Alexandre Machado

Análise de Tempo de Execução em alto nível para Sistemas de Tempo Real utilizando-se o framework LLVM

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do grau de bacharel em Sistemas de Informação.

Orientador:
Olinto José Varella Furtado

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Florianópolis-SC

2008

Sumário

Lista de Figuras

Resumo

Os sistemas chamados “de tempo real” são aqueles que têm tarefas que precisam ser executadas em intervalos (ou “janelas”) pré-determinados de tempo, ou seja, que tem tarefas que, uma vez iniciadas, têm um tempo máximo para terminar.

No projeto de sistemas de tempo real, é necessário prever, a priori, se a tarefa de tempo real vai poder ser executada a tempo em um determinado processador, realizando-se o que se chama de “análise de tempo de execução”, que é realizada através de medidas de tempo de execução do programa para um conjunto de entradas (análise dinâmica), ou através do cálculo do tempo de execução a partir da análise do código (fonte ou executável) do programa (análise estática).

O uso da análise dinâmica em sistemas de tempo real “crítico” (*hard real time*) traz consigo o risco de que as medidas ignorem os extremos (tempo de execução no melhor caso — BCET – *Best Case Execution Time* — e tempo de execução no pior caso — WCET – *Worst Case Execution Time*), que são as duas medidas de tempo de maior interesse, e que podem não estar presentes nas medidas pela dificuldade de se identificar os dados de entrada que provocam o melhor e o pior caso. Em virtude disto, há um interesse crescente pela pesquisa em métodos estáticos de cálculo de tempo de execução.

O cálculo de tempo de execução através de métodos estáticos é dividido em duas fases: a análise de fluxo de execução do programa ou *análise de alto nível* e a simulação do comportamento de tempo do processador ou *análise de baixo nível*.

Este trabalho realiza um breve estudo das técnicas de análise de alto nível, avalia o framework LLVM como ferramenta para construção de um analisador de tempo de execução e implementa a técnica de análise de fluxo de execução do programa por interpretação abstrata do código utilizando estre framework.

A LLVM (*Low Level Virtual Machine*) é um framework para o desenvolvimento de compiladores, voltado para otimização e análise de código, que tem como principal característica uma representação intermediária de código (LVIS – *LLVM Virtual Instruction Set*) de baixo nível (próximo ao assembly do processador MIPS), mas com estruturas e informações de tipos de dados de alto nível, adequadas às técnicas de análise de fluxo de execução do programa geralmente empregadas em código fonte.

O protótipo implementado neste trabalho e os resultados obtidos a partir dele consistem em uma indicação da viabilidade do uso da LLVM para o desenvolvimento de ferramentas de análise de tempo, e aponta caminhos para o desenvolvimento desta ferramenta.

Abstract

The so called “Real Time Systems” are systems that have at least one task that need to be executed within a specific “time interval” or “time window”. This kind of task, once started, need to be concluded before a time interval is finished.

During the design of real time systems, it’s necessary to know if the system (processor, memory, bus,etc) is able to conclude the real time tasks in the previously determined time windows. The process of studding the system to determine if it is able to meet the time constraints is called “time analysis”, and it can be done in two ways: measuring the execution time to a set of input data (the dynamic analysis) or analyzing the code (source or executable) of the program to calculate the execution time (static analysis).

The use of dynamic analysis is unsafe for “hard real time” systems, once the measurements can miss the extreme values (the BCET – *Best Case Execution Time*, and the WCET – *Worst Case Execution Time*). These two values are the most important measurements during the design of real time systems, and the dynamic methods can miss it because the input data for the best and worst case are, usually, unpredictable. This is the reason that leads the research community to invest in the development of static analysis methods.

The Static Time Analysis is done in two steps: the control flow analysis (or *high level analysis*) and the simulation of the processor’s behavior (*low level analysis*).

The present paper does some research on the methods and techniques of static analysis, focused on *high level analysis*, evaluates the LLVM framework as a tool and starting point to the construction of a time analysis tool and implements a prototype tool to the *high level analysis* using this framework and a technique of abstract interpretation of the program’s code.

The LLVM (*Low Level Virtual Machine*) is a compiler development framework, focused on code analysis and optimization. The main feature of the LLVM is the intermediary code representation, (LVIS – *LLVM Virtual Instruction Set*). The LVIS is a low level code representation (it resembles the MIPS processor’s assembly), but it have high level type information and type constructions, enabling the use of analysis techniques usually applied on source code level.

The prototype tool implemented to this paper, and the results that can be obtained using this tool show us that it’s possible to use the LLVM to the development of time analysis tools, and show some ways to to that.

1 Introdução

Vivemos em um mundo com uma dependência crescente de computadores, muito além do que podemos perceber a um primeiro olhar. Os computadores do nosso dia a dia não se parecem em nada com os "computadores pessoais" que estamos acostumados a utilizar. Eles estão nos nossos telefones celulares, carros, fornos de microondas, refrigeradores e máquinas de lavar roupa, entre outros equipamentos.

Estes computadores são componentes dos chamados "sistemas embarcados"[1]. Um sistema embarcado é, segundo Ermendahl (2003), "um computador que não se parece com um computador". E estes computadores estão quase onipresentes: 98% dos processadores vendidos são utilizados em sistemas embarcados (ERMENDAHL, 2003), como automóveis, brinquedos, equipamentos de telecomunicações ou controle industrial.

Sistemas embarcados são sistemas de processamento de informações construídos a partir da combinação de software e hardware de computador com outros componentes eletrônicos e/ou mecânicos, com fim pré-determinado. De modo geral, são componentes de sistemas ou produtos maiores, como, por exemplo, roteadores para redes de computadores e/ou controladores para equipamentos industriais. (BARR, 2007)

Os sistemas embarcados contrastam com os computadores de uso geral, como os PCs (*personal computers*) ou Estações de Trabalho (*workstations*) pelo fato de serem construídos com propósitos específicos e pré-definidos, e, em função disto, terem características que favorecem o uso para este propósito e dificultam o uso para outros fins.

Muitos sistemas embarcados são também sistemas de tempo real (*Real Time Systems*), ou seja, sistemas que têm restrições de tempo para sua execução. Uma maneira prática de identificar um sistema de tempo real é responder a seguinte pergunta: Se o sistema der uma resposta após um tempo máximo isto é tão ruim quanto ou pior que o sistema dar uma resposta errada? Se a resposta for afirmativa, o sistema pode ser considerado de tempo real. (BARR, 2007)

[1] *"embedded systems"*

Por exemplo, se uma câmera de vídeo está gravando imagens a uma taxa de 40 quadros por segundo, o sistema que codifica um quadro de vídeo tem $\frac{1}{40}$ segundos para codificar cada quadro. Se levar um tempo maior, parte do quadro não será codificado, e aparecerá como falha na imagem.

Os sistemas embarcados e de tempo real podem ser classificados em dois tipos: *sistemas de tempo real brando* (*soft real time systems*) e *sistemas de tempo real crítico* (*hard real time systems*).

Os sistemas de tempo real *brando* são aqueles nos quais, se o sistema não for capaz de terminar a tarefa dentro do intervalo pré-determinado de tempo, o usuário percebe uma queda na qualidade do serviço, mas esta queda não impossibilita o uso do serviço ou causa maiores prejuízos. É o caso do exemplo citado da codificação ou decodificação de vídeo. Por outro lado, em sistemas de tempo real *crítico*, uma falha em executar a tarefa dentro do tempo previsto é tão ruim quando não executar a tarefa, ou seja, causa uma interrupção do serviço, percebida pelo usuário como uma falha grave. Em alguns casos, se o sistema falha em atuar dentro da janela de tempo previsa pode, inclusive, colocar em risco vidas humanas (é o caso de uma falha em um sistema de "*air bag*").

O projeto de sistemas embarcados de tempo real envolve o projeto conjunto de software e hardware, principalmente no contexto de sistemas desenvolvidos em um único circuito integrado (SOCs - *Systems On a Chip*), compostos de um ou mais processadores, memória, meios de comunicação e outros componentes (SCHULTZ, 2007), e este projeto é muito facilitado com o uso de ferramentas de desenvolvimento de software redirecionáveis, que permitam portar o mesmo sistema para diversas arquiteturas de processadores sem necessidade de recodificação ou adaptações específicas. Esta característica torna desejável que qualquer ferramenta que venha a ser utilizada no projeto/construção de sistemas de tempo real seja redirecionável para o maior número possível de plataformas/arquiteturas.

Um dos desafios do projeto de sistemas de tempo real é determinar se o sistema vai cumprir as restrições de tempo em uma determinada plataforma de hardware para qualquer que seja o conjunto de entradas ou a condição de carga do sistema. Para auxiliar os desenvolvedores de sistemas de tempo real a superar este desafio é que foram criadas as ferramentas de análise de tempo de execução.

A análise de tempo de execução, ou, mais especificamente, análise de tempo de execução no pior caso (WCET - *Worst Case Execution Time*) é uma área de pesquisa bastante ativa, que tem como meta o desenvolvimento de técnicas para estimar, de forma relativamente independente dos dados de entrada, qual o tempo máximo que cada componente ou processo de software leva

para executar, a fim de garantir, na fase de projeto de sistemas de tempo real, que o sistema possa obedecer às restrições de tempo impostas pela aplicação.

