Explorando a estratégia híbrida modular para determinar o Worst-Case Execution Time

Felipe D. A. Brito[1], Gustavo C. E. Santo[1], Ingrid F. V. Oliveira[1], Lucas G. B. Cunha[1], Paulo Maciel[1], João C. da C. Davison[2]

[1]Centro de Informática - Universidade Federal de Pernambuco (UFPE) - Pernambuco, Brasil
Programa de Especialização em Software Embarcado

[2]Embraer - Brasil

{fdab,gces,ifvo,lgbc}@cin.ufpe.br

Abstract. *Critical real-time systems have stringent temporal requirements, and missing deadlines is a severe concern. Therefore, understanding the Worst-Case Execution Time (WCET) of a task is essential for these systems. Acknowledging the issues in estimating WCET and the advantages and challenges of the approaches presented in the literature, this article describes a modular hybrid approach to obtain upper bounds on WCET. Measurement methods, static analysis, and concepts of abstract execution and probabilistic analysis are employed to explore hardware and software variabilities. The proposed framework examined 23 benchmarks, and 15 provided a WCET for a probability of exceeding* 10^{-9}.

Resumo. *Sistemas de tempo real críticos possuem requisitos temporais rigorosos. O não cumprimento dos prazos pode ter consequências graves. Portanto, conhecer o Pior Tempo de Execução (WCET) de uma tarefa é essencial. Conhecida a problemática na determinação do WCET e as vantagens e desafios das abordagens existentes na literatura, este artigo descreve uma abordagem híbrida modular para obtenção dos limites superiores do WCET. Métodos de medição, análise estática e conceitos de execução abstrata e análise probabilística são utilizados na exploração das variabilidades de hardware e software. Dos 23 benchmarks explorados pelo framework proposto, 15 entregaram o WCET para uma probabilidade de excedência de* 10^{-9}.

1. Introdução

Os sistemas de tempo real (STRs) são sistemas computacionais que atendem a requisitos temporais. Esses requisitos são expressos como prazos (*deadlines*) para execução de tarefas do sistema, onde a confiabilidade do sistema está diretamente relacionada com sua habilidade de executar as tarefas dentro dos prazos, mesmo no pior cenário de execução. Caso esses prazos não sejam atendidos, dependendo da criticalidade do sistema de tempo real, as consequências podem ser desde inexistentes até acarretar na perda de vidas.

Um passo fundamental para quantificar a confiabilidade de um STR é determinar os limites superiores para os tempos de execução das tarefas no pior caso (*Worst Case Execution Time - WCET*), ou seja, o tempo máximo que pode ser gasto pelo hardware alvo (*target hardware*) para executá-las [Arcaro et al. 2018]. Entretanto, determinar o WCET tem se tornado um desafio a medida que os sistemas modernos adotam arquiteturas de computadores mais complexas. Desta forma, diversas abordagens foram propostas na literatura para determinação dos limites de WCET.

A abordagem estática para determinar o WCET é conhecida por gerar informações confiáveis, resultantes da modelagem do sistema, baseando-se em quatro etapas principais: reconstrução do fluxo de controle do software, análise de valor, análise da microarquitetura e obtenção do pior caminho. No entanto, a medida em que os processadores tornam-se mais complexos (memória cache, *branch predictors*, *pipelines*), a modelagem completa é intratável devido à grande quantidade de estados. Como resultado, a modelagem imprecisa aumenta o pessimismo, tornando-se um fator dominante neste tipo de análise [Silva 2015]. Já a abordagem baseada em medição analisa os tempos de execução das tarefas produzidos em *runtime*, levando à redução de esforços quanto a análise e sendo potencialmente aplicável a arquiteturas complexas. Porém, neste caso, o valor obtido para o maior tempo de execução (*High Water Mark - HWM*), dificilmente será o WCET. Deste modo, é necessário estabelecer margens de segurança que levam em consideração os casos de tempo possivelmente não observados durante o processo de coleta das medições [Arcaro et al. 2018].

A análise de tempo probabilística baseada em medição (*Measurement-Based Probabilistic Timing Analysis - MBPTA*) visa estabelecer limites para os WCETs das tarefas através da análise de medições de tempo de execução usando abordagem estatística e, desta forma, estimar um limite superior de tempo de execução, no qual a probabilidade de excedência esteja abaixo de um limiar aceitável para o projeto ($10^{-9} \sim 10^{-15}$). A teoria dos valores extremos (*Extreme Value Theory - EVT*) é uma das abordagens mais utilizadas, consistindo na análise estatística capaz de estimar a probabilidade de acontecimento de eventos extremos incomuns, a partir da observação de desvios da modelagem do comportamento típico do fenômeno analisado [Arcaro et al. 2018].

Por fim, a análise híbrida combina elementos de análise estática e análise base-

ada em medição. O objetivo dessa análise é superar as desvantagens da análise estática e dos métodos dinâmicos. A utilização de técnicas de análise estática como determinação do grafo de fluxo de controle, em conjunto com a análise de valor (execução abstrata), permite explorar o problema da variabilidade de software e assim encontrar os valores de entrada que exercitam o caminho do pior caso. Além disso, com a execução de um conjunto de medições no hardware alvo, utilizando as entradas obtidas para o caminho crítico, a variabilidade causada pelo hardware do sistema pode ser contemplada, e desta forma, um pWCET, a probabilidade de excedência para um determinado valor de WCET, pode ser obtido a partir de um tratamento estatístico adequado [Davis and Cucu-Grosjean 2019].

Deste modo, este trabalho descreve a utilização de um método híbrido e modular para a obtenção de um limite superior para o WCET de programas escritos em linguagem C. O termo "Híbrido" deriva do fato de que métodos de análise dinâmica e estática foram concebidos para serem utilizados simultaneamente, aproveitando assim as vantagens que cada um oferece, ao mesmo tempo que as deficiências são compensadas de maneira análoga. O termo "Modular" expressa a característica de generalização da abordagem, que se apresenta como um conjunto de etapas coesas com saídas bem definidas e que podem ser substituídas conforme as necessidades de cada caso, novas ideias e novos trabalhos surgirem.

A sequência deste artigo está organizada da seguinte forma: na Seção 2 são abordados os trabalhos relacionados disponíveis na literatura. Na Seção 3 é apresentada a metodologia utilizada no desenvolvimento desse trabalho, sendo dividida em cinco principais etapas. Através da aplicação da metodologia, na Seção 4 são apresentados e discutidos os testes realizados e os resultados obtidos. Por fim, na Seção 5 são apresentadas as conclusões desse trabalho.

2. Trabalhos Relacionados

A literatura explora o problema do WCET em três principais vertentes: métodos de medição, métodos estáticos e uma técnica de otimização denominada IPET (*Implicit Path Enumeration Technique*). Cada estratégia apresenta suas vantagens e desafios. Além disso, diversas ferramentas foram desenvolvidas com o objetivo de abordar a problemática da obtenção do WCET, tanto no âmbito comercial quanto acadêmico. Um levantamento e análise das ferramentas disponíveis na literatura é realizado em [Wilhelm et al. 2008]. Esse artigo reúne os principais aspectos relacionados ao desafio da determinação do WCET, além de proporcionar uma visão abrangente das principais práticas da indústria nesse contexto.

Em [Ermedahl et al. 2011] é mencionado que a abordagem do IPET é a preferida na análise do WCET atualmente. Nessa técnica, a análise de fluxo de um programa é traduzida em um conjunto de restrições lineares envolvendo variáveis contadoras para partes

do programa, por exemplo, blocos básicos. Essas restrições, em conjunto com uma função objetivo, formam um problema de programação linear inteira (*Integer Linear Programming Problem - ILP*), cuja solução maximiza o valor do WCET. O autor ainda cita que o desafio dessa técnica decorre do fato de que, para estimativas mais próximas do WCET real, as restrições de fluxo precisam ser detalhadas. Entretanto, é complexo incorporar no IPET os efeitos da variabilidade no tempo de execução provocados pelo hardware, como memória cache e *pipeline*. Além disso, considerando que o ILP possui uma complexidade *NP-hard*, o problema pode tornar-se computacionalmente inviável.

Além dos principais métodos, apresentados acima, que exploram o problema da determinação do pior tempo de execução, um método alternativo é encontrado na literatura: a utilização de uma abordagem híbrida [Wilhelm et al. 2008] [Davis and Cucu-Grosjean 2019]. Essa abordagem procura, através da junção de duas ou mais técnicas, potencializar as suas vantagens e reduzir os seus aspectos problemáticos, resultando assim em uma estratégia híbrida, que devido a sua versatilidade foi a opção escolhida e desenvolvida neste trabalho.

A determinação explícita dos caminhos é uma etapa geralmente presente em métodos de análise estática, mas que pode ser vantajosamente utilizada na estratégia modular híbrida. Os desafios associados a essa etapa estão relacionados com o crescimento exponencial de possíveis caminhos quando estruturas, como loops e condicionais, são utilizadas de maneira aninhada [Wilhelm et al. 2008]. Embora o problema de enumerar todos os possíveis caminhos seja desafiador [Yen et al. 1989], a determinação de um subconjunto de caminhos que atenda a determinados critérios pode ser uma estratégia viável.

Considerando a possibilidade de representação de um software a partir da modelagem de um grafo (por exemplo, *Control Flow Graph - CFG*), de acordo com [de Rezende 2014] a busca pelo caminho mais longo em grafos acíclicos e direcionados possui uma complexidade conhecida e uma abordagem matemática formalizada, tratando-se de um problema algorítmico amplamente explorado na literatura. Em [Yen et al. 1989] é proposta a busca dos k maiores caminhos em um grafo direcionado acíclico, sendo que o critério de tamanho é determinado pelo tempo total necessário para percorrer cada caminho.

No geral, o fluxo de controle em software é naturalmente direcionado devido a sua característica de execução sequencial de instruções, o que em parte valida a modelagem por meio de grafos. No entanto, a segunda restrição, referente a aciclicidade do grafo, limita a modelagem do software, visto que estruturas como loops precisariam ser desdobradas em segmentos sequenciais.

Em [Altenbernd 1996], o autor discute que simplesmente encontrar o maior ca-

minho estrutural pode levar à superestimação do WCET caso esse caminho não seja realizável em uma execução real. Ademais, o autor propõe a ideia de analisar as instruções do código, como cabeçalhos de condicionais, na busca pelo maior caminho realizável através da utilização da **"execução simbólica"**.

A execução simbólica consiste na exploração de todos os caminhos de execução possíveis/viáveis de um programa. A cada condicional verificada no programa, novos caminhos (estados) de execução são criados com restrições de viabilidade. Essas restrições são analisadas e, caso não sejam atendidas, a execução do caminho é interrompida [Martins 2018]. Por fim, [Ermedahl et al. 2011] propõe e implementa, em uma ferramenta acadêmica chamada SWEET, o que denominou de **Execução Abstrata**, uma forma de execução simbólica baseada no *framework* de interpretação abstrata e que é utilizada para obtenção de *flow-facts* e exploração dos possíveis caminhos de execução do software.

3. Metodologia

Conforme visto na Seção 1, o objetivo deste trabalho é conceber um processo que, a partir de um código-fonte em linguagem C, atinja o limite superior do WCET. Considerando as questões levantadas na Seção 1 a respeito da problemática na determinação do WCET e as vantagens e desafios das abordagens existentes na literatura discutidas na Seção 2, optou-se por desenvolver esse processo utilizando uma abordagem híbrida adotando a estratégia de modularização das camadas.

A abordagem modular em camadas possui como características principais: a manutenibilidade e modificabilidade. A manutenibilidade define que eventuais problemas identificados estão restritos a um subconjunto de camadas. Já a modificabilidade prevê que a substituição de um subprocesso, definido em qualquer camada, pode ocorrer conforme as necessidades do momento, desde que a interfaces padronizadas entre as camadas sejam preservadas. Dessa forma, projetos futuros podem reutilizar a estrutura geral do processo e adaptá-la conforme seja necessário.

A metodologia desenvolvida pode ser dividida em cinco principais etapas que envolvem conceitos de análise estática para o estudo do fluxo do programa e de análise dinâmica com o uso de medições. As três primeiras etapas são aplicáveis para programas que apresentam fluxo múltiplo de execução dependente de entradas. Enquanto que no caso de programas de fluxo único de execução, a metodologia deve ser observada a partir da quarta etapa, conforme ilustrado no fluxograma da Figura 7 no Apêndice A. Essas etapas são elencadas abaixo e serão exploradas com mais detalhes ao decorrer desta seção.

- **Etapa 1:** Medição dos blocos básicos do código.
- **Etapa 2:** Execução abstrata para obtenção do pior caminho (*Worst-Case Execu-*

tion Path - WCEP).

- **Etapa 3:** Obtenção das entradas que provocam o WCEP.
- **Etapa 4:** Medição do código completo no Hardware.
- **Etapa 5:** Análise estatística (MBPTA).

Para realizar as etapas descritas acima, buscou-se o auxílio de ferramentas de software e hardware para montar o processo. Essa busca resultou no desdobramento de um conjunto de passos. Primeiramente, buscou-se a conversão do código em linguagem C para um arquivo ALF (linguagem de representação intermediária). Em seguida é realizada a criação do arquivo TDB, um arquivo *template* para o tempo de execução dos blocos básicos (BBs). Então, cria-se o arquivo ANN, um arquivo *template* para as anotações no código. Após a criação desses arquivos, realiza-se a análise de correspondência dos BBs do programa, ALF para C. A partir de então é realizada a medição do tempo de execução de cada bloco básico e realiza-se a execução abstrata utilizando a ferramenta SWEET.

Na ferramenta SWEET são inseridos os arquivos ALF, TDB e ANN para obtenção do WCEP, em termos dos BBs do código ALF, além de uma estimativa do WCET. A partir do conhecimento obtido em uma primeira execução do SWEET, realiza-se uma nova execução abstrata para inferir as entradas que estimulam o pior caminho. Através do uso das entradas resultantes do passo anterior, realiza-se a medição do tempo de execução do WCEP, de ponta a ponta, em hardware físico. E por fim, é realizado um tratamento estatístico dos dados de WCET obtidos. Esse conjunto de passos podem ser observados no fluxograma ilustrado na Figura 8 do Apêndice A.

3.1. Etapa 01 - Identificação e Medição dos blocos básicos

Para estimar o WCET, primeiramente é necessário definir o tempo de execução de cada um dos blocos básicos (BBs) do código em análise. Portanto, o primeiro passo é identificar os BBs, que são as partes/seções do código compostas apenas por instruções sequenciais sem desvios de fluxo (condicionais e chamadas de funções), e após a identificação dos BBs, realizar as medições de tempo de execução.

Como abordado na seção Seção 3, a execução abstrata é realizada através da utilização da ferramenta SWEET. A ferramenta SWEET utiliza como entrada um arquivo no formato ALF. O ALF é uma linguagem intermediária desenvolvida com o intuito de ser adotada em análises de fluxo para cálculos de WCET. A execução abstrata será abordada em mais detalhes na Subseção 3.2 e maiores informações sobre o ALF são apresentadas no Apêndice E.

Para esta primeira etapa, é necessário realizar a conversão do código em C para o formato ALF. Para isso, utiliza-se a ferramenta **AlfBackend** [Gustafsson 2019]. Além

```
/* ------------------------------------------------------- */
/* Annotation templates for imported data (frefs)          */
/* ------------------------------------------------------- */

/* Given annotations should provide a safe upper bounds on what
   value a data, or specific fields of a data, may hold.
   The annotations should be valid for any execution of the
   program (for all its possible input value combinations).
   Please run: sweet -h topic=annot  for annotation syntax   */

FUNC_ENTRY "binary_search" ASSIGN "%x" INT 1 20;
```

Figura 1. Exemplo ilustrativo do arquivo formato ANN.

de realizar essa conversão, a ferramenta permite a criação de um arquivo de mapeamento que realiza a correspondência entre os BBs no código ALF e o código C. Detalhes sobre a instalação das ferramentas são introduzidos no Apêndice F e detalhados no *Github* [Felipe D. A. Brito 2023a], onde é fornecido um tutorial para instalação das ferramentas, manuais, referências técnicas, comandos úteis e algumas dicas de uso, como o uso de *scripts* para Linux que automatizam a geração dos arquivos realizada nessa primeira etapa.

O arquivo ALF é então utilizado na ferramenta SWEET para a geração de dois arquivos *template*. Um dos arquivos gerados é o *template* para anotação de código, o arquivo ANN. Esse arquivo permite indicar quais valores, ou intervalos de valores, a execução abstrata deverá explorar para certas variáveis. A Figura 1 apresenta um exemplo de anotação indicando que, na entrada da função *binary_search*, a variável x poderá assumir os valores inteiros de 1 a 20.

