo	Se condição for verdadeira, atualizar CP com endereço alvo

Tabela 8: Especificação das instruções de controle de fluxo (Apêndice).

modo deseja-se determinar um *pipeline* que seja análogo ao mínimo múltiplo comum de todos os recursos necessários. Além disso, maximiza-se a utilização de todos os estágios em diferentes instruções diminuindo a fragmentação externa. Se um estágio ocioso é inevitável, tenta-se colocá-lo no final do *pipeline* permitindo que a instrução termine antes e para reduzir sua latência.

Em termos de implementação, a representação lógica da figura 28 possui algumas particularidades. Os seis diferentes estágios significam que haverá seis instruções ao mesmo tempo em execução, portanto, *register file* necessitará suportar duas leituras (dois operandos no estágio LT) e uma escrita (estágio REG) simultaneamente. A I-*cache* deverá suportar uma leitura a cada ciclo a menos que seja interrompida por uma instrução de controle de fluxo, bem como a D-*cache* com uma leitura ou escrita (estágio MEM). O subsistema de memória é relativamente simples considerando duas *cache*s isoladas com uma porta de acesso. Contudo, se ocorrer qualquer falta de *cache* (*cache miss*), haverá flutuação no *pipeline*. Neste modelo, o acesso à *cache* possui uma latência de um ciclo, mas conforme a *cache* torna-se maior e a lógica do processador aumenta de complexidade, manter uma latência de um ciclo torna-se muito difícil.

A.2 MINIMIZAÇÃO DE FLUTUAÇÕES DO *PIPELINE*

Um programa é uma sequência de instruções em *assembly*. As instruções podem ser especificadas, conforme (SEHN; LIPASTI, 2005) como uma função $i : T \leftarrow S_1 op S_2$, onde o domínio é $D(i) = \{S_1, S_2\}$ e a imagem é

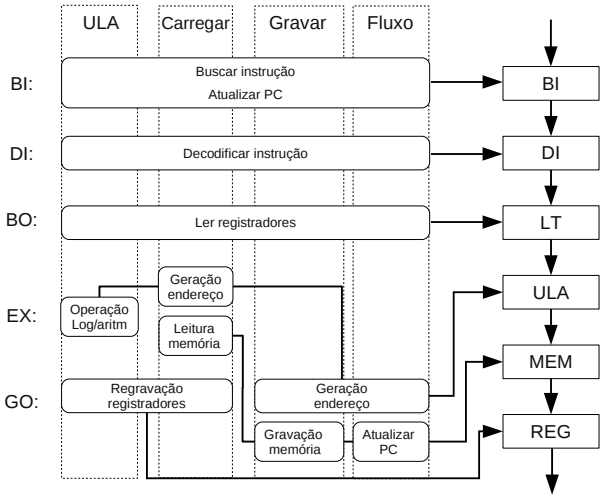


Figura 28: Exemplo de unificação de recursos de um *pipeline* (Apêndice).

$I(i) = \{T\}$, sendo que a relação entre eles é definida pelo operador op. Dadas duas instruções i e j sendo j precedido de i, j pode ser dependente de i se uma das três condições descritas pelas equações A.1, A.2 ou A.3 existirem.

$$I(i) \cap D(j) \neq \emptyset \tag{A.1}$$

$$I(j) \cap D(i) \neq \emptyset \tag{A.2}$$

$$I(i) \cap I(j) \neq \emptyset \tag{A.3}$$

A condição da equação A.1 implica que a instrução j requer um operando que está na imagem de i, denotada como dependência *read-after-write* (RAW) ou dependência verdadeira, ou seja, instrução j não pode executar até a completude de i. Pode-se visualizar um exemplo na equação A.4, onde registrador R_3 é compartilhado por i e j.

$$i : R_3 \leftarrow R_1 op R_2$$

$$j : R_5 \leftarrow R_3 op R_4 \tag{A.4}$$

A equação A.2 denota que a instrução j irá modificar uma variável que é operando de i, denotada como *write-after-read* (WAR) ou dependência anti-dados, ou seja, a existência dessa dependência requer que a instrução j não complete antes da execução de i, ou i irá utilizar o operando errado. Pode-se visualizar esta situação na equação A.5, onde registrador R_1 é utilizado por i e j.

$$\begin{aligned} i &: R_3 \leftarrow R_1 op R_2 \\ j &: R_1 \leftarrow R_4 op R_5 \end{aligned} \tag{A.5}$$

A terceira condição, equação A.3 indica que as instruções i e j irão modificar a mesma variável, conhecida como dependência *write-after-write* ou dependência de saída, ou seja, a instrução j não pode completar antes de i. Um exemplo de dependência de saída é ilustrado na equação A.6, onde registrador R_3 é escrito por i e j.

$$\begin{aligned} i &: R_3 \leftarrow R_1 op R_2 \\ j &: R_3 \leftarrow R_6 op R_7 \end{aligned} \tag{A.6}$$

Além das dependências de dados há as dependências de controle. Dadas as instruções i e j, com j precedido por i, a execução de j depende do resultado de i. As dependências de controle são resultados da estrutura de controle de fluxo do programa e saltos geram incerteza no sequenciamento das instruções.

Como a semântica do programa requer que as dependências acima mencionadas sejam respeitadas e a execução de instruções por um *pipeline* pode facilmente romper o sequenciamento, deve haver mecanismos de identificação e resolução de dependências.

Uma potencial violação das dependências é conhecida como *pipeline hazzards*. As dependências verdadeiras, anti-dados e de saída devem ser identificadas e resolvidas através de mecanismos de *hardware*.

Visualizando-se o *pipeline* de seis estágios da figura 28 e considerando a dependência de dados relacionados com a memória, percebe-se que não há violação do sequenciamento neste estágio. Como a arquitetura é baseada em carregar/gravar (*load/store*), haverá dependência somente nestas duas instruções que possuem acesso à memória. Como há somente um estágio MEM que acessa a D-*cache*, há o sequenciamento completo das instruções não havendo

nenhum tipo de violação de dependência de dados.

Com relação à dependência de dados dos registradores, identifica-se os estágios LT e REG que acessam o *register file* na leitura e gravação dos operandos do *pipeline* da figura 28. A dependência de saída é garantida pois a escrita é sequencial e realizada somente pelo estágio REG. A dependência anti-dados é garantida pois toda a instrução i faz a leitura dos operandos antes da instrução j gravar o resultado, pois o estágio LT é anterior ao REG. Contudo a dependência de dados verdadeira não é garantida automaticamente pelo *pipeline* modelado. Se a instrução i é seguida diretamente por j, quando j alcançar o estágio LT, i ainda estará no ULA, portanto j não pode ler os operandos até i terminar o estágio REG.

Uma dependência de controle pode ser analisada com uma dependência verdadeira envolvendo o contador de programa (CP). Uma instrução de salto condicional escreve no CP e o estágio de busca de instrução faz a leitura. Se a condição é verdadeira, a instrução atualizará o CP com o endereço do salto, caso contrário será a próxima instrução sequencial. Contudo, conforme o *pipeline* apresentado na figura 28, a atualização do CP ocorrerá no estágio MEM e o estágio BI utiliza seu conteúdo para a busca da próxima instrução. A ordenação entre BI e MEM garante uma violação da dependência verdadeira no CP, já que o estágio BI fará a leitura antes da atualização de CP por MEM, após a avaliação da condição de controle.

Uma das formas de garantir o cumprimento das dependências verdadeira e de controle apontadas acima é impedir que o registrador entre no estágio crítico até a próxima instrução termine. Baseando-se nisso, a tabela 9 aponta o pior caso de ciclos de penalidade para que se resolva as violações.

	ULA	Memória	Fluxo
Estágio da instrução j	ULA, Carr./Gravar, Fluxo	ULA, Carr./Gravar, Fluxo	ULA, Carr./Gravar, Fluxo
Registrado envolvido	R_i	R_i	CP
Estágio escrita (i)	REG (6)	REG (6)	MEM (5)
Estágio leitura (j)	LT (3)	LT (3)	BI (1)
Penal.	3 ciclos	3 ciclos	4 ciclos

Tabela 9: Máxima penalidade envolvendo violações semânticas (Apêndice).

Obviamente, o pior caso é quando a instrução j é igual a $i+1$. Se

há dependência de instruções entre i e j, mas estas não estão em sequência direta a penalidade é $3 - s$, onde s representa o número instruções entre i e j. A mesma ideia é válida para dependência de controle sendo a penalidade calculada como $4 - s$.

Porém para as possíveis violações de controle, a penalidade de 4 ciclos somente acontece quando a avaliação de controle for realmente verdadeira. Assim, pode-se construir um mecanismo que assume que a condição de salto será falsa e, portando, o pipeline pode continuar a buscar instruções sequencialmente. Quando efetivamente se resolve o condicionamento, o endereço do CP é atualizado no estágio MEM, as instruções internas ao *pipeline* são removidas. Nesta situação a penalidade de quatro ciclos acontece apenas quando o salto condicional é realmente tomado.

A inclusão de polarização das tomadas de controle de fluxo não resolve o problema de ciclos de penalização, apenas ameniza o caso médio. Assim, inclui-se no *pipeline* caminhos de atalho (*forwarding paths*) conforme a figura 29. Os caminhos de atalho antecipam os resultados já calculados enviando-os aos estágios anteriores sem a necessidade de completar todo o *pipeline*. Por exemplo, se duas instruções aritméticas são dependentes e diretamente sequenciais, o resultado da primeira pode ser utilizado pela segunda assim que esta terminar o estágio ULA, evitando a espera de 3 ciclos decorrente do estágios MEM e REG. Esta situação é ilustrada pela seta (caminho de atalho) ligando a saída do estágio ULA à ele mesmo. A mesma ideia é utilizada interligando os estágios MEM e ULA e MEM e BI.

Com a adição dos atalhos e a respectiva lógica de controle pôde-se reduzir os ciclos de penalidade, conforme a tabela 10.

	ULA	Memória	Fluxo
Instrução j	ULA, Carr./Gravar, Fluxo	ULA, Carr./Gravar, Fluxo	ULA
Registrador	R_x	R_x	CP
Estágio i	REG (6)	REG (6)	MEM (5)
Estágio j	LT (3)	LT (3)	BI (1)
Saída do atalho	ULA, MEM, REG	MEM, REG	MEM
Entrada do atalho	ULA	ULA	BI
Penalidade	0 ciclos	1 ciclo	4 ciclos

Tabela 10: Máxima penalidade com implementação de atalhos (Apêndice).

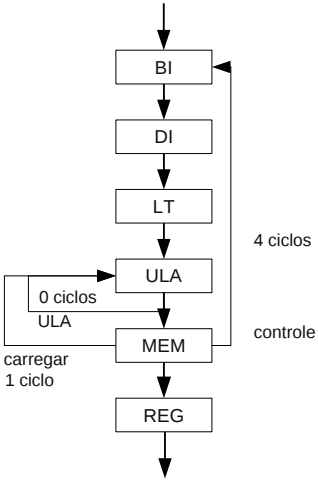


Figura 29: *Pipeline* com inclusão de atalhos (Apêndice).

A resolução dos problemas de dependência por mecanismos de *hardware* é conhecido como *pipeline interlock*. Este mecanismo deve detectar todos os possíveis *pipeline hazzards* e garantir que todas as dependências sejam satisfeitas. Conforme visto, este mecanismo envolve a criação de flutuações no *pipeline* em certos estágios bem como controle da lógica dos caminhos de atalho. Detalhamentos da implementação do mecanismo podem ser visualizados em (SEHN; LIPASTI, 2005).

