Cálculo do Tempo de Execução de Códigos no Pior Caso (WCET) em Aplicações de Tempo Real: Um Estudo de Caso

Bruna C. Oliveira[1], Max Mauro Santos[1], Fernando Deschamps[2]

[1]Laboratório de Sistemas de Tempo Real, Centro Universitário do Leste de Minas Gerais, Campus Universitário, Bairro Universitário, Coronel Fabriciano, MG, Brasil
brunerrima@alunos.unilestemg.br, maxmauro@unilestemg.br

[2]Pontifícia Universidade Católica do Paraná, Prado Velho, Curitiba, PR, Brasil
fernando.deschamps@pucpr.br

Resumo

O *WCET* vem sendo estudado há poucos anos por pesquisadores da área de sistemas de tempo real para tentar melhorar ou minimizar os defeitos ocorridos em aplicações de tempo real. Uma vez que, o *WCET* aprofunda-se a nível de *hardware* e *software*, fazendo cálculos em aplicações de baixo nível. Logo, este trabalho tem como principais objetivos, dar uma visão geral sobre o que seria *WCET*, apresentar as principais técnicas para se fazer o cálculo da análise de fluxo, apresentar os componentes da análise estática e contribuir para disseminação do assunto.

Palavras-chave: Pior Caso do Tempo de Execução, Melhor Caso do tempo de Execução, Caso Médio do tempo de Execução, Técnica de Enumeração do Caminho Implícito, Sistema de Tempo Real, Pipeline.

Abstract

The *WCET* come being studied – few years for researchers of the real time systems area for try to improve or minimize the defects occurred in real time applications. Once, the *WCET* it is treated the level of *hardware* and *software*, making calculations in level low applications. Soon, this work have as main objectives, give one vision generality about the that it would be *WCET*, to present the main techniques for if make the calculation of the flow analysis, to present the components of the static analysis and contribute for dissemination of the subject.

Key-word: Worst Case Execution Time (*WCET*), Best Case Execution Time (*BCET*), Average Case Execution Time (*ACET*), Implicit Path Enumeration Technique (*IPET*), Real Time System (*STR*), Pipeline.

1 Introdução

As aplicações de tempo real [8], [10] estão se tornando cada vez mais importante nos dias atuais, pois a maioria dos processadores utilizados tem suporte para aplicações de tempo real. Uma vez que, a maioria dos aparelhos eletro-eletrônicos possui um sistema embarcado [4], [5] em seu interior. Sendo assim, é de extrema importância relatar, que os STR estão cada vez mais ligados aos sistemas embarcados.

Os sistemas de tempo real podem ser divididos em dois sistemas: crítico e brando, nos quais quando o tempo de execução ultrapassa seu deadline, pode ou não causar grandes estragos.

Um exemplo para sistemas crítico seria o acionamento de airbag de automóveis. Caso este não seja

acionado no momento imediato da colisão, poderá ocasionar a perda das vidas dos passageiros. Um exemplo para sistemas brando seria a transmissão de um vídeo via web. Neste caso, o não cumprimento em tempo exato poderá ocasionar apenas falhas na transmissão, na qual a imagem não chegará no mesmo instante em que ela foi computada.

Por isso, em sistemas de tempo real o comportamento temporal de cada tarefa é muito importante. Tem que ser garantido que neste sistema, as tarefas sejam finalizadas antes de seu deadline, mesmo no pior caso. Por este motivo o *WCET* (Worst Case Execution Time) [6] vem sendo estudado há poucos anos pela comunidade de pesquisadores da área de sistemas de tempo real, para tentar melhorar ou minimizar os defeitos ocorridos em tais aplicações.

A aplicação do *WCET* nesses sistemas permite trilhar os caminhos de execução, e fazer o cálculo através dos caminhos encontrados ao longo dos programas. Ele aprofunda-se no nível de hardware e software, fazendo cálculos em aplicações com instruções de baixo nível. Para tais cálculos, ele utiliza-se de três estágios para fazer a análise do fluxo que são: extração, representação e cálculo.

2 Visão Geral do WCET

Para se obter uma aplicação segura, é muito importante saber quais são as principais estimativas do tempo de execução de um programa.

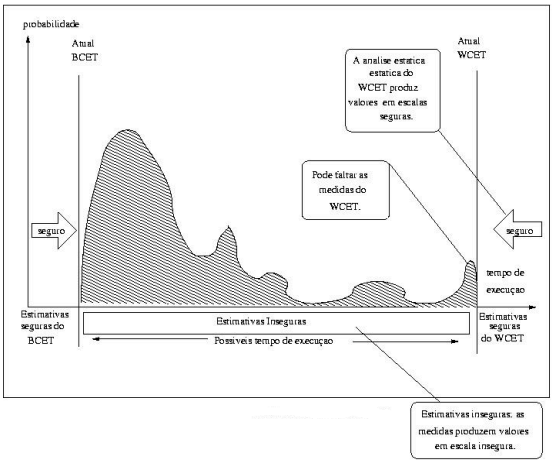

Figura 1: Estimativas do tempo de Execução

As três estimativas do tempo de execução são: *WCET*, *BCET* e o *ACET*. O *WCET* pode ser definido como sendo o maior tempo (caminho) que o tempo de execução de um programa leva para ser executado em sua plataforma alvo (hardware). O *BCET* é então definido como sendo o menor tempo, e o *ACET* seria a média dos tempos de execução entre o *WCET* e o *BCET*.

O objetivo da análise do tempo de execução é produzir estimativas do *WCET* e *BCET*. Para ser válido o uso das estimativas em *STR* crítico, a estimativa do *WCET* deve ser totalmente segura, garantindo assim o bom desempenho dos programas. A Figura 1 mostra as estimativas do *WCET* e *BCET* em um programa.