Segundo Petters, Zadarnowski e Heiser (2007), as duas principais abordagens para a análise do tempo de execução são o cálculo (análise estática) e a medição de tempo (análise dinâmica), e os autores sugerem uma tendência de abordagens híbridas, utilizando o melhor das duas técnicas.

Em Wilhelm et al. (2007), este problema é apresentado com detalhes, e são apresentadas as principais técnicas e ferramentas disponíveis para atacar este problema.

Este trabalho vai concentrar-se na análise estática do tempo de execução, que é, geralmente, dividida em duas fases: a análise de alto nível, dependente apenas de características do programa analisado, sem levar em conta particularidades da plataforma na qual o programa irá executar, e a análise de baixo nível, modelando o comportamento de recursos específicos da plataforma, como caches e pipelines.

Algumas técnicas de análise estática de tempo de execução estão intimamente ligadas aos compiladores, e, desta forma, a implementação de ferramentas de análise de tempo pode ser facilitada pelo uso de uma infra-estrutura de desenvolvimento de compiladores que favoreça a análise do programa como um todo (análise em tempo de ligação) e que seja facilmente redirecionável (ENGBLOM; ERMEDAHL; ALTENBERND, 1998). Esta característica nos levou à escolha da LLVM (LATTNER, 2002) como infra-estrutura para a implementação das técnicas de análise de tempo.

A LLVM é uma infra-estrutura para a criação de compiladores (estáticos e dinâmicos) que caracteriza-se pelo suporte a análise e transformações no código em todas as fases do processo de compilação, inclusive em tempo de execução, através de uma representação intermediária em SSA (Single Assignment Form – forma de representação mais utilizada para otimização de código (SINGER, 2005)) e do uso de informações de tipo de alto nível até o momento da geração final do código objeto.

A Importância dos Sistemas Embarcados de Tempo Real

Considerando que a maior parte dos processadores vendidos são relacionados com sistemas embarcados, e que a convergência tecnológica está levando aplicações tradicionais dos computadores pessoais para dispositivos móveis com conexão sem fio, a diversidade deste tipo de sistema deve aumentar ainda mais nos próximos anos.

O estudo de sistemas embarcados e sistemas de tempo real em conjunto é justificado pela afirmação de Engblom (2002b): “a maioria dos sistemas embarcados é de tempo real, e a maioria dos sistemas de tempo real é embarcado”, e pelo fato de que as aplicações usuais de computadores costumam priorizar o desempenho em detrimento da previsibilidade, dificultando a aplicação de sistemas de tempo real nestes ambientes.

Muitos sistemas embarcados são formados por vários processadores, cada um com uma função específica e executando seu próprio conjunto de programas. A medida que os processadores vão ficando mais baratos e mais poderosos, várias aplicações controladas por dispositivos mecânicos ou eletrônicos devem passar a ser controladas por sistemas embarcados. Um exemplo típico disto são os automóveis. Em um Mercedes Classe S existem mais de 60 processadores comunicando-se através de várias redes (ERMENDAHL, 2003).

Uma característica marcante dos sistemas embarcados é a enorme variedade de processadores utilizados. Pode-se encontrar sistemas embarcados com processadores de 4 bits com maior freqüência do que sistemas com processadores de 32 bits.

Processadores embarcados costumam ter um projeto mais simples e mais barato do que os processadores de uso geral. Outro requisito importante, principalmente relacionado a sistemas móveis, é o consumo de energia. Alguns recursos como caches são pouco utilizados neste contexto, pois aumentam o consumo do processador e incluem comportamentos de difícil previsão em relação a tempo e consumo de energia.

A maioria dos sistemas embarcados são programados em linguagens como C, C++ e Assembly, e as construções mais comuns diferem dos programas para computadores de uso geral (desktop e servidores), privilegiando variáveis estáticas e globais, com poucos dados alocados dinamicamente, e com maior enfoque em operações lógicas do que aritméticas (ENGBLOM, 1999).

Boa parte do código encontrado em programas embarcados, principalmente de tempo real, é gerado automaticamente por ferramentas de desenvolvimento, e apresenta estruturas atípicas em programas convencionais codificados manualmente, como uso freqüente da instrução “go to” provocando desvios de forma não estruturada no código. Esta característica, aliada às transformações realizadas pelos compiladores-otimizadores, dificulta a análise deste tipo de programa, motivando a criação de ferramentas automáticas de análise, como a proposta neste trabalho.

2 *Objetivos*

O presente trabalho está incluso no contexto da criação de ferramentas de desenvolvimento para sistemas de tempo real, com foco em sistemas embarcados de tempo real. Neste contexto, a análise de tempo ocupa um lugar importante na comunidade acadêmica, mas ainda não é largamente utilizada no meio industrial.

Em virtude da limitação de tempo para o desenvolvimento deste trabalho, não estamos propondo o desenvolvimento de uma ferramenta completa de análise de tempo, limitando-nos a analisar a viabilidade da construção de tal ferramenta.

2.1 Objetivo Geral

O objetivo deste trabalho é analisar a viabilidade, pontos fortes e pontos fracos do uso do framework *LLVM* para a construção de ferramentas redirecionáveis de análise de tempo de execução.

2.2 Objetivos Específicos

Para atingir o objetivo geral proposto, é necessário realizar uma série de passos intermediários, cada qual com seus próprios objetivos e resultados. Desta forma, este trabalho se propõe a:

- Realizar um estudo das principais técnicas de análise de tempo em uso por ferramentas comerciais e acadêmicas
- Realizar um estudo do framework *LLVM* e das suas potencialidades e forma de utilização e extensão

- Implementação de técnicas de análise do fluxo de controle do programa (primeiro passo da análise estática de tempo de execução), utilizando-se LLVM, e análise desta implementação do ponto de vista de viabilidade do uso desta infra-estrutura.
- Proposição de abordagens para análise de baixo nível redirecionável utilizando-se LLVM, sem, porém, realizar implementações destas análises (aspecto deixado para trabalhos futuros).

3 O Cálculo Estático do Tempo de Execução Para o Pior Caso

O objetivo da análise do tempo de execução é determinar dois valores de tempo: o tempo de execução no pior caso (WCET - *Worst Case Execution Time*) e o tempo de execução no melhor caso (BCET - *Best Case Execution Time*).

O tempo de execução no melhor caso é útil para escalonamento de tarefas em sistemas de tempo real, identificando qual o tempo mínimo que uma tarefa precisa para ser executada. Já o tempo de execução no pior caso é útil para determinar se, dada uma configuração de software e hardware, a tarefa é capaz de terminar dentro da janela de tempo que ela precisa ser executada. Neste trabalho, nos concentramos na análise de tempo de execução para o pior caso (WCET).

Uma característica importante dos sistemas de tempo real é que "*rápido o suficiente é rápido o suficiente*", ou seja, não é objetivo dos sistemas de tempo real realizar a tarefa no menor tempo possível, e sim realizar a tarefa dentro de uma janela de tempo, com o mínimo possível de consumo de recursos como processador, memória e energia elétrica (importantíssimo para sistemas embarcados em dispositivos móveis). Desta forma, o uso da análise de tempo pode ajudar a dimensionar o processador ou tipo de memória mínimo para atender às necessidades de tempo da aplicação.

Em um sistema de tempo real, é usual existirem tarefas que são de tempo real *crítico*, de tempo real *brando* ou sem restrições de tempo. Por isto, as ferramentas de análise geralmente precisam que seja indicado que partes do programa constituem tarefas de tempo real, ou, no caso mais comum, consideram que cada função do programa delimita uma tarefa de tempo real, e fazem a análise por função.

O WCET pode ser estimado por métodos dinâmicos (a partir de medições de tempo de execução do sistema real ou emulador, para conjuntos de dados de entrada) ou por métodos estáticos (sem execução do sistema real, com o cálculo realizado a partir de análises no código – fonte ou executável – da aplicação). Os dois tipos de métodos são utilizados na prática.

Porém, segundo Ermendahl (2003), há restrições para o uso dos métodos dinâmicos, sobretudo Para sistemas de tempo real “crítico”, conforme indicado na figura 3.1.