Outro arquivo gerado é o *template* de tempos de execução dos BBs, o arquivo TDB. Esse arquivo organiza, lado a lado, os BBs no formato ALF com o tempo de execução de cada bloco, conforme ilustrado na Figura 2. Inicialmente esses tempos de execução estão zerados e serão preenchidos após o passo de medição. Essas medidas são obtidas, nessa metodologia, através da instrumentação do código.

A instrumentação consiste na inserção de códigos que visam capturar a quantidade de tempo, ou ciclos, que um determinado conjunto de instruções precisa para ser executado. No cenário estudado, cada conjunto de instruções instrumentado é um bloco básico. Esse processo pode ser realizado em diferentes níveis de abstração, desde cenários mais simplistas, porém menos precisos, como o código-fonte, até a instrumentação do próprio *assembly* que, apesar de mais complexo, possibilita uma maior precisão.

A instrumentação do código-fonte em C consiste em computar, através do auxílio de funções, o número de ciclos gastos entre uma instrução inicial, a qual demarca o começo de um BB equivalente no ALF, e uma instrução final que corresponde ao fechamento desse BB, através do arquivo de mapeamento de código ALF para código C.

```
main::bb 0
main::bb::1 0
binary_search::bb 0
binary_search::bb1 0
binary_search::bb2 0
binary_search::bb9 0
binary_search::bb14 0
binary_search::bb19 0
binary_search::bb21 0
binary_search::bb23 0
binary_search::bb24 0
binary_search::bb25 0
```

Figura 2. Exemplo ilustrativo do formato tdb.

Ou seja, para medir o custo de um BB no ALF utiliza-se a medição no código C em um trecho que corresponda ao BB em questão. Para isso, utiliza-se a biblioteca disponível em [Malich 2017], que implementa um contador de *ticks* para microcontroladores AVR. Essa biblioteca implementa as seguintes funções:

- *void ResetTickCounter(void)*;
- *void StartTickCounter(void)*;
- *void StopTickCounter(void)*;
- *ticks_t GetTicks(void)*;

A função *ResetTickCounter* é responsável por reiniciar o contador de *ticks*. A função *StartTickCounter* tem como objetivo iniciar a contagem de *ticks*. *StopTickCounter* pausa a contagem de *ticks* até que a mesma seja novamente iniciada. Por último, *GetTicks* retorna a quantidade de *ticks* contabilizados no período entre a chamada de *StartTick-Counter* e *StopTickCounter*.

Com o código instrumentado, é necessário realizar o *cross-compiling*, visando as configurações do microcontrolador AVR desejado. Para isso, utilizou-se o compilador avr-gcc, que possibilita a geração de código para AVR a partir de um código-fonte em linguagem C, especificando a geração para o microcontrolador Atmega328, o qual não possui memória cache e *pipeline*.

Após a compilação, é possível realizar a execução no microcontrolador. Para isso utiliza-se o programa **SimulAVR**, um simulador para microcontroladores da família AVR que possibilita a instanciação de um Atmega328 e a subsequente execução do código compilado. Por conseguinte, é possível obter o número de ciclos gastos para a execução do trecho de código em C que corresponde a um BB no código intermediário ALF. A relação de tempos medidos dos BBs é então reunida em um arquivo TDB para a realização da execução abstrata na ferramenta SWEET Subseção 3.2.

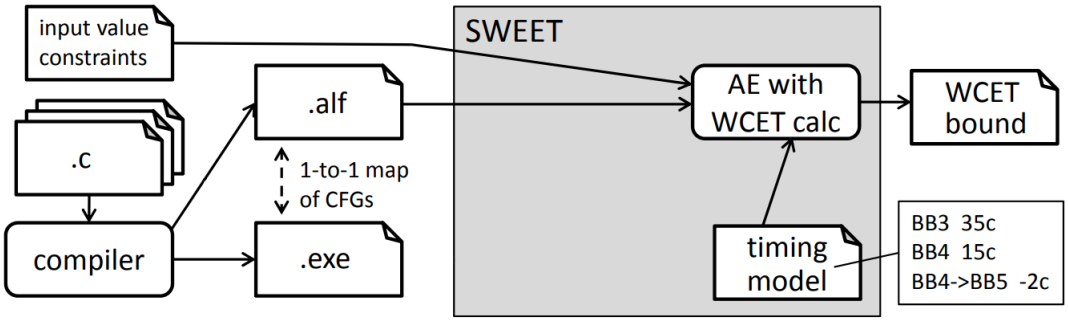

Figura 3. Fluxo de uso da Execução Abstrata na ferramenta SWEET.

3.2. Etapa 02 - Execução Abstrata e o SWEET

A execução abstrata é um método que permite o cálculo de restrições de fluxo em um código, a geração de fatos de fluxo (*flow facts*) e, ao mesmo tempo, uma análise de valor nas variáveis do programa. Ao adicionar o tempo como uma variável explícita e incrementá-la para cada bloco básico processado, o método da execução abstrata permite determinar os limites superiores e inferiores para *loops* e *loops aninhados*, os limites superiores para a execução de BBs, os pares de BBs inviáveis, os caminhos inviáveis, além de devolver o melhor e o pior tempo de execução, BCET e WCET, junto dos respectivos caminhos que os realizam, BCET e WCET [Ermedahl et al. 2011].

A ferramenta SWEET, *Swedish Execution Time Analysis Tool*, foi desenvolvida na *Mälardalen University* na Suécia para prover métodos de cálculo do WCET para software de tempo real crítico [Lisper 2014]. O software implementa estratégias de análise de fluxo (como a execução abstrata, foco dessa etapa) e análises de baixo nível para os processadores NECV850E e ARM9.

Com o código no formato ALF e os *templates* de anotações e custos de BBs preenchidos, a segunda etapa da metodologia propõe encontrar uma estimativa do caminho crítico (WCEP) e seu tempo (WCET) por meio da execução abstrata. A ferramenta consegue, ao executar o programa em um domínio abstrato, calcular possíveis valores para variáveis em diferentes pontos do programa de maneira contextual, associando-os a valores abstratos. Por exemplo, se a execução abstrata utilizar o domínio de intervalos, o conteúdo de uma variável é o intervalo com todos os valores que ela pode assumir em um ponto do programa. A coleção de todas as variáveis abstratas e seus possíveis valores correspondem a um estado abstrato.

A Figura 3 resume o fluxo do uso da ferramenta. Evidencia-se o uso dos arquivos TDB, ANN e ALF como entradas do programa, identificados, respectivamente, como *timing model*, *input value constraints* e **.alf**. A saída da execução abstrata, identificada como *WCET bound*, é o valor para o WCET e o pior caminho de execução.

3.3. Etapa 03 - Determinação das entradas

O último desafio, antes da medição no hardware, é a obtenção das entradas que provocam o WCEP obtido na etapa anterior. Dada a modularidade que é proposta neste trabalho, não foi determinado um único método que seja abrangente para qualquer tipo de fluxo de controle. Pelo contrário, derivou-se um conjunto de processos que podem ser utilizados, caso a caso, à medida em que se adéquem melhor ao problema.

Em geral, esses processos operam de forma iterativa, isto é, realizam a execução abstrata, capturam as informações produzidas e as utilizam em execuções subsequentes do método. O fato do método retornar os *flow-facts*, que indicam o número máximo de iterações de *loops* e caminhos/combinações de BBs inviáveis, e o próprio caminho crítico (WCEP) e seu tempo (WCET), pode já fornecer informação suficiente para o testador restringir a busca pelas entradas necessárias ou já determiná-las. Uma inspeção visual do código, seguindo o WCEP e auxiliado pelos *flow-facts*, pode fornecer indícios de quais são as entradas que geram o caminho ou pelo menos pode restringir os intervalos que as contém. Além disso, ao realizar novamente a execução abstrata, restringindo os intervalos das entradas, com as informações do WCEP, do WCET e um maior conhecimento das próprias entradas dado pela execução anterior, é possível descobrir se a restrição foi feita no intervalo correto.

Dessa forma, a determinação das entradas pode ser feita por um método iterativo no próprio SWEET, seguindo uma estratégia manual de busca binária. Como explicado anteriormente, a execução abstrata funciona sobre um intervalo de possibilidades para as entradas que é fornecido no arquivo de anotações. Após a primeira execução, é retornado o BCEP, o WCEP, o BCET e o WCET. Sabendo que o WCET é o pior tempo de execução para o conjunto de entradas escolhido, ao reduzir o intervalo de entradas pela metade no arquivo de anotação e realizar novamente a execução abstrata, necessariamente o novo valor do WCET será:

- **Menor que o WCET anterior**: Isso significa que as entradas que provocam o caminho crítico, com o maior WCET possível, estavam no intervalo das entradas rejeitado e não estão mais definidas no arquivo de anotações.
- **Igual ao WCET anterior**: Isso significa que as entradas que provocam o caminho crítico continuam no intervalo definido no arquivo de anotações.

O objetivo é que a cada redução do intervalo de entradas, no arquivo de anotações, e nova execução abstrata, o WCET permaneça o mesmo. A técnica permite então reduzir o intervalo de busca pela metade a cada execução, isolando as entradas que provocam o WCEP. O fluxograma representativo dessa busca pode ser consultado na Figura 9 no Apêndice A.

3.4. Etapa 04 - Medida no hardware

Com as entradas que levam ao WCEP definidas, o *benchmark* está pronto para ter seu tempo de execução medido no hardware. Para essa etapa, utilizou-se o *BeagleBone Black*, um kit de desenvolvimento baseado no processador AM3358X que conta com um ARM Cortex-A8 super escalar, o qual não suporta execuções fora de ordem. Esse último contendo uma memória cache de dois níveis, sendo a L1 de 32KB e a L2 de 256KB, e um *pipeline* de 13 estágios. Além disso, a cache L1 é do tipo *write-back* e opera com a política de substituição aleatória. A memória cache e sua política de substituição são descritas em maiores detalhes na Subseção E.3.

Para evitar o *overhead* proporcionado por sistemas operacionais, as medições foram conduzidas em um ambiente *bare-metal*. Tendo em vista tal objetivo, foi utilizada uma imagem do Uboot, um *bootloader* de primeiro e segundo estágio capaz de inicializar o hardware, carregar e executar o código da aplicação em um endereço desejado.

Com o ambiente de execução configurado, é necessário preparar o código a ser medido, isto é, instrumentá-lo a fim de obter o número de ciclos decorridos de seu fluxo de execução. A instrumentação também será responsável por desativar a cache L2 e realizar uma rotina de manutenção da cache L1 a cada fluxo completo. Essa rotina consiste na invalidação da cache de instruções e na ocupação da cache de dados, de forma que seu conteúdo não seja útil para a próxima execução do programa. Uma análise mais detalhada da cache pode ser encontrada no apêndice E.3.

3.5. Etapa 05 - Measurement-Based Probabilistic Timing Analysis - MBPTA

O MBPTA consiste em um método para determinação de um limite superior do pior tempo de execução. Esse método baseia-se em análises probabilísticas de medidas dos tempos de execução utilizando a Teoria do Valor Extremo (TVE), que é o ramo da estatística inicialmente projetado para estimar a probabilidade de eventos naturais incomuns/extremos.

A análise permite obter uma estimativa do WCET associado a uma baixa probabilidade de excedência aleatória, ou seja, o pWCET (*Probabilistic Worst-Case Execution Time*) [Arcaro et al. 2018]. Essa estimativa é derivada de uma amostra de observações de tempo de execução de um programa realizadas em um hardware alvo, ou em um simulador de hardware, de acordo com um protocolo de medição apropriado. O protocolo de medição executa o programa múltiplas vezes de acordo com alguma(s) sequência(s) de estados de entrada viáveis e estados iniciais de hardware, amostrando um ou mais cenários possíveis de operação [Davis and Cucu-Grosjean 2019]. Portanto, neste trabalho, o cenário explorado consiste na combinação de entradas e estado do hardware que representam o pior caso de execução do programa.

Após a coleta da amostra de observações de tempo de execução, é necessário

verificar se a aplicação da TVE atende a alguns requisitos. Esses requisitos tem como propósito verificar se o comportamento dos dados analisados é de fato compatível com o modelo estatístico empregado na análise. Primeiramente, o fenômeno analisado deve ser modelável por variáveis aleatórias. E por fim, a amostra de observações utilizada deve ser independente e identicamente distribuída (i.i.d.).

A verificação da aplicabilidade da TVE é feita a partir da realização de testes estatísticos apropriados. Para verificação da propriedade i.i.d. dos dados foram implementados os seguintes testes em um código *python* [Costa 2021]: o teste de *Anderson-Darling*, que testa se os valores da amostra pertencem a uma mesma distribuição, o teste *Kolmogorov-Smirnov*, que avalia se duas amostras são oriundas de uma mesma distribuição, o teste *Ljung-Box*, que testa a independência nas observações a partir da análise de autocorrelação de resíduos, e o teste *Wald-Wolfowitz runs*, que testa a aleatoriedade da amostra.

Após comprovação da aplicabilidade da TVE, seleciona-se os dados, a partir da amostra, por meio de dois métodos empregados na literatura sobre MPBTA. Esses métodos são o de máximo de blocos (*Block Maxima - BM*), que é baseado no teorema de Fisher-Tippett-Gnedenko [Davis and Cucu-Grosjean 2019], e o de picos acima do limiar (*Peaks-over-Threshold - PoT*), que é baseado no teorema de Pickands-Balkema-de Haan [Davis and Cucu-Grosjean 2019]. Esses teoremas, nos quais o TVE se baseia, mostram que a distribuição assintótica da cauda de uma amostra de variáveis aleatórias, independentes e identicamente distribuídas, converge para famílias de distribuições conhecidas como Valor Extremo Generalizado (*Generalized Extreme Value - GEV*) e Distribuição de Pareto Generalizada (*Generalized Pareto Distribution - GPD*), respectivamente para o uso dos métodos BM e PoT [Costa 2021].

O método BM divide a amostra em blocos de tamanho fixo e obtém o valor máximo para cada bloco. A partir dos dados selecionados, é obtida a função densidade de probabilidade (PDF), sendo ajustada pela curva GEV, como ilustrado na Figura 4.

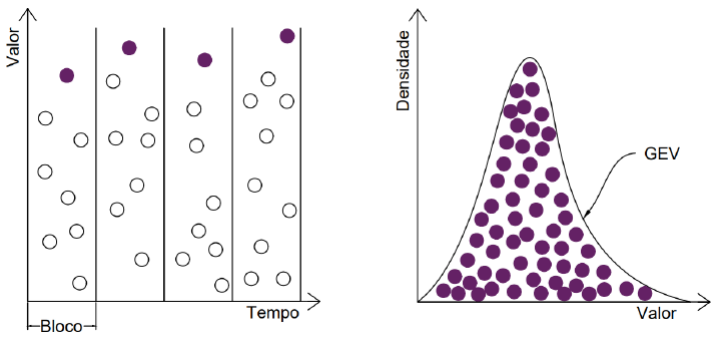

Figura 4. Exemplo de aplicação do método BM.

Fonte: [Costa 2021].

A GEV é uma família de distribuições que, de acordo com os valores dos seus parâmetros, converge para uma distribuição específica, podendo ser uma invertida de Weibull, Gumbel ou Fréchet, dependendo do parâmetro de forma ξ. A forma paramétrica da GEV é dada por [Costa 2021]:

$$G(x;\xi,\mu,\sigma) = \frac{1}{\sigma}\left[1+\xi\left(\frac{x-\mu}{\sigma}\right)\right]^{-1/\xi-1} e^{-\left[1+\xi\left(\frac{x-\mu}{\sigma}\right)\right]^{-1/\xi}} \tag{1}$$

$$G(x;\xi=0,\mu,\sigma) = \frac{1}{\sigma}\exp\left[-\exp\left(-\frac{x-\mu}{\sigma}\right)\right]\exp\left(-\frac{x-\mu}{\sigma}\right) \tag{2}$$

Os parâmetros ξ, σ e μ, apresentados nas equações acima, são conhecidos como forma, escala e local, respectivamente. De acordo com o valor de ξ, a distribuição resultante pode ser:

- $\xi = 0$: Gumbel (exponencial ou de cauda leve)
- $\xi > 0$: Fréchet (cauda pesada)
- $\xi < 0$: Weibull (cauda curta)

Já o método PoT utiliza um valor escolhido como limite para desempenhar o papel de filtro, em que apenas observações que excedam o valor limite são selecionadas. De maneira análoga, obtendo a PDF dos dados, a curva é ajustada por uma curva GPD, como ilustrado na Figura 5.