As curvas da Figura 1 demonstram as estimativas inseguras, na qual as medidas produzem valores em escalas inseguras. Também nela podemos presenciar que o lado esquerdo antes das curvas, e o lado direito após as curvas produzem valores em escala segura através da análise do *BCET* e da análise estática do *WCET* [14].

Logo, para se obter uma estimativa de execução segura, o *WCET* [2] tem que ser medido. Para que isto ocorra, a medida é dividida em duas fases que são: a metodologia e a definição do problema.

Esta metodologia é basicamente a seguinte: executar o programa várias vezes e tentar diferentes entradas de valores de piores casos para o cálculo do *WCET*. Isto será o tempo consumido e o trabalho difícil, no qual nem sempre oferece resultados que podem ser garantidos. Então quando usamos medidas, uma margem segura pode ser adicionada para o resultado obtido, na espera de que o pior caso seja abaixo da estimativa do resultado do *WCET*, como mostrado na Figura 1. Entretanto, se muitas margens forem adicionadas, os recursos serão desperdiçados, e se poucas margens forem adicionadas, o sistema resultante será inseguro.

Para que isto não ocorra, temos que usar a análise estática do *WCET* [12], pois ela evita a necessidade de executar o programa simultaneamente considerando os efeitos de todas possíveis entradas de valores, possíveis fluxos de programas e a maneira na qual o programa interage com o hardware. E isto é feito utilizando modelos de cálculos matemáticos [9] que envolvem o software e o hardware. Sendo assim, o resultado da estimativa do *WCET* deve ser igual que o pior caso, tendo então uma segurança em todas as circunstâncias.

É válido lembrar também que, quando usamos a análise estática, não é necessário setar o atual alvo do sistema. Com isso, fica claro perceber que um dos principais problemas da definição do *WCET* [6] é a interação com o resto do sistema, pois ela é válida somente para um programa isolado. Quando o programa tenta interagir com o sistema, ele produz estimativas inseguras. Logo, os meios que interferem nas atividades de segundo plano (*background*) como: *DMA* (*Direct Memory Access*) ou refresh da *DRAM* (*Dynamic Memory of Random Access*), e os meios que interferem diretamente no *SO* (*Operational System*) como: as interrupções ou preempções, não podem ser levados em consideração. A questão é então conseguir fornecer uma estimativa do *WCET* segura para um programa simples, no qual pode ser executado em um ambiente específico de uma plataforma de hardware particular.

Essa análise possui alguns usos. Seu principal uso é no desenvolvimento e análise de *STR*. Nesses sistemas as estimativas do *WCET* são usadas para fornecer análises de escalonamento e correção temporal, no qual a análise de escalonamento e a análise do *WCET* formam a base de um *STR*. Dessa forma, a análise de *WCET* fornece a garantia do tempo para todos os comportamentos do sistema, sendo considerada então como uma ferramenta natural para aplicações em sistemas de tempo real.

As áreas de aplicação nas quais tais ferramentas do *WCET* podem ser usadas incluem controladores de tempo real e protocolos de comunicação. Em particular aquelas áreas nas quais os fatores de tempo são críticos.

Em *STR* críticos, a análise do *WCET* tem que garantir o comportamento temporal em todas as circunstâncias, principalmente no pior caso. Em *STR* brandos esta análise não precisa garantir o pior caso, uma vez que caso as tarefas ultrapassem seus *deadlines*, isto não implicará em perdas catastróficas do comportamento do sistema. Logo, o domínio tradicional e de maior importância do *WCET* seria o *STR* crítico.

Para partes de códigos de *STR* crítico [10], as estimativas do *WCET* podem ser usadas para verificar que o tempo de execução é bem pequeno, que manipuladores de interrupção são finalizados rapidamente, ou que o teste padrão de um controle de laço pode ser seguro. Um exemplo de *STR* crítico seria em desenvolvimento de sistemas embarcados, no qual a análise do *WCET* seria usada para selecionar o hardware apropriado (alvo).

Outros usos da análise do *WCET* seriam na execução de caminhos de programas. Esses caminhos de programas podem ser vistos como: laços limitados (*For*, *While*, *Do-While*), sequências de laços, estruturas condicionais (*If*, *Else*) e recursividade de laços.

3 Componentes da Análise Estática do WCET

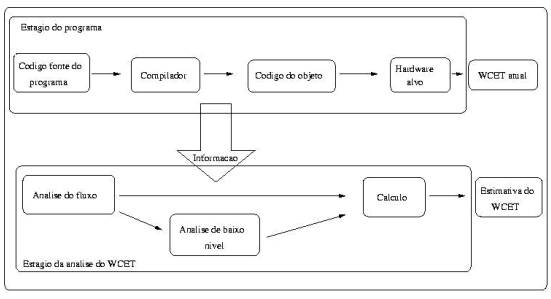

Figura 2: Componentes da Análise Estática do WCET

Como pode ser visto na Figura 2, o tempo de execução do programa depende de vários fatores. Os fatores estão distribuídos em dois estágios que são: estágio do programa e estágio da análise do *WCET*.

No estágio do programa existe: o código fonte do programa, o compilador, o código do objeto, e o hardware alvo. O código fonte do programa define as possíveis instruções e caminhos de execução para serem executados. O compilador transforma o código do programa de alto nível para um código de objeto equivalente. O código do objeto é então executado no hardware alvo, e o atual *WCET* equivale então ao maior tempo de execução que pode sempre ser observado quando o programa é executado.

No estágio da análise do *WCET* existem três fases: a análise do fluxo, a análise de baixo nível e o cálculo.