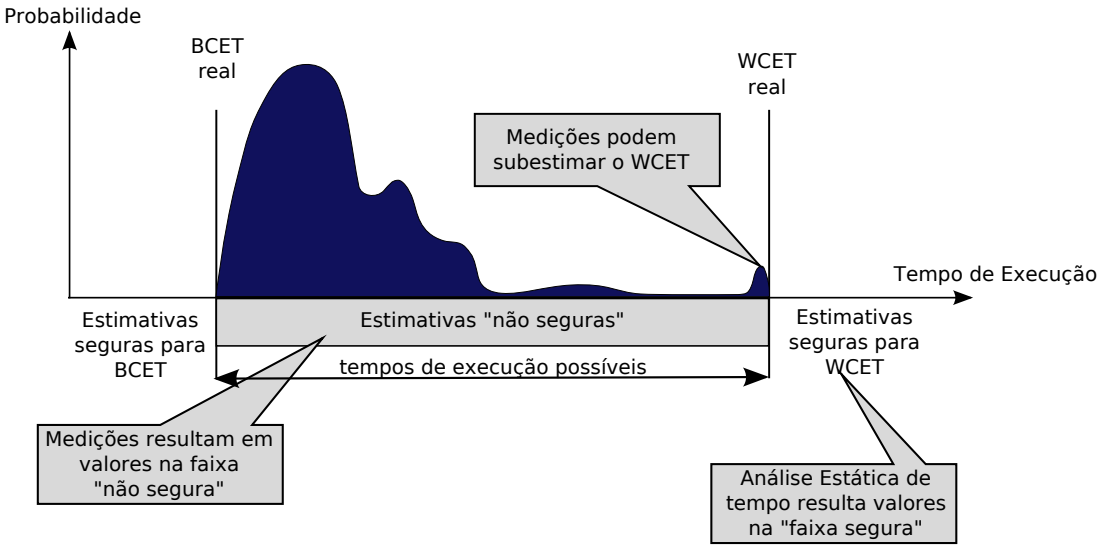


Figura 3.1: Estimativas de Tempo de Execução obtidas através de métodos estáticos e dinâmicos, adaptado de (ERMENDAHL, 2003)

Kirner e Pushner (2003) e Petters, Zadarnowski e Heiser (2007) fazem uma análise mais profunda desta questão, indicando possibilidades de uso de técnicas dinâmicas em sistemas de tempo real “brandos” e do uso da combinação das duas técnicas. Reconhecemos a importância e a aplicação das técnicas dinâmicas, porém, neste trabalho, ficaremos restritos a análise estática do tempo de execução.

Uma vez que determinar o tempo máximo de execução exato para o pior caso implica em identificar a priori este pior caso, ou, conforme apresentado por Engblom (1997), “*encontrar o máximo tempo de execução em que o programa termina*”, e que esta identificação só é possível para programas triviais[1], as ferramentas de análise estática de tempo de execução buscam fazer estimativas seguras dos tempos mínimo e máximo de execução, o mais próximo possível dos tempos mínimo (BCET) e máximo (WCET) reais. Desta forma, quando uma ferramenta deste tipo estima o tempo mínimo de execução, ele seguramente é menor que o tempo mínimo real, e, da mesma forma, o tempo máximo estimado é maior que o tempo máximo real, levando a um superdimensionamento dos sistemas projetados utilizando-se estas medidas como parâmetros.

Para realizar estas tarefas, as ferramentas de análise fazem as seguintes simplificações (ENGBLOM, 2002b):

[1] determinar se um programa arbitrário termina ou não já é um problema indecidível (ENGBLOM, 1997), encontrar o pior caso em que ele termina é ainda mais complexo

- O programa executa de forma isolada em um processador durante toda a execução da tarefa de tempo real.
- O programa está sendo executado em uma CPU específica (suportada pela ferramenta) a uma velocidade de clock específica.
- O programa foi compilado por um determinado compilador (suportado pela ferramenta).
- Não há atividades em segundo plano interferindo na atividade usada para o cálculo.
- Não há chaveamento de tarefas (multitasking).

Como as tarefas de tempo real geralmente correspondem a apenas uma parte do programa, e que o escalonamento de tarefas em sistemas de tempo real procura garantir que não haja chaveamentos de tarefas no meio de uma tarefa crítica, estas simplificações não estão longe da realidade deste tipo de sistema.

Ermendahl (2003) propõe a divisão da análise estática do tempo de execução em três componentes distintos: a análise do fluxo de controle do programa (análise de alto nível), a análise de baixo nível, levando em conta as particularidades do hardware, e o cálculo, juntando o resultado das duas outras tarefas.

3.1 A análise do Fluxo de Controle

A análise do fluxo de controle busca extrair do programa informações da sua estrutura, independente das características da máquina alvo, construindo um grafo orientado que represente todos os caminhos dentro do programa. Cada nodo deste grafo é um bloco básico e cada ramo aponta de um nodo para um bloco básico que pode ser executado na seqüência. Desta forma, em cada decisão (desvio condicional), um nodo pode levar a dois nodos, um para cada resultado da decisão.

No decorrer da construção deste grafo, a análise deve identificar o seguinte:

- Caminhos possíveis e impossíveis no fluxo de controle (*unfeasible paths*), a partir da análise dos valores possíveis das variáveis que controlam cada decisão.
- Quantidade máxima de iterações de cada loop do programa.
- Quantidade máxima de recursões para cada função recursiva.

Para ser efetiva, a análise do fluxo de controle deve ser global, ou seja, deve levar em conta todas as chamadas de função a partir de cada função analisada.

As técnicas propostas para realizar este tipo de análise têm como resultado estimativas para os valores de números de iterações de loops e recursões, mas, em função da entrada para o programa no pior caso ser desconhecida, geralmente não é possível determinar o valor exato para estes resultados. Em cada situação em que o conjunto de valores que uma variáveis que influencia em uma decisão do programa não pode ser determinado, decide-se por estimativas seguras, superestimando o tempo de execução. Algumas destas técnicas são apresentadas em Lokuciejewski et al. (2007).

Engblom (2002a) indica algumas abordagens para esta análise, como a execução simbólica do programa, a interpretação abstrata do código fonte ou simulação em nível de instrução para o código objeto, e todas estas abordagens procuram determinar duas coisas em relação ao programa:

Caminhos impossíveis: são enumerados todos os caminhos possíveis no programa, e, para cada decisão, são criados dois caminhos: um para a decisão verdadeira e outro para a decisão falsa. Ao longo da análise, todas as decisões que foram seguidas são lembradas, para determinação de contradições e identificação de caminhos impossíveis[2].

laços (loops e recursões): Para cada loop identificado, procura-se identificar qual a variável ou variáveis que controlam o loop e os possíveis valores que estas variáveis podem assumir, e procura-se identificar qual o número máximo de iterações deste loop.

O capítulo 4 faz um estudo mais aprofundado destes problemas e das técnicas de análise geralmente usadas para resolvê-lo.

3.2 Análise do comportamento do processador

A análise do comportamento do processador, ou análise de baixo nível, modela características específicas da arquitetura da máquina alvo, como, por exemplo, a influência de memória cache, pipelines e "*branch predictors*" no tempo de execução de cada bloco básico de instruções.

Conforme apresentado em Wilhelm et al. (2007), um processador típico é formado por vários componentes que podem influenciar no tempo de execução de uma instrução específica, o que torna este tempo dependente do contexto de execução da instrução.

[2] *Unfeasible Paths*

As ferramentas de análise utilizam modelos abstratos de processador para realizar esta tarefa, que procuram aproximar, de forma conservativa, o comportamento de tempo e a influência de componentes como memória, barramentos e periféricos, gerando estimativas de tempo garantidamente maiores do que o maior tempo possível no sistema real.

Para prever o comportamento do processador na execução de uma instrução específica, estas ferramentas precisam analisar esta instrução no contexto de todos os possíveis caminhos no fluxo de controle do programa em que esta instrução é executada, estimando um tempo de execução em cada um destes caminhos (WILHELM et al., 2007).

A complexidade da análise de baixo nível é proporcional à complexidade do processador a ser analisado. Processadores com recursos de melhoria de performance (pipelines, caches, *branch predictors*) tendem a apresentar um número maior de *Anomalias de Tempo* do que processadores mais simples.

As *Anomalias de Tempo*, conforme definidas por Wilhelm et al. (2007), são *"influências contra-intuitivas do tempo de execução (local) de uma instrução no tempo total de execução de uma tarefa"*, ou, em termos mais simples, são situações em que uma ferramenta de análise de tempo pode ser induzida a erros em virtude da complexidade do hardware e software que executa um sistema. Nem todos os dados de entrada estão disponíveis, e nem todos os contextos de execução de cada instrução são enumeráveis para serem simulados. Desta forma, quando calculamos o tempo de execução de uma instrução, assumimos um contexto de execução e um estado abstrato para o sistema como um todo (hardware e software), e este estado pode não refletir o estado real em que a instrução será executada, fazendo-nos estimar um tempo muito diferente do real. Exemplos destas anomalias podem ser encontrados em (WILHELM et al., 2007), e estão relacionados, principalmente, a incertezas na predição de comportamento dos recursos de melhoria de performance do processador.

3.3 O cálculo do WCET

A fase final da análise estática consiste em estimar o tempo máximo de execução para cada tarefa de tempo real do sistema. Este cálculo utiliza-se do resultado das duas fases anteriores (análise do fluxo de execução e análise do comportamento do processador) para produzir esta estimativa.

A literatura – (ERMENDAHL, 2003) (WILHELM et al., 2007) – apresenta três abordagens para a fase de cálculo:

- Métodos baseados em *análise da estrutura do programa*[3].
- Métodos baseados em *caminhos no grafo de fluxo de execução* do programa.
- Métodos baseados na técnica *IPET – Implicit Path Enumeration Technique* ou "técnica de enumeração de caminhos implícitos".

Nos métodos de cálculo baseados em análise da estrutura do programa, a árvore sintática do mesmo é percorrida de baixo para cima (*bottom-up*), combinando o tempo calculado para cada componente das instruções (*statements*) de acordo com regras pré-definidas, unindo vários nós da árvore sintática em um único nó. O processo é repetido até reduzir a árvore sintática toda para um único nó (a tarefa para a qual o tempo de execução está sendo estimado). Para avaliar todos os contextos de execução, são necessárias transformações na árvore sintática, refletindo diferentes contextos de iteração em loops, por exemplo.