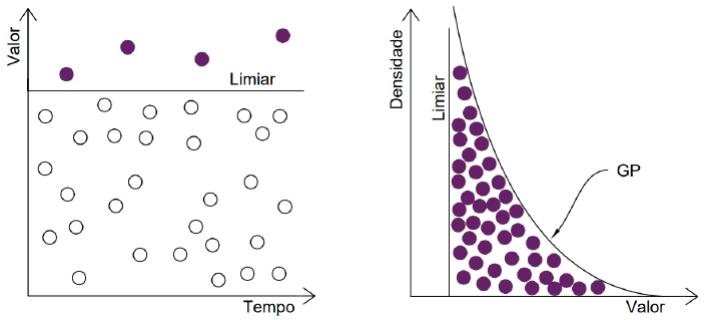

Figura 5. Exemplo de aplicação do método PoT.

Fonte: [Costa 2021].

A GPD é uma família de distribuições que incluem, Exponencial, Beta e Pareto, dependendo do parâmetro de forma ξ. A forma paramétrica da GPD é dada por [Costa 2021]:

$$H(x;\xi,\beta,\sigma) = \frac{1}{\sigma}\left(1+\frac{\xi(x-\beta)}{\sigma}\right)^{-1/\xi-1} \tag{3}$$

$$H(x;\xi=0,\beta,\sigma)=\frac{1}{\sigma}e^{-\frac{x-\beta}{\sigma}} \tag{4}$$

Analogamente a GEV, os parâmetros ξ, σ e β são conhecidos como forma, escala e local. De acordo com o valor de ξ, a distribuição resultante pode ser:

- $\xi = 0$: Exponencial
- $\xi > 0$: Pareto
- $\xi < 0$: Beta

Neste trabalho, optou-se por utilizar o método de Máximo de Blocos. Na literatura é possível encontrar discussões quanto à confiabilidade e acurácia desse método comparado ao método PoT. Alguns trabalhos afirmam que PoT é uma técnica mais complexa e limitado do que a BM [Machado 2021].

O valor de WCET é estimado a partir da função de confiabilidade (*Reliability Function*) dada por $R(t) = 1 - CDF$, onde CDF é a função de distribuição acumulada. Logo, o WCET é dado pela Equação 5, onde $pWCET$ é a probabilidade de excedência fornecida. De acordo com a literatura, são utilizados valores entre $10^{-9} \sim 10^{-15}$.

$$WCET = arg(R(t) = pWCET) \tag{5}$$

4. Testes e Resultados

Este trabalho possuía como meta submeter 33 *benchmarks* ao processo desenvolvido. Entretanto, apenas 13 desses *benchmarks* revelaram-se adequados para serem submetidos ao processo completo das cinco etapas da metodologia. Ao passo que, 17 foram considerados independentes das entradas, sendo submetidos ao processo metodológico a partir da etapa 04. Essas decisões são ilustradas pela Figura 8 e são descritas na Seção 3.

4.1. Análise Execução Abstrata

Conforme visto anteriormente, neste trabalho a execução abstrata é realizada com auxílio da ferramenta SWEET. Para isso, utilizou-se as seguintes máquinas:

- **Máquina 01:** 8 GB de RAM, processador Intel I5-6200U 2.30GHz, 1Tb de SSD;
- **Máquina 02:** 54 GB de RAM, processador Ryzen 5 5600G 3.90GHz, 2Tb de SSD M.2.

Verificou-se, em uma primeira rodada de testes, que durante a execução abstrata alguns *benchmarks*, que produziram mais estados e/ou fizeram uso de variáveis do tipo *float* ou *double*, demandaram um maior tempo de processamento e um maior uso de

memória RAM da máquina de testes. Essa diferença do custo da análise levou à necessidade de utilização de um ambiente de testes apropriado para o *benchmark* em estudo, isto é, que possuísse hardware suficiente para sua execução. Desta forma, dos 13 *benchmarks* submetidos à execução abstrata, 7 foram processados pela Máquina 01. Os 6 *benchmarks* restantes, por demandarem mais recursos, foram explorados na Máquina 02, sendo descritos a seguir.

O *benchmark* **33-Ud** foi explorado utilizando um intervalo de 0 a 50 para a entrada n. Após cerca de 40 minutos de análise, a execução do algoritmo foi finalizada com sucesso. Para o *benchmark* **15-Insertsort**, a primeira tentativa foi explorar o intervalo de valores de 0 a 10 para as 11 posições do array. Portanto, o SWEET deveria testar as 11^{11} possibilidades de entradas e lidar com os estados gerados. Após cerca de 2 horas de execução e 11219665 estados explorados, a Máquina 02 ocupou os 54GB de RAM disponíveis, assim encerrando o terminal junto do processo do SWEET, impossibilitando a conclusão do algoritmo.

A experiência inexitosa de execução do **15-Insertsort** remete ao discutido na Subseção D.2, que diz que quando a complexidade do programa é grande, o tratamento completo da execução abstrata se torna computacionalmente inviável. Então, nesse *benchmark* seguimos uma estratégia chamada de *merge*, que trata-se da junção de estados ao longo da execução com o intuito de reduzir a complexidade da execução e o uso de recursos da máquina às custas da precisão nos resultados. O *benchmark* utiliza *loops while* aninhados, por isso escolheu-se a estratégia de *merge* em pontos de saída de *loops*, utilizando o parâmetro $merge = le$ do SWEET.

O *benchmark* **28-qurt** possui declarações de variáveis do tipo *float* ou *double*. Consequentemente, assim como ocorreu no algoritmo anterior, a análise sem estratégia de *merge* não produziu resultados. No geral, após cerca de 1 hora e meia de execução na máquina 02, a memória RAM foi completamente consumida e o processo do SWEET encerrado sem a conclusão do algoritmo. Portando, aplicou-se a estratégia de *merge*. Esse *benchmark* realiza algumas chamadas de função em meio aos cálculos, funções essas que calculam o absoluto e a raiz quadrada de um número. Portanto, escolheu-se a estratégia de *merge* em pontos de saída de retorno de funções, através da utilização do parâmetro $merge = fr$ do SWEET. Já os *benchmarks* **30-sqrt**, **20-ludcmp** e **29-select**, mesmo envolvendo o uso de variáveis do tipo *float*, obtiveram resultados sem o uso de merge. O primeiro finalizou a execução após 1 hora, o segundo em cerca de 15 minutos e o terceiro em menos de 1 segundo.

A conclusão da execução abstrata em cada um dos *benchmarks* foi condição para a metodologia de busca das entradas responsáveis pelo WCEP, apresentada na Subseção 3.3. Os resultados da análise dos 13 *benchmarks* submetidos à execução abstrata podem ser observados na Tabela 1.

Programa	Máquina AE	Nº de comb. de entradas	Entradas	Estratégia de Merge	Tempo AE
bs	Máquina 01	20	a = 0	-	<1 s
cover	Máquina 01	1000	cnt = 0	-	<5 s
expint	Máquina 01	2500	n = 50, x = 1	-	<10 s
insertsort	Máquina 02	1x11^11	a[0] = 0, a[1] = 9, a[2] = 8, a[3] = 7, a[4] = 6, a[5] = 5, a[6] = 4, a[7] = 3, a[8] = 2, a[9] = 1, a[10] = 0	LE	<5 s
janne_complex	Máquina 01	900	a = 0, b = 5	-	<20 s
lcdnum	Máquina 01	16	a=0	-	<1 s
ludcmp	Máquina 02	10	n = 49	-	∼15 m
ns	Máquina 01	500	x = 5	-	<2 s
prime	Máquina 01	1x10^6	x = 997, y = 997	-	<5 s
qurt	Máquina 02	1000	a = 7, b=9, c=[2..3]	FR	<5 s
select	Máquina 02	20	k = 5	-	<1 s
sqrt	Máquina 02	10	valor = 1	-	∼1 h
ud	Máquina 02	50	n = 49	-	∼40 m

Tabela 1. Resultados das análises dos Benchmarks após AE.

4.2. Análise de Medição no Hardware

Com a conclusão da execução abstrata e as entradas, que excitam o pior caminho de cada *benchmark*, avaliadas, os códigos são submetidos ao processo de instrumentação para então serem executados no hardware. Este processo consiste na mudança das entradas com base no resultado da execução abstrata, a adesão do código de ativação do contador de ciclos, a rotina de manutenção da cache e a contabilização do número de ciclos, inicial e final, entre o fluxo avaliado.

O fluxo analisado é então colocado em um *loop* para ser executado 50000 vezes. Cada iteração do *loop* consistindo no armazenamento do número de ciclos inicial, seguido da realização do fluxo avaliado e o armazenamento do número de ciclos final. A diferença entre o número de ciclos final e inicial é então calculada e, após a remoção do *overhead* causado pela chamada da função de leitura do número de ciclos, armazenada em um endereço livre da memória. Antes que o código passe para a iteração seguinte, a função de manutenção da cache é executada, visando invalidar a cache de instruções e desfavorecer a cache de dados. Um exemplo do código instrumentado pode ser encontrado no repositório do projeto [Felipe D. A. Brito 2023a], assim como uma explicação mais técnica em relação às funções de instrumentação.

A Tabela 2 apresenta a relação de medições bem-sucedidas de cada *benchmark*. É possível notar que nem todos os *benchmarks* foram medidos com sucesso. Isso ocorre, na maioria das vezes, devido à arquitetura ou o funcionamento do algoritmo, como no

caso do *benchmark* **fir**, o qual possui a inicialização de suas estruturas de dados fora da função *main*, fazendo com que não seja possível reinicializá-las a cada iteração sem que haja uma grande mudança no código. Tal comportamento tem um efeito negativo nesse processo de medição, uma vez que o *benchmark* em questão executa inúmeras operações e salva os resultados em suas matrizes, que por sua vez não são reinicializadas no fluxo principal, o que faz com que o resultado da iteração anterior seja levado como entrada para a próxima iteração. Consequentemente, há um aumento de número de ciclos a cada iteração, visto que o tamanho das operações tende a aumentar. Além deste cenário, alguns *benchmarks*, por motivos não explorados, não foram capazes de completar todas as 50000 iterações e reprovaram na etapa de medição.

Benchmarks	**Medição bem-sucedida**	**Benchmarks**	**Medição bem-sucedida**
bs	sim	jfdctint	sim
bsort100	sim	lcdnum	não
cnt	sim	lms	não
compress	não	ludcmp	não
cover	não	matmult	sim
crc	sim	minver	não
duff	sim	ndes	não
edn	não	ns	não
expint	não	nsichneu	sim
fdct	sim	prime	sim
fft1	não	qurt	sim
fibcall	não	select	sim
fir	não	sqrt	não
insertsort	sim	statemate	sim
janne_complex	sim	ud	sim

Tabela 2. Resultados da Medição dos *Benchmarks*

4.3. Análise MBPTA

A primeira etapa na análise MBPTA é a verificação da aplicabilidade dos dados coletados ao ajuste pelo modelo TVE. A avaliação dos teste estatísticos foi feita pela comparação do **pvalue** com o nível de significância α ($0.01 \leq \alpha \leq 0.05$), onde tem-se que:

- $pvalue \geq \alpha$: Hipótese-nula do teste não pode ser rejeitada;
- $pvalue < \alpha$: Hipótese-nula do teste pode ser rejeitada;

Os testes foram implementados em um código *python*, retornando os valores de **pvalue**, para os testes executados, para a amostra de dados fornecida ao programa. Neste

trabalho, considera-se um nível de significância $\alpha = 0.05$, o tamanho de bloco igual a 200 e um $pWCET = 10^{-9}$. Os resultados dos testes, aplicados aos dados coletados dos *benchmarks*, estão apresentados na Tabela 3.

Benchmarks	KS	AD	WW	LB	Benchmarks	KS	AD	WW	LB
bs	0,187	0,059	0,373	0,950	select	0,585	0,250	0,063	1,000
bsort100	0,001	0,001	0,022	0,980	ud	0,001	0,001	0,013	0,964
cnt	0,464	0,250	0,063	0,980	janne_complex	0,720	0,250	0,000	0,997
crc	0,105	0,094	0,000	0,558	prime	0,000	0,001	0,000	0,980
duff	0,869	0,250	0,000	0,132	statemate	0,000	0,001	0,087	0,999
fdct	0,622	0,250	0,061	0,998	nsichneu	0,017	0,001	0,000	0,003
jfdctint	0,886	0,250	0,109	0,984	insertsort	0,163	0,097	0,695	0,960
matmult	0,392	0,250	0,267	0,398	qurt	0,526	0,250	0,000	0,201
fir	0,000	0,000	0,000	0,988	-	-	-	-	-

Tabela 3. Resultados dos testes estatísticos para verificação de aplicabilidade da TVE.

De acordo com os resultados observados na Tabela 3, os testes foram atendidos apenas para os *benchmarks* **bs**, **cnt**, **fdct**, **jfdctint**, **matmult**, **select** e **insertsort** ($pvalue \geq \alpha$). Em seguida, empregou-se a análise da TVE nas medições. Ressalta-se que para os arquivos de dados que não atenderam aos requisitos verificados pelos testes estatísticos, não é possível garantir confiabilidade para os resultados de WCET gerados, ao passo que, para os que atenderam, é possível garantir tal aspecto.

Os ajustes das curvas de distribuição de probabilidade foram realizados a partir de dois métodos de ajuste: *Levenberg-Marquardt (LM)* e *Maximum Likelihood Estimation (MLE)*. O primeiro método é amplamente utilizado para ajuste de curvas não-lineares, enquanto o segundo é comumente adotado para ajustes de curvas de distribuição, sendo mais empregado na TVE. Para utilização da LM, baseado no histograma de distribuição de densidade de probabilidade, gerado a partir dos dados selecionados por BM, foi obtida a curva PDF utilizando estimativa de densidade Kernel, que trata-se da forma de estimar a função de densidade de probabilidade (PDF) de uma variável aleatória de forma não paramétrica. Um exemplo é ilustrado na Figura 6. A partir dessa estimativa, realizou-se o ajuste de uma distribuição GEV pelo método LM.

Para os *benchmarks* que atenderam aos testes, foram calculados os parâmetros de ajuste para a curva GEV utilizando cada método (LM e MLE). Nos casos em que se obteve parâmetro de forma $\xi < 0$ em algum dos ajustes, realizou-se o ajuste da distribuição por uma curva Gumbel ($\xi = 0$). Por fim, os resultados foram dispostos na Tabela 4.

De modo geral, ambos os métodos de ajustes mostraram-se válidos para determinação dos parâmetros da curva GEV, tendo em vista que o gráfico Quantil-Quantil

Benchmark	Parâmetro	GEV - LM	GEV - MLE	Gumbel - LM	Gumbel - MLE
cnt	ξ	0,1688	0,1605	-	-
	μ	5219,2648	5219,1645	-	-
	σ	21,8694	20,8875	-	-
	WCET	5344,9186	5344,6390	-	-
fdct	ξ	0,1528	0,0292	-	-
	μ	7429,3715	7429,9127	-	-
	σ	37,8606	33,4480	-	-
	WCET	7666,6650	7950,0448	-	-
matmult	ξ	0,2869	0,1161	-	-
	μ	97012,3769	97009,2448	-	-
	σ	145,1511	133,3853	-	-
	WCET	97517,0557	98054,5580	-	-
select	ξ	0,1908	0,0922	-	-
	μ	7076,8208	7076,1816	-	-
	σ	22,3148	21,3767	-	-
	WCET	7191,5201	7273,6603	-	-
jfdctint	ξ	-0,0243	-0,0379	0	0
	μ	9280,2233	9280,1747	9279,8155	9279,8155
	σ	54,1514	49,4162	54,1890	54,1890
	WCET	10738,3779	10836,6258	10402,7884	10318,8289
bs	ξ	-0,258223622	0,369156619	0	0
	μ	1006,461352	1004,293521	1004,592413	1004,592413
	σ	18,89168713	27,3146257	18,68080816	18,68080816
	WCET	16036,5106	1078,250273	1391,719767	1678,732049
insertsort	ξ	-0,2371	0,0948	0	0
	μ	2125,3474	2125,5565	2122,3349	2122,3349
	σ	38,4155	39,9326	37,6754	37,6754
	WCET	24033,0462	2487,6221	2903,0927	2940,0047

Tabela 4. Parâmetros obtidos a partir dos ajustes GEV pelos métodos LM e MLE, e seus respectivos WCET obtidos.

exibiu boa aderência do modelo ajustado aos dados experimentais. No entanto, em alguns casos observou-se que a existência de picos secundários convoluídos ao pico de maior densidade nas curvas PDF, ou inomogeneidade na distribuição dos dados, tem efeito negativo nos ajustes pelos métodos LM e MLE, assim como na confiabilidade dos resultados de WCET gerados. Isso ocorre nos casos dos *benchmarks* ***bs*** e ***insertsort***, apresentados nas Figura 26 e Figura 30, localizados no Apêndice C, juntamente com os demais resul-

tados. Além disso, observa-se na Tabela 4, que para esses benchmarks, os resultados de WCET obtidos pelo modelo GEV-LM são superestimados quando comparados com os calculados a partir do GEV-MLE, Gumbel-LM e Gumbel-MLE. Portanto, conclui-se que a confiabilidade e qualidade dos ajustes estão associadas à homogeneidade da distribuição de probabilidade dos dados, característica inerente da plataforma alvo de execução dos *benchmarks*.