Conforme visto até então, um *pipeline* de k-estágios pode potencialmente alcançar um desempenho de um fator k em relação ao projeto normal. Contudo, com a construção de *pipelines* longos as penalidades relacionadas com dependências e suas resoluções tornam-se maiores. As penalidades envolvendo instruções de busca de operandos, memória e operações aritméticas geralmente são resolvidas por caminhos de atalho e incorporam poucos ciclos, mas penalidades relacionadas a instruções de controle de fluxo podem incorporar muitos ciclos de ociosidade. Portando, no projeto de microprocessadores, sempre busca-se um equilíbrio entre a quantidade de estágios, frequência de operação, desempenho e custo objetivando o melhor *throughput* possível nestas condições.

A.3 PROCESSADORES SUPERESCALARES

Enquanto máquinas com *pipeline* são efetivas como microarquitetura, esses sistemas escalares possuem algumas limitações. Devido a procura por aumento de desempenho, essas limitações são superadas com o desenvolvimento de *pipelines* superescalares.

As máquinas superescalares são capazes de avançar múltiplas instruções pelos estágios do *pipeline* incorporando múltiplas unidades funcionais. Tais sistemas aumentam a capacidade de processamento concorrente a nível de instrução aumentando o *throughput*. Outro atributo fundamental é a habilidade de executar instruções em ordem diferente ao programa original. A ordem sequencial das instruções no programa implica em algumas precedências desnecessárias entre as instruções. A capacidade de executar as instruções fora de ordem alivia a imposição sequencial e permite mais processamento paralelo sem modificação do programa original.

Um *pipeline* de k-estágios teoricamente pode aumentar o desempenho em um fator k. De modo alternativo, pode-se atingir o mesmo fator de velocidade empregando k cópias do mesmo *pipeline* processando as k instruções em paralelo. Estas duas maneiras de paralelismo são conhecidas como paralelismo temporal e paralelismo espacial respectivamente (SEHN; LIPASTI, 2005). *Pipelines* paralelos empregados nos processadores superescalares utilizam as duas técnicas de paralelismo.

O fator de velocidade de um *pipeline* escalar é medido em relação a um projeto sem *pipeline* e é primeiramente determinado pelo seu comprimento. Para os *pipelines* superescalares o fator de velocidade é determinado pela sua largura. Um *pipeline* com largura s pode processar concorrentemente s instruções, levando a fator de velocidade teórico s sobre um *pipeline* escalar.

A quantidade de recursos de *hardware* utilizados na implementação do *pipeline* superescalar aumenta consideravelmente. A complexidade lógica de cada estágio pode aumentar com o fator s. No pior caso, os circuitos para interconexão dos estágios podem aumentar em um fator s^2, se um *crossbar* de $s \times s$ for utilizado para conectar todos os s *buffers* de instrução de um estágio para todo os s *buffers* do próximo estágio. Além disso, para suportar acesso concorrente aos registradores pelas s instruções, o número de portas de leitura e escrita do *register file* deve aumentar em um fator s, bem como deve haver portas adicionais na *cache* de dados e de instrução.

Na figura 30 apresenta-se o modelo do *pipeline* escalar utilizado no Intel i468 e o seu sucessor superescalar Pentium.

O Pentium basicamente implementa um *pipeline* paralelo de largura $s = 2$, ou seja, dois i486 em paralelo. Esta estrutura permite que múltiplas instruções sejam buscadas e decodificadas pelo dois primeiros estágios em

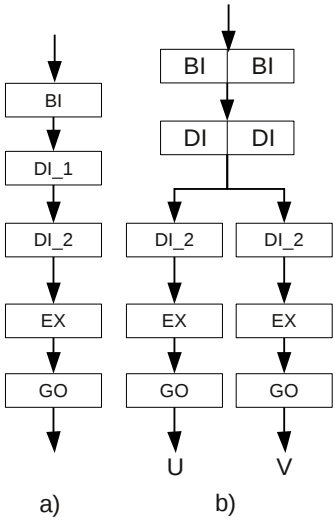


Figura 30: *Pipeline* do i486 (a) e Pentium (b) (Apêndice).

cada ciclo de máquina. Além disso, potencialmente duas instruções podem ser iniciadas nas unidades *U* e *V* de execução.

Comparado com o i486, o Pentium requer recursos de *hardware* significativos. Primeiramente os cinco estágios dobraram de largura. As duas unidades de execução podem acomodar até duas instruções em cada um dos três últimos estágios. O estágio de execução pode realizar uma operação ULA ou acessar os dados da D-*cache*. Portanto, portas adicionais aos registradores são necessárias para suportar duas operações ULA concorrentes em cada ciclo. Se as duas instruções em execução são operações de memória, então a *cache* de dados deve promover acesso dual, sendo isto caro de implementar. A D-*cache* do Pentium é implementada com uma única porta com acesso intercalado em oito níveis (*eight-way interleaving*). Acesso simultâneo a dois bancos diferentes pelas duas instruções de memória nas unidades *U* e *V* podem ser suportados. Se há um conflito de banco, as instruções tornam-se sequenciais.

A ineficiência da unificação de um único *pipeline* é naturalmente incorporada à *pipelines paralelos*. Ao invés de implementar *s* unidades idênticas, pode-se implementar unidades diversificadas na unidade de execução, formando um *pipeline* diversificado. Tal *pipeline* pode ser visualizado na figura 31. Neste exemplo quatro unidades funcionais são implementadas e o estágio BO envia as instruções para as unidades corretas baseando-se nos ti-

pos de instrução.

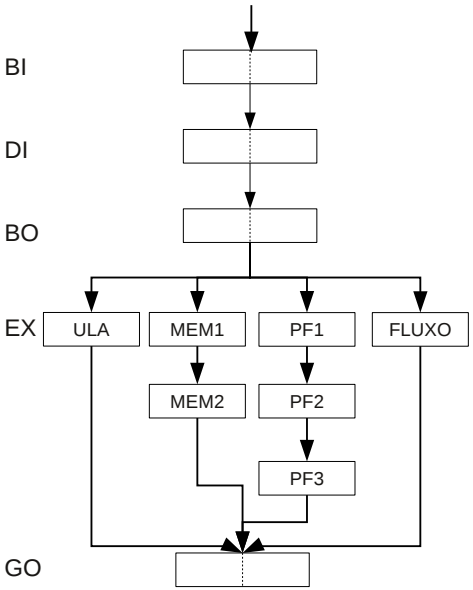


Figura 31: Exemplo de *pipeline* diversificado (Apêndice).

Neste projeto, cada unidade funcional pode ser customizada para uma instrução particular resultando em um *hardware* eficiente. Cada instrução possui sua própria latência e utiliza todos os estágios de sua unidade funcional. Se as dependências intra-instruções são resolvidas antes do envio às unidades funcionais, não há flutuações no *pipeline*. Este esquema permite controle independente e distribuído de cada *subpipeline* de execução.

Em qualquer projeto de *pipeline*, *buffers* são necessários entre os estágios. Nos *pipelines* escalares um *buffer* de única entrada é colocado nos estágios consecutivos e este mantém todos os *bits* de controle e dados da instrução que está atravessando os estágios. Todas as instruções entram e saem de cada *buffer* na mesma ordem que especificadas no código sequencial.

No *pipeline* paralelo, *buffers* com múltiplas entradas são necessários. Se estes *buffers* seguirem o mesmo conceito dos *buffers* do projeto escalar, haverá latências desnecessárias em algumas instruções. Assim, os *pipelines* superescalares utilizam *buffers* multi-entrada complexos que permitem acesso independente a cada entrada e instruções podem permanecer armazenadas por vários ciclos, formando uma pequena memória *cache* associativa em cada inter-estágio do *pipeline*.

Para evitar ciclos de flutuação desnecessários no *pipeline* diversificado, permite-se que novas instruções sejam adiantadas quando há ciclos de flutuação, por dependências ou faltas na memória *cache*. Este adiantamento pode mudar a ordem de execução das instruções da ordem sequencial do código do programa. A execução fora de ordem permite que estas sejam executadas assim que os operandos estejam disponíveis. Um *pipeline* paralelo que permite execução fora de ordem é chamado de *pipeline* dinâmico, onde a execução dinâmica é possível utilizando-se *buffers* multi-entrada complexos. Tal sistema permite que as instruções entrem e saiam em ordens diferentes.

Na figura 32 encontra-se um exemplo de *pipeline* dinâmico de largura $s = 2$. O estágio de execução é constituído de quatro unidades funcionais, sendo estas antecedidas e seguidas por dois *buffers* multi-entrada de ordenação. O primeiro é chamado de *buffer* despachante e recebe as instruções decodificadas e na ordem sequencial do programa mas as despacha em ordem diferente dependendo da utilização das unidades funcionais.

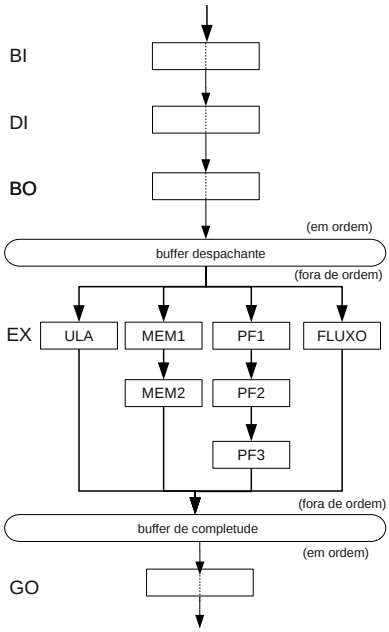


Figura 32: Exemplo de *pipeline* dinâmico (Apêndice).

Além do possível despacho fora de ordem, as unidades funcionais possuem latências diferentes permitindo que as instruções terminem a execução fora da ordem. Para certificar-se que as exceções podem ser servidas

de acordo com a ordem original do programa, as instruções devem ser completadas na ordem do programa. Quando as instruções terminam foram de ordem, um outro *buffer* multi-entrada de reordenação é utilizado no final das unidades funcionais. Tal *buffer* é conhecido como *buffer* de completude e armazena as instruções que terminaram antes enviando-as ao próximo estágio na ordem correta.

Com um *pipeline* largo, o estágio de busca acessa *s* instruções da I-*cache* em cada ciclo de máquina. Isto implica que a organização da I-*cache* deve ser larga o suficiente para que cada linha armazene *s* instruções e que toda a linha possa ser acessada em cada ciclo de instrução. Contudo, não é possível que se garanta que a banda máxima de busca seja atendida devido ao desalinhamento das instruções na I-*cache* e também a presença de instruções de controle de fluxo. O problema de maximização de busca de instrução é resolvido de maneira diferente em cada microarquitetura, por *hardware* ou mesmo por diretivas de alinhamento em tempo de compilação do programa.

O estágio de decodificação em um ISA RISC deve identificar dependências das instruções e verificar quais as instruções que podem ser despachadas em paralelo. Além disso, deve identificar mudanças do controle de fluxo das instruções decodificadas para promover uma rápida resposta do estágio de busca. Para um ISA CISC a tarefa de decodificação pode ser ainda mais complexa precisando de múltiplos sub-estágios. Tanto o Pentium quanto o AMD K5 empregam dois sub-estágios para decodificação da instrução IA32 e o Intel Pentium Pro precisa de um total de cinco ciclos de máquina para acessar a I-*cache* e decodificar uma instrução. A utilização de instruções de tamanho variável (ISA CISC) impõe sequenciamento na decodificação, pois uma instrução deve ser decodificada e seu tamanho conhecido para que a próxima instrução possa ser identificada.