Nos métodos baseados em caminhos no grafo de fluxo de controle, é construído um grafo com todos os caminhos possíveis na execução da tarefa, e este grafo é percorrido para determinar o maior caminho. O tempo estimado para este caminho é considerado a estimativa de WCET para a tarefa.

Na técnica IPET, o fluxo de controle e os tempos de execução são representados por um sistema lógico ou algébrico de restrições (*constraints*). Para cada bloco básico ou ramo ligando blocos básicos no grafo de fluxo de execução é atribuído um tempo de execução ($t_{entidade}$ – estimado a partir da análise do comportamento do processador) e um contador de iterações ($x_{entidade}$ – a quantidade máxima de vezes que aquele bloco básico ou ramo é executado, estimada a partir da análise de fluxo de controle), e é estabelecido um conjunto de restrições associadas a estes nodos e/ou ramos. O WCET é calculado maximizando-se a soma $\sum_{i \in entidades} x_i * t_i$.

As restrições aplicadas às entidades na técnica IPET podem ser resolvidas utilizando-se técnicas de *Programação Linear* ou *Programação por Restrições* (WILHELM et al., 2007).

[3]Tratados por Ermendahl (2003) como baseados em árvore – "*tree based*"

4 *Análise de Fluxo de Controle do Programa*

O propósito da análise de fluxo de controle do programa é obter todos os caminhos de execução possíveis no código do programa. O resultado desta análise consiste em um conjunto de informações compreendendo (ERMENDAHL, 2003):

- que funções são chamadas;
- quantas vezes cada laço (*loop*) é executado;
- que dependências existem entre as decisões e desvios(*if*) do programa.

Como já foi citado anteriormente neste trabalho, e conforme evidenciado por Ermendahl (2003), no caso geral, este problema é indecidível, e, portanto, efetua-se uma análise aproximada. Esta aproximação precisa dar resultados seguros (*safe*), ou seja, incluir todos os caminhos de execução possíveis dentro do programa, mas precisa, também, dar resultados com o menor desvio da realidade (*tight*), ou seja, incluir o menor número possível de caminhos impossíveis (*unfeasible paths*).

Ermendahl (2003) propõe uma abordagem de três fases para a análise do fluxo de controle do programa:

1. Extração da informação do fluxo de controle;
2. Representação dos resultados do fluxo de controle;
3. Conversão dos resultados para uma forma que possam ser utilizados pelo método de cálculo usado pela ferramenta.

4.1 Extração da informação do fluxo de controle

A extração da informação do fluxo de controle do programa consiste na análise do mesmo para a construção de um grafo de fluxo de controle, e da identificação, a partir deste grafo, de informações sobre o comportamento dinâmico do programa: número de vezes que cada loop executa, profundidade das recursões e caminhos possíveis e impossíveis no fluxo de execução do programa.

Wilhelm et al. (2007) e Ermendahl (2003) citam várias abordagens e técnicas para extração da informação do fluxo de controle de um programa, combinando análise automática com anotações manuais no código fonte do programa ou na representação intermediária.

4.1.1 Definições

Gustafsson, Bermudo e Sjöberg (2002) definem os seguintes grafos relacionados a análise de fluxo de controle:

Grafo de Fluxo de Controle (**CFG** – *Control Flow Graph*): definido para cada função do programa, com os nodos representando blocos básicos e o fluxo de controle sendo representado pelos ramos do grafo.

Grafo de Invocação (**IG** – *Invocation Graph*): definido para o programa todo, em que cada invocação de função (*call site*) é representada por um nodo, com um único ramo ligando-o uma única instância da função que é chamada. Cada instância de função representa uma cadeia de chamadas (*call string*), ou seja, representa o contexto da chamada de uma função, com toda a cadeia de chamadas desde o ponto de entrada do programa (função main) e um ambiente de chamada (conjunto de valores possíveis para valores globais).

Grafo de Fluxo de Controle Estendido (**ECFG** – *Extended Control Flow Graph*): formado pela combinação do grafo de invocação (IG) com o grafo de fluxo de controle (CFG) de cada instância de função invocada.

Escopo : um subconjunto do grafo de fluxo de controle estendido (ECFG).

Grafo de Escopos (**SG** – *Scope Graph*): um grafo sobreposto ao ECFG permitindo a navegação por todos os escopos de uma função.

Cadeia de Chamada (*call-string*): A cadeia de chamada de uma função `foo` é formada pela concatenação de todos os nomes das funções chamadas, desde a função main até a função `foo`.

4.1.2 Análise Automática de Fluxo de Controle

O nosso principal interesse neste trabalho são as técnicas de detecção automática do fluxo de controle, obtendo os seguintes resultados:

- grafo de fluxo de controle estendido (ECFG), com informações de
  - Grafo de Invocação (IG) de funções, incluindo chamadas indiretas através de ponteiros[1].
  - Grafo de fluxo de controle (CFG) para cada função
- caminhos possíveis e impossíveis (ou infactíveis) no grafo de fluxo de controle[2], eliminando do SG os caminhos não alcançáveis.
- identificação dos loops, com estimativa do número máximo de iterações de cada loop em cada escopo de execução.
- identificação das chamadas recursivas, com estimativa da profundidade máxima da recursão (número máximo de vezes que a recursão acontece).

Em Engblom et al. (2001), é apresentado o projeto de uma ferramenta modular de cálculo de tempo de execução, cujo módulo de análise de fluxo de controle do programa é detalhado em Gustafsson, Bermudo e Sjöberg (2002). Neste último, os principais problemas, limitações e técnicas para análise de fluxo de controle com o objetivo de cálculo de tempo de execução são apresentados.

A implementação realizada por Gustafsson, Bermudo e Sjöberg (2002) está inserida no processo de compilação do programa, e boa parte dos algoritmos são definidos para operar sobre uma representação intermediária do código, que é transformada para assumir a forma SSA[3].

A partir do código representado na forma SSA, Gustafsson, Bermudo e Sjöberg (2002) definem algoritmos para construir o grafo de fluxo de controle estendido (ECFG) e o grafo de escopos (SG), e, a partir destes, através de técnicas de reconhecimento de padrões e interpretação abstrata do código, anotar o grafo de escopos com fatos relativos ao fluxo de controle (*flow-facts* – número de iterações de loops – e restrições – constraints – para execução de cada bloco

[1] Esta é uma limitação no trabalho de Gustafsson, Bermudo e Sjöberg (2002) para a qual proporemos uma solução parcial neste trabalho

[2] *unfeasible paths*

[3] SSA – *Single Static Assignment*: A representação SSA obriga que cada registrador virtual seja definido apenas uma vez (ou seja, tenha seu valor atribuído apenas uma única vez) e que esta atribuição domine todos os usos deste registrador. (SINGER, 2005)

básico). Estes fatos serão utilizados futuramente pelo método de cálculo para obtenção do WCET.

4.1.3 Identificação de Escopos

Conforme apresentado em Gustafsson, Bermudo e Sjöberg (2002), os escopos para análise de tempo podem ter uma das seguintes formas:

- Um conjunto de funções, em casos de funções recursivas (um componente fortemente conectado — SCC – *Strongly Connected Component*[4] — no grafo de invocação);
- Uma função;
- Uma parte de uma função, quando o fluxo de controle da função apresenta mais de um caminho possível:
  - loops;
  - trechos irredutíveis do grafo de fluxo de controle.

Desta forma, um escopo pode conter diversos escopos, e a mesma função pode aparecer em vários escopos, dependendo do contexto de invocação.

Uma vez que a identificação dos escopos é dependente do contexto de invocação, o grafo de escopos pode ser visualizado como:

- Um grafo de escopo (SG) completo, mostrando simultaneamente o SC e o ECFG.
- Um grafo de escopo simplificado, mostrando apenas o SG, mas não o ECFG ou apenas uma parte do ECFG.
- Uma visão hierárquica do SG, mostrando apenas a hierarquia de escopos.

Os algoritmos para identificação de escopos percorrem o ECFG, avaliando as seguintes situações:

Funções não recursivas: A cada função não recursiva corresponde um escopo, pois podem ser identificados fatos relacionados à função que são independentes dos vários contextos de invocação.

[4] Definido em Ramalingam (2002)

Se uma função é invocada em vários contextos diferentes, seu escopo vai ser duplicado no grafo de escopos para cada invocação, fazendo com que o SG de um programa não recursivo seja uma árvore e todas as strings de invocação sejam finitas.

Chamadas Recursivas: Em situações de recursão, a string de invocação pode crescer indefinidamente. Em função disto, as chamadas recursivas devem ser identificadas e tratadas de um modo especial.

As recursões são identificadas no IG como componentes SCC. Desta fora, define-se um escopo para cada SCC no IG, e este escopo é analisado de forma semelhante a um loop, e a profundidade da recursão (número de vezes que o trecho recursivo é repetido) é calculada.

Análise de escopos dentro de funções: Em um contexto de programas escritos em linguagens que permitem códigos não estruturados, é possível encontrar casos em que o CFG de uma função é não redutível[5].

Quando o programa é estruturado, e o CFG é redutível, cada loop natural é um escopo dentro do escopo da função. No caso de loops aninhados, os escopos também estão aninhados.

Em casos em que o CFG é não redutível, são necessárias análises adicionais para identificar loops em código não estruturado, e Gustafsson, Bermudo e Sjöberg (2002) sugere o uso da técnica DJ (*Dominator/Join-graphs*), definida em Sreedhar, Gao e Lee (1996) para identificação destes loops e associação de escopos a eles.