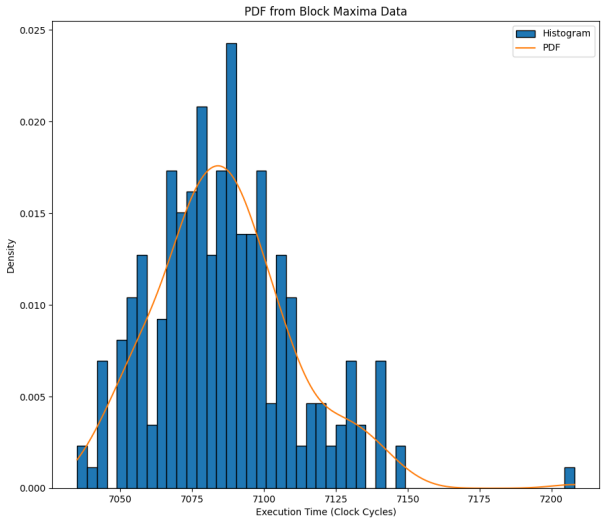

Figura 6. Histograma de densidades de probabilidade e PDF gerado.

Para os *benchmarks* que não passaram nos testes de i.i.d., foram estimados valores de WCET, no entanto a confiabilidade desses resultados não é garantida pela hipótese de aplicabilidade da TVE. Uma tabela resumo foi criada e está disponível em Tabela 5. Para os *benchmarks* **fir** e **nsichneu** o programa não conseguiu ajustar os parâmetros, sendo assim, não foi possível calcular um WCET.

5. Conclusão

A metodologia proposta divide-se em duas frentes de estudo: a investigação da variabilidade de software e a de hardware. Ao estudar o fluxo de programa, o objetivo é encontrar o caminho crítico de execução. Com o WCEP definido e a variabilidade de software, consequentemente, explorada, o desafio torna-se medir e analisar o tempo de execução total no hardware final, visando obter o pior tempo de execução.

A abordagem de tratar os problemas de forma modularizada tem vantagens, como explorado na Seção 3, em especial a separação de responsabilidades e resolução de problemas. No entanto, essa abordagem pode levar a perda da precisão dos resultados, pois a variabilidade de software e hardware estão fortemente relacionadas.

O pior caminho do software pode depender dos mecanismos presentes no hardware (cache, *pipeline*, previsão de saltos e execução fora de ordem) e como esses se aproveitam de características como localidade espacial e temporal para acelerar a execução. E

o hardware pode aproveitar as particularidades de cada código para utilizar efetivamente os já citados mecanismos para a execução de certos trechos. Nesse panorama, tratar a variabilidade de software e hardware de formas separadas pode ser visto como um *trade-off*, se por um lado há o ganho em modularização, modificabilidade e simplificação em cada uma das etapas, por outro há a possibilidade de perda de informação no processo.

Durante a etapa de identificação e medição dos blocos básicos, ficou evidente a facilidade do estudo de uma arquitetura mais simples, proporcionada pelo Atmega328, visto que a variabilidades de *pipeline* e de cache são descartadas da equação. Porém, no processo de medições estabelecido, há a possibilidade de que informações sejam perdidas e de que alguns ciclos acabem não sendo contabilizados. Isso ocorre por limitações da instrumentação no código e seu uso em códigos de alto nível.

Após a análise dos *benchmarks* com a execução abstrata, conclui-se que seu sucesso dependerá da complexidade do fluxo de execução, do(s) tipo(s) e tamanho do intervalo da(s) entrada(s) em estudo. Programas com variáveis do tipo $Inteiro$ e com menos de 4 entradas demandaram menos tempo e recursos do que aqueles que lidam com muitas combinações de entrada e com variáveis do tipo $Float$ ou $Double$. Nesses, o custo para a execução abstrata subiu consideravelmente, tanto em tempo, quanto em recursos computacionais, especialmente a quantidade necessária de memória RAM. O uso de estratégias de *merge*, nesses casos, é dificilmente evitado, caso contrário a análise pode simplesmente não finalizar ou inviabilizar as etapas seguintes.

Na etapa de exploração da variabilidade de hardware, realizada no *BeagleBone Black*, ficaram visíveis os desafios trazidos pelo aumento da complexidade da arquitetura, uma vez que para deixar o processador em sua pior condição, rotinas de manutenção da cache foram implementadas e executadas entre cada medição do programa. A opção de efetuar as execuções em *bare-metal* manteve a simplicidade do ambiente, evitando distúrbios nas medições advindos das operações de um sistema operacional.

Por fim, na etapa de MPBTA a análise permitiu obter um valor de WCET para os benchmarks medidos. No entanto, além da necessidade de verificação da aplicabilidade dos dados ao TVE, a análise depende da homogeneidade da distribuição dos dados e das características da plataforma alvo, a qual deve viabilizar a obtenção de medições adequadas ao TVE.

Referências

Altenbernd, P. (1996). On the false path problem in hard real-time programs. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pages 102–107.

Arcaro, L. F., Silva, K., and de Oliveira, R. (2018). A reliability evaluation method for probabilistic wcet estimates based on the comparison of empirical exceedance densities. In *Anais do VIII Simpósio Brasileiro de Engenharia de Sistemas Computacionais*, pages 129–133, Porto Alegre, RS, Brasil. SBC.

Costa, J. J. S. (2021). Emprego de medição na estimação do tempo de execução no pior caso para sistemas de tempo real -orientador, rômulo silva de oliveira, coorientador, luís fernando arcaro. https://repositorio.ufsc.br/handle/123456789/229274. Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Engenharia de Automação e Sistemas, Florianópolis, 2021.

Davis, R. I. and Cucu-Grosjean, L. (2019). A survey of probabilistic timing analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems*, 6(1):03:1–03:60.

de Rezende, S. F. (2014). Caminhos mais longos em grafos. Master's thesis, USP, https://repositorio.ufsc.br/xmlui/handle/123456789/133239. Karila Palma Silva ; orientador, Rômulo Silva de Oliveira - Florianópolis, SC, 2015. Dissertação (mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo.

Ermedahl, A., Gustafsson, J., and Lisper, B. (2011). Deriving wcet bounds by abstract execution. In Healy, C., editor, *Proc. 11th International Workshop on Worst-Case Execution Time (WCET) Analysis (WCET 2011)*. Austrian Computer Society (OCG).

Felipe D. A. Brito, Gustavo C. E. Santo, I. F. V. O. e. L. G. B. C. (2023a). Pes-tcc-estratégia-híbrida-modular. https://github.com/sw3luke/PES_TCC_WCET-Estrategia_Hibrida_Modular/tree/master.

Felipe D. A. Brito, Gustavo C. E. Santo, I. F. V. O. e. L. G. B. C. (2023b). Rastreamento de requisitos, casos de testes e oráculo. https://docs.google.com/spreadsheets/d/1tN1qobqN7eIpG4Zlqa67YX1dnGO739SYWEtYIIPu5a0/edit?usp=sharing.

Gustafsson, J. (2019). Sweet manual. In *SWEET manual*. Västerås Sweden.

Gustafsson, J., Ermedahl, A., Lisper, B., Sandberg, C., and Källberg, L. (2009). ALF - A

Language for WCET Flow Analysis. In Holsti, N., editor, *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, volume 10 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–11, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6.

Lisper, B. (2014). Sweet – a tool for wcet flow analysis. In Steffen, B., editor, *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 482–485. Springer-Verlag.

Machado, G. I. D. (2021). Analysis of the extreme value theory on the estimation of probabilistic wcet. Master's thesis, PUC-RS, https://tede2.pucrs.br/tede2/handle/tede/10022.

Malich, M. (2017). Avr-tick-counter. https://github.com/malcom/AVR-Tick-Counter.

Martins, G. N. (2018). Validando modelos para verificação de programas p4 por execução simbólica. Master's thesis, UFRGS, https://lume.ufrgs.br/bitstream/handle/10183/190199/001088714.pdf?sequence=1.

Silva, K. P. (2015). Análise de valor para determinação do tempo de execução no pior caso (wcet) de tarefas em sistemas de tempo real. Master's thesis, UFSC, https://repositorio.ufsc.br/xmlui/handle/123456789/133239. Karila Palma Silva ; orientador, Rômulo Silva de Oliveira - Florianópolis, SC, 2015. Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenstrom, P. (2008). The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7.

Yen, S., Du, D., and Ghanta, S. (1989). Efficient algorithms for extracting the k most critical paths in timing analysis. In *26th ACM/IEEE Design Automation Conference*, pages 649–654.

Apêndices

A. Fluxogramas

A.1. Fluxogramas da metodologia

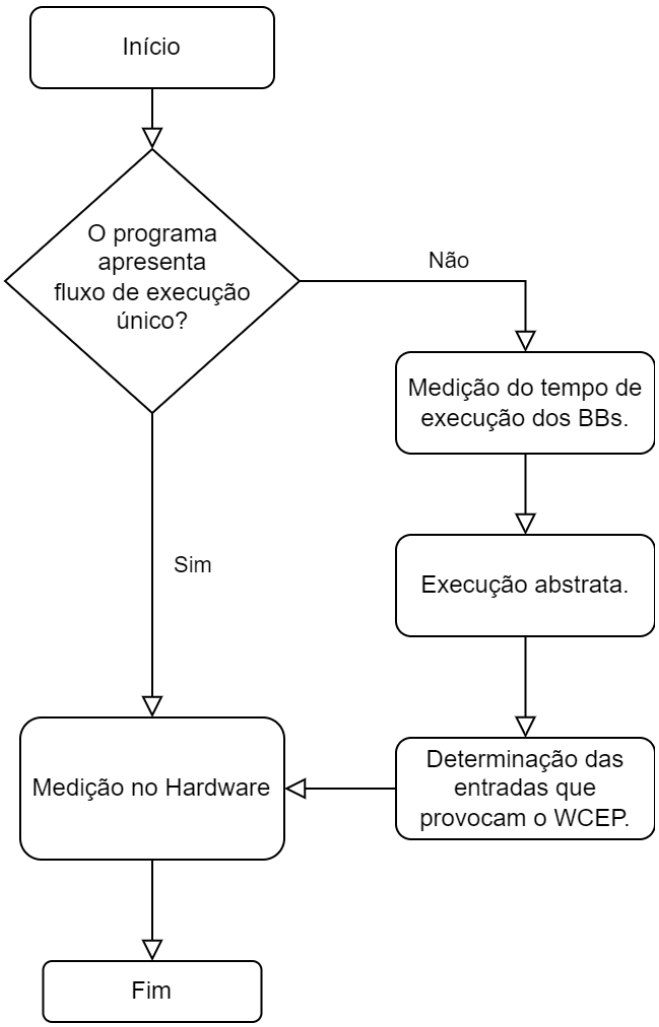

Figura 7. Fluxograma geral da metodologia.

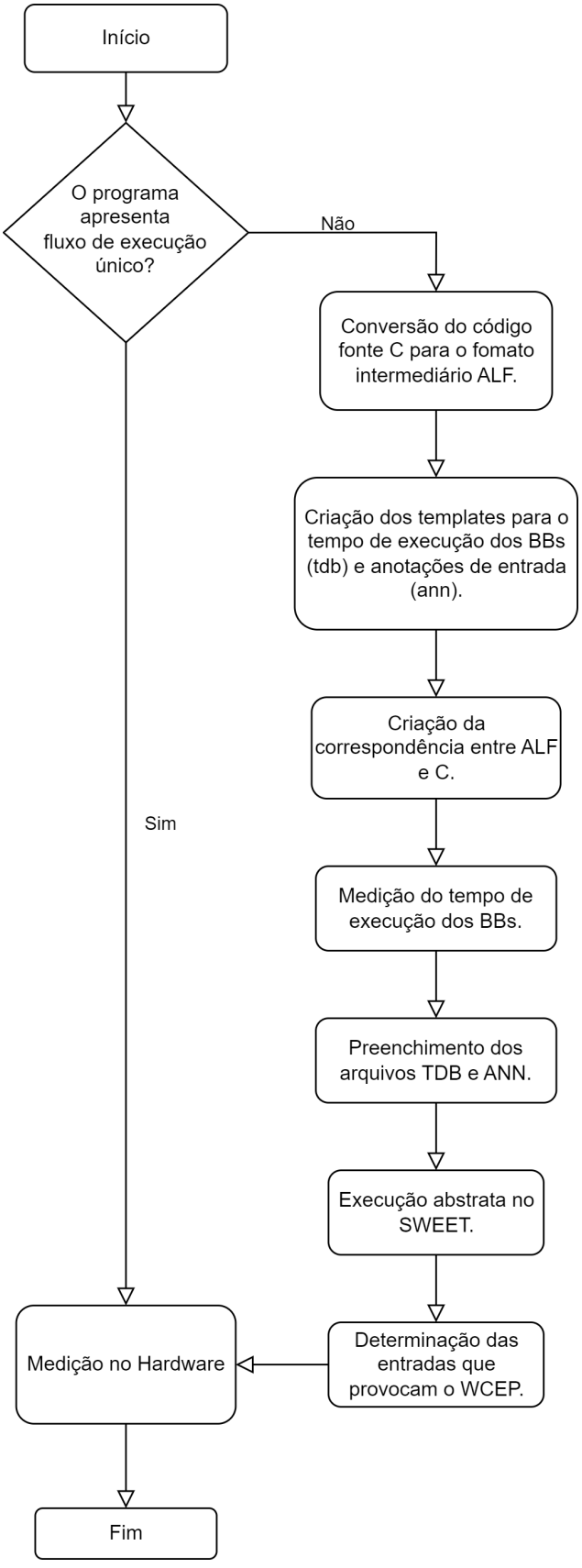

Figura 8. Fluxograma detalhado da metodologia.

A.2. Fluxograma para busca de entradas

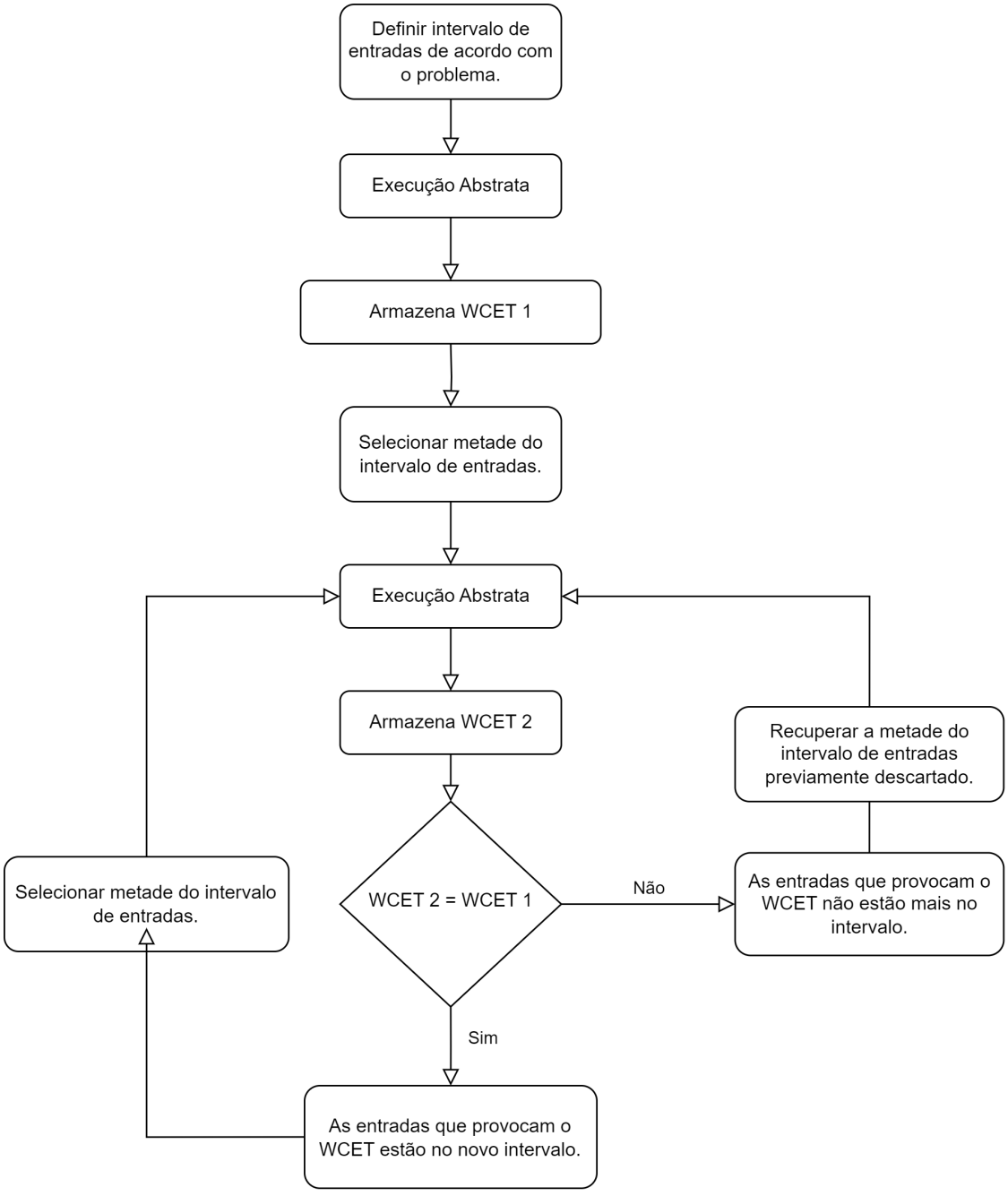

Figura 9. Fluxograma da técnica de busca binária para isolar as entradas que provocam o WCEP.