Na decodificação das instruções CISC, o decodificador deve traduzir as instruções da arquitetura para operações de baixo nível que são executadas diretamente pelo *hardware*. Estas operações internas relembram as instruções RISC e são vistas como micro instruções. Na microarquitetura AMD são conhecidas como operações RISC (*ROPs – RISC operations*) e no Intel são identificadas como micro operações (μ-ops – *micro-operations*). Cada instrução IA32 é traduzida como uma ou mais micro operações sendo que, na média, uma instrução possui 1,5 a 2,0 μ-ops. Além disso as μ-ops utilizam o modelo de memória *load/store*. Para muitos processadores superescalares o *hardware* de decodificação é bastante complexo e requer particionamento em múltiplos estágios. Quando o número de estágios aumenta, a penalidade de instruções de controle de fluxo aumenta, por essa razão, utiliza-se a técnica de pré-decodificação que pode ser consultada em detalhes em (SEHN; LIPASTI, 2005).

Durante a decodificação é possível que alguns dos operandos não estejam prontos pois instruções anteriores não terminaram a execução. Para evitar ciclos de ociosidade com flutuações no *pipeline*, busca-se os operandos que estão prontos e se avança estas instruções a um *buffer* separado. Quando todos os registradores forem atualizados o estágio de despacho envia as instruções às unidades funcionais. Este *buffer* separado é conhecido como estações de reserva – *reservation stations* – promovendo o desacoplamento entre a execução e decodificação das instruções. As estações de reserva provém um *buffer* que permite resolver disparidade temporal devida à variação de *throughput* entre os estágios de execução e decodificação.

Uma instrução é considerada completa quando termina sua execução e atualiza o estado da máquina. Após o término nas unidades funcionais a instrução entra no *buffer* de completude e após este, torna-se completa. A completude da instrução envolve a atualização do estado da máquina enquanto a remoção envolve a atualização do estado da memória. Assim em uma máquina com estações de reserva uma instrução passa pelas fases de busca (*fetch*), decodificação (*decode*), despachante (*dispatch*), início (*issue*), execução (*execute*), término (*finish*), completude (*complete*) e remoção (*retire*).

Na ocorrência de interrupções, o processador para a busca de novas instruções e permite que as instruções internas ao *pipeline* continuem até o término para salvar o estado. Quando a interrupção é servida pelo sistema operacional, o estado anterior é retomado e continua a execução do programa.

Quando ocorrem exceções matemáticas, que são induzidas pela execução do programa, os resultados das subcomputações não são mais válidos e o sistema operacional intervém para reportar essas exceções. No caso de faltas de páginas, dependendo da microarquitetura, o tempo para trazer as páginas alocadas no disco induz muitos ciclos, e ocorre suspensão temporária do programa.

Em termos gerais, uma máquina superescalar é apresentada na figura 33.

A.4 TÉCNICAS DE PREVISÃO DE FLUXO

O *pipeline* dos processadores é construído para evitar ao máximo ciclos de ociosidade relativos a dependências de dados entre as instruções. Esta característica é adquirida com a utilização de caminhos de atalho que minimizam os ciclos de penalidade, conforme visto anteriormente. Contudo, as dependências de controle ainda induzem uma quantidade significativa de flutuações e técnicas de previsão do controle de fluxo (*branch prediction*) são

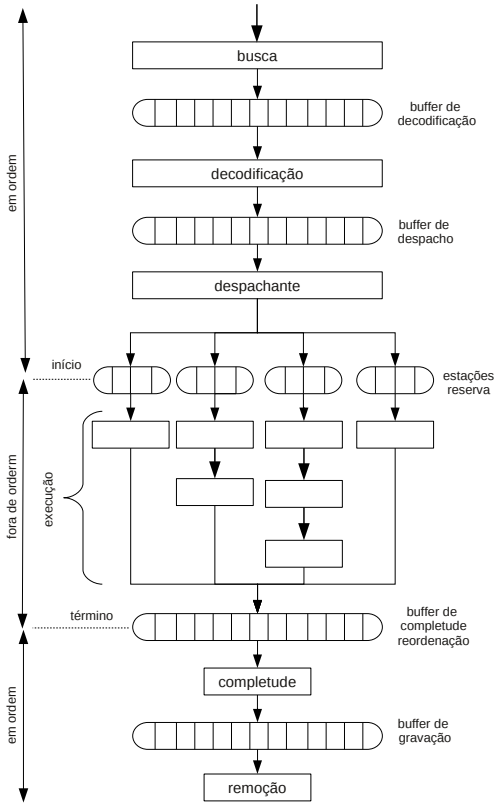


Figura 33: Exemplo de *pipeline* superescalar (SEHN; LIPASTI, 2005) (Apêndice).

adotadas.

Estudos apontam que o comportamento desse tipo de instrução é altamente previsível (SEHN; LIPASTI, 2005). A técnica chave para minimizar as penalidades e maximizar o desempenho é especular tanto o endereço quanto a condição do salto nas instruções de controle. Uma instrução de controle estático é repetidamente executada, enquanto as de comportamento dinâmico podem ser rastreadas. Obviamente que a utilização de técnicas especulativas requerem mecanismo de recuperação de erros de previsão.

A especulação do endereço envolve o uso de um *buffer – Branch Target Buffer* (BTB) – para armazenar o endereço alvo da instrução de controle anterior. O BTB é uma memória *cache* pequena acessada durante o estágio

de busca de instruções. Cada entrada no BTB contém dois campos: endereço da instrução – *Branch Instruction Address* (BIA) – e o endereço alvo – *Branch Target Address* (BTA). Quando uma instrução de controle estático é executada na primeira vez, uma entrada na BTB é gerada, armazenando o seu endereço e o endereço alvo. O BTB é acessado concorrentemente a I-*cache* e quando o contador de programa (CP) bate com o BIA, ocorre um acerto no BTB. Assim o campo BTA é utilizado para a busca da próxima instrução se a condição é especulada como verdadeira para o salto.

Se acerta-se a previsão do salto não há ciclo de penalidade. A versão não especulativa de execução continua em processo para avaliação. A instrução de controle continua sendo buscada da I-*cache* e é executada normalmente. A avaliação do endereço e condicionamento são comparadas com a versão especulada e se concordam a previsão estava correta. Caso contrário um erro de previsão ocorreu e se deve iniciar a recuperação. O resultado da execução não especulativa é utilizada para atualizar os conteúdos do BTB.

Existem várias maneiras de implementar a especulação da condição. A maneira mais fácil é polarizar o *hardware* para sempre considerar que o salto não será tomado. Outra maneira é utilizar suporte do compilador para dar dicas sobre o controle do fluxo, mas requer modificação do ISA. Outro método mais agressivo de previsão dinâmica utiliza o endereço alvo como deslocamento. Esta forma de previsão primeiramente calcula o deslocamento relativo entre o endereço atual e o endereço alvo. Um valor positivo polariza o *hardware* para considerar falsa a condição e um valor negativo indica laço, polarizando para considerar como verdadeiro. A técnica mais comum utilizada nos processadores superescalares levam em conta o histórico das execuções de controle anteriores.

A previsão de fluxo baseada em histórico faz a previsão da direção do salto, com tomado (T) e não tomado (N), baseado nas direções observadas anteriormente. O algoritmo baseado em histórico é baseado na máquina de estados finitos apresentada na figura 34. Cada estado representa um padrão de histórico em termos de sequências de salto. A saída da lógica gera a previsão baseado no estado atual da máquina.

A máquina de estados da figura 34 utiliza dois *bits* para manter histórico das duas execuções anteriores do salto e é polarizada para tomar o salto. Os dois *bits* de histórico constituem as variáveis de estado da máquina podendo estar em: NN, NT, TT ou TN, representando as direções tomadas nas duas últimas execuções do salto. O estado NN é considerado inicial e um valor de saída T ou N é associado a cada um dos quatro estados representando a direção que o previsor fará em tal estado. Quando um salto é executado, a direção tomada é utilizada como entrada da máquina de estados e a transição de estado atualiza o histórico que será utilizado na próxima previsão.

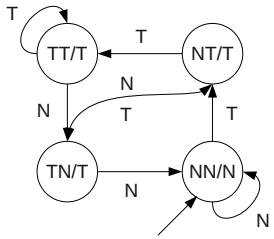


Figura 34: Exemplo de máquina de estados finitos para previsão de controle de fluxo (Apêndice).

A máquina de estados apresentada antecipa longas cadeias de N ou T. Desde que uma das duas execuções anteriores são consideradas como “tomadas”, haverá previsão para tomar o salto da próxima execução. A previsão somente será mudada quando encontrar cadeias de dois N seguidos. Esta máquina apresenta um tipo de previsor de fluxo e inúmeros projetos diferentes são possíveis, dependendo da arquitetura e pesquisas realizadas.

A especulação de fluxo envolve a previsão do fluxo de controle e a busca de novas instruções baseia-se no endereço previsto. Na escolha de um caminho, instruções adicionais de fluxo podem ser encontradas e a previsão destas pode ser similarmente realizada, potencialmente especulando múltiplos caminhos antes da primeira especulação ser avaliada. Quando tal situação ocorre as instruções adicionais especuladas já residentes na máquina são identificadas através de uma etiqueta (*tag*). Quando ocorre a avaliação da condição do salto e esta foi corretamente especulada, as identificações são desalocadas e as instruções do caminho especulativo podem ser completadas. Quando a previsão especulativa erra o caminho, este é terminado e a busca pelo caminho correto é iniciada com a atualização do contador de programa (CP). Se o caminho correto é o sequencial (sem o salto) o CP correto é obtido pelo endereço da instrução do salto. Após a atualização do CP e a busca de instruções do caminho correto se reinicia, o previsor de caminhos recomeça. A remoção das instruções do caminho errado é realizada utilizando-se etiquetas de identificação para remoção das instruções, tanto aquelas que se encontram nos estágios de decodificação e despacho quanto as das estações reserva. Entradas dos *buffers* de reordenação também são desalocadas. Pode-se notar que um erro de previsão é custoso, além dos ciclos perdidos no processamento de instruções não utilizadas, deve-se realizar toda a limpeza das instruções falsamente previstas. Porém testes de desempenho apontam que cerca de 82,5% a 96,2% das predições são corretas (SEHN; LIPASTI, 2005).

As técnicas de previsão de caminhos apresentados até então são sim-

ples e apresentam limitações. Implementações como o Pentium Pro apresentam previsão de caminho especulativo adaptativo de dois níveis (*two level adaptative branch prediction*) atingindo 95% de previsões corretas. Detalhes específicos da previsão de caminhos de cada implementação de microarquitetura atual não são revelados (Intel Nehalem, por exemplo) pois é uma peça chave no desempenho dos processadores atuais.

A.5 ARQUITETURA DE MEMÓRIA

Devido a propriedade de localidade de referência, é possível satisfazer as questões relacionadas ao custo e velocidade das memórias de forma hierárquica formando a estrutura de memória utilizada nos computadores atualmente, conforme a figura 35.

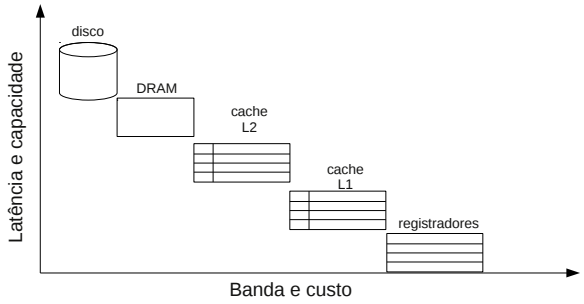


Figura 35: Relação e hierarquia de memória (Apêndice).

A localidade de referência é a propriedade segundo a qual os programas acessam o mesmo ou endereços próximos frequentemente e repetidamente. A localidade é dividia em localidade espacial e localidade temporal.

A localidade temporal refere-se à acessos ao mesmo endereço ocorrendo em intervalos de tempo pequenos. Muitos programas reais possuem essa tendência tanto para referências de instruções quanto de dados. A localidade de instruções é facilmente explicada relacionando-se com laços: para cada interação do laço, as instruções que formam o corpo são buscadas novamente. Quanto aos dados, variáveis amplamente utilizadas conduzem à localidade temporal, além da utilização da pilha para passagem de parâmetros em chamadas de funções. A pilha é desalocada no retorno dos procedimentos e realocada em uma chamada subsequente, os endereços correspondentes ao topo da pilha são acessados repetidamente para passagem dos parâmetros, registradores e retorno dos resultados.