A análise do fluxo [7] determina o comportamento dinâmico de um programa, ou seja, ela determina os possíveis fluxos através do programa e as possíveis seqüências de instruções que podem ser executadas. Nesta fase, além de se ter uma determinação do comportamento dinâmico do programa, ela também informa sobre quais funções serão chamadas ao longo do programa com seus respectivos componentes, como por exemplo: número de interações de laços, profundidade ou tamanho das recursões, dependências das entradas (*If*, *For*, etc) e caminhos impraticáveis. Para se ter uma boa informação do fluxo, a informação de execução tem que ser segura para todos os caminhos praticáveis e impraticáveis. Esta fase é então sub-dividida em três sub-fases que são: a extração do fluxo, a representação do fluxo e a conversão do cálculo.

A extração do fluxo fornece informações sobre o comportamento do programa através de anotações manuais ou métodos automáticos da análise do fluxo. Nesta sub-fase, é muito difícil às vezes fornecer informações dos fluxos, pois diferentes aproximações tem diferentes significados e ainda, geram diferentes somas de informação. Para evitar isto, devemos complementar e/ou adicionar as anotações manuais nos métodos automáticos da análise de fluxo, pois as anotações manuais permitem ao programador comentar o programa com maiores detalhes.

A representação do fluxo transforma os resultados da extração do fluxo em esquemas de representações de árvores para interagir com os resultados dos vários métodos automáticos da análise do fluxo e/ou anotações manuais. Aqui, a informação do fluxo é extraída, e será representada através de gráficos, árvores, ou código do programa. Ela deve ser dada em relação ao código fonte ou ao código do objeto.

E a conversão do cálculo usa a informação do fluxo, convertendo-a para o cálculo final do *WCET*. Nesta sub-fase, o fluxo de informação representado é convertido para a fase final do cálculo do *WCET*.

A análise de baixo nível determina o tempo de execução para partes do programa no hardware, ou seja, determina o comportamento de sincronismo para as instruções que estão sendo executadas no hardware alvo.

Nesta fase é determinado o tempo de execução para cada unidade atômica do fluxo tais como: uma instrução, um bloco básico, ou um caminho de execução longo. É importante fixar que um bloco básico é uma seqüência de instruções que podem somente ser começadas na primeira instrução da seqüência e finalizadas somente na ultima instrução da seqüência. Esta fase é sub-dividida em 2 sub-fases: a análise global de baixo nível e a análise local de baixo nível.

Na sub-fase da análise global de baixo nível é determinado o efeito do sincronismo da máquina, dependente de fatores que necessitam ser modelados sobre todo o programa. Os exemplos de algumas características dessa sub-fase são: instruções de *cache*, dados de *cache* [11], [13] e partes da predição [1]. Para alguns comportamentos de sincronismo de microprocessadores, é requerido que se obtenha um resultado seguro do programa. Por exemplo: para se determinar o comportamento de uma instrução da *cache*, a análise deve ser feita de modo que se deva considerar várias instruções para a instrução considerada, ou seja, levar em conta as instruções do programa.

Na sub-fase da análise local de baixo nível é determinado o efeito do sincronismo da máquina, dependente de fatores que podem ser manipulados localmente para instruções adjacentes de um programa simples. Alguns exemplos são: acesso rápido a memória e sobreposição de *pipeline*.

O cálculo combina os resultados dos tempos da análise do fluxo e de baixo nível para assim poder dar uma estimativa do *WCET* para o programa. Nesta fase, é calculada uma estimativa do *WCET* para o programa, dado o fluxo do programa e os resultados da análise global e local de baixo nível. Esta fase pode ser sub-dividida em três tipos de cálculos que são: cálculo baseado na árvore, cálculo baseado no caminho e o cálculo baseado na *IPET*.

Na sub-fase do cálculo baseado na árvore é feita uma representação de todo o programa em nodos que descrevem a estrutura do programa como: seqüências, laços ou condições, que resultam em blocos básicos.

Na sub-fase do cálculo baseado no caminho é feita a estimativa do *WCET* que é gerada pelo cálculo dos tempos dos diferentes caminhos executados no programa, procurando sempre o caminho completo com o maior tempo de execução. Neste tipo de cálculo a principal característica é que os possíveis caminhos de execução são explicitamente representados.

Na sub-fase do cálculo baseado na *IPET* é feita uma representação do fluxo do programa e dos tempos de execução usando a álgebra e/ou lógica. Para cada bloco básico do programa e/ou extremidade no gráfico do bloco básico é dado uma variável de tempo (t), que denota no tempo de execução de cada bloco básico, e uma variável de contagem (x), que denota o número de tempos que cada bloco básico ou extremidade é executado. O *WCET* é então encontrado pela maximização da soma das variáveis (t) e (x):

$$WCET = \max \left(\sum t * x \right),$$

que resulta na estrutura do programa ou possíveis fluxos.

4 Representação do Fluxo do Programa

O fluxo do programa pode ser representado através de algumas características específicas como pode ser visto na Figura 3. Existe um código fonte do programa, uma representação do código através do gráfico do bloco básico e uma relação entre as possíveis execuções e as informações do fluxo.

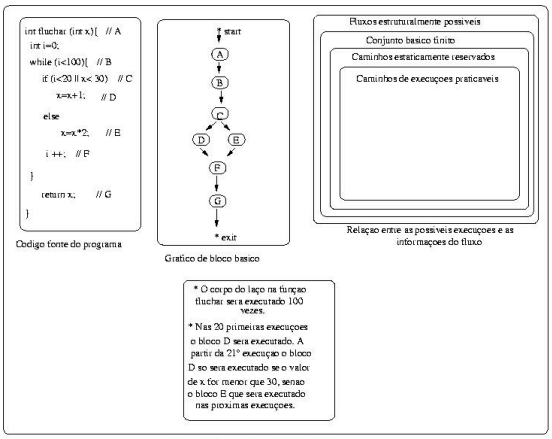

Figura 3: Representação do Fluxo do Programa

Analisando a conexão entre a representação do programa e a informação do fluxo, notamos que algumas informações de fluxo são importantes, enquanto outras não.