B. Requisitos e arquitetura de ferramentas

Os requisitos de alto nível que guiaram o desenvolvimento do processo e *framework* foram:

- HLR01: O *framework* deve identificar os blocos básicos relacionados ao código fonte em C fornecido como entrada.

- HLR02: Caso o programa analisado apresente múltiplos caminhos, cuja execução depende das entradas, os blocos básicos devem ser instrumentados para medição de seus respectivos tempos de execução.
- HLR03: O *framework* deve identificar o WCEP baseado na técnica de Execução Abstrata, caso o programa apresente múltiplos caminhos.
- HLR04: A técnica de Execução Abstrata deve ter como entrada os seguintes arquivos: código a ser analisado, tempos de execução dos blocos básicos e arquivo de anotações com os *ranges* possíveis das entradas.
- HLR05: O *framework* deve determinar as entradas que provocam o WCEP.
- HLR06: O fluxo de execução correspondente ao WCEP deve ser provocado pelas entradas responsáveis e medido em Hardware.
- HLR07: O *framework* deve realizar a análise probabilística utilizando a Teoria do Valor Extremo.
- HLR08: O *framework* deve permitir a realização de testes estatísticos (K-Sample Anderson Darling, Kolmogorov-Smirnov, Wald Wolfwitz Runs e Ljung-Box) para verificação da aplicabilidade da TVE, seguindo requisitos mínimos de dados I.I.D..
- HLR09: A análise MBPTA para estimação do WCET deve empregar o método de seleção de dados Máximo de Blocos (BM).
- HLR10: A análise MBPTA, para estimação do WCET, deve receber como entradas o arquivo de dados de tempo de execução, tamanho de bloco referente ao método BM, e valor de probabilidade de excedência (pWCET), retornando como saída o valor de WCET associado à pWCET.
- HLR11: Para realização dos testes de I.I.D., a análise de aplicabilidade TVE deve receber como entradas o arquivo de medições e número de amostras, e deve retornar os resultados de cada teste.
- HLR12: O *framework* deve permitir o cálculo de WCET pela TVE, independente dos dados passarem nos testes de aplicabilidade.
- HLR13: A estimação do WCET e análise de aplicabilidade da TVE devem ser implementadas em código *python*.
- HLR14: O hardware escolhido deve possibilitar a desativação de sua cache secundária(L2), caso a mesma exista.
- HLR15: O hardware escolhido não deve possuir execução fora de ordem.
- HLR16: O processador deve possuir um registrador que permita a leitura da quan-

tidade de ciclos de *clock* decorridos.

- HLR17: O hardware deve permitir ser configurado para executar o programa em *bare-metal*.
- HLR18: O framework deve ser capaz de ajustar a configuração do hardware para o pior caso ao medir cada execução do programa em análise.
- HLR19: O código com os blocos básicos instrumentados deve ser executado por um simulador para coleta dos tempos relacionados.
- HLR20: O simulador escolhido para a medição de blocos básicos deve possibilitar a instanciação de uma arquitetura sem cache e sem *pipeline*.
- HLR21: O *framework* deve ser desenvolvido a partir da composição de componentes modulares.

Os requisitos do *framework* são mapeados para as etapas, tecnologias e *scripts* utilizados e desenvolvidos para o presente trabalho em [Felipe D. A. Brito 2023b]. Já a arquitetura geral do processo/*framework*, que reflete diretamente os requisitos, é apresentada na Figura 10.

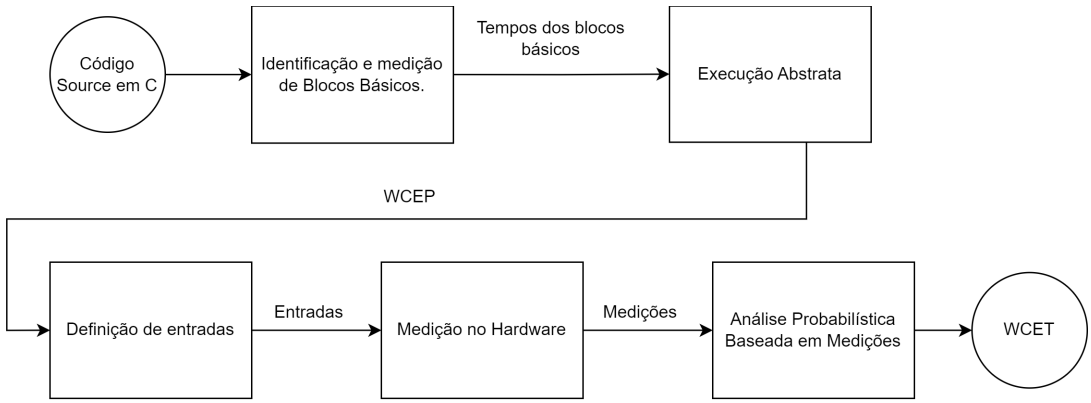

Figura 10. Arquitetura do Framework.

Na etapa de **Identificação e Medição de Blocos Básicos** são utilizados os programas **ALF-Backend**, **Python** e **SimulAVR**. O ALF-Backend atua na conversão do código em C para o formato intermediário ALF, utilizado posteriormente na etapa de execução abstrata. O Python é utilizado para a execução de um *script* que entrega a equivalência entre os dois formatos C e ALF. Por fim, SimulAVR é o simulador responsável por medir os BBs no código-fonte C. Essa etapa é ilustrada na Figura 11.

Na etapa da **Execução Abstrata** é utilizado o programa **SWEET** que realiza a

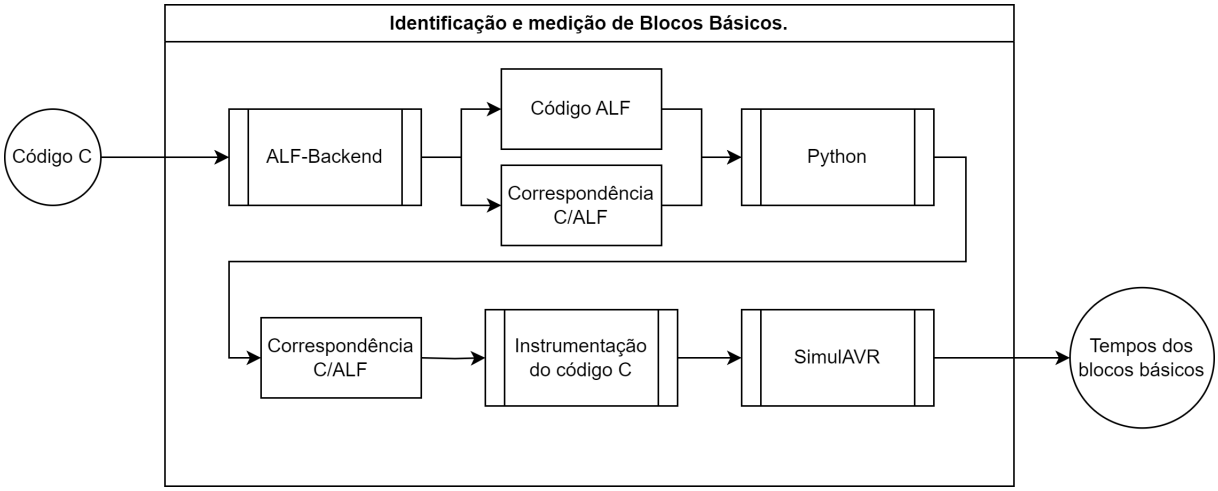

Figura 11. Etapa de Identificação e Medição dos Blocos Básicos.

execução abstrata cujas entradas são: o código no formato intermediário e os arquivos TDB e ANN. O foco dessa etapa é determinar as entradas que provocam o WCEP, assim a execução abstrata é feita de forma iterativa restringindo, a cada repetição, a faixa de entradas possíveis para o WCEP. A Figura 12 ilustra a etapa de execução abstrata descrita acima.

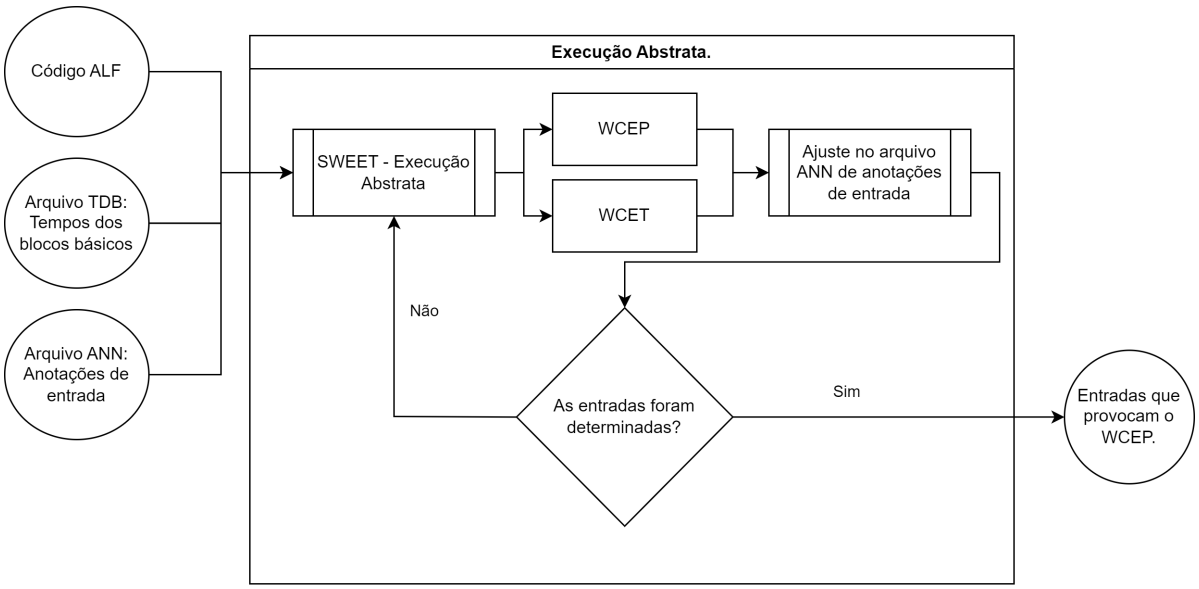

Figura 12. Etapa da Execução Abstrata.

Na etapa da **Medidas no Hardware**, apresentada na Figura 13, é utilizado o código fonte em C e as entradas que provocam o caminho crítico no software. O programa é executado sucessivamente de início a fim e mensurado. Entre cada execução há o cuidado de invalidar as caches de instruções e dados com o objetivo de colocar o ambiente no pior estado possível para a execução. O resultado do processo de medidas são 2 blocos de 50 mil medições cada.

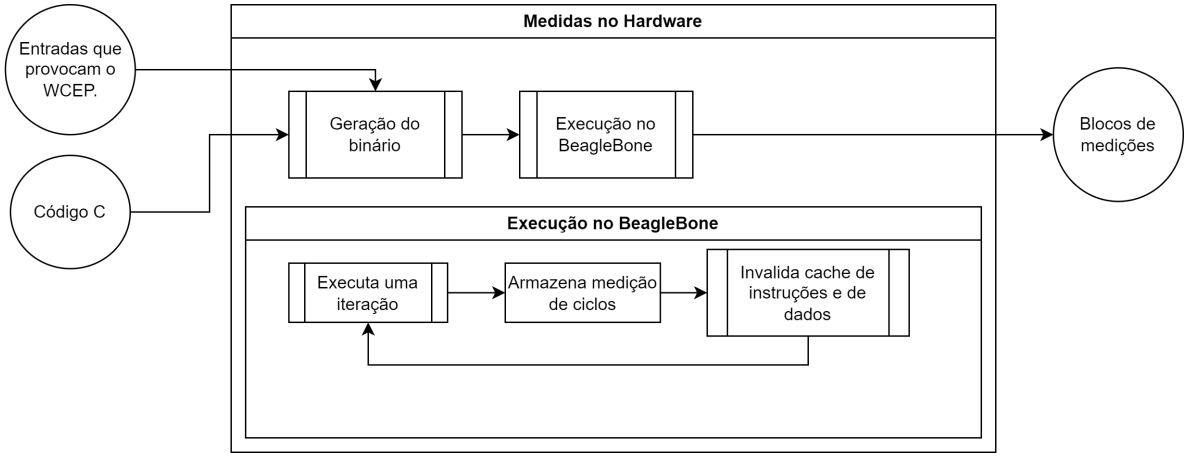

Figura 13. Etapa de Medição no Hardware.

Na etapa da **Análise MBPTA**, utilizando o conjunto de dados coletado, são realizados os testes estatísticos de I.I.D. e estacionariedade dos dados para posterior aplicação da modelagem TVE. Essa etapa é apresentada na Figura 14. Analisando os resultados dos testes, uma vez que os requisitos de aplicação são atendidos, a modelagem é feita, fornecendo a amostra de observações, valor de tamanho de bloco, e valor de probabilidade de excedência, retornando, por fim, o valor de WCET.

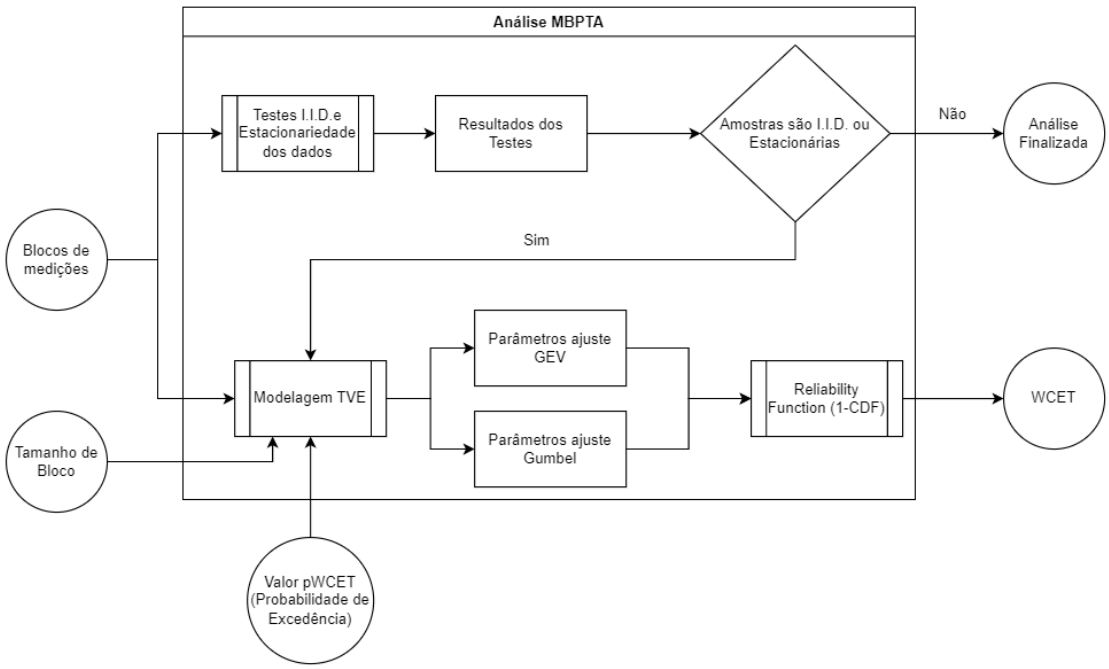

Figura 14. Etapa da Análise MBPTA.