A localidade espacial refere-se à acessos aos endereços próximos ocorrendo em intervalos de tempos pequenos: uma referência anterior à algum endereço é seguida por uma referência a um endereço adjacente ou bem próximo. Em termos de instrução, a execução sequencial de programas leva ao acesso sequencial à memória, portanto excluindo-se saltos de controle de fluxo, o acesso a memória é linear. Quando se considera os saltos, ocasionando descontinuidade na busca de instruções, os endereços alvos normalmente encontram-se próximos. A localidade espacial de dados frequentemente ocorre por razões algorítmicas. Aplicações numéricas que atravessam grandes matrizes de dados possuem acesso aos elementos de maneira serial. Desde que os dados das matrizes são gravados na memória na mesma ordem que são acessados (serialmente), grande localidade espacial pode ser alcançada.

Obviamente, programas que possuem baixa localidade espacial ou temporal existem. Nestes casos é muito difícil projetar hierarquias de memórias eficientes levando-se em conta o sistema de memória *caches*, paginação de memória e memória RAM (*Random Access Memory*).

A.5.1 Memória *cache*

A memória *cache* foi criada para explorar as propriedades de localidade espacial e temporal, criando uma ilusão de memória rápida de grande capacidade. O princípio básico é colocar uma memória da baixa capacidade, rápida e de alto custo entre o processador e a memória principal, sendo esta mais barata e com grande capacidade. Carrega-se nesta memória *cache* as instruções e dados que possuem localidade espacial e temporal. As referencias à memória que estão em *cache* são satisfeitas rapidamente, reduzindo a latência média e consequentemente garante-se uma alta largura de banda. Considerando que a propriedade de localidade é alta, mesmo a memória *cache* de nível primário (L1) consegue satisfazer 90% das referências na maioria dos casos. Referências na *cache* são consideradas acertos (*hit*) e referências não satisfeitas são consideradas faltas (*misses*).

O mecanismo de *cache* pode ser estendido a múltiplos níveis adicionando-as em forma hierárquica aumentando a capacidade e a latência. Desde que cada nível de *cache* consiga capturar uma quantidade de referências, a latência induzida do processador é substancialmente menor que se todas as referências fossem enviadas ao nível mais baixo da hierarquia de memória. Através da equação A.7 (SEHN; LIPASTI, 2005) é possível calcular a latência média de referências de memória, onde l_i representa a latência correspondente ao nível i e h_i representa a fração de referências satisfeitas por i.

$$l = \sum_{i=0}^{n} h_i \times l_i \tag{A.7}$$

Percebe-se que, desde que a taxa de acerto h_i dos níveis superiores (L1 e L2) for alta, a latência média será baixa. Contudo a equação requer que as taxas de acertos sejam especificadas de forma global. Também é interessante utilizar as taxas de acertos locais, que especificam a fração de referências de memória servidas em tal nível da memória. Para o primeiro nível de *cache*, nível L1, as taxas locais e globais são idênticas. Contudo, no segundo nível, apenas as referências que resultam em falta no primeiro nível (m_1) são servidas, no terceiro, apenas as faltas do segundo, e assim por diante. Assim, a taxa de acerto de um nível *i* pode-se ser calculada através da equação A.8.

$$l_i = \frac{h_i}{m_{i-1}} = \frac{h_i}{1 - \sum_1^{i-1} h_i} \tag{A.8}$$

Pode-se ainda calcular a taxa de faltas em termos de instrução. Essa métrica reporta a taxa de falta normalizada em termos de números de instruções executadas ao invés de quantidade de referências da memória. Dado uma taxa de falta por instrução im_i e uma penalidade específica p_i para uma falta no sistema de *cache*, pode-se estimar efeito de desempenho do sistema hierárquico usando a métrica "tempo de memória por instrução" – *memory-time-per-instruction* (MTPI) (SEHN; LIPASTI, 2005), definida na equação A.9.

$$MTPI = \sum_{i=0}^{n} im_i \times p_i \tag{A.9}$$

Nesta equação p_i reflete a penalidade associada com uma falta no nível *i* assumindo que será satisfeita no próximo nível. Desta forma, p_i é calculado como a diferença das latências dos níveis adjacentes: $p_i = l_{i+1} - l_i$.

A taxa de falta por instrução im_i é convertida da taxa de falta global m_i, onde $m_i = 1 - h_i$, e o número de referências realizadas por instrução. Assumindo que cada instrução é buscada individualmente e que 40% destas são instruções de memória (*load/store*), tem-se um total de $n = 1 + 0.4 = 1.4$ referências por instrução. A taxa de faltas por instrução é calculada pela equação A.10 (SEHN; LIPASTI, 2005).

$$im_i = \frac{(1 - \sum_{j=1}^{i} h_j) faltas}{referencias} \times \frac{n\_referencias}{instrucao} \quad \text{(A.10)}$$

Considerando um sistema com três níveis de memória, com $h_1 = 95\%$, $l_1 = 1ns$, $h_2 = 4\%$, $l_2 = 10ns$, $h_3 = 1\%$ e $l_3 = 100ns$, $im_1 = (1 - 0,95) \times 1.4 = 0.07$ faltas por instrução e $im_2 = (1 - 0,95 + 0,04) \times 1.4 = 0.014$ faltas por instrução. O cálculo do tempo de memória por instrução é $MTPI = [(0.07 \times (10ns - 1ns)] + [0.014 \times (100ns - 10ns)] = 1.89ns$ por instrução.

Ainda, se considerar que cada ciclo do processador possui $1ns$, o ciclo de memória por instrução (MCPI) será de 1.89. Em conjunto com o MCPI e o CPI do núcleo do processador, pode-se calcular o desempenho total do sistema, conforme a equação A.11.

$$CPI_{total} = CPI_{pipeline} + MCPI \quad \text{(A.11)}$$

Cada nível de *cache* deve ser projetado para suportar os requerimentos de banda e latência. Como os níveis mais altos devem operar em velocidades comparáveis ao núcleo do processador, estes devem ser implementados usando *hardware* rápido e com complexidade limitada.

Ainda, cada nível deve implementar um mecanismo que permita uma procura, de baixa latência, para verificar se dado bloco está ou não contido na *cache*. O tamanho do bloco ou tamanho da linha descreve a granularidade na qual a *cache* opera. Cada bloco é uma sequência contínua de *bytes* e inicia em alinhamento comum. O menor tamanho utilizável de um bloco possui o tamanho natural de palavras do processador, ou seja, 4-*bytes* para máquinas de 32-*bits* ou 8-*bytes* para máquinas de 64-*bits*.

Devido ao alinhamento dos blocos, os *bits* menos significativos do endereço são sempre zero. Se é preciso acessar um *byte* diferente do primeiro, esses zeros menos significativos são utilizados como deslocamento para encontrar o *byte* correto. O número de *bits* necessários para o deslocamento são calculados pelo log_2 do tamanho do bloco, por exemplo, se o bloco é de 64-*bytes*, $log_2(64) = 6$ *bits* são utilizados como deslocamento.

Em termos de localização de blocos há três possíveis organizações: mapeamento direto, totalmente associativo e associativo por conjunto.

No mapeamento direto, um endereço particular pode residir apenas em uma única localização na *cache*. Esta localização é normalmente determinada extraindo os *n bits* do endereço e os utilizando para índice direto das 2^n possíveis localizações na *cache*. Como há um mapeamento de muitos endereços para somente um lugar na *cache*, cada localização necessita de uma

etiqueta (*tag*) que corresponde aos *bits* restantes do bloco gravados naquela localização. Em cada procura, o *hardware* precisa ler a etiqueta e comparar com o endereço da referência e determinar se há um acerto ou falta.

A organização totalmente associativa permite um mapeamento de múltiplos endereços a múltiplas localizações na *cache*, ou seja, qualquer endereço de memória pode residir em qualquer localização na *cache*. Como todas as localizações da *cache* devem ser verificadas para encontrar os dados, não há *bits* de índice para determinação da localização. Novamente cada localização deve possuir uma etiqueta do endereço dos dados que está armazenando, para o *hardware* comparar e detectar uma falta ou acerto.

Na associatividade por conjunto há um mapeamento de múltiplos endereços para algumas localizações na *cache*. Em cada procura, um subconjunto de *bits* de endereço são utilizados para geração do índice igualmente ao caso de mapeamento direto. Contudo, o índice corresponde a um conjunto de entradas que são procuradas em paralelo em busca da etiqueta correta, ou seja, a associatividade por conjunto é um compromisso entre o mapeamento direto e a associatividade total.

Como cada nível de *cache* possui uma capacidade finita, deve haver uma política para despejar ocupantes atuais possibilitando espaço para blocos com referências mais recentes. Uma política de substituição determina que algoritmo utilizar para identificar um candidato para ser despejado. Em um mapeamento direto o problema é trivial já que há somente um lugar para cada endereço.

Na associatividade completa ou por conjunto, há escolhas a serem feitas: um novo bloco pode ser colocado em várias localizações, e todas as entradas são candidatas para substituição. Há basicamente três políticas implementadas no projeto de *cache*: FIFO (primeiro a entrar, primeiro a sair), menos recentemente utilizado – *Least Recently Used* (LRU) – e aleatoriamente.

A política FIFO simplesmente mantém a ordem de inserção e despeja blocos que são residentes por mais tempo. Pode-se utilizar uma fila circular para implementação tanto para a associatividade completa, quanto para formação de conjuntos. Contudo a política FIFO nem sempre mantém a localidade temporal inerente ao padrão de acesso do programa, como por exemplo, variáveis globais que são acessadas continuamente. Tais referencias sofrerão faltas frequentes, pois tais blocos serão substituídos em intervalos regulares.

A política LRU tenta minimizar este problema mantendo uma lista ordenada das recentes referências de cada bloco que são candidatos à substituição. Toda vez que um bloco é referenciado como acerto ou falta, este é colocado no início da lista, enquanto os outros são empurrados para baixo. Quando um bloco precisa ser despejado, o último da lista ordenada é esco-

lhido. Empiricamente, esta política funciona muito bem, mas possui difícil implementação: requer manutenção de lista em *hardware* e atualização desta em cada acerto e falha na *cache*. Geralmente implementa-se uma política aproximada de LRU, conhecida como não-mais-recente-utilizado – *not-most-recently-used* (NMRU) – onde se mantém histórico do bloco recentemente mais utilizado e despeja algum dos outros, escolhido aleatoriamente se houver mais de um candidato. No caso de associação de dois níveis, LRU e NMRU são equivalentes, mas quando se aumenta o conjunto, o NMRU é menos exato, mas mais simples de implementar.

A política aleatória escolhe um bloco aleatoriamente para ser substituído. Apesar de parecer ineficiente, estudos revelam que a política aleatória é pouco pior que LRU e melhor que FIFO. Obviamente que a implementação de uma política aleatória verdadeira é muito difícil, então mecanismos práticos implementam aproximações.

A utilização de memória *cache* implica em múltiplas cópias do bloco. Enquanto há apenas leitura não ocorre nenhum problema, contudo deve haver mecanismos para atualização dos blocos nos outros níveis da memória. Existem basicamente dois mecanismos conhecidos como *write-through* e *write-back*.