Para a representação de um dado programa, a quantidade das possíveis execuções são dadas pela estrutura do programa, na qual todas as execuções dos caminhos podem ser traçadas através dos comandos do código do programa, são consideradas possíveis ou praticáveis.

O conjunto das possíveis execuções são ditas finitas quando todos os laços no código do programa são limitados por algum limite superior no número de execuções. Quando adicionamos mais informações de fluxos, fica-se então permitido que o conjunto de execuções seja estreitado para um conjunto de caminhos estaticamente reservados. Para este tipo de conjunto, o cálculo irá extrair o caminho de execução com o pior caso.

Neste estágio, o *WCET* encontra um excesso de estimação. Uma boa representação da informação do fluxo pode permitir fazer o conjunto de caminhos estaticamente reservados seguro em relação ao conjunto dos atuais caminhos de execuções praticáveis. Logo no

estágio dos atuais caminhos de execuções praticáveis, o *WCET* encontrará um resultado desejado.

É válido lembrar que a análise da representação do fluxo pode ser estática, como mencionado acima, ou dinâmica. A diferença entre a representação estática e a dinâmica, é que na análise estática a informação não depende do estado atual de execução do programa enquanto que na análise dinâmica, a informação depende. E um bom exemplo disto pode ser vista na Figura 3 acima: a primeira instrução é dita estática e a última instrução é dita dinâmica.

5 A Área do Gráfico

A área do gráfico é marcada por um conjunto de nodos, ramificações, conjunto de sub-áreas, *start* e *exit*. Cada nodo na área do gráfico faz referência a um bloco básico, e um bloco básico pode ser referenciado por vários nodos diferentes.

Os nodos e as ramificações na área do gráfico são particionados dentro de áreas, correspondendo a diferentes ambientes encontrados no gráfico do bloco básico como: laços limitados, funções, chamadas recursivas, parte de códigos, etc.

As ramificações são os possíveis caminhos de passagem de um bloco básico para outro.

As áreas são organizadas dentro de uma hierarquia de área, na qual existe uma área cabeça que contém as instruções e execuções de seus blocos básicos, e existem as áreas filhas que também contém as instruções e execuções de seus blocos básicos. É válido lembrar que uma área filha pode conter outras áreas filhas também, e que um nodo somente pode ser iniciado e finalizado na área principal.

Enfim, existe uma área principal, também chamada de área mãe, e as áreas secundárias, também chamadas de descendentes ou filhas. Sendo assim, podemos exemplificar os nodos e as ramificações através da Figura 3 acima, na qual os nodos seriam os blocos:

start, A, B, C, D, E, F, G e *exit*,

e as ramificações seriam as passagens de

start→A,
A→B,
B→C,
C→D,
C→E,
D→F,
E→F,
F→G,
F→B,
B→G e
G→*exit*.

Pode-se exemplificar a área do gráfico e a hierarquia pelas Figuras 4 e 5 abaixo respectivamente.

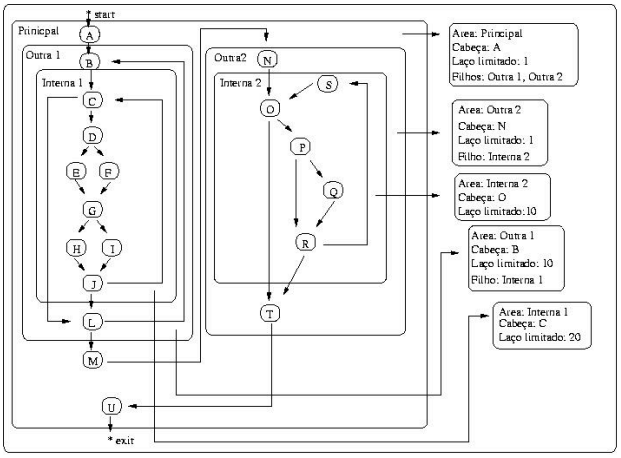

Figura 4: A Área do Gráfico

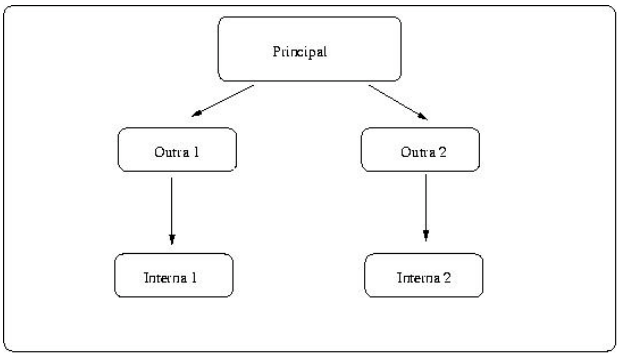

Figura 5: Hierarquia da Área

6 Análise de Baixo Nível

A análise de baixo nível é dada no código do objeto de um programa para obter o comportamento atual de sincronismo tendo-se as execuções que estão sendo executadas no hardware alvo. Para se obter este comportamento, o hardware alvo tem que ser conhecido.

Como se sabe, a análise de baixo nível é dividida em duas sub-fases: a análise global de baixo nível, na qual usa-se os efeitos de sincronismo da máquina que são dependentes de fatores que necessitam ser modelados sobre todo o programa, e a análise local de baixo nível, na qual usam-se os efeitos de sincronismo da máquina que são dependentes de fatores que podem ser manipulados localmente para cada instrução de um programa. Alguns exemplos da análise global de baixo nível são: instruções de *cache*, dados de *cache*, entre outras. E alguns dos exemplos da análise local são: acesso rápido à memória, sobreposição de *pipeline* [3], entre outras.