C. Resultados

Neste apêndice são apresentados os gráficos, referentes às curvas de ajuste e de comparação de ajustes, gerados na etapa de Análise MBPTA da Subseção 3.5. Além

disso, uma tabela do resumo dos resultados das análises do *benchmarks* pode ser encontrada ao final desse apêndice.

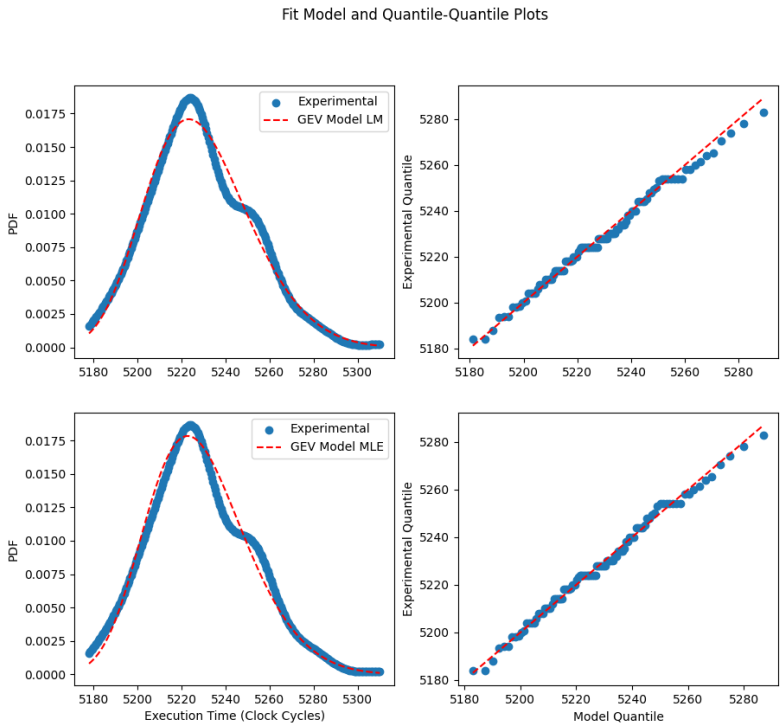

Figura 15. Curvas de Ajuste do modelo GEV pelo método LM e MLE e seus respectivos gráficos Quantil-Quantil para o benchmark *cnt*.

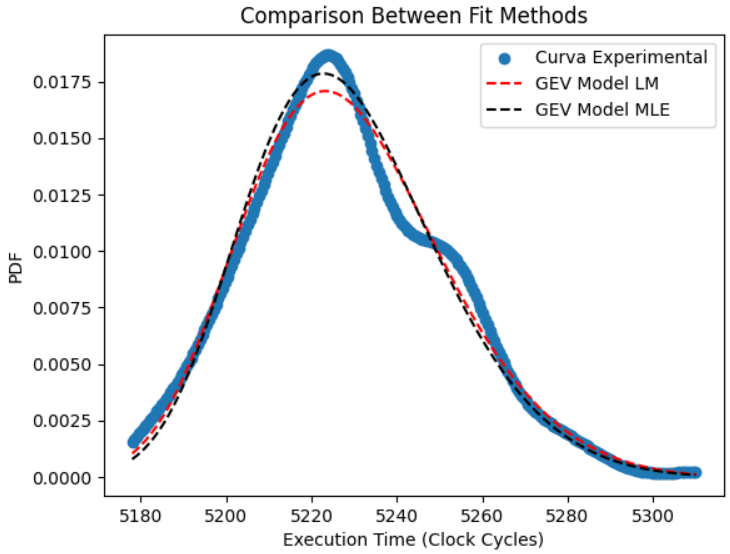

Figura 16. Comparação entre os ajustes do modelo GEV pelo método LM e MLE para o benchmark *cnt*.

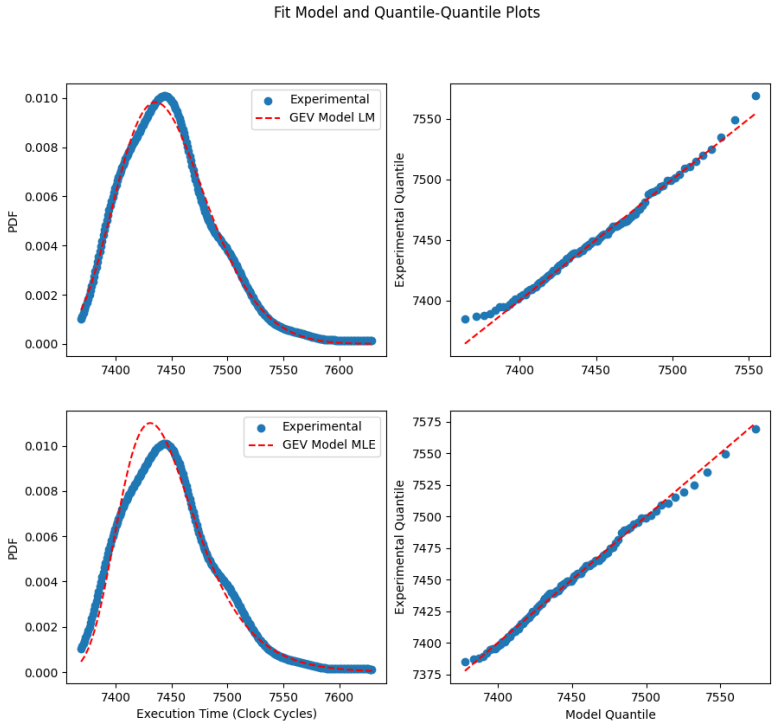

Figura 17. Curvas de Ajuste do modelo GEV pelo método LM e MLE e seus respectivos gráficos Quantil-Quantil para o benchmark *fdct*.

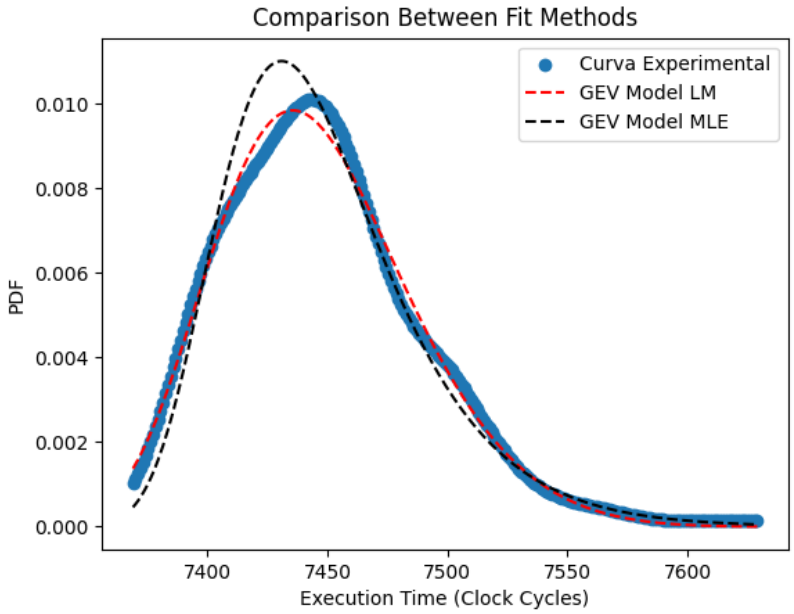

Figura 18. Comparação entre os ajustes do modelo GEV pelo método LM e MLE para o benchmark *fdct*.

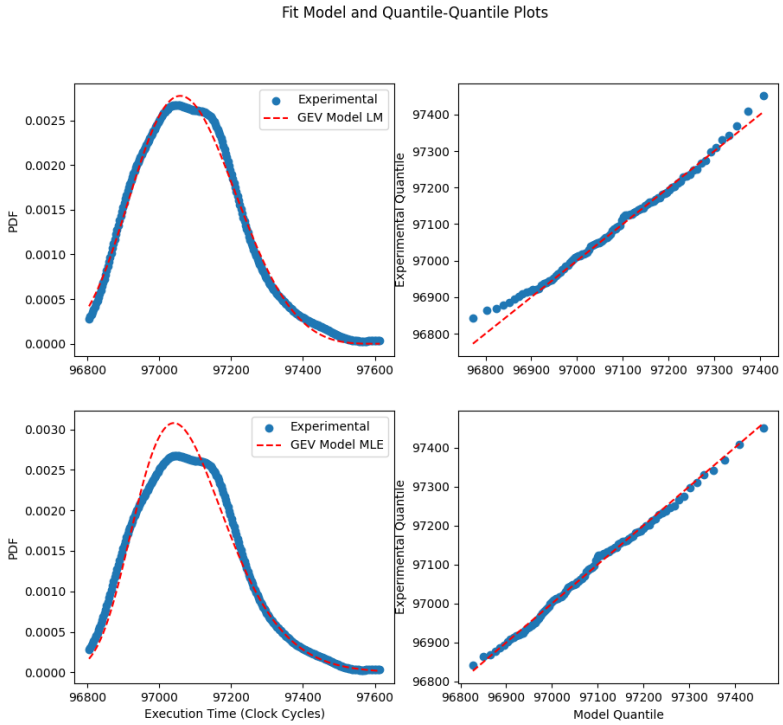

Figura 19. Curvas de Ajuste do modelo GEV pelo método LM e MLE e seus respectivos gráficos Quantil-Quantil para o benchmark *matmult*.

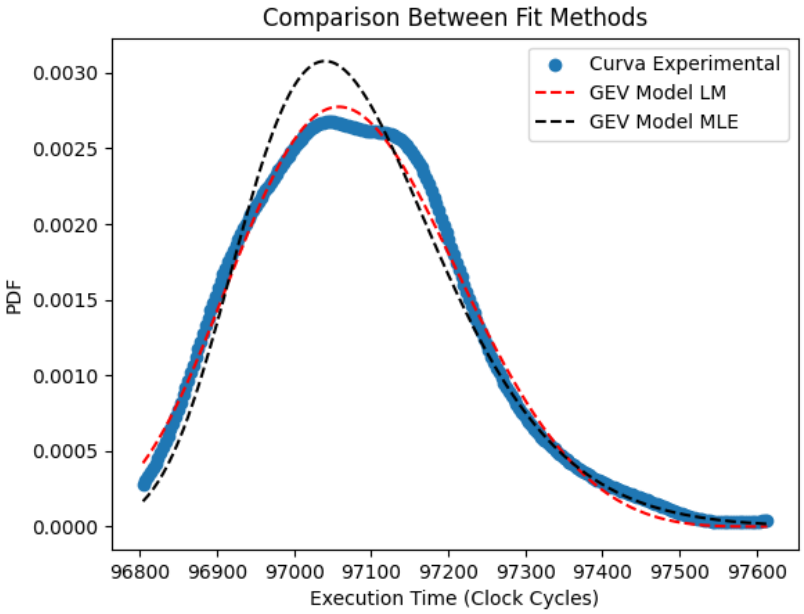

Figura 20. Comparação entre os ajustes LM e MLE para o benchmark *matmult*.

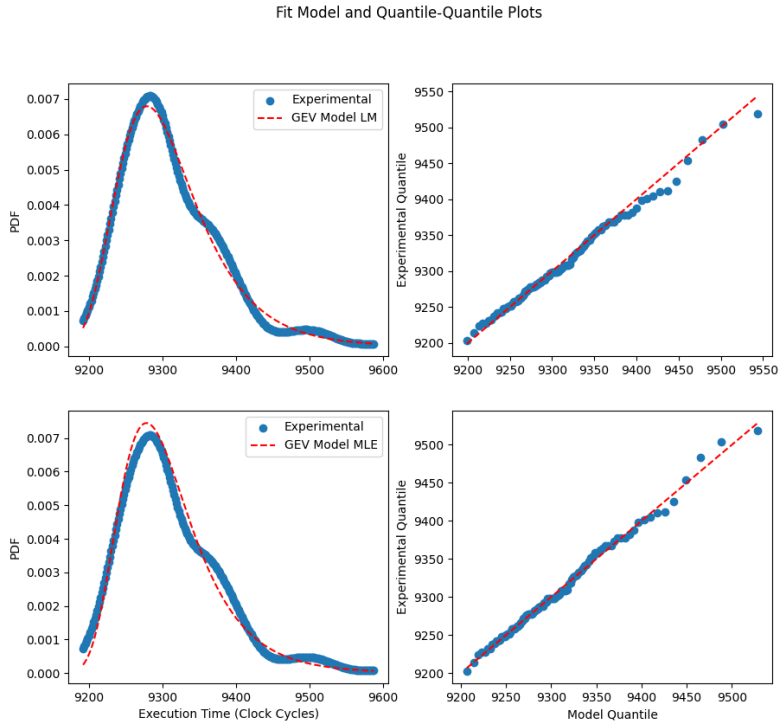

Figura 21. Curvas de Ajuste do modelo GEV pelo método LM e MLE e seus respectivos gráficos Quantil-Quantil para o benchmark *jfdctint*.

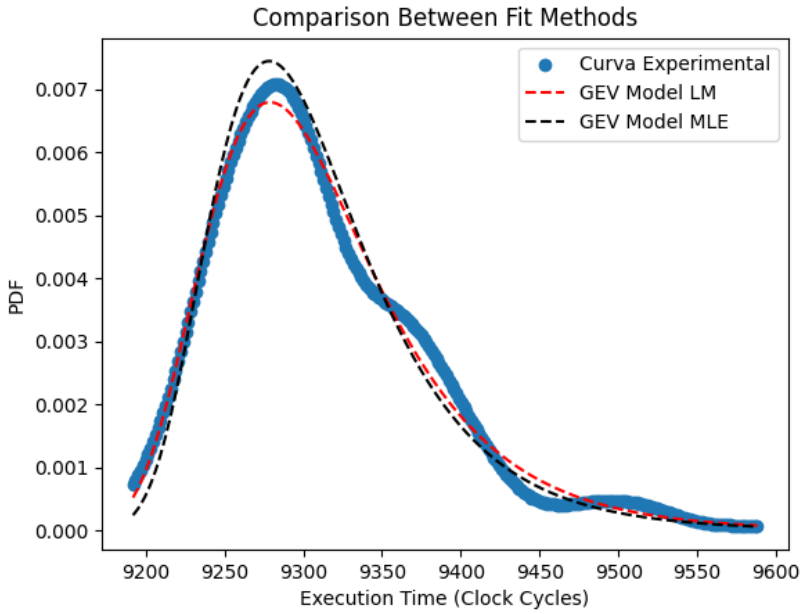

Figura 22. Comparação entre os ajustes do modelo GEV pelo método LM e MLE para o benchmark *jfdctint*.

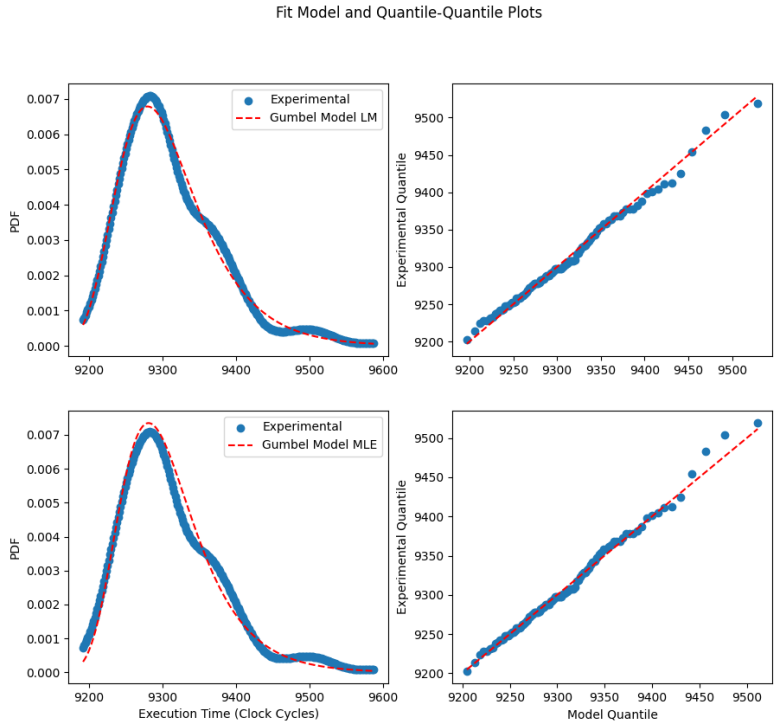

Figura 23. Curvas de Ajuste do modelo Gumbel pelo método LM e MLE e seus respectivos gráficos Quantil-Quantil para o benchmark *jfdctint*.