O mecanismo *write-through* simplesmente propaga a atualização para os níveis inferiores. É fácil implementação e não há ambiguidade sobre a versão dos dados. Contudo exige muita demanda de barramento e a disparidade da frequência entre o processador e a memória principal torna impossível a utilização em todos os níveis da hierarquia. Este mecanismo ainda deve especificar se vai ou não alocar espaço na *cache* quando ocorre uma falta em escrita de um bloco. A política *write-allocate* implica em alocar o bloco na *cache*, enquanto a *write-no-allocate* evitará a instalação do bloco na *cache*, realizando esta operação somente em faltas de leitura.

O mecanismo *write-back* prorroga a atualização das outras cópias até o ponto que se precisa manter a ordem de correção. Neste tipo de política, uma ordem de prioridade implícita é utilizada para encontrar a cópia mais atualizada do bloco e somente esta cópia é atualizada. Se um bloco é encontrado no nível mais elevado da *cache*, esta cópia é atualizada, enquanto as cópias dos níveis mais baixos permanecem desatualizados. Se uma cópia é apenas encontrada um nível mais baixo, esta é promovida ao topo e é ali atualizada. As cópias atualizadas são marcados com um *bit* (*dirty bit*) sinalizando que os outros níveis possuem cópias desatualizadas. Quando um bloco sinalizado é substituído, este é atualizado para o nível inferior, certificando que a cópia mais recente permaneça. A cópia deste nível é agora marcada com *bit* sinalizador para que quando for despejado, seja atualizado nos níveis inferiores. Na figura 36 encontra-se a relação das propriedades da memória *cache*.

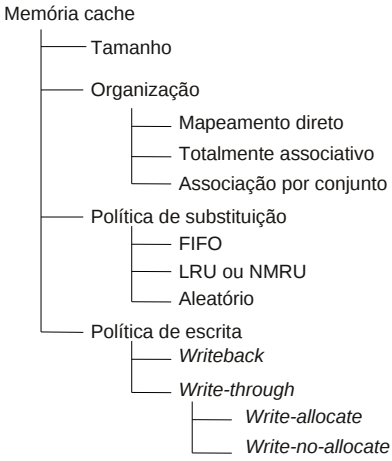


Figura 36: Propriedades da memória *cache* (Apêndice).

Em relação às faltas, há basicamente três causas:

- faltas compulsórias ou *cache* fria: ocasionadas pela primeira referência de um programa a memória. Essas faltas são fundamentais por não poderem ser evitadas por nenhuma técnica;
- faltas por capacidade: ocasionadas pela falta de capacidade de uma dada *cache*. Aumentando a capacidade pode eliminar alguns ou todas as faltas de capacidade;
- faltas por conflito: acontecem devido a alocação imperfeita de entradas particulares na *cache*. Mudanças na associatividade ou função de indexação podem melhorar ou piorar o número de conflitos.

As faltas compulsórias são fundamentais e são determinadas pelo conjunto de trabalho (*working set*) do programa em questão. Contudo a variação do tamanho do bloco afeta diretamente o número de faltas. Uma *cache* com o tamanho do bloco igual a uma palavra do processador irá enfrentar uma falta para cada palavra referenciada pelo programa (forma-se um limite superior do número de faltas compulsórias em termos de organização de *cache*), enquanto uma *cache* com tamanho de bloco infinito irá enfrentar somente uma única falta compulsória. A última afirmação é válida pois a primeira referência irá instalar todos os endereços na *cache*, evitando qualquer falta adicional. Obviamente que blocos infinitos não é uma aplicação prática, portanto usualmente emprega-se blocos de 16 a 512-bytes.

As faltas por capacidade são determinadas pelo tamanho do bloco e capacidade da *cache*. Com o aumento da capacidade, o número de faltas diminui em função da possibilidade de captura de um maior conjunto de trabalho do programa. Em contraste, com blocos maiores, o número de blocos únicos que podem residir simultaneamente na *cache* de tamanho fixo diminui. Blocos maiores tendem a pior utilização, devido a probabilidade de acesso a todas as palavras de um bloco particular diminuir com o aumento do bloco, levando a uma menor capacidade efetiva. Assim, com menos blocos únicos e menor probabilidade de que todas as palavras são úteis em cada bloco, blocos maiores normalmente resultam em aumento das faltas por capacidade. Contudo, programas que eficientemente utilizam todo o conteúdo de blocos grandes não experimentarão tal aumento de faltas.

As faltas por conflito são determinadas pelo tamanho do bloco, capacidade e associatividade da *cache*. O aumento da capacidade reduz o número de conflitos diretamente: como há mais espaço, mais blocos podem residir simultaneamente. Da mesma forma que as faltas por capacidade, blocos menores reduzem a probabilidade de conflito e aumentam a capacidade efetiva. De forma similar, o aumento da associatividade irá reduzir o número de faltas por conflito. Na tabela 11 apresenta-se efeitos da variação de parâmetros da *cache* e possíveis consequências, conforme (SEHN; LIPASTI, 2005).

A.5.2 Memória Virtual

Este mecanismo virtualiza a memória principal separando a perspectiva do programa do atual endereço físico real dos blocos de memória. Isto é realizado adicionando uma camada de cooperação de *software* e *hardware* que gerencia o mapeamento entre o endereço virtual e o endereço físico do programa. Esta camada de cooperação é chamada de sistema de memória virtual e é responsável por manter a ilusão que todos os endereços virtuais são residentes na memória e podem ser transparentemente acessados pelo programa. Obviamente que a memória principal é limitada e dispositivos de paginação são utilizados, mapeando endereços para o disco, conforme a disponibilidade.

Os programas geralmente consomem espaços de memória virtual em três regiões: texto – *text* (binário do programa e bibliotecas compartilhadas), pilha e *heap* (alocação de memória dinâmica). Essas regiões não são contínuas e são acessadas de maneira esparsa. Em termos práticos, somente as regiões que estão em acesso corrente precisam residir na memória. Assim, o sistema de paginação deve administrar as regiões de memória com certa granularidade, escolhendo páginas que serão despejadas da memória principal.

	Faltas			
Parâmetro	**Compul.**	**Capac.**	**Conflito**	**Geral**
Redução capacidade	sem efeito	mais	pouco mais	pouco mais
Aumento capacidade	sem efeito	menos	pouco menos	pouco menos
Redução tam. Bloco	mais	pouco menos	pouco menos	variável
Aumento tam. Bloco	menos	pouco mais	pouco mais	variável
Redução associatividade	sem efeito	sem efeito	pouco mais	pouco mais
Aumento associatividade	sem efeito	sem efeito	pouco menos	pouco menos
Write-back vs Write-through	sem efeito	sem efeito	sem efeito	sem efeito
Write-no-allocate	possiv. menor	possiv. menor	possiv. menor	possiv. menor

Tabela 11: Relação de mudança de parâmetros sobre faltas (Apêndice).

O sistema de memória virtual conta com mecanismo de alocação "preguiçosa": ao invés de alocar espaço para a necessidade do programa, espera-se até que o programa realmente faça a referência. Quando acontece uma referência, ocorre uma exceção de página – *page fault*, onde *hardware* registra a falta e transfere o controle ao sistema operacional. Como os dados possivelmente se encontram no disco, a latência do sistema de paginação é grande. Desta forma, possivelmente o sistema operacional escalona outro processo até que os dados encontrem-se disponíveis.

A.5.3 Proteção de memória

Em alguns cenários é desejável que múltiplos processos acessem a mesma página física de memória permitindo comunicação ou evitando duplicação de cópias de bibliotecas compartilhadas ou binários na memória. Contudo, o sistema operacional também é residente na memória e deve proteger

suas estruturas de dados internas dos programas normais. Desta forma, o sistema de memória virtual deve promover mecanismos de proteção de páginas compartilhadas de programas maliciosos ou defeituosos além de garantir que acessos errôneos não corrompam o estado do sistema.

O sistema de memória virtual típico permite que cada página tenha suas próprias permissões de leitura, escrita e execução. O *hardware* é responsável por verificar se instruções de busca ocorrem em páginas com permissão de execução, carregamentos com permissão de leitura e gravações com permissão de escrita. Estas permissões são mantidas em paralelo com o sistema de mapeamento entre endereços virtuais e reais e podem ser apenas manipulados pelo código do sistema operacional. Qualquer referência que viole as permissões são bloqueadas e uma exceção de *hardware* é invocada executando código do sistema operacional que irá lidar com o problema.

Bits de permissão são um mecanismo eficiente para o compartilhamento de objetos somente leitura como binários e bibliotecas. Se há múltiplos processos executando o mesmo binário, o sistema operacional pode mapear a mesma cópia física ao espaço de memória de cada processo. Isto resulta em múltiplas páginas virtuais mapeadas ao mesmo endereço físico. O mesmo princípio é válido para páginas com permissão de escrita compartilhadas entre processos para comunicação.

O mapeamento da memória virtual à física deve ser gravado em uma memória de tradução. O sistema operacional é responsável por atualizar esse mapeamento quando necessário, enquanto o processador deve acessar a memória de tradução para determinar o endereço físico de cada acesso virtual que ocorre. Cada entrada de tradução deve conter o endereço virtual, o endereço real correspondente, *bits* de permissão para leitura, escrita e execução, além de *bits* de referência e modificação. Possivelmente há ainda um *bit* de inibição de *cache*. O *bit* de referência é utilizado para o algoritmo de despejo encontrar páginas para substituição, enquanto o *bit* de modificação sinaliza que o candidato a substituição deve ser gravado no nível inferior antes de ser despejado. O *bit* de inibição de *cache* serve para inibir que páginas de memória sejam gravadas na hierarquia de *cache* do processador. Todas as referências a estes endereços devem ser satisfeitas com acesso direto pelo barramento, como por exemplo dispositivos de entrada e saída mapeados em memória.

As memórias de tradução são comumente chamadas de tabelas de páginas e podem ser organizadas em tabelas de páginas diretas (*forward page tables*) ou tabelas de páginas invertidas (*inverted page tables/hashed page tables*). A primeira organização, mais simples, possui uma entrada para cada bloco do espaço de endereço virtual do processo usando a tabela de páginas, resultando em uma enorme estrutura com muitas entradas não utilizadas.

Desta forma, as tabelas diretas são implementadas em múltiplas seções. Os *bits* de alta ordem dos endereços são adicionados ao registrador base que aponta ao primeiro nível da tabela de página. Outros conjuntos de *bits* são adicionados formando uma espécie de lista encadeada de tabelas. A procura termina quando se encontra a folha da estrutura de dados que corresponderá ao endereço real em conjunto com os *bits* de acesso procurados. Além disso, as entradas da tabela de páginas podem ser armazenadas no sistema de memória virtual, permitindo que estas sejam paginadas no disco. Em função disso podem haver falhas de páginas aninhadas quando a primeira falta experimenta uma segunda falta tentando encontrar a informação de tradução de seu próprio endereço. Se paginação da tabela de página é permitida, a raiz da tabela deve permanecer na memória para evitar falhas de páginas impossíveis de resolver.

Na tabela de página invertida há apenas entradas suficientes para mapear todos os endereços da memória física ao invés de mapear todos os endereços da memória virtual. Como a tabela de páginas contém menor número de entradas, esta pode ser armazenada confortavelmente na memória principal sem necessidade de paginação. Desta forma, o sistema operacional pode acessá-la diretamente pelo endereço físico. A tabela de páginas invertida é basicamente uma tabela *hash* onde o endereço virtual é tipicamente a entrada de função ou-exclusiva e adicionada com endereço base da tabela. O endereço resultante indica um conjunto de endereços que devem ser procurados sequencialmente até encontrar a entrada correta, já que podem haver múltiplas entradas geradas pela função *hash*. Um ponto negativo desse esquema de paginação é que há apenas mapeamentos para as páginas residentes na memória. O mapeamento das páginas despejadas para o disco devem ser gravadas pelo sistema operacional que mantém uma tabela separada.