A sub-fase da análise global de baixo nível considera os efeitos das características de performance da máquina que são analisadas sobre todo o programa para poder dar um resultado seguro. Para determinar como tais características irão afetar a execução de uma instrução, não podemos considerar poucas instruções vizinhas.

Nesta sub-fase determina-se somente como as características investigadas afetam a execução de instruções, mas não geram o tempo de execução atual. O resultado da informação pode ser uma estimativa segura sobre como a característica investigada irá afetar a execução das instruções. È importante fixar que, as características para análise nesta sub-fase são dependentes do hardware alvo, e conseqüentemente, a informação coletada será diferente para diferentes CPU's. Logo, a complexidade nesta parte da análise depende do tipo de características usadas no alvo, e como elas interagem. Esta análise pode às vezes beneficiar a análise da informação do fluxo para produzir melhores estimativas. Por exemplo, se temos informações que um certo bloco básico nunca será executado, ele é seguro para assumir que essas instruções do bloco não serão carregadas, e assim não irão interferir em outras instruções da *cache*.

A informação gerada neta fase necessita comunicar com a análise local de baixo nível. Isto inclui informações seguras sobre a análise da *cache* e seus acertos e erros. É muito importante saber sobre algumas informações na análise da *cache*.

A *cache* é usada para acelerar o acesso de instruções na memória. Ela é muito menor e mais rápida que a memória principal. Quando o processador vai acessar o bloco da memória, ele primeiro checa se a *cache* contem o bloco. Se a resposta for sim, o acesso é um acerto na *cache* e o resultado será um acesso rápido. Se a resposta for não, o acesso é um erro na *cache* e o bloco é então copiado para a memória principal dentro da *cache*, onde é armazenado para usos futuros. Conseqüentemente, um erro nela leva um tempo maior de processamento.

Existem as instruções da *cache* que são usadas para fornecer um acesso rápido para as instruções executadas, os dados da *cache* que são usados para fornecer um acesso rápido para os dados manipulados pelas instruções. Quando suas instruções e os seus dados são armazenados na mesma *cache*, dá-se o nome união da *cache*. Quando vários níveis de *cache* são guardados entre o processador e a memória principal, forma-se uma hierarquia de *cache*. A memória pode então ser organizada em quatro etapas, vista na Figura 6.

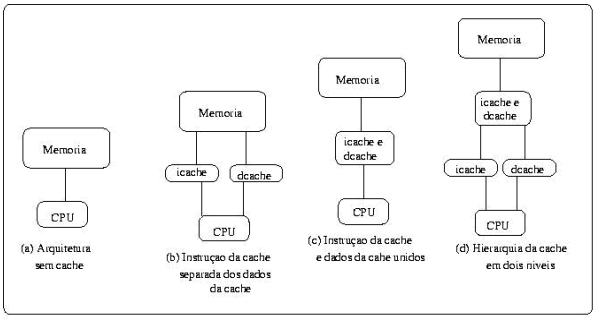

Figura 6: Possíveis Organizações da Memória

Uma *cache* consiste de várias localizações onde os blocos da memória podem residir. Se um bloco da memória pode residir somente em uma localização particular nela, ela é chamada de mapeada direta. Se um bloco de memória pode ser colocado em várias localizações da *cache*, ela é então chamada de conjunto associado.

Outros tipos de informações que são válidos lembrar são o comportamento das instruções da *cache* e o comportamento dos seus dados. O primeiro tipo de comportamento é bastante fácil para se analisar, uma vez que a busca do comportamento da instrução pode ser determinada no fluxo do programa. Mas o segundo tipo de comportamento é difícil de se determinar, pois o acesso de dados padrão não é fixado, mas depende do comportamento de tempo de execução do programa.

Logo para expressar os resultados da análise global temos que fazer uma categorização de modos de execução das referências da *cache* para as instruções como: sempre acerto (*hit*), sempre erro (*miss*), persistente e indefinida. Os significados dessas categorizações são: *hit* – se a referência da memória irá sempre resultar em um acerto da *cache*; *miss* – se a referência da memória irá sempre resultar em um erro da *cache*; persistente – se a primeira referência da memória não pode ser classificada nem como um acerto ou erro da *cache*, mas todas as execuções posteriores irão resultar em acerto da *cache*; indefinida – se a referência da memória não pode ser classificada por qualquer uma das categorizações acima. Um exemplo do modo de execução das referências da *cache* pode ser visto na figura 7 abaixo.

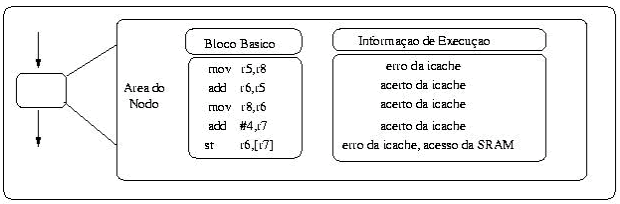

Figura 7: Área do Nodo com Bloco Básico e Execução

A sub-fase da análise local de baixo nível manipula os efeitos da máquina que dependem de uma simples instrução e das instruções vizinhas. Esta fase determina os efeitos de *pipeline* e gera os tempos de execução para partes do programa. Um dos exemplos desta fase é a análise de *pipeline*, na qual se faz o uso do modelo do hardware para extrair os tempos das seqüências das instruções anotadas. A análise trata o modelo do hardware como um bloco básico. É importante dizer que um modelo de hardware é um modelo do processador atual onde o programa é executado. Para ser capaz de se criar um modelo de hardware, as informações anotadas sobre o comportamento de sincronismo do processador tem que ser conhecidas.