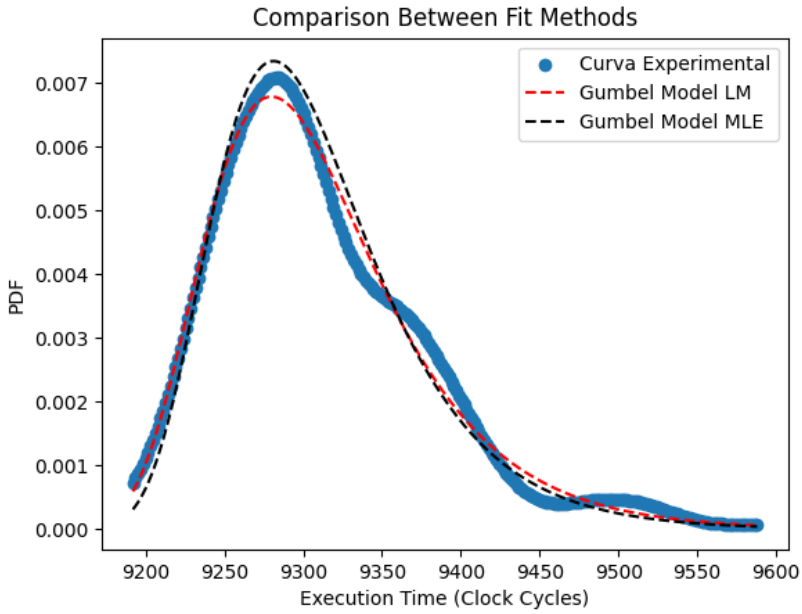

Figura 24. Comparação entre os ajustes do modelo Gumbel pelo método LM e MLE para o benchmark *jfdctint*.

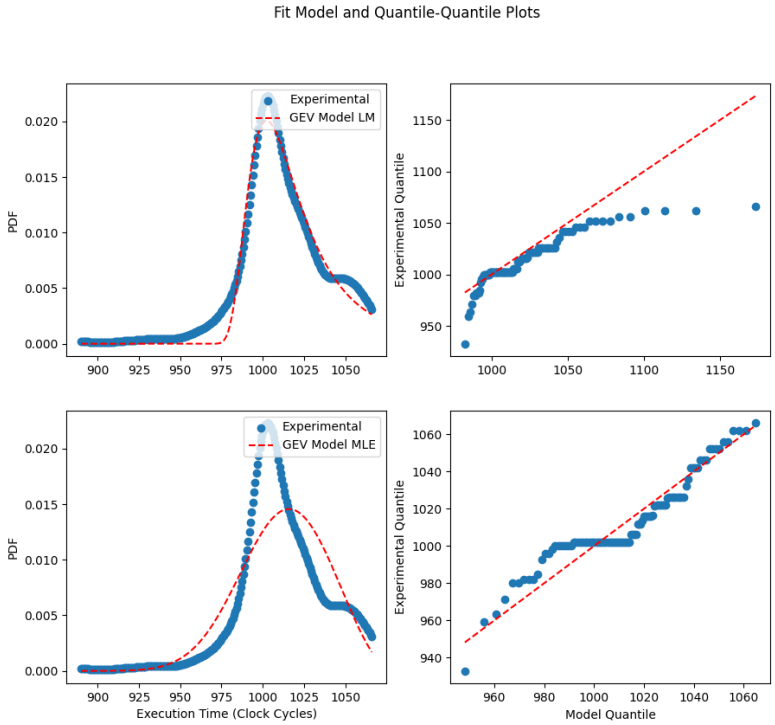

Figura 25. Curvas de Ajuste do modelo GEV pelo método LM e MLE e seus respectivos gráficos Quantil-Quantil para o benchmark *bs*.

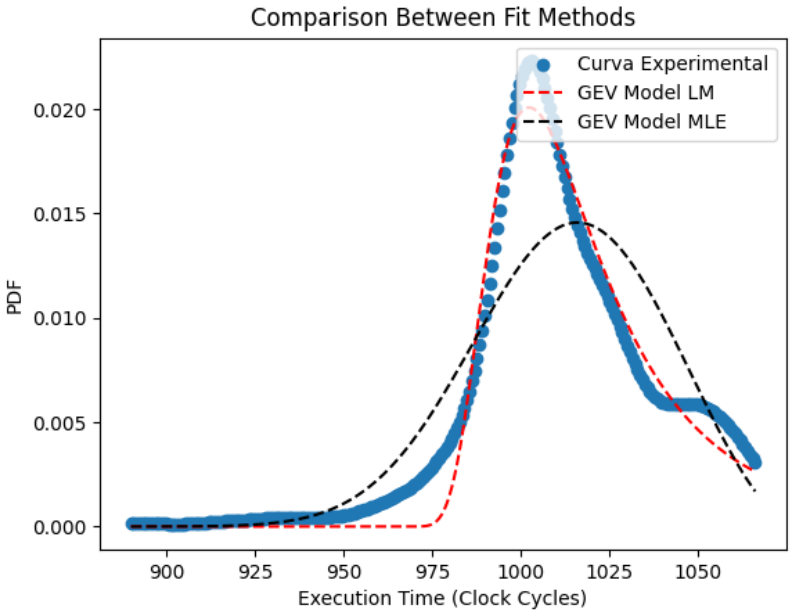

Figura 26. Comparação entre os ajustes do modelo GEV pelo método LM e MLE para o benchmark *bs*.

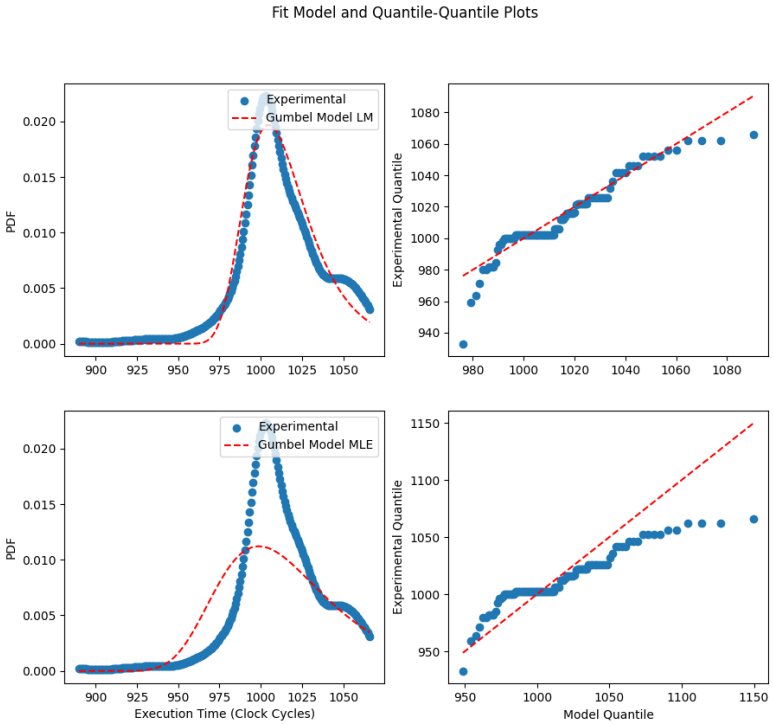

Figura 27. Curvas de Ajuste do modelo Gumbel pelo método LM e MLE e seus respectivos gráficos Quantil-Quantil para o benchmark *bs*.

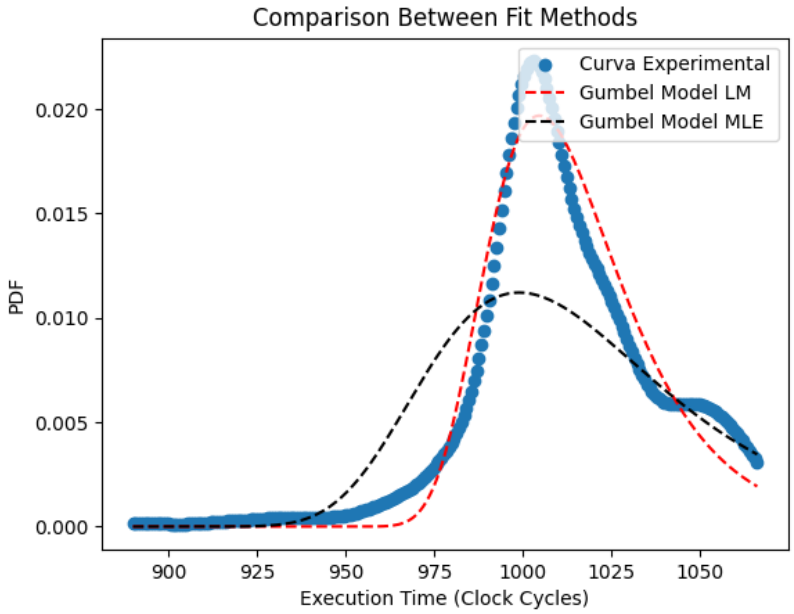

Figura 28. Comparação entre os ajustes do modelo Gumbel pelo método LM e MLE para o benchmark *bs*.

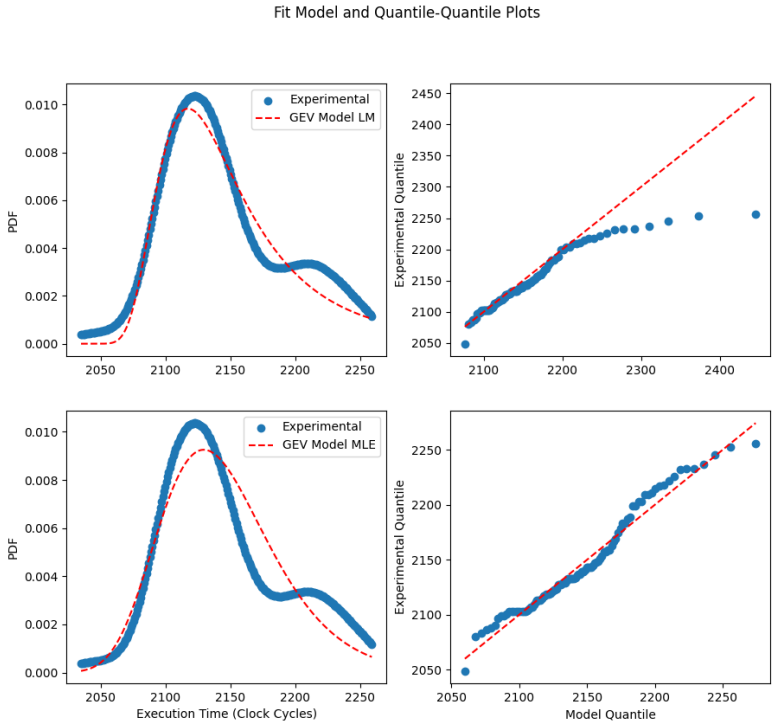

Figura 29. Curvas de Ajuste do modelo GEV pelo método LM e MLE e seus respectivos gráficos Quantil-Quantil para o benchmark *insertsort*.

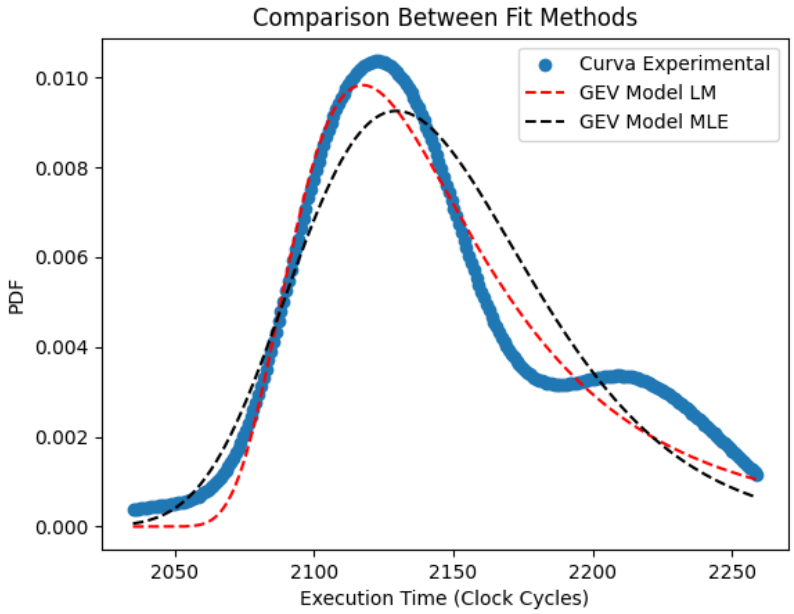

Figura 30. Comparação entre os ajustes do modelo GEV pelo método LM e MLE para o benchmark *insertsort*.

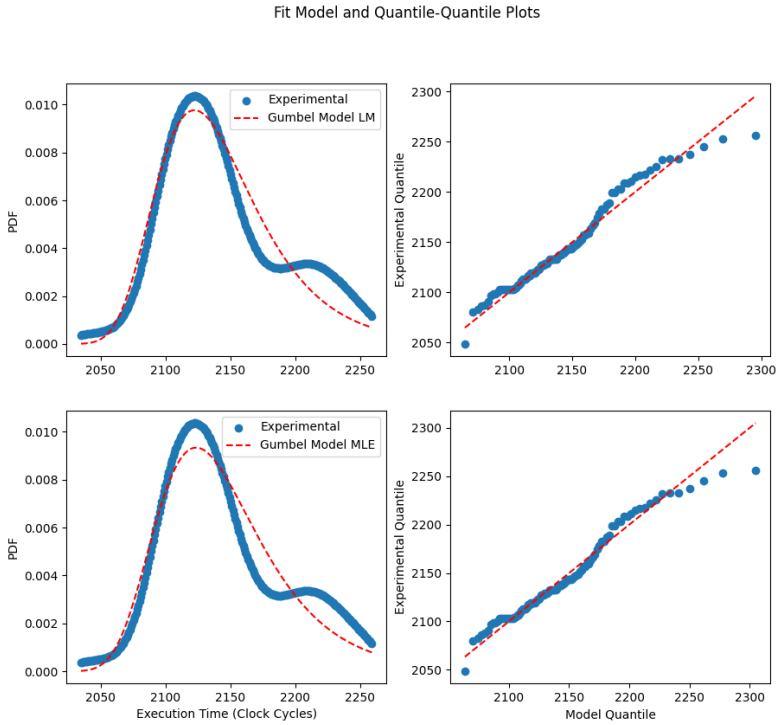

Figura 31. Curvas de Ajuste do modelo Gumbel pelo método LM e MLE e seus respectivos gráficos Quantil-Quantil para o benchmark *insertsort*.

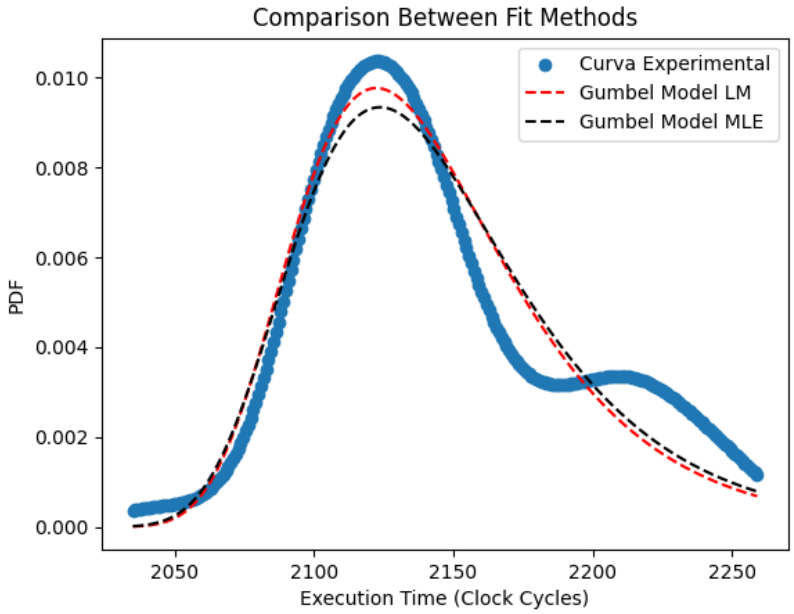

Figura 32. Comparação entre os ajustes do modelo Gumbel pelo método LM e MLE para o benchmark *insertsort*.