4.1.4 Fatos do ao fluxo de controle (flow-facts)

Reconhecimento de Padrões

Gustafsson, Bermudo e Sjöberg (2002) apresentam, como método mais geral para analisar o número de iterações dos loops é a interpretação abstrata. Este método, porém, tem um custo (*tempo*) elevado, e estes autores propõem uma técnica para acelerar esta análise, a partir da identificação de construções usuais nos loops do programa.

Uma vez identificado um dos padrões conhecidos pelo programa de análise, este é utilizado para determinar o número de iterações do loop, eliminando este loop da análise por interpretação abstrata.

[5]Os algoritmos para identificação se um CFG é redutível ou não e para identificação dos loops naturais é descrito em Aho, Sethi e Ullman (1988)

A identificação de padrões de construção dos loops, leva em conta a forma da expressão de controle, dos incrementos nas variáveis de controle e nas possíveis condições de saída do loop, e é capaz de reconhecer os loops estruturalmente mais simples, descrevendo-os através do que os autores chamaram de “equações de recorrência”: equações que descrevem o comportamento da(s) variável(is) de controle do loop ao longo das iterações. Estas equações são resolvidas, a priori, para os loops mais comuns (padrões), e, cada vez que um loop destes é encontrado no código, a solução da equação é aplicada aos parâmetros do loop, gerando as anotações para o escopo do loop, e, em seguida, o loop é substituído por um trecho de código que, do ponto de vista das variáveis de controle do loop e *variáveis de indução*[6] é equivalente à execução do loop no pior caso.

Com esta técnica, analisando-se do loop mais interno para o mais externo, é possível, em alguns casos, eliminar completamente os loops do restante da análise, otimizando o tempo de análise para a interpretação abstrata.

Gustafsson, Bermudo e Sjöberg (2002) apresenta o algoritmo da figura 4.1 para analisar os loops em uma função.

```
Remover todas as variáveis que não influenciam no fluxo de controle.

Para cada loop, do mais interno para o mais externo, faça:
   se o resto das variáveis são variáveis de indução
    e o loop pode ser expresso em uma forma padrão
    e existe uma solução em forma fechada para este padrão
       então calcule os fatos do fluxo de controle
             calcule o valor final das variáveis
             substitua o loop por uma construção mais simples

Para o programa reduzido, remover as variáveis que
não influenciam no fluxo de controle.
```

Figura 4.1: Algoritmo para análise de loops em uma função

Através deste algoritmo, é possível identificar, para os loops que correspondem aos padrões reconhecidos pela ferramenta, os números mínimo e máximo de iterações.

Interpretação Abstrata

Algumas construções possíveis na maioria das linguagens de programação pode levar a loops que não sejam facilmente reconhecíveis, ou para os quais não seja possível determinar, a priori, uma forma fechada para o uso de reconhecimento de padrões.

A abordagem usual para tratar destes casos é tentar simular a execução do código para uma faixa de valores possíveis de entrada, para determinar, simultaneamente, o pior caso de

[6] induction variables

execução do programa (o caminho executado no pior caso), como o número de iterações nos loops neste caminho.

Para fins de estimativas de tempo, não é necessário simular completamente a arquitetura alvo. Ao invés disto, cria-se um modelo abstrato do processador alvo, com as operações definidas de forma conservadora (ou seja, simulando a performance no pior caso).

Conforme Engblom (1997), a interpretação abstrata aproxima a semântica de uma linguagem de programação, geralmente representando valores como conjuntos de valores ou intervalos.

Ermedahl e Gustafsson (1997) apresentam esta técnica com detalhes e um exemplo da aplicação, e discutem algumas das suas vantagens e desvantagens.

4.2 O cálculo do WCET

Os resultados da análise do fluxo de controle (também chamada, por alguns autores, de análise de alto nível) servem de entrada para o cálculo do tempo de execução para o pior caso.

Conforme apresentado em Wilhelm et al. (2007), as principais técnicas de cálculo utilizadas em ferramentas de cálculo estático de tempo de execução podem ser divididas em técnicas *baseadas em estrutura* do programa, *baseadas em caminhos* de execução do programa, e técnicas de *enumeração implícita de caminhos — IPET – Implicit Path Enumeration Technique*.

A técnica IPET, apresentada originalmente em Li e Malik (1995), consiste em associar restrições a cada nodo do ECFG, e posterior interpretação destas restrições para cálculo do tempo. Desta forma, são percorridos todos os caminhos no ECFG sem, entretendo, que estes sejam completamente enumerados.

Para uso desta técnica, a análise de fluxo de controle deve criar as restrições como anotações no ECFG, e estas anotações serão interpretadas durante o processo de cálculo, juntamente com a simulação do comportamento do processador (análise de baixo nível).

5 *LLVM - Low Level Virtual Machine*

Conforme apresentado em Engblom, Ermedahl e Altenbernd (1998), uma das dificuldades para o cálculo do tempo de execução está relacionada com o fato de que a análise do comportamento do programa é facilitada quando se utiliza o código fonte como representação do mesmo, que a análise da influência do hardware no tempo de execução é facilitada quando a esta é realizada no código executável (objeto) diretamente. Assim, para cada fase a análise existe uma representação mais adequada. Porém, com o uso de compiladores-otimizadores, são realizadas transformações no código durante a compilação, dificultando o mapeamento direto das estruturas do código objeto nas estruturas que os originaram no código fonte.

Esta situação nos levou a propor uma ferramenta de cálculo de tempo integrada a um compilador, idéia apresentada no artigo citado. Avaliamos, para esta tarefa, o GCC – *GNU Compiler Collection* (Free Software Fondation, 2007) e o LLVM – *Low Level Virtual Machine* (LATTNER, 2008), e optamos pelo segundo pelas seguintes razões:

- Modularização do compilador, facilitando a implementação de novas ferramentas baseadas nele;
- Representação intermediária do código em um nível mais alto, facilitando a análise de alto nível;
- Framework de transformação e análise que permite identificar e selecionar explicitamente as otimizações realizadas no código;
- Comunidade ativa de desenvolvedores, provendo documentação e suporte ao desenvolvimento de uma nova ferramenta;
- Separação entre otimizador e gerador de código, facilitando a geração de ferramentas redirecionáveis.

Neste capítulo apresentamos esta ferramenta com detalhes, e indicamos como ela pode ser utilizada para o desenvolvimento de uma ferramenta de análise de tempo de execução.

5.1 Descrição e Metas da LLVM

A LLVM pode ser descrita como um *framework para construção de compiladores*, ou uma *linguagem para representação intermediária de código*, com uma *máquina virtual* que interpreta esta linguagem e um *just-in-time compiler* implementado nesta máquina virtual, ou ainda um compilador completo (com um front-end experimental para a linguagem C/C++, e uma adaptação do GCC para atuar como front-end).

A motivação para a criação da LLVM foi o desenvolvimento de uma infra-estutura de compilação capaz de favorecer o processo de de análise e transformação no código, no intuito de gerar o melhor código possível [1].

Entre os critérios utilizados por Lattner (2002) ao propor a LLVM está a constatação de que a evolução das linguagens de programação (com maior carga semântica associada às instruções e operadores) aliada aos novos recursos presentes nas arquiteturas de processadores (pipelines, caches multinível, executores especulativos, etc.) aumentou a complexidade da geração do código com o melhor desempenho possível, e que a arquitetura do compilador deve ajudar o projetista de compiladores a enfrentar esta complexidade.

Em Lattner (2002), o autor apresenta as estratégias geralmente adotadas pelos compiladores para atacar o problema de geração do código com a melhor performance possível. Nesta análise, ele constata que as técnicas são semelhantes para otimizações estáticas e dinâmicas, e evidencia como principal desafio a *otimização global*, entendida, neste contexto como a otimização a partir da análise do programa todo, ou *análise inter-procedural*.

Constatamos que um dos problemas principais da análise do tempo de execução é a análise inter-procedural para determinação dos caminhos factíveis no grafo de execução do programa, e que esta análise é, em muitos aspectos, semelhante a análise realizada pelos compiladores para otimizações inter-procedurais.

Lattner (2002) constatou que a estratégia de análise está fortemente vinculada com a representação intermediária utilizada pelo compilador na execução da mesma. Em representações de alto nível (como *AST – Abstract Syntax Tree*), o compilador dispõe de um conjunto importante de informações que permitem fazer análises inter-procedurais agressivas, mas várias otimizações pontuais não podem ser realizadas enquanto não for gerado o código para a arquitetura alvo, o que só ocorre no final do processo de ligação. Em casos de compiladores *Just in Time*, as transformações na estrutura do programa e a própria análise podem exigir um grande número de ciclos de processamento, penalizando a própria aplicação, e, em compiladores off-

[1] O problema de gerar código ótimo é NP-Completo (AHO; SETHI; ULLMAN, 1988)

line (estáticos), a performance do processo de compilação pode ser crítica. Por outro lado, em compiladores que trabalham com representações de nível mais baixo, as otimizações locais são favorecidas e podem acontecer mais cedo, porém, eventualmente, não há informação suficiente para alguns tipos de análise de alto nível necessária para otimizações inter-procedurais.