O modelo do hardware retorna o tempo de execução de um nodo ou de uma seqüência de nodos na área do gráfico. E assim, o resultado é usado para

construir um modelo de sincronismo. Este resultado reserva a estimativa do *WCET* de um programa, para o cálculo da menor parte do sincronismo do programa. O modelo representa os tempos executados por uma função T, usando os tempos de nodos, denotados por t_{nodo}, e os efeitos de sincronismo para seqüências de nodos, denotados por δ_{seq}. A Figura 8 exemplifica um modelo de sincronismo.

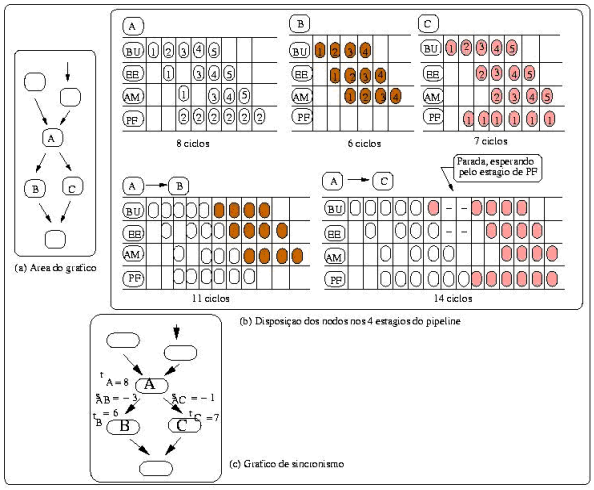

Figura 8: Modelo de Sincronismo com os Estágios de *Pipeline*

A Figura 8 (a) acima mostra uma área de gráfico com três nodos A, B e C, cada um referenciando um bloco básico. A Figura 8 (b) mostra a disposição dos nodos de *pipeline*, quando descritos nos quatro estágios da Figura 10 abaixo. O t_{nodo} representa o tempo de execução de um nodo isolado, e o δ_{seq} representa a mudança no tempo de execução que ocorre devido ao efeito do *pipeline* quando os nodos são executados em seqüência. O nodo δ_{seq} pode ser negativo para indicar que a seqüência de nodos é rápida, e o δ_{seq} pode ser positivo para indicar que a seqüência de nodos é mais lenta. Os nodos A e C tem instruções usando o estágio de PF, enquanto o nodo B não tem. O tempo de execução para uma seqüência de instrução, pode ser dada quando a primeira instrução inicia o *pipeline* até quando a última instrução finaliza o *pipeline*. Usando esta definição, a sobreposição do *pipeline* entre as instruções dentro do mesmo bloco básico é capturado, e os tempos de execuções para os nodos são 8 ciclos para A, 6 ciclos para B e 7 ciclos para C. Quando executamos o bloco básico em seqüência, o total do tempo de execução para a seqüência é menor que a soma dos tempos para os blocos básicos individualmente. Por exemplo, o tempo para a seqüência de A→B é 11 ciclos, enquanto que a soma dos tempos de execuções para A + B é igual a 8 + 6 = 14 ciclos. O tempo de execução para a seqüência de A→C é igual a 14 ciclos, enquanto que a soma separada de A + C é igual a 8 + 7 = 15 ciclos.

Um meio para achar o tempo de execução para os efeitos de sincronismo entre os bloco básicos (δ_{seq}) é subtraindo o tempo da seqüência menos a soma dos tempos de execução para os blocos separados. Por exemplo, na Figura 8 (c), o tempo do efeito de sincronismo da seqüência δ_{seq} A→B é (11 – 14 = -3 ciclos), e o tempo de sincronismo da sequência δ_{seq} A→C é (14 – 15 = -1 ciclo). O resultado negativo indica que a sequência de nodos é rápida. Dado o valor dos nodos separados e os efeitos de sincronismo das seqüências, podemos achar o valor do tempo de execução da seqüência dos nodos pela fórmula:

$$T(seq) = \sum t_{nodo} + \sum \delta_{seq}.$$

Logo, o tempo de execução para a seqüência A→B é igual a (8 + 6 – 3 = 11 ciclos) e o tempo de execução para a seqüência A→C é igual a (8 + 7 -1 = 14 ciclos). É importante fixar que o tempo para uma seqüência vazia é zero.

Para uma melhor visualização sobre a área do gráfico e o modelo de sincronismo, veja a Figura 9 (a) e (b) respectivamente abaixo.

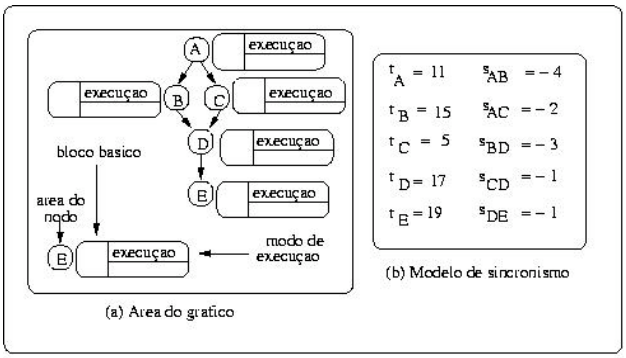

Figura 9: Modelo de Sincronismo

Para sabermos o valor do efeito de sincronismo das ramificações da Figura 9 (a) tais como: A→B, A→C, B→D, C→D e D→E, devemos fazer a soma de cada bloco básico separado, mais o valor da respectiva seqüência (δ_{seq}). Por exemplo: A →B = 26 – 4 = 22 ciclos, A→C = 16 – 2 = 14 ciclos, B→D = 32 – 3 = 29 ciclos, C→D = 22 – 1 = 21 ciclos e D→E = 36 – 1 = 35 ciclos..