Programa	Nº de comb. de entradas	Entradas	Merge	Tempo AE	HWM	Passou Testes I.I.D?	WCET (pWCET = 1e-9)
bs	20	a = 0	-	<1 s	1066	sim	1678,7320
bsort100	-	-	-	-	50292	não	50350,7195
cnt	-	-	-	-	5278	sim	5344,6390
cover	1000	cnt = 0	-	<5 s	-	-	-
crc	-	-	-	-	5568	não	5656,1378
duff	-	-	-	-	2744	não	3830,7799
expint	2500	n = 50, x = 1	-	<10 s	-	-	-
fdct	-	-	-	-	7629	sim	7950,0448
fir	-	-	-	-	245658	não	-
insertsort	1x10^11	a[0] = 0, a[1] = 9, a[2] = 8, a[3] = 7, a[4] = 6, a[5] = 5, a[6] = 4, a[7] = 3, a[8] = 2, a[9] = 1, a[10] = 0	LE	<5 s	2259	sim	2940,0047
janne_complex	900	a = 0, b = 5	-	<20 s	944	não	970,5971
jfdctint	-	-	-	-	9588	sim	10836,6258
lcdnum	16	a=0	-	<1 s	-	-	-
ludcmp	10	n = 49	-	∼15 m	-	-	-
matmult	-	-	-	-	97614	sim	98054,5580
ns	500	x = 5	-	<2 s	-	-	-
nsichneu	-	-	-	-	43788	não	-
prime	1x10^6	x = 997, y = 997	-	<5 s	4894	não	4896,9505
qurt	1000	a = 7, b=9, c=[2..3]	FR	<5 s	20431	não	20543,0854
select	20	k = 5	-	<1 s	7208	sim	7273,6603
sqrt		valor = 1	-	∼1 h	-	-	-
statemate	-	-	-	-	19019	não	19143,2278
ud	50	n = 49	-	∼40 m	399567	não	401929,5976

Tabela 5. Resumo dos resultados das análises nos Benchmarks.

D. Problemas, Limitações e Ressalvas

D.1. Etapa 01 - Identificação e Medição dos blocos básicos

Durante a realização das medições, notou-se uma limitação no SimulAVR, relacionada ao tamanho da RAM de 32 KB, visto que alguns *benchmarks* ultrapassavam esse valor e portanto não era possível carregá-los na memória. A fim de abordar esse problema, os *benchmarks* que se enquadravam nesse cenário foram submetidos a uma das seguintes abordagens:

1. Não prosseguir com o *benchmark*;
2. Ajustar o código do *benchmark* para ocupar menos espaço na RAM.

Para a última alternativa, abordagens, como reduzir o tamanho das estruturas de dados instanciadas pelo *benchmark*, foram utilizadas. Entendendo o objetivo do código avaliado, era possível alterá-lo, sem modificar o WCEP obtido pelos custos de tempo dos BBs que o compõem. Além disso, é importante frisar que os valores obtidos pelo cálculo dos BBs servem como uma estimativa do custo relativo dos blocos, custo esse que passa a depender da plataforma em que as medições foram realizadas, nesse caso relacionadas ao Atmega328, emulado pelo SimulAVR.

Quanto à instrumentação a nível do código-fonte, utilizando como referência os BBs definidos pelo mapeamento gerado a partir do ALFBackend. Ressalta-se que concessões foram feitas para trechos em que a medição não poderia ser realizada, como nos casos em que o mapeamento de um bloco básico, gerado a partir do código ALF, apontava para instruções em código C, como *return*, *else if* ou chaves de abertura/fechamentos - {}.

Como o processo de instrumentação é feito a partir dos BBs definidos no arquivo ALF, e devido às concessões necessárias comentadas anteriormente, a etapa de instrumentação dos blocos básicos em C foi realizada manualmente. Fez-se necessário checar o arquivo ALF para uma melhor interpretação da correspondência dos BBs, como nos casos em que mais de um BB apontava para uma mesma instrução do código C.

D.2. Etapa 02 - Execução Abstrata e o SWEET

Como introduzido na Subseção 3.2, a Execução Abstrata lida com estados abstratos, que é uma coleção de todos as variáveis abstratas e seus possíveis valores em um determinado ponto no programa. Cada caminho de execução pode corresponder a um estado. Por exemplo, se houver uma estrutura condicional ***if***, que pode ser tanto avaliada como verdadeiro ou falso, a execução abstrata seguirá os dois fluxos de forma independente, cada um representando um estado diferente.

Na execução abstrata, todos os caminhos de execução possíveis (estados) são tratados simultaneamente. Além disso, certas estruturas como *loops* com condicionais internas resultam em um rápido aumento no número de estados [Wilhelm et al. 2008]. Técnicas de *merge* permitem a fusão dos estados abstratos em pontos específicos, acelerando a execução abstrata, no entanto, ao custo da perda de informação, afetando a precisão da estimativa do WCET, tornando-a mais pessimista. Portanto, há um *trade-off* entre o tempo de análise da execução abstrata e a precisão dos valores obtidos para o WCET e WCEP [Ermedahl et al. 2011].

Dado que a busca se baseia em obter um WCEP e as entradas que o estimulam, é importante que o WCEP encontrado seja realizável e não apenas uma aproximação. Isso garante que as entradas obtidas possam ser efetivamente utilizadas posteriormente na etapa de medição no hardware final, e realmente manifestem o WCEP. No entanto, nem sempre é possível atingir esse determinismo. Dependendo do estilo de programação,

quantidade de combinações de entrada e o uso de entradas do tipo *float* (Subseção D.3), a expansão de todos os estados abstratos se torna computacionalmente inviável, impossibilitando a conclusão da execução abstrata. Assim, optou-se por inicialmente não utilizar as técnicas de *merge* nos códigos *benchmarks*, aumentando a precisão do WCEP obtido. Caso a análise não seja concluída, então a estratégia de *merge* é adotada com a consciência de que o WCEP retornado pode não ser realizável semanticamente no software.

Um grande obstáculo no uso do SWEET é a falta de documentação detalhada sobre a ferramenta. Há a falta, por exemplo, de uma seção em que sejam explicadas as condições e códigos de erro, e o que significam. Por exemplo, em algumas análises a execução abstrata finaliza com o retorno *"Killed"* sem maiores detalhes do motivo. A seção sobre *Abstract Input Annotations* documenta bem os usos, mas é pobre em exemplos práticos, o que levou a dúvidas sobre como assinalar valores em estruturas compostas como *Arrays*. A seção sobre *Debug* também é deficiente e as possibilidades de monitorar o que realmente ocorre na execução abstrata é limitada.

Por fim, o maior dos problemas é que há casos em que a execução abstrata não finaliza e causa travamentos na máquina. Isso ocorreu em programas que apresentaram muitos estados a serem trabalhados simultaneamente, principalmente devido à abundante quantidade de combinações de entradas e uso de variáveis de ponto flutuante, o que levou ao esgotamento dos recursos do sistema. Na execução abstrata dos exemplos **que trabalhavam com tipos *float***, além do terminal do SWEET, vários outros processos abertos encerraram e a interface gráfica do Linux foi reiniciada. Notou-se no entanto, ao analisar os mesmos *benchmarks* em um computador com maior quantidade de recursos (principalmente memória RAM) que a execução abstrata passou a ser bem sucedida. Apenas em dois casos, nos *benchmarks* **15-insertsort** e **28-qurt**, observou-se o SWEET consumir os 54GB de RAM disponíveis da máquina sem alcançar a conclusão da análise.

D.3. Etapa 03 - Determinação das entradas

Há uma ressalva sobre os tipos de dados que a execução abstrata suporta manipular quando as estratégias de *merge* estão desabilitadas. No capítulo da documentação [Gustafsson 2019], que explica sobre o **arquivo de anotações**, é indicada a possibilidade do uso de *floats* para as variáveis. Na prática, no entanto, execuções de códigos do SWEET com *inputs* desse tipo não atingiram a conclusão da execução do algoritmo, finalizando em estado de erro. Isso restringe o uso de entradas para o tipo INTEIRO para uso da execução abstrata em máxima precisão, isto é, obter o WCEP que é semanticamente realizável.

D.4. Etapa 04 - Medida no hardware

Durante a etapa de medição no hardware, um grande obstáculo inicial foi a configuração do dispositivo para a execução *bare-metal*, o que foi fortemente facilitado pelo projeto *StarterWare*, disponibilizado pela *Texas Instrument*. Dito isso, o projeto em questão, in-

felizmente, apesar de ainda disponível, não é mais mantido pela empresa. Além disso, os fóruns oficiais do projeto não existem mais, o que torna desafiador a busca por informações relacionadas.

Com a etapa de execução em *bare-metal* resolvida, outro problema se tornou evidente durante as medições, visto que a amostra de medições extraída para alguns *benchmarks* não se mostrou apta para os testes estatísticos, o que parece estar relacionado a um problema inerente do hardware, que faz com que os dados não sejam suficientemente randomizados. Além disso, a rotina de manutenção da cache também traz ressalvas, tendo em vista que dependendo do tamanho da cache e do tamanho da memória RAM, a manutenção, da maneira concebida pode se tornar inviável, tendo em vista que a RAM talvez não tenha disponível endereços de memória suficientes para que todos os índices da cache sejam visitados um número suficiente de vezes.

D.5. Etapa 05 - Measurement-Based Probabilistic Timing Analysis - MBPTA

Uma observação importante a ser destacada, além do cumprimento dos requisitos mínimos para aplicação da TVE, é a escolha do tamanho de bloco. Não existe ao certo um tamanho de bloco ideal para o método BM, no entanto, é reportado na literatura tamanhos de bloco entre 50 a 250 medições, a depender do tamanho total da amostra. Deve-se atentar que tamanhos de bloco muito pequenos acabam por capturar ruídos indesejáveis à curva de distribuição, enquanto que tamanhos de bloco muito grandes levam a perda de informações relevantes, além de resultar em um baixo número de dados para a composição da curva PDF. Desse modo, o ideal é analisar caso a caso qual tamanho de bloco permite a obtenção da curva PDF suficiente para o ajuste do modelo GEV.

E. Informações Complementares

E.1. Sobre o ALF

Para utilização da ferramenta SWEET, a qual é discutida na Subseção 3.2, é necessária a preparação de arquivos auxiliares, a partir do código C a ser avaliado, os quais são utilizados como arquivos de entrada para execução da ferramenta. Dentre eles, o principal é o arquivo de representação intermediária chamado ALF (*ARTIST2 Language for WCET Flow Analysis*).

O ALF é uma linguagem intermediária, desenvolvida pela equipe da Universidade de Mälardalen, no intuito de ser usada em análises de fluxo para cálculo do WCET (*Worst Case Execution Time*). De modo geral, a análise de fluxo pode ser realizada tanto em código-fonte, código intermediário ou código binário. No entanto, a análise feita em cada um dos formatos de código pode trazer diferentes informações referentes a um mesmo artefato em questão:

- **Código Binário:** A análise do código binário garante que seja encontrado o fluxo de informações correto que é executado no processador. No entanto, as

informações disponíveis no código-fonte podem ser perdidas no binário, como as informações acerca do tipo de dado (inteiro, ponto flutuante, ponteiros) e estrutura de controle, o que pode levar a uma análise de fluxo menos precisa;

- **Código-Fonte:** O código-fonte normalmente pode ser analisado com mais detalhes, uma vez que há muito mais informações acerca do tipo de dados, fluxo de controle, estruturas de dados, definição de funções e chamadas. Por outro lado, as otimizações do compilador podem fazer com que a estrutura de controle do código binário gerado seja diferente daquele do código fonte;
- **Código Intermediário:** O código-fonte e o código intermediário gerado a partir dele geralmente são aproximadamente isomórficos. Além disso, após as otimizações, o código intermediário tem um fluxo de programa próximo ao fluxo do código binário gerado. Essas propriedades tornam o código intermediário um candidato interessante para análise de fluxo, uma vez que pode ser analisado quanto às propriedades de fluxo do código-fonte e binário.

Desse modo, a proposta do ALF é ser uma linguagem genérica para análise de fluxo utilizada em WCET. Pode ser gerada a partir de qualquer uma das representações de código mencionadas acima, agrupando as principais informações que a análise isolada em cada formato de código trariam, em um único código [Gustafsson et al. 2009].

ALF faz parte do desenvolvimento do projeto ALL-TIMES. A proposta desse projeto consiste na interoperabilidade das diversas ferramentas dos principais fornecedores comerciais (Rapita, AbsInt) e universidades, a fim de desenvolver ferramentas integradas usando estruturas e interfaces de ferramentas *open-source*.

E.2. MBPTA - Máximo de Blocos

De acordo com [Davis and Cucu-Grosjean 2019], o método de Máximo de Blocos pode ser resumido da seguinte forma:

- Obtenha uma amostra de observações de tempo de execução usando um protocolo de medição apropriado.
- Verifique, através de testes estatísticos apropriados, se as observações coletadas do tempo de execução são analisáveis usando TVE.
- Divida a amostra em blocos de tamanho fixo e obtenha o valor máximo para cada bloco.
- Ajuste uma distribuição de valor extremo generalizado (GEV) à distribuição dos valores máximos obtidos.
- Alternativamente, caso o parâmetro de forma obtido no ajuste anterior seja $\xi \leq 0$, ajuste uma distribuição Gumbel, ou seja, parâmetro de forma fixado em zero ($\xi = 0$).

- Verifique a qualidade do ajuste entre os máximos e o GEV ajustado (por exemplo, usando gráficos de quantis).
- A partir da distribuição GEV obtida para os valores extremos, estime a distribuição pWCET.

E.3. Análise da Cache

Visto que a memória cache aumenta o desempenho da execução, é de suma importância que, para a coleta das amostras, a cache seja colocada sempre em seu pior estado antes de cada nova execução do programa alvo, a fim de evitar medições que possam acarretar em um *WCET* muito otimista. Sendo assim, o pior estado da cache pode variar entre uma cache suja, para caches que possuam políticas *write-back*, e cache limpas ou sujas, para caches com políticas *write-through*.

A rotina de manutenção da cache tem uma grande dependência de sua modelagem, isto é, de seu tamanho total, da quantidade de índices, do número de linhas, do tamanho das linhas e da política de substituição. No cenário estudado, a cache possui um total de 518 linhas, cada uma com um tamanho de 64 *bytes*. Por ser uma cache *4-way associative*, cada índice irá agrupar 4 linhas, o que resultará em uma cache com 128 índices únicos. Levando em consideração que uma *word* possui 4 *bytes*, cada linha na cache será capaz de armazenar 16 *words*, logo a rotina de manutenção da cache deve ser capaz de acessar endereços de memoria suficientes, pertencentes a diferentes blocos, de forma que todos os índices da cache sejam visitados.

Dadas as características da cache apresentada, se um carregamento de 4 blocos de memória for mapeado para um mesmo índice e ocorrer um *Cache Miss* nos quatro eventos, todas as linhas seriam substituídas no caso de uma política de substituição LRU (*Least Recently Used*). No entanto, neste caso a cache possui a política de substituição aleatória, o que afeta o determinismo da escolha da linha. Para aumentar as chances de todas as linhas serem visitadas, aumenta-se o número de carregamentos em um mesmo índice, diminuindo a probabilidade de uma linha não ser substituída. Assim, a rotina de manutenção foi implementada para cada índice ser visitado 16 vezes, fazendo com que a chance de uma linha do índice não ser visitada seja de aproximadamente 4%.

Visto que a política de substituição da cache prejudica o determinismo da manutenção, no cenário estudado se torna extremamente custoso alcançar a substituição de 100% das linhas. É possível aumentar as chances de que todas as linhas sejam de fato alteradas, porem isso se torna dependente do numero de acessos a um mesmo índice, o que por sua vez gera um *trade-off* entre endereços de memoria pertencentes a blocos diferentes utilizados na operação e a porcentagem de linhas únicas visitadas. Dito isso, na operação de manutenção implementada, após 8192 eventos, a probabilidade de que pelo menos 126 dos 128 índices tenham tido seu conteúdo completamente afetado é de

90%.

F. Instalações

O passo a passo das instalações e procedimentos realizados nas ferramentas, além dos principais comandos estão disponíveis em [Felipe D. A. Brito 2023a]. As instruções de instalação para o ALF-Backend e o SWEET se aplicam a distribuição Ubuntu 22.04.3 do Linux.