Nestas circunstâncias, ele propõe a LLVM (*Low Level Virtual Machine*), com as seguintes metas:

- Permitir otimização agressiva em múltiplos estágios do processo de compilação.
- Servir de *host* (no sentido de hospedeiro ou infra-estrutura) para pesquisas em compiladores e otimização de código.
- Operar de forma transparente para o usuário, como um "drop-in replacement" (substituição direta) para ferramentas tradicionais. Particularmente, a ferramenta visada pelo autor é o *Gnu Compiler Collection - GCC* (Free Software Fondation, 2007).

5.2 Arquitetura da LLVM

Para ser utilizável na prática, a LLVM precisou ser implementada de forma a permitir uma substituição com baixo ou nenhum impacto de ferramentas usuais de desenvolvimento. Desta forma, o processo de compilação com a LLVM fica ilustrado na Figura 5.1: Ao invés de gerar diretamente código nativo, o front-end gera código intermediário LLVM (*LLVM Virtual Instruction Set*) como código objeto. Algumas otimizações já são realizadas neste processo. O código intermediário é então ligado com bibliotecas LLVM (mesma representação) e com bibliotecas nativas (as otimizações do LLVM não alteram bibliotecas nativas, mesmo em tempo de execução). O executável gerado contém, além do código nativo (gerado para a arquitetura alvo durante o processo de ligação) otimizado, uma representação comprimida do código intermediário, que pode ser utilizado para otimizações on-line (baseada em medições ou *profiling* em tempo de execução) ou off-line (no caso de transformações que exijam maior quantidade de recursos - tempo de processamento e memória).

O que Lattner não chega a explicitar na sua dissertação, mas que pode ser considerado um aspecto importante da arquitetura do compilador desenvolvido com LLVM é a sua organização em múltiplos *PASSOS*. Cada otimização é implementada como um passo, com um escopo definido: alguns atuam apenas em um bloco básico, outros em uma função, outros em um módulo (correspondente a um arquivo fonte), e outros ainda no programa todo. Como todos os passos

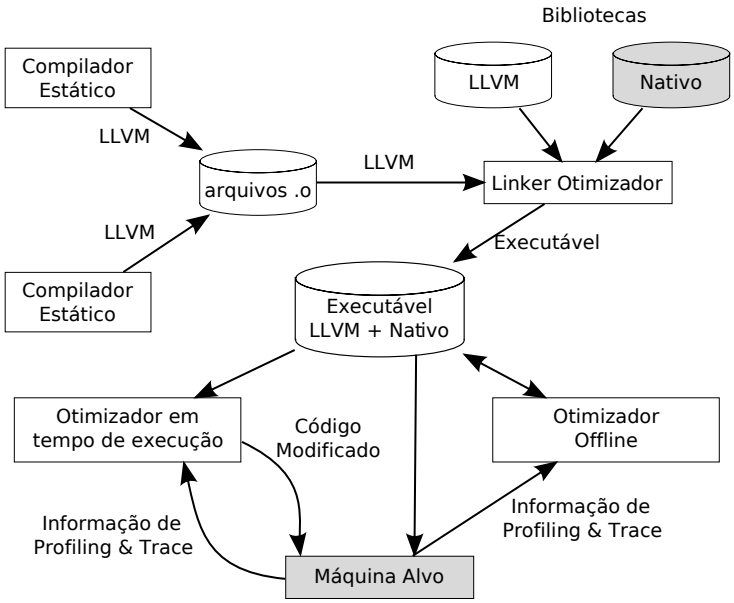


Figura 5.1: Arquitetura do compilador usando LLVM, adaptado de Lattner (2002)

têm uma interface bem definida, a reutilização de passos ou do resultado de análises é simplificada, permitindo, por exemplo, executar uma otimização como "*dead code elimination*" (eliminação de código inalcançável no programa) global em um módulo, e depois reutilizá-la, em tempo de ligação, no programa todo.

5.2.1 Front-end

O front-end de um compilador LLVM tem como responsabilidade a transformação do programa da linguagem fonte para a representação intermediária (*LLVM Virtual Instruction Set*). Como a representação intermediária é independente de linguagem, é possível a criação de vários front-ends que compartilham o linker e as estruturas e passos de análise e otimização.

A representação intermediária da LLVM não têm construções de mais alto nível, como modelo de objetos ou passagem de parâmetros por referência. Desta forma, o front-end tem que mapear as instruções de alto nível da linguagem em instruções de mais baixo nível da representação intermediária. O mesmo vale para as estruturas e tipos de dados, que devem ser mapeados no conjunto de tipos de dados definidos pela LLVM.

É responsabilidade do front-end na LLVM a execução do máximo possível de tarefas de análise e otimização já na primeira fase da compilação, para otimizar o tempo do linker.

5.2.2 Linker e Otimizador Inter-procedural

A ligação é o primeiro momento em que a maior parte (ou quase todo) do programa está disponível para análise pelo compilador. Desta forma, a maior parte das análises e transformações globais são executadas nesta fase.

Esta fase permite análises e transformações mais agressivas, e todas as transformações operam diretamente na representação intermediária, fazendo com que elas possam ser combinadas em qualquer ordem ou até reexecutadas para corrigir possíveis efeitos colaterais de outra transformação.

A geração de código é o último passo do linker, e pode conter sub-passos de otimização específicos da arquitetura alvo.

Como a LLVM mantém a representação intermediária até o último momento possível, ela é facilmente redirecionável para novas arquiteturas, pois as atividades de geração de código e as otimizações dependentes de arquitetura são agrupadas em módulos com interfaces bem definidas e estáveis.

5.2.3 *Profiling* e Otimizações em Tempo de Execução

A estratégia de reotimização em tempo de execução da LLVM é possível graças ao armazenamento, no executável, da representação intermediária do programa.

Um programa instrumentalizado para coletar informações de desempenho durante a execução detecta os pontos do programa em que este utiliza a maior parte do tempo de execução, em uso real[2], e o suporte de execução do programa pode realizar otimizações diretamente no código executável, ou, em transformações cujo escopo vai além de algumas instruções, se referenciar à representação intermediária armazenada no executável para realizar as transformações e regerar o código. Este comportamento permite ao programa adaptar-se a mudanças no padrão de uso: a cada vez que o usuário muda o seu comportamento, o programa detecta os novos “hot-spots” e reotimiza para obter a melhor performance.

Nem todas as transformações possíveis podem ser executadas sem um impacto considerável à performance da aplicação. Por isto, é possível ao suporte de execução da LLVM serializar as

[2]Este é um dos aspectos que diferencia o profiling na LLVM: ao invés de instrumentalizar uma versão de desenvolvimento do programa para coletar informações de execução e simular a execução, os dados são coletados em campo, pela versão de produção, e a reotimização pode acontecer durante a execução do programa, sem necessidade de voltar ao processo de link. Este tipo de comportamento também é alcançado por máquinas virtuais de alto nível, como Java e CLR, mas, no caso destas, não é possível rodar otimizações off-line para as transformações que exigirem mais processador e/ou memória

informações de profiling para um arquivo em disco, e um otimizador off-line lê, do executável, a representação intermediária, e executa as transformações, guiado pelas informações de profiling, gerando um novo executável em substituição daquele que foi lido.

5.3 LLVM Virtual Instruction Set

O conjunto virtual de instruções é o núcleo da arquitetura da LLVM, e foi projetado para ser "baixo nível" (próximo à máquina) o suficiente para permitir o uso das principais técnicas de otimização disponíveis na literatura e nos compiladores convencionais, mas, da mesma forma, ser "alto nível" o suficiente para permitir otimizações globais mais agressivas usando a mesma representação.

Este conjunto de instruções captura operações comuns à maioria dos processadores, evitando restrições comuns de máquinas, tais como a quantidade e tipo dos registradores, o uso de pipelines e memória cache, e as convenções de chamadas de procedimento específicas de cada arquitetura. É responsabilidade do gerador de código (no processo de ligação) adaptar a representação intermediária às características da máquina alvo, gerando código nativo para esta máquina.

A LLVM Virtual Instrucion Set (LVIS) utiliza um conjunto infinito de registradores virtuais, usados na forma de *Single Static Assignment - SSA*. Esta forma é usual para os algoritmos de análise de fluxo de dados. Cada instrução que computa um valor, cria um novo registrador para armazenar este valor. Para garantir que a forma SSA seja mantida mesmo em instruções de seleção (desvios condicionais), a LVIS define a instrução `phi`, que tem a semântica da função Φ da representação SSA (para mais detalhes da semântica da função Φ, ver (SINGER, 2005)). A representação SSA é restrita aos registradores, ou seja, aos escalares (valores e ponteiros) manipulados pelo programa. O conteúdo da memória não segue às restrições desta representação.

O LVIS tem um sistema rígido de tipos: todos os valores têm um tipo definido e as instruções têm regras rígidas para operar com os diversos tipos de dados. Conversões são realizadas apenas via a instrução `cast`. Os tipos de dados suportados pela LLVM são:

- tipos primitivos, como inteiros (com ou sem sinal, de 8 a 64 bits[3]), reais (precisão simples ou dupla), booleano, void e opaque (sem tipo definido, usado para passagem de parâmetros).

[3]Estes valores foram mantidos por serem os referenciados na dissertação de Lattner. A versão atual da LLVM suporta inteiros de precisão arbitrária

- tipos construtivos (arrays, estruturas, ponteiros e funções).