Como se pode vê, a análise de *pipeline* não é muito complicada, entretanto, existem alguns problemas que são envolvidos nela. Como se sabe, a análise do *pipeline* modela os efeitos do tempo de execução da sobreposição entre as instruções nos processadores *pipeline*. A sobreposição entre as instruções faz o tempo de execução para uma seqüência de instruções (ou bloco básico) ser menor que a soma dos tempos de execuções das instruções individuais (ou bloco básico). Entretanto, nem todas as instruções necessitam usar todos os estágios de *pipeline*, e as instruções podem ser paradas, esperando por algum estágio do *pipeline* ou pelos dados de outras instruções. Para demonstrar os seus estágios, usaremos um *pipeline* escalar com unidades paralelas que contém quatro estágios. Esses estágios podem ser vistos na Figura 10.

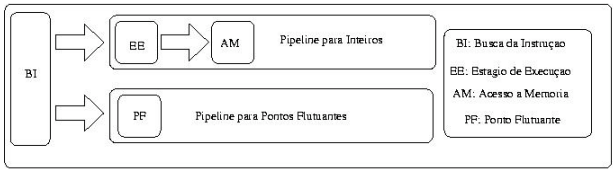

Figura 10: Unidades Paralelas de um Pipeline

7 Cálculo Baseado no Caminho

No cálculo baseado no caminho, os diferentes caminhos ao longo do programa são calculados, e usados para fornecer o caminho total com o maior tempo de execução. Uma vez que o maior caminho de execução, resulta no *WCET*.

A característica deste caminho é que os possíveis tempos de execuções são explicitamente definidos ou determinados. Existem duas entradas que são: a área do gráfico com os possíveis fluxos e o modelo de sincronismo. A entrada dos possíveis fluxos é representada na área do gráfico, mas este método pode somente manipular um subconjunto de fluxos (blocos básicos). Podem existir várias áreas, mas cada área na área do gráfico é estruturalmente restrita para conter somente um nodo cabeça, desse modo o cálculo fica limitado para cada laço de cada área. A entrada do modelo de sincronismo, é o resultado da análise de baixo nível (local e global), tendo o modelo de sincronismo representado como um gráfico global de sincronismo.

Um exemplo de como o cálculo baseado no caminho funciona, pode ser visto na Figura 11 (b) abaixo, uma vez que o cálculo é baseado no controle do fluxo do gráfico como mostrado na Figura 11 (a).

Primeiro o maior caminho no laço é encontrado e anotado. Depois o tempo do nodo cabeça também é anotado. Em seguida, calcula-se a estimativa do *WCET* através da multiplicação do tempo do maior caminho encontrado com o número de interação do laço menos uma unidade de interação. Depois deste resultado, soma-se ele com o valor de execução do nodo cabeça. No exemplo da Figura 11, definiu-se que a interação do laço seria de 100 execuções. Logo, através desta fórmula podemos obter o cálculo do *WCET* que é: 31 * (100 -1) + 3 = 3072 ciclos.

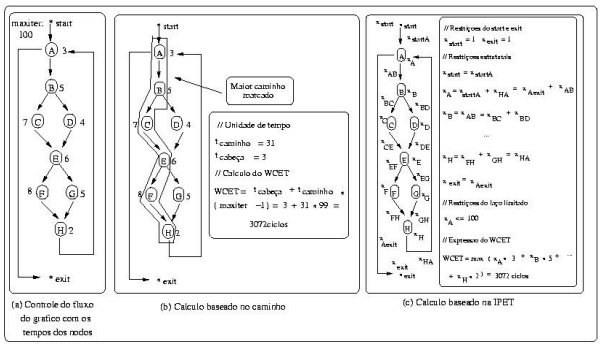

Figura 11: Cálculo Baseado no Caminho

8 Cálculo Baseado na IPET

No cálculo baseado na *IPET*, a cada bloco básico do programa e/ou ramificação, é dado uma variável de tempo (t), que denota no tempo de execução de cada bloco básico e/ou ramificação, e uma variável de contagem (x), que denota no número de tempos que o bloco básico e/ou ramificação é executado.

As variáveis de contagem são consideradas globais para o programa (partes), e seus valores refletem no número total de execuções dos nodos do programa. A característica definida é que os possíveis tempos de execuções são implicitamente definidos ou determinados. Por exemplo, o nodo E na Figura 12 (b), tem uma variável de tempo $t_E = 6$ e uma variável de contagem x_E. A variável x_E tem o número de tempos que o nodo E é executado sobre todo o programa.

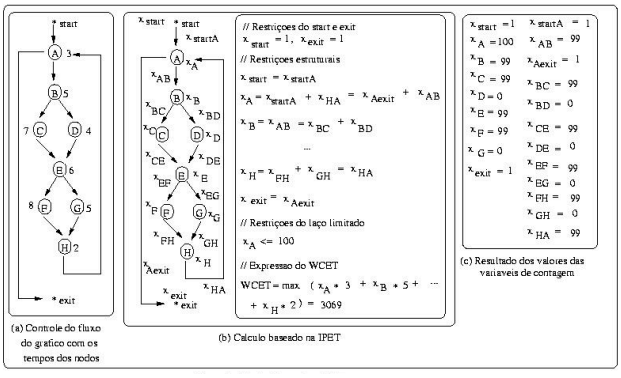

Figura 12: Cálculo Baseado na IPET

É válido acrescentar que para cada nodo t, o valor de sua(s) entrada(s) mais o valor de sua(s) saída(s), tem que ser igual ao valor do respectivo nodo. Por exemplo, o nodo A na Figura 12 (c) é igual a 100. Logo a soma de sua entrada ($x_{startA} = 1$) mais a sua saída ($x_{AB} = 99$) tem que ser igual ao valor do nodo A ($x_A = 100$).