A última alternativa é não utilizar estruturas de dados particulares residentes em memória para manter as tabelas de páginas. Ao invés disso, utiliza-se uma estrutura chamada de *Translation Lookaside Buffer* (TLB) que pode ser definida como uma parte do modo de proteção do processador. O TLB contém um número pequeno de entradas de páginas (tipicamente 64) organizadas por associatividade completa. O processador provê um rápido *hardware* de pesquisa associativa para converter referências de cada instrução de busca, carregamento ou gravação. Faltas no TLB resultam em faltas de página que invocam o sistema operacional. Este usa sua própria tabela de página, podendo ser outra estrutura de dados para encontrar a tradução correta e atualizar o TLB, utilizando instruções privilegiadas. Neste esquema, o sistema operacional pode estruturar a tabela de páginas da melhor maneira possível, em função das procuras ocorrerem somente no *software* tratador de exceções de página. Esta estrutura de manipulação do TLB é especificada

pelas arquiteturas MIPS, Alpha e Sparc, conhecido como *software TLB miss handler*. Contudo, processadores que implementam uma arquitetura que especifica a organização da tabela de páginas, incluem uma máquina de estados para acesso à memória por *hardware* para procura da tabela de páginas. Tais processadores promovem tradução de endereços para todas as referências de memória, onde a estrutura da tabela de página é fixa, acessada tanto pelo sistema operacional como pelos mecanismos de *hardware*. Este sistema é utilizado nos ISA PowerPC e Intel IA-32, conhecido como *hardware TLB miss handler*.

A.5.4 Interação *cache* e TLB

Um dos problemas do sistema de tradução de endereços de memória é o fato de dois acessos à memória serem necessários para cada referência à memória principal por uma instrução. Primeiramente a tabela de páginas deve ser acessada para obtenção do número da página física, e então chega-se à memória física usando o endereço de tradução. Dependendo da implementação, a tabela de páginas é tipicamente armazenada na memória principal (em endereços alocados pelo sistema operacional), assim cada referência à memória por uma instrução requer dois acessos sequenciais à memória física. Esta situação torna-se um sério gargalo de desempenho. Por essa razão, porções da tabela de página são armazenadas no TLB.

O TLB é basicamente uma memória *cache* para a tabela de páginas. Por esta razão, o TLB pode ser implementado por qualquer esquema de memória *cache*: mapeamento direto, associatividade completa e associatividade por conjunto. O TLB com mapeamento direto é simplesmente uma versão menor e rápida da tabela de páginas.

Para garantir uma utilização mais flexível e eficiente do TLB, utiliza-se associatividade. Na associatividade por conjunto, o endereço virtual é particionado em três campos: índice, etiqueta e deslocamento. O tamanho do deslocamento da página é relacionado com o tamanho da página, sendo esta especificada pela arquitetura e pelo sistema operacional. Os outros campos, índice e etiqueta, constituem o número de página virtual. Na associatividade completa o campo índice é omitido e o campo etiqueta contém o número da página virtual.

A *cache* de porções da tabela de páginas no TLB permite rápida tradução de endereços, contudo faltas no TLB (*TLB misses*) podem ocorrer. Nem todos os mapeamentos de página podem estar simultaneamente presentes no TLB. Quando acessado o TLB, uma falta pode ocorrer e neste caso o TLB deve ser abastecido com as traduções presentes na memória principal. Esta

situação pode resultar ciclos de flutuação no *pipeline*. Além disso, é possível que uma falta no TLB resulte em uma falta de página, devido a possibilidade do endereço nem mesmo existir na tabela de páginas. Neste caso a página referenciada não existe na memória e deve ser buscada no disco ou em outro dispositivo secundário. Para servir a falta de página precisa-se invocar o sistema operacional para acessar o disco e acarretará em dezenas de milhares de ciclos de latência.

A *cache* de dados é utilizada para armazenar porções da memória principal e o TLB é utilizado para armazenar porções da tabela de páginas. A interação entre o TLB e a *cache* de dados pode ser vista na figura 37.

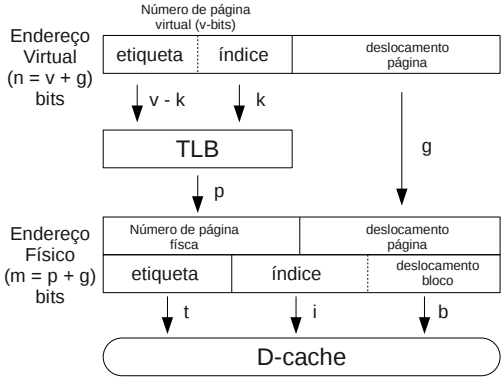


Figura 37: Interação entre *cache* e TLB (Apêndice).

O endereço virtual consiste em um número de página (*v-bits*) e um deslocamento (*g-bits*). Se o TLB tem associatividade por conjunto, os *v-bits* do número de página virtual são divididos em índice (*k-bits*) e etiqueta ($v-k$-*bits*). O segundo estágio da unidade de memória corresponde ao acesso ao TLB utilizando o número de página. Assumindo que não há falta no TLB, o TLB trará o número da página física (*p-bits*), que é então concatenado ao deslocamento (*g-bits*) para produzir o endereço físico de *m-bits*, onde $m = p+g$ e m não é essencialmente igual a n. Durante o terceiro estágio do *pipeline* o endereço físico é utilizado para acessar a *cache* de dados. A exata interpretação do endereço físico depende da arquitetura da *cache* de dados. Se os blocos contém múltiplas palavras, os *b-bits* de baixa ordem são usados como deslocamento para selecionar a palavra específica do bloco selecionado. O bloco selecionado é determinado pelos *bits* restantes ($m-b$) *bits*. Se a *cache* tem associatividade por conjunto esses *bits* restantes são divididos em uma etiqueta (*t-bits*) e um índice (*i-bits*). O valor i é determinado pelo tamanho total da *cache* e sua associatividade: deverá haver i conjuntos na *cache* de as-

sociatividade por conjunto. Se não houve falta da *cache*, no final do terceiro estágio do *pipeline* os dados estarão disponíveis.

A serialização do TLB e *cache* imposta na figura 37 introduz uma latência desnecessária. Uma solução é implementar uma *cache* de dados virtualmente indexada que permite acesso em conjunto com o TLB. Como os *bits* de deslocamento não são traduzidos, eles podem ser utilizados sem a tradução. Os *g-bits* do deslocamento podem ser utilizados como deslocamento do bloco (*b-bits*) e os campos de índice (*i-bits*) da *cache* de dados. Maiores detalhes da implementação da *cache* virtualizada podem ser encontrados em (SEHN; LIPASTI, 2005).

A.5.5 Protocolos de coerência de *cache* para multiprocessadores

Atualmente, protocolos de atualização de *cache* não são mais utilizados devido o consumo excessivo de banda nas interconexões. Desta forma, para sistemas multiprocessados com memória compartilhada utiliza-se protocolos de invalidação. Neste protocolo apenas um processador pode escrever em uma linha de *cache* em qualquer ponto. Esta política é reforçada certificando-se que a linha que será modificada será a única cópia do sistema invalidando as outras presentes nos outros processadores.

O protocolo de invalidação requer que o diretório de *cache* mantenha no mínimo dois estados para cada linha de *cache*: modificado (M) e inválido (I). No estado inválido, o endereço requisitado não está presente e deve ser buscado da memória. No estado modificado o processador sabe que a cópia em questão é exclusiva no sistema e permite-se realizar leituras e escritas na linha. Obviamente que cada linha modificada deve ser regravada na memória. Uma otimização simples incorpora um *bit* sinalizador (*dirty bit*) no estado da linha da *cache* que permite que o processador diferencie as linhas que são exclusivas do processador (conhecido como estado E) e outras que são exclusivas e estão desatualizadas devido uma escrita (estado M). Este protocolo é conhecido como MEI e foi utilizado em processadores PowerPC G3.

Pelo protocolo MEI, não se permite que linhas de *cache* existam em mais de um processador ao mesmo tempo. Para resolver este problema e permitir leituras de linhas, inclui-se o estado compartilhado (estado S (*Shared*)). Este estado indica que uma ou mais cópias remotas existem. Se deseja-se modificar uma linha no estado S, primeiramente modifica-se o estado para o estado M invalidando as cópias remotas. A inclusão do estado S nomeia o protocolo como MESI e as ações de cada estado relacionadas com eventos locais e do barramento podem ser consultadas na tabela 12 onde s e s' são os estados atual e próximo estado do protocolo respectivamente.

	Local			**Barramento**		
Estado Atual	Leitura	Escrita	Despejo	Leitura	Escrita	Atualiz.
Inváll. (I)	ler barr. se $s \neq S$ então s'=E senão s'=S	escrever no barr. s'=M	s'=I	-	-	-
Compar. (S)	-	atualiz. do barr. s'=M	s'=I	Resp. S	s'=I	s'=I
Exclus. (E)	-	s'=M	s'=I	Resp. S s'=S	s'=I	Erro
Modif. (M)	-	-	grava mem. s'=I	Resp. *dirty* grava mem. s'=S	Resp. *dirty* grava mem. s'=I	Erro

Tabela 12: Ações do protocolo de coerência MESI (Apêndice).

Um aprimoramento do protocolo MESI pode ser obtido adicionando o estado *owned* (O), resultando no protocolo MOESI. O estado O é atingido quando acontece uma leitura remota do bloco sinalizado (*dirty*) no estado M. Assim, o estado O significa que múltiplas cópias válidas do bloco existem, onde o requisitor remoto recebeu uma cópia válida satisfazendo a leitura, enquanto o processador local também mantém uma cópia. Contudo ele é diferente do estado convencional S pois se evita a atualização dos dados na memória. Este estado também é conhecido como *shared-dirty*, pois o bloco é compartilhado, mas aguarda a atualização. O sistema que implementa o estado O pode colocar tanto o processador remoto quanto o local em estado S: basta uma cópia ser sinalizada para atualização.

Outro aprimoramento do protocolo MESI é a implementação MESIF (Intel Corporation, 2009). Os estados M, E, S e I representam as mesmas caraterísticas do protocolo MESI clássico, mas adiciona-se o estado *forward* (F). O estado F é um estado S especializado que indica que a *cache* deverá

atuar como fornecedor fixo de uma dada linha compartilhada. O protocolo garante que uma requisição de linha do estado F é enviada somente uma vez e não por todas as *cache*s que contém a linha compartilhada (S). A utilização do protocolo diminui a utilização das interconexões de *cache*.

Uma comparação dos protocolos de coerência de *cache* MOESI e MESIF pode ser encontrada no trabalho de Hackenberg et al. (2009).

A.6 EXECUTANDO MÚLTIPLAS *THREADS* – HYPERTHREADING

O sistema de múltiplas *threads* é dividido em granularidade grossa e fina. Um processador *multithread* de granularidade fina mantém dois ou mais contextos de *threads* internamente. Este sistema faz o escalonamento em granularidade fina usualmente processando instruções das diferentes *threads* em cada ciclo. Contudo esse sistema de *multithread* sacrifica o desempenho em regiões sequenciais.

A granularidade grossa mantém todos os benefícios da baixa. Contudo, ao invés de mudar o contexto de *threads* a cada ciclo, somente o faz quando uma *thread* flutua devido a eventos de alta latência como faltas de *cache*. Para evitar penalidade de mudança de contexto, replica-se os registradores de *pipeline* do processador para cada *thread* e salva-se o estado do *pipeline* a cada mudança de contexto. Desta forma pode-se alternar contextos de *threads* em cada ciclo de máquina.