A maioria das instruções do LVIS é polimórfica: tem sua semântica definida a partir do tipo de dado em que opera. Por exemplo, a instrução `add` representa a adição inteira quando seus parâmetros são inteiros, e a adição em ponto flutuante quando seus parâmetros são números reais. Porém, é impossível misturar os tipos. Se for necessário adicionar um inteiro a um real, um dos dois vai ter que ser convertido com a instrução `cast`.

O acesso a memória é feito apenas através de duas instruções: `load` para carregar um valor da memória para um registrador virtual e `store` para armazenar o conteúdo de um registrador virtual em uma locação de memória.

O acesso a componentes de tipos construtivos é feito através de ponteiro para a locação de memória do tipo construtivo. O endereço do elemento específico é calculado com a instrução `getelementptr`.

A área de memória é dividida em três partes: dados globais (incluindo código), heap e pilha. A memória do heap é alocada com uma chamada à função `malloc`, e precisa ser liberada explicitamente com a função `free`. A memória da pilha é alocada com uma chamada à função `alloca`, e é liberada automaticamente quando o programa sai do escopo da função. Toda memória usada pelo programa deve ser alocada diretamente via `alloca` ou `malloc`.

O tratamento de exceções é suportado pelo LVIS em um esquema de "ZERO COST Exception Handling", em que o programa não executa nenhuma instrução adicional se não houver exceção. Este processo é realizado através de uma chamada específica de função e de uma instrução de retorno específica: Se a função for chamada com `call`, ela não pode lançar exceções, e se ela for chamada com `invoke`, é passado um parâmetro adicional para o `invoke` que é o endereço de retorno no caso de ocorrer exceção. O suporte de execução da LLVM mantém uma tabela destes endereços em memória e realiza os desvios quando ocorre um retorno de exceção de alguma função.

O LVIS tem três representações equivalentes entre si: uma em texto legível, utilizada para depuração e inspeção do código gerado pelo front-end, uma interna em memória, acessada através de uma API específica, e uma compactada[4] (bytecode), que é armazenada no executável, junto com o código gerado para a arquitetura alvo, para possibilitar otimizações em tempo de execução.

[4]Esta representação compactada foi modificada na versão 2.0 da LLVM, e é agora chamada de bitcode. Esta modificação foi feita para permitir um armazenamento mais compacto e um formato mais extensível, suportando os números inteiros de precisão arbitrária.

5.4 Otimizações e Análises Implementadas na LLVM

Lattner descreve um conjunto de otimizações implementadas por ele na LLVM, e alguns trabalhos independentes, de outros pesquisadores, utilizando a LLVM como infra-estrutura. A lista de otimizações apresentada por ele está desatualizada, pois há um grupo bastante ativo de pesquisadores utilizando a LLVM como infra-estrutura para realizar seu trabalho.

Dentre as transformações citadas, merece destaque o passo de *Análise de Estruturas de Dados*, não só pela sua aplicação no cálculo do tempo de execução do programa, motivo da escolha desta ferramenta, mas também pela eficácia com que foi implementado, sendo utilizado em várias otimizações globais implementadas na LLVM.

Este passo realiza uma análise extensiva do heap, sensível ao contexto[5]. Esta análise tem como resultado a indicação, para cada locação de memória, das funções, registradores e instruções que podem acessar esta locação ("may point to"), representada através de um grafo de estrutura de dados, que é criado para cada função do programa. Cada escalar no programa que pode representar um ponteiro é mapeado para um nodo do grafo, e cada locação de memória é também mapeada para um nodo, e os ramos indicam que nodo pode apontar para outro nodo na memória.

Como subproduto desta análise, é criado o grafo de chamada de funções, e, para cada função é computada uma lista de "call sites", ou seja, instruções que invocam esta função, através de um algoritmo de identificação de componentes fortemente conectados (*SCC - Strongly Connected Components*).

5.5 Projetos utilizando LLVM

Do ponto de vista do usuário, a LLVM pode ser vista de várias formas diferentes:

- Um compilador, utilizando o gcc como front-end e com recursos de otimização configuráveis;
- Uma infra-estrutura para construção de compiladores, permitindo o desenvolvimento de novos front-ends, back-ends ou passos de otimização;
- Uma infra-estrutura para construção de ferramentas de análise de código.

[5] Os dados são analisados com base no fluxo de controle da aplicação

Para os dois últimos pontos de vista, existem duas abordagens possíveis: realizar alterações na LLVM para suportar novos recursos ou utilizar a LLVM como uma API e implementar os novos recursos como projetos separados.

Sempre que possível, é preferível a segunda abordagem, menos intrusiva, fazendo com que a API e as estruturas de dados da LLVM, utilizada por vários projetos, permaneça estável. Mesmo ferramentas internas, e que fazem parte da API da LLVM, são implementadas desta forma (passos de otimização, ferramenta `opt`, *just-in-time compiler*).

A documentação da LLVM recomenda a organização dos projetos em duas partes: uma biblioteca com todas as funcionalidades na forma de uma API e um *driver* para realizar a interface com o usuário ou com outras aplicações em linha de comando. Desta forma, a funcionalidade criada para uma ferramenta pode ser reaproveitada por outras. Nesta documentação (LATTNER, 2008) é sugerida uma estrutura de diretórios e de `makefiles` para que o projeto possa se aproveitar dos parâmetros configurados pelo usuário quando este compilou a LLVM.

Um driver típico de um projeto LLVM tem a seguinte estrutura:

Suporte a opções de linha de comando: através da instanciação de objetos globais da classe `llvm::cl::opt`. Com isto, o programa ganha suporte às opções usuais das ferramentas LLVM, como "`-statistic`" (calcula estatísticas de uso de cada passo de análise ou otimização).

Criação dos módulos referentes à entrada: a API da LLVM disponibiliza uma classe chamada `BitCodeReader` para leitura de arquivos da forma binária da representação LVIS. Se a ferramenta trabalha com transformações ou análise de código, é usual fazer a leitura do arquivo através desta classe. Por outro lado, se a ferramenta é um fron-end para uma nova linguagem, então a leitura da entrada é feito pelo parser da própria ferramenta. para criação do módulo.

Criação de um `PassManager`: objeto que vai coordenar as dependências entre passos e escalonar a execução dos mesmos.

Criação dos passos necessários: os passos definidos para realizar a tarefa proposta pela ferramenta. A LLVM não distingue passos definidos pela API e passos definidos pelo usuário, e as dependências entre passos são extraídas automaticamente da estrutura dos próprios passos.

Registro dos passos no `PassManager`: através de chamadas ao método `addPass`. O PassManager vai procurar manter a ordem dos passos especificada pelo usuário, mesmo que,

para isto, precise duplicar algum passo (no caso do usuário especificar passos em ordem inversa à dependência entre eles, e o passo que for executado primeiro invalidar os dados do passo que foi especificado pelo usuário como passo seguinte).

Execução dos passos do PassManager: para cada módulo, com a chamada do méotodo de execução da classe PassManager (`PassManager.run(Módulo)`).

O trabalho de análise ou transformação de código propriamente dito é realizado pelos diversos *passos*, que são objetos que atuam sobre trechos específicos de código. Cada passo é executado de forma independente dos outros, a não ser que o próprio passo expresse que há uma dependência, ou seja, que o passo utiliza resultados de análise de um passo anterior. Neste caso, esta dependência é respeitada na execução.

Os passos possíveis são derivados de:

ModulePass: Um passo que analisa ou transforma um módulo todo de cada vez. Um módulo é, geralmente, correspondente a um arquivo fonte do programa. Este passo é recomendado para análises e transformações inter-procedurais, e não tem restrições em relação ao que pode fazer ao programa.

O método principal deste passo, que deve ser sobrescrito pelo passo definido pelo usuário, é `runOnModule`, e é chamado para cada módulo do programa. A mesma instância de ModulePass é utilizada para analisar todos os módulos, podendo manter informações sobre todo o programa. O *linker* da LLVM, por exemplo, é implementado como um destes passos.

CallGraphSCCPass: Passo que é executado para cada SCC (*Strongly Connected Component*) no grafo de chamadas de funções do programa. Através deste passo, pode-se facilmente detectar recursividade indireta. É o segundo passo mais abrangente, e pode manter informações sobre um trecho do código quando está analisando outro, ou seja, tem a visão global interprocedural do programa.

O método principal deste passo e o `runOnSCC`, e é chamado para cada componente SCC do IG, tendo como parâmetro a lista de funções que fazem parte deste SCC. Um passo deste tipo pode analisar ou transformar funções que estão neste SCC ou funções que são chamadas por estas.

A ordem de execução dos passos é a ordem inversa do grafo de invocação (ou seja, primeiro são tratadas as funções que não invocam outras, depois, as que invocam estas, até a raiz do grafo de invocação — geralmente, a função `main`).

FunctionPass: Executado para cada função do programa. O usuário deverá sobrescrever o método `runOnFunction`, que será chamado para cada função. Durante a análise, um passo deste tipo só pode alterar a função que está analisando, mas pode inspecionar ou utilizar resultados de análise das funções chamadas a partir dele.

Análises de função estão organizadas para poderem, em uma versão futura da LLVM, serem executadas em paralelo em máquinas multiprocessadas. Por isto, este tipo de passo deve considerar que funções diferentes podem ser interpretadas por instâncias diferentes do método, e que não será possível reutilizar resultados calculados por outras execuções do método *runOnFunction*. Todas as análises ou transformações são restritas à função sendo analisada.