Os possíveis fluxos através da estrutura do programa são modelados usando a estrutura de restrições. As estruturas de restrições são necessárias para garantir o limite (finito) do programa a conclusão do cálculo. Logo, com estas restrições, para um nodo ser executado, o número de vezes de execução deve ser começado em um nodo *start* e finalizado no nodo *exit*. Estes tendo uma variável de contagem (x) contendo os mesmos valores.

Na Figura 12 é mostrado esses valores do *start* e *exit*, que são iguais a 1. Para restringir os possíveis fluxos através do programa, restrições adicionais podem ser dadas na variável de contagem (x).

O *WCET* pode então ser encontrado através da maximização da soma das variáveis:

$$WCET = \max \left(\sum t * x \right) \quad (1),$$

resultando na estrutura do programa ou possíveis fluxos. Para cada nodo com suas variáveis (t) e (x), é atribuído um novo somatório para tais variáveis. Com isso, para cada nodo t_{nodo}, é atribuído o produto de ($t_n * x_n$) onde x_n é a variável de contagem do nodo, e para cada seqüência δ_{seq}, é atribuído o produto de ($\delta_s * x_s$) onde x_s é a variável de contagem do número de vezes que a seqüência é executada.

Dessa forma, o *WCET* é então encontrado pela maximização dos seguintes somatórios:

$$WCET = \max \left(\sum t_n * x_n + \sum \delta_s * x_s \right)$$

9 Conclusão

Pode-se dizer então que, as estimativas do *WCET* são requeridas para fornecer a garantia de sincronismo dos programas usados em sistemas de tempo real, uma vez que, seu *deadline*, ou seja, seu pior caso é muito importante.

A análise estática de baixo nível determina o pior caso dos programas, utilizando o comportamento atual de sincronismo do hardware e os modelos de fluxos de programas.

Os tipos de cálculos mostrados são utilizados para fornecer um resultado satisfatório e ao mesmo tempo seguro, para as aplicações que envolvem um medida de tempo precisa e segura. Tais como as aplicações de *STR hard*, em que, o comportamento temporal dos programas é de extrema importância.

Referências

[1] COLIN, A, and PUAUT, I. *Worst Case Execution Time Analysis for a Processor with Branch Prediction*, Real-Time Systems, vol. 18, no. 2-3, pages 249-274, May 2000.

[2] ENGBLOM, J. *Worst Case Execution Time Analysis*. Disponível em: http://user.it.uu.se/~jakob/presentations/kth-wcet-feb2002.pdf .

[3] ENGBLOM, J. *Processor Pipelines and Static Worst Case Execution Time Analysis*. Thesis (Doctor of Philosophy in Computer Systems) – Department of Information Technology. Uppsala: Uppsala University, 2002.

[4] ENGBLOM, J, ERMEDAHL, A, SJODIN, M, GUSTAFSSON, J, and HANSSON, H. *Worst Case Execution Time Analysis for Embedded Real Time Systems*. Journal of Software and Transfer Technology (STTT), vol 4, p. 437-455, no. 4, 2003. Disponível em: http://www.csd.uu.se/~jakob/publications/astec-wcet-sttt.ps.gz .

[5] ENGBLOM, J, ERMEDAHL, A, and STARPPET, F. *A Worst Case Execution Time Tool Prototype for Embedded Real Time Systems*. In: Workshop on Real-Time Tools (RT-TOOLS 2001), 20, August, 2001, Aalborg Denmark.

[6] ERMEDAHL, A. *A Modular Tool Architecture for Worst Case Execution Time Analysis*. Thesis (Doctor of Philosophy in Computer Systems) - Department of Information Technology. Uppsala: Uppsala University, 2003.

[7] ERMEDAHL, A, and ENGBLOM, J. *Modeling Complex Flows for Worst-Case Execution Time Analysis*. In: 21st IEEE Real-Time Systems Symposium (RTSS 2000), December, 2000, Orlando, Florida, USA. December 2000.

[8] FARINES, Jean Marie ; FRAGA, Joni da Silva ; OLIVEIRA, R. S. de . Sistemas de Tempo Real. 1. ed. São Paulo-SP: Escola de Computação 2000, IME-USP, 2000. v. 1. 200 p.

[9] FAUSTER, J, KIRNER, R, and PUSCHENER, P. *Intelligent Editor for Writing Worst Case Execution Time Oriented Programs*. Lecture Notes in Computer Science, vol 2855, p. 190-205, 2003. Disponível em: http://www.vmars.tuwien.ac.at/php/pserver/extern/docdetail.php?DID=1245&viewmode=published&year=2003 .

[10] BERNAT, G, COLIN, A, and PETTERS, S. *WCET Analysis of Probabilistic Hard Real-Time Systems*,

In Proceedings of the 23rd Real-Time Systems Symposium (RTSS'02), December 2002.

[11] WHITE, R, MUELLER, F, HEALY, C, WHALLEY, D, and HARMON, M. *Timing Analysis for Data Caches and Set-Associative Caches*, In Proceedings of the 3rd Real-Time Technology and Applications Symposium (RTAS '97), June 1997.

[12] SANDELL, D. *Evaluating Static Worst Case Execution Time Analysis for a Commercial Real Time Operating System*. Thesis (Master Computer Science) – Department of Computer Science. Malardalen: Malardalen University, 2004.

[13] KIM, S, and HA, R. *Efficient Worst Case Timing Analysis of Data Caching*, In Proceedings of the 2nd Real-Time Technology and Applications Symposium, June 1996.

[14] PETTERS, S, BETTS, A, and BERNAT, G. *A New Timing Schema for WCET Analysis*, In Proceedings of 4th International Workshop on Worst Case Execution Time Analysis, June 2004.