A tecnologia *HyperThreading* incorporada inicialmente no Pentium 4 permite que dois processadores lógicos compartilhem alguns dos recursos do processador. As partes do *pipeline* com execução sequencial, busca e decodificação são *multithread* de granularidade fina. Assim, duas *threads* podem buscar, decodificar e retirar instruções em ciclos alternados, a não ser que uma *thread* é parada por alguma razão. Neste caso, a outra pode consumir todos os recursos unicamente. O Pentium 4 também implementa uma lógica de escalonamento de dois estágios, onde as instruções são colocadas em cinco filas de início no primeiro estágio, mas somente são iniciadas nas unidades funcionais no segundo estágio. O primeiro estágio possui escalonamento *multithread* de granularidade fina: apenas uma *thread* pode colocar instruções nas cinco filas em um ciclo. Novamente, se uma *thread* sofre flutuação, outra pode colocar instruções nas filas até que a flutuação seja resolvida.

Na essência, o Pentium 4 implementa uma combinação de *multithread* granularidade fina e grossa em todos os estágios do *pipeline*. Há *threads* simultâneas no segundo estágio de início (antes das unidades funcionais) bem como nos estágios de execução e memória, permitindo que instruções de ambas as *threads* combinem-se de forma arbitrária.

Nos sistemas com *HyperThreading* habilitado, haverá disponível o dobro de processadores do que há fisicamente. Como há duplicação completa do estado da arquitetura envolvendo os registradores de propósito geral, registradores de controle, controladores de interrupção e alguns registradores de estado da máquina, o sistema operacional controla os processadores lógicos da mesma forma que os processadores físicos. Porém otimizações como parar um processador lógico (instrução *halt* que desliga compartilhamento dos recursos) quando não há tarefas disponíveis e escalonar tarefas primeiramente nos processadores físicos garantem um melhor desempenho.

O compartilhamento dos recursos do Pentium 4 é relativamente complicado. A maior parte dos *buffers* na porção "fora de ordem" do *pipeline* são particionados pela metade ao invés do compartilhamento arbitrário. Este particionamento fixo de recursos sacrifica o desempenho na execução de *threads* únicas para melhorar quando há duas disponíveis. Os recursos que utilizam pouca área são duplicados (registradores, controladores interrupção e TLB de instrução) enquanto outros mais complexos são particionados ou totalmente compartilhados (*cache* de dados, TLB de dados, unidades funcionais). Em (MARR et al., 2002) há o detalhamento completo dos recursos particionados e compartilhados do processador.

De um nível mais alto, este particionamento de recursos pode funcionar muito bem quando duas *threads* são simétricas em comportamento, mas baixo desempenho se elas são assimétricas e tem necessidade de recursos diferentes. Esta situação pode ser ilustrada pelo compartilhamento da *cache* de dados: como os processadores lógicos compartilham os dados, há condições para faltas de conflito, contudo a simetria de *threads* pode indicar um modelo produtor-consumidor que acessa os mesmo dados. De forma geral, o Pentium 4 suporta um modo de única *thread* onde particionamento dos recursos é desabilitado.

O *HyperThreading* é uma implementação particular da técnica de execução de múltiplas *threads* conhecido como *Simultaneous Multithreading* (SMT). Maiores detalhes e variações podem ser encontrados em (SEHN; LIPASTI, 2005).

A.7 DESEMPENHO DE PROCESSADORES

O objetivo principal no desenvolvimento de microarquiteturas é aumentar o desempenho, mas recentemente tem-se visado a redução de consumo de energia dos processadores. A equação A.12 (SEHN; LIPASTI, 2005) apresenta termos relacionados com a métrica desempenho.

$$\frac{1}{desemp.} = \frac{tempo}{programa} = \\ = \frac{instrucoes}{programa} \times \frac{ciclos}{instrucao} \times \frac{tempo}{ciclo} \quad (A.12)$$

Primeiramente, o desempenho é relacionado com quanto tempo o processador leva para executar dado programa ($tempo/programa$). Esta métrica pode ser formulada pela multiplicação dos três termos seguintes, onde o primeiro ($instrucoes/programa$) indica o número de instruções necessárias para executar o programa particular; o segundo representa a média de quantos ciclos de máquina são consumidos por cada instrução – *cycles per instruction* (CPI); e o terceiro representa o tempo de cada ciclo de máquina.

Verificando a equação A.12, o desempenho pode ser aumentado reduzindo qualquer uma das três partes da equação. Quanto menos instruções, menos tempo. Se o CPI é reduzido, instruções executarão mais rapidamente e, em último caso, reduz-se o tempo de cada ciclo. Contudo aumentar o desempenho não é uma tarefa trivial, pois os três termos não são independentes havendo interações complexas entre eles: redução de qualquer um deles pode favorecer o aumento de outro.

Em termos de otimização, algumas técnicas podem reduzir um dos termos mantendo outros constantes, como por exemplo, otimização executada por compiladores e aumento da frequência de operação. Para ilustrar a dependência dos termos, considere reduzir o número de instruções utilizando instruções mais complexas, resultando em um ISA CISC – *Complex Instruction Set Computer*. Enquanto o número de instruções decresce, a complexidade das unidades de execução aumenta, levando a um potencial aumento do tempo de ciclo. Para compensar, utiliza-se um *pipeline* com mais estágios. Este *pipeline* requer uma unidade de previsão de fluxo que pode errar elevando a quantidade de ciclos por instrução. Técnicas de otimização de laços podem reduzir o número de instruções mas aumentam o tamanho estático do código. Esse aumento pode influenciar negativamente a taxa de acerto da memória *cache* que aumenta o CPI.

O desenvolvimento de processadores avançados sempre leva em consideração a redução dos ciclos por instrução. Os pontos chave de redução levam em conta a paralelização através de *pipelines* longos, utilização de memórias *cache* para redução da latência de memórias e utilização de previsão de fluxo avançado. Este último recurso reduz o problema de flutuação (*stall*) no *pipeline* mas incorpora uma maior complexidade em cada sub-estágio.

A técnica chave para reduzir o tempo de ciclo é efetivamente o *pipe-*

line. Esta técnica particiona o processamento de cada instrução em múltiplos estágios: um estágio decodifica a instrução, outro acessa a memória, outro é responsável por operações aritméticas, etc. A latência de cada estágio determina o tempo de ciclo da máquina. Com a utilização de *pipelines* longos, a latência de cada estágio e o tempo de ciclo pode ser reduzida. Atualmente a utilização de *pipelines* agressivos é a técnica comum em processadores com elevada frequência de operação.

Há pontos interessantes sobre a elevação da frequência de operação utilizando *pipelines* longos, sendo o CPI influenciado por três aspectos. Primeiramente, como a entrada (*front end*) do *pipeline* torna-se longa, o número de estágios entre operações de busca e execução de instruções (*fetch/execute*) aumenta. Esta situação aumenta os ciclos de penalidade relacionados com o erro de previsão do fluxo de controle (*branch miss-prediction*), resultando em um aumento do CPI. Em segundo lugar, se o *pipeline* é tão longo que uma operação aritmética primitiva requer múltiplos ciclos, a latência necessária entre duas instruções dependentes será de múltiplos ciclos. Em terceiro lugar, como a frequência de operação é alta, a latência da memória em termos de ciclos aumenta significativamente. Consequentemente, há aumento do tempo médio de operações de memória, e novamente, há aumento do CPI.

De certo modo, a utilização de *pipelines* desenvolve o paralelismo temporal. Processadores sequenciais tradicionais executam uma instrução no tempo, onde uma instrução deve terminar até a execução da próxima. Como há tantas instruções em execução quanto estágios de *pipeline*, esta técnica desenvolve o processamento paralelo a nível de instrução.

Em processadores escalares com *pipeline* há ainda a limitação de busca de no máximo uma instrução para o *pipeline* a cada ciclo de máquina. Com esta limitação, o melhor CPI possível é 1, ou inversamente, desempenho (*throughput*) máximo é 1 instrução por ciclo – *Instruction per cycle* (IPC). Uma forma mais agressiva de processamento paralelo pode ser desenvolvida buscando e iniciando múltiplas instruções dentro de um *pipeline* mais largo a cada ciclo de máquina. Desta maneira o objetivo é reduzir o CPI abaixo de 1 ou obter um *throughput* maior que 1. Processadores capazes de realizar um IPC maior que 1 são conhecidos como processadores superescalares.

Assim, processadores escalares são processadores com *pipeline* capazes de buscar e iniciar no máximo uma instrução a cada ciclo. Processadores superescalares são aqueles que buscam e iniciam múltiplas instruções a cada ciclo.

Fazendo-se o inverso da equação A.12, tem-se a medida de desempenho médio em função do IPC.

$$
\begin{aligned}
desempenho &= \frac{1}{numero\_instrucoes} \times \frac{instrucoes}{ciclo} \times \frac{1}{ciclo} \\
&= \frac{IPC \times frequecia}{numero\_instrucoes} \qquad \text{(A.13)}
\end{aligned}
$$

Percebe-se claramente na equação A.13 que o desempenho é diretamente proporcional ao IPC e à frequência e inversamente ao número de instruções. O número de instruções é determinado pelo conjunto de instruções, compilador e pelo sistema operacional. O ISA e a quantidade de trabalho codificado em cada instrução influencia enormemente a quantidade de instruções executadas por um programa. A eficiência do compilador e as funções do sistema operacional invocadas durante a execução também contribuem para o desempenho do sistema. O IPC médio reflete o *throughput* de instruções alcançado pelo processador e é uma medida fundamental de efetividade de microarquitetura.

Conforme utiliza-se processamento paralelo é interessante verificar as limitações descritas pela de Lei de Amdahl (SEHN; LIPASTI, 2005). Computadores tradicionais são processadores paralelos que realizam operações escalares e vetoriais. As computações escalares são sequenciais e computações vetoriais com N unidades paralelas são utilizadas.

Considerando, conforme a figura 38, que s é o tempo que a máquina executa operações escalares e $(1-s)$ o tempo que realiza operações vetoriais, a equação A.14 representa a eficiência de um supercomputador em termos da Lei de Amdahl.

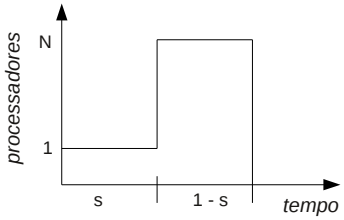


Figura 38: Processamento escalar e vetorial de um supercomputador (Apêndice).

$$
E = \frac{s + N.(1-s)}{N} = 1 - s.\left(1 - \frac{1}{N}\right) \qquad \text{(A.14)}
$$

Com o crescimento de N a equação A.14 tende a $1-s$ que é o tempo que a máquina passa executando operações vetoriais. Com N grande, este tempo tende a ficar cada vez menor. Portanto, a eficiência da máquina tende a zero, ou seja, grande parte do tempo de computação é gasto em computações escalares e o aumento de N tem baixo impacto na redução do tempo total de execução.

Seguindo este mesmo princípio, considera-se na figura 39 a porção v de trabalho que pode ser vetorizado, e consequentemente a porção $(1-v)$ que se executa sequencialmente. Sendo T o tempo total para executar o programa, o fator de velocidade S é representado pela equação A.15.

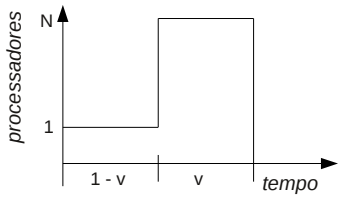


Figura 39: Porções de trabalho vetorizado e sequencial (Apêndice).

$$S = \frac{1}{T} = \frac{1}{(1-v)+(\frac{v}{N})} \tag{A.15}$$

Novamente, com N grande, o fator de velocidade depende basicamente da porção $(1-v)$ que é a parte sequencial do programa. Esta situação é conhecida como gargalo sequencial, que demonstra que a parte escalar da computação torna-se um limite de quanto de desempenho pode-se alcançar com a exploração do paralelismo.