LoopPass: Executado para cada loop identificado pela LLVM no programa. A identificação automática de loops é realizada pelo passo "Natural Loop Constructor".

O médodo de análise de loops é `runOnLoop`, e é chamado do loop mais interno para o mais externo. Da mesma forma que os passos derivados de `FunctionPass`, o escopo de análise é restrito ao loop sendo analisado. Um `LoopPass` só pode alterar o loop atual e os loops internos a este.

Basic Block Pass: Executado para cada bloco básico do programa. É o passo mais restrito, pois o método `runOnBasicBlock` só pode alterar o bloco atual e só pode inspecionar o bloco atual e seus sucessores e antecessor diretos.

Este tipo de passo costuma ser utilizado para otimizações tipo *peep-hole*.

Machine Function Pass: Executado pelo gerador de código, a cada função, após o passo de seleção de instruções. Permite ao gerador de código fazer otimizações específicas para a arquitetura alvo, com acesso às instruções definidas para esta arquitetura e depois da alocação de registradores.

5.6 Estratégias para uso da LLVM em Cálculo de Tempo de Execução

Em virtude da sua organização modular, e dos diversos passos de análise já disponíveis, a LLVM oferece recursos para facilitar a resolução do problema de cálculo de tempo de execução nos seus vários aspectos: a análise de loops, a análise de chamadas recursivas, a análise de caminhos não realizáveis (*unfeasible paths analysis*), e, com algumas alterações na fase de geração de código, a análise do efeito de caches e pipelines.

5.7 Análise de Loops

Quando a LLVM executa um `LoopPass` para uma função, o CFG para esta função já foi previamente computado, e os blocos básicos estão organizados de acordo com este grafo (ou seja, cada bloco básico conhece seu precedessor e seus sucessores). Com isto, é possível identificar, na instrução final de cada bloco básico, para que bloco o fluxo de controle pode ser desviado e qual a condição deste desvio.

A representação SSA facilita a análise das condições de controle dos loops e a análise de estruturas de dados permite avaliar se alguma função chamada de dentro do loop pode alterar alguma locação de memória cujo valor faz parte da expressão de controle do loop. Com isto, identifica-se os loops "não limitados", e é possível solicitar ao programador parâmetros para limitar estes loops ou acusar erro no cálculo.

5.8 Análise de Chamadas Recursivas

O passo `CallGraphSCCPass` identifica as recursões indiretas, e, quando é executado, o grafo de chamada de funções (Call Graph) já foi construído. Desta forma, pode-se percorrer este grafo e identificar todas as recursões.

A recursão representa um problema complexo para a análise do tempo de execução, pois, freqüentemente, não é possível determinar o número máximo de chamadas recursivas que uma função irá executar a partir de uma entrada qualquer sem simular o comportamento desta para um grande conjunto de entradas.

Com o uso da LLVM, pode-se realizar esta simulação diretamente na representação intermediária, e utilizar-se o resultado da análise de estruturas de dados para identificar se alguma outra função chamada a partir da atual pode alterar alguma locação de memória que influencie no controle da recursão, da mesma forma que o loop.

5.9 Análise da influência de uma arquitetura específica de hardware

O gerador de código da LLVM utiliza algumas informações do hardware para realizar alocação de registradores e seleção de instruções. Estes módulos podem ser modificados para acrescentar passos de cálculo do tempo de execução para cada arquitetura.

6 Abordagem para Implementação

Para validação da LLVM como plataforma para desenvolvimento de ferramenta de análise de tempo de execução, foi realizada a implementação de um analisador de tempo, restrito à análise de alto nível, ou seja, sem realizar a análise do comportamento de um processador específico.

Na implementação realizada, consideramos que cada instrução da LLVM é executada pelo processador em uma unidade de tempo, e, com isto, podemos assumir que o maior tempo de execução é dado pelo caminho com o maior número de instruções.

Esta simplificação nos permite concentrar o estudo na análise do fluxo de controle, e deixar em aberto o problema da simulação do processador, que pode ser acrescentada em um trabalho futuro.

Com a utilização de análises já disponíveis na LLVM, parte dos algoritmos necessários para o cálculo do tempo de execução são simplificados.

A análise de fluxo de controle já disponível na ferramenta gera um grafo de chamada de funções, computando chamadas diretas e indiretas (a partir de ponteiros para função), e, dentro de cada função, um grafo com os blocos básicos representando o fluxo de controle dentro desta função. A representação LVIS já é na forma SSA, poupando mais um passo da análise.

A LLVM é apresentada com detalhes no capítulo 5 e o uso que é feito dela para análise de fluxo de controle para cálculo do tempo de execução é mostrado neste capítulo.

6.1 Reutilização de resultados de análise

A riqueza de informações da representação intermediária da LLVM e das análises já disponíveis para esta plataforma fornecem ao usuário um conjunto de informações valioso sobre o código.

A estrutura da representação intermediária (LVIS) já organiza os blocos básicos como um grafo, com uma extensa API de navegação. A partir de um bloco básico, é possível identificar a instrução que termina o mesmo (médoto `getTerminator`), e, a partir desta, todos os *sucessores* deste bloco (blocos para os quais o fluxo de controle pode avançar ao final do bloco básico). Da mesma forma, cada bloco básico tem uma lista de *predecessores* (blocos básicos a partir dos quais o fluxo de controle pode avançar para este bloco).

A LVIS tem duas instruções que marcam a invocação de funções:

`call`: chamada de função, que pode aparecer no meio de um bloco básico. A semântica desta instrução é uma chamada normal de função, ou seja, quando a instrução `ret` da função é executada, o fluxo de controle retorna à instrução seguinte ao `call`, no mesmo bloco básico.

`invoke`: chamada de função com implementação de *zero cost exception handling*[1]. É uma chamada de função que sempre finaliza um bloco básico, e tem um parâmetro adicional, que é o endereço para retorno de exceção. Se a função chamada terminar com `ret`, o fluxo de controle avança para o bloco básico que representa a continuação normal da função (o próximo bloco no fluxo de controle principal), e, se a instrução de retorno for `unwind`, o fluxo de controle avança para o bloco de tratamento de exceção.

Além da possibilidade de navegar no código através das estruturas de dados definidas pela LVIS, a LLVM ainda provê resultados de análises do grafo de invocação, do grafo de fluxo de controle e das variáveis acessadas por cada função ou bloco básico.

Neste trabalho, utilizamos resultados de algumas destas análises.

Análise do Grafo de Invocação: A maneira mais simples de navegar no grafo de invocação gerado pela LLVM é especializar a classe `CallGraphSCCPass`. Esta especialização passa pela sobreposição do método `runOnSCC`, que será chamado para cada componente SCC do grafo de invocação.

Neste trabalho, um passo deste tipo foi utilizado para a criação do grafo de escopos, identificando os escopos nas situações de chamada de função recursiva.

[1] Mecanismo de tratamento de exceções em que nenhuma instrução adicional é executada no retorno normal da função (na inexistência de exceção).

Análise de Loops: A LLVM possui um passo de análise que identifica os *loops naturais*[2], chamado `LoopInfo`, que foi utilizado neste trabalho para a criação do grafo de escopos para cada função.

6.2 Ferramenta de cálculo de tempo usando LLVM

A arquitetura proposta por este trabalho para uma ferramenta de cálculo de tempo com base da LLVM é ilustrada na figura 6.1.

Uma das vantagens de utilizar a LLVM como base para executar este trabalho é que muitas das otimizações podem ser realizadas de maneira independente do cálculo de tempo, antes da atuação desta ferramenta.

O código fonte do programa é compilado com um dos front-ends possíveis para a LLVM (por exemplo, o llvm-gcc, conforme ilustrado na figura 6.1). O código gerado, na representação intermediária da LLVM (*LVIS*, descrita na seção 5.3) pode ser transformado pela ferramenta de otimização (opt), e vários módulos fonte podem ser ligados ainda na representação intermediária, utilizando-se a ferramenta `llvm-link`. Desta forma, a entrada para o cálculo de tempo de execução é um único arquivo, contendo todo o programa já otimizado[3] na representação LVIS.

A partir da representação do programa em LVIS, podemos dividir a ferramenta de cálculo de tempo em duas partes: a análise de alto nível, independente da plataforma alvo (programa `llflow` da figura 6.1, objeto deste trabalho) e outra dependente da plataforma alvo (programa `llvm-wcet` na figura, indicado como possível desdobramento deste trabalho).

Esta ferramenta dependente da plataforma alvo será responsável não só pelo cálculo de tempo como pela geração do código objeto final, com instruções nativas.

6.3 Implementação da análise de fluxo de controle

O objetivo da análise de fluxo de controle é gerar as seguintes saídas:

[2]Loops que atendem às seguintes restrições (AHO; SETHI; ULLMAN, 1988):

- Possuem um único ponto de entrada (*header*), que *domina* todos os nodos (blocos básicos) no loop.
- Possuem pelo menos um caminho de volta ao *header*, ou seja, um meio de iterar no loop;

[3]por “otimizado” entendemos que já foram aplicadas ao programa as análises e transformações possíveis na representação intermediária. Algumas otimizações adicionais (*peephole optimization*) podem ser executadas no momento da geração do código para a arquitetura alvo.