Conforme Sehn e Lipasti (2005), o modelo de desempenho de um *pipeline* segue as mesmas características da Lei de Amdahl para processamento paralelo. Desta forma, na figura 40 apresenta-se o perfil de desempenho de um *pipeline* com N estágios onde a porção $(1-p)$ representa o tempo de inicialização (execução parcialmente sequencial) e p representa as partes em operação total (totalmente paralelo) e término de execução das instruções.

O crescimento inicial em degrau representa a evolução dos estágios em cada ciclo e esta situação também ocorre no término da execução das instruções. Em termos de modelagem, crescimento inicial é idêntico ao final formando a área limitada pelos traços tracejados. Portanto, o perfil de desempenho apresentado na figura 39 é idêntico ao perfil da figura 40 e con-

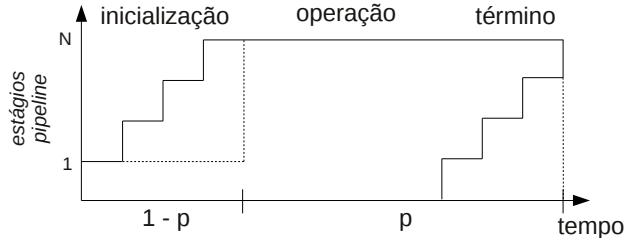


Figura 40: Perfil de execução de um *pipeline* de *N* estágios (Apêndice).

sequentemente pode ser calculado pela equação A.16.

A modelagem da figura 40 não leva em conta possíveis flutuações (*stalls*) do *pipeline* que devem ser considerados. Uma modificação sucinta pode ser realizada, mudando o estágio de operação, conforme a figura 41, incluindo vários estágios de inicialização.

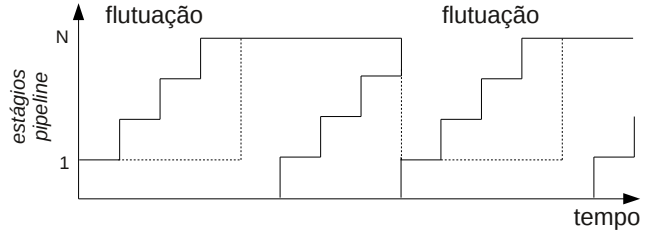


Figura 41: Perfil de execução de um *pipeline* com flutuação (Apêndice).

Na modelagem com flutuações, o parâmetro p torna-se uma fração na qual o *pipeline* está em operação. Como o aumento de estágios, os ciclos de flutuação podem ser muito custosos em termos de desempenho, e portanto são similares ao gargalo sequencial do processamento paralelo.

$$S = \frac{1}{(1-p)+(\frac{p}{N})} \tag{A.16}$$

A equação A.16 representa um modelo simples pois não considera possíveis otimizações que o *hardware* moderno contém. Quando ocorre uma flutuação, assume-se que há apenas uma instrução em execução, contudo, em função da utilização de caminhos de atalhos e outros mecanismos, os ciclos de penalização devido a flutuações são menores. A equação A.17 é

uma generalização da anterior, onde g_1 representa a fração do tempo na qual há apenas uma instrução no *pipeline*, g_2 duas instruções e g_N é a fração do tempo em operação normal.

$$S = \frac{1}{\frac{g_1}{1} + \frac{g_2}{2} + ... + \frac{g_N}{N}} \tag{A.17}$$

Para ilustrar a aplicação da equação A.17 considera-se um *pipeline* de 6 estágios que possui um fator de velocidade teórico igual a 6. Considerando que 13% das instruções executadas geram penalidade de 4 ciclos, 25% geram penalidade de 1 ciclo e os restantes 62% não geram penalidade, tem-se um fator de velocidade conforme equação A.18.

$$S = \frac{1}{\frac{0.13}{6-4} + \frac{0.25}{6-1} + \frac{0.62}{6}} = 4.5 \tag{A.18}$$

Verifica-se portanto, que com um *pipeline* de 6 estágios, perde-se 25% da efetividade quando se considera possíveis flutuações causadas por dependência de dados, controle ou acesso a memória.

As equações até então apontam o desempenho de máquinas com vários estágios de *pipelines* mas não contam com a possibilidade de superescalabilidade. Isto significa que por mais complexo e rápido que seja o *hardware* o IPC mínimo teórico é limitado em 1. A suposição que toda a parte não vetorizada do programa deve ser executada sequencialmente reflete um pessimismo enorme e não necessário. Se algum nível de paralelismo pode ser alcançado na parte sequencial, os impactos severos do gargalo podem ser significativamente moderados. Supõe-se agora dois casos de fator de velocidade com $N = 6$ apontados pelas equações A.19 e A.21 e um caso com $N = 100$ apontado pela equação A.20.

$$S_1 = \frac{1}{(1-v) + \frac{v}{6}} \tag{A.19}$$

$$S_2 = \frac{1}{(1-v) + \frac{v}{100}} \tag{A.20}$$

$$S_3 = \frac{1}{\frac{(1-v)}{2} + \frac{v}{6}} \tag{A.21}$$

Em ambos os casos com $N = 6$, o nível de vetorização máximo é 6, contudo na equação A.21 considerou-se que a porção não vetorizada tem uma paralelização mínima de 2.

Examinando-se as curvas das equações A.19, A.20 e A.21, a curva S_3 é sempre superior a S_1 e também superior em 75% do tempo a S_2. Isto significa que para casos de vetorização menor de 75% , é muito mais benéfico construir um sistema com vetorização de apenas 6, mas garantindo que a parte sequencial tenha um paralelismo mínimo de 2.

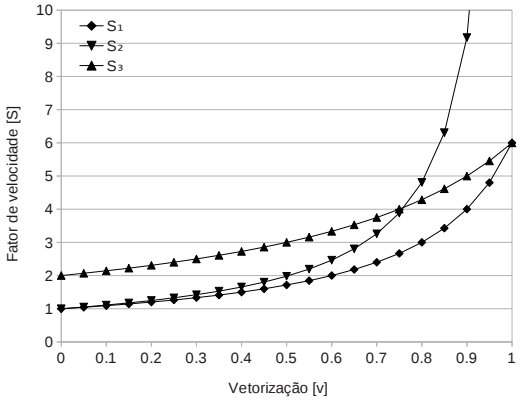


Figura 42: Modelo de um *pipeline* com flutuação (Apêndice).

A vetorização de um programa é uma função complexa que envolve linguagem de programação, algorítmica, compilador e a arquitetura. Assim grande parte dos programas de propósito geral possuem uma parcela vetorizada menor que 75%. Portanto, o objetivo dos processadores superescalares é obter um paralelismo genérico a nível de instrução e consequentemente um fator de velocidade superior para qualquer tipo de programa, incluindo partes que são sequenciais.

A.8 CONCLUSÕES E TENDÊNCIAS

Devido às barreiras de aumento de frequência ocasionada pelo fuga de tensão dos componentes internos do *chip* e altas perdas térmicas, fabricantes têm desenvolvidos processadores com mais núcleos ou com mais *threads* de *hardware* (*HyperThreading*, etc) (SODAN et al., 2010).

Antigamente, os desenvolvedores de processadores usavam a área adi-

cional devido a miniaturização para o desenvolvimento de processadores superescalares com replicação das unidades de execução e *pipelines* profundos para explorar o paralelismo ao nível de instrução. Contudo, a atual direção emprega o espaço disponível para adicionar mecanismos de *multithread* ou *multicore*. Esses processadores suportam multitarefa rodando programas em paralelo ou ainda diversos programas concorrentes.

Os CPUs *multithread* suportam execução concorrente de *threads* ao nível de instrução, com o objetivo de melhorar a utilização do recurso enviando instruções de várias *threads*. CPUs *multicore* possuem uma concorrência em um nível mais alto, focando em menos utilização por núcleo objetivando a escalabilidade com replicação de núcleos.

Cada *thread* de *hardware* precisa de seus próprios componentes de estado, como registradores de controle e ponteiros de instrução. Contudo, o Intel Xeon precisa apenas de um adicional de 5% de espaço para suportar uma segunda *thread*. Atualmente, processadores comerciais apenas enviam instruções de duas *threads* por core por ciclo, mas existem protótipos como o Tera/Cray MTA com 128 *threads*, apropriado para *High performance computing*.

O *multithread* por núcleo tem limitação de escalabilidade, limitada pela saturação das unidades de execução e o custo de *threads* adicionais. Por isso, os CPUs *multicore* são mais escaláveis. Estes *chips* mantém muito da arquitetura dos seus antecessores, replicando apenas as unidades de controle e execução e compartilhando outras, como: *cache*, controlador de memória, FPUs, componente de refrigeração, pinos, etc. No entanto, compartilhamento tem a desvantagem da necessidade de controle e da saturação desses recursos.

Tendências de desenvolvimento indicam a replicação de componentes adicionais em CPUs *multicore*, como controladores de memória e *caches*. A arquitetura de memória varia, havendo *caches* privadas nos níveis próximos ao processador e compartilhada no nível sucessor à memória principal. Colocando o controlador de memória no *chip* aumenta a banda e diminui a latência, o que justifica a atual tendência. Outra recente pesquisa, condiz com a integração de GPUs ao CPUs, formando processadores híbridos.

Devido a grande diferença de desempenho das memórias e dos CPUs, estes atualmente contém grande quantidade de memória *cache* no *chip*. Outras arquiteturas não utilizam *cache*, aliviando o problema da memória com altos níveis de *multithreading* (Tera/Cray MTA), ou utilizando memórias de alta velocidade diretamente acessadas (IBM Cell SPE).

O benefício de *caches* privadas ou compartilhadas não depende somente do espaço disponível no *chip*, mas também das características da aplicação. *Caches* compartilhadas são importantes se as *threads* da mesma aplicação executando em múltiplos núcleos compartilham uma quantidade sig-

nificativa de dados. Contudo, *caches* compartilhadas impõe alta demanda de interconexão. Aplicações que não compartilham muitos dados vão competir por *cache*. Esta situação dificulta a previsão de cada *thread* pois depende de detalhes como padrões de acesso, localidade de memória como também a carga do sistema. Novas pesquisas propõem uma abordagem híbrida de *cache* que permite particionamento dinâmico.

A interconexão dos componentes impacta diretamente na performance do CPU. Inicialmente, utilizava-se um barramento para essa conexão. Contudo, a tendência é utilizar mecanismos avançados como *crossbar* para reduzir a latência e contenção. A AMD emprega o *crossbar* enquanto a Tilera implementa um *fast nonblocking multilink mesh*. Esse tipo de interconexão é relativamente caro. Por exemplo, um *crossbar* de 8 x 8 consome tanta área de *chip* quanto cinco núcleos e consome energia equivalente a dois (SODAN et al., 2010).

A comunicação para componentes exteriores evolui de barramento (*Front Side Bus* – FSB) para comunicação de pacotes, com conexão ponto a ponto. A AMD implementa esse conceito com o *HyperTransport* seguido pela Intel como o *QuickPath Interconnect*. Estas conexões podem interligar vários processadores em diferentes *chips* como também todo o sistema de entrada/saída.

Processadores que focam no desempenho por *thread* possuem frequências mais elevadas dos que focam no *multithread*. Contudo, a utilização do espaço do *chip* para aumento por *thread* resulta em ganhos não lineares, onde a experiência sugere que a performance apenas dobra com o quádruplo da complexidade dos componentes adicionais para suporte de *threads* extras.

Grandes quantidades de núcleos simples são preferíveis desde que a parte sequencial da aplicação seja pequena, caso contrário, CPUs complexos têm provado ser mais efetivos. O desempenho não domina mais o objetivo do desenvolvimento: custos de fabricação, tolerância a falhas, eficiência energética e dissipação de calor são também considerações críticas. Como os núcleos são simplificados, o consumo energético diminui linearmente, o que é uma grande vantagem de processadores *multicore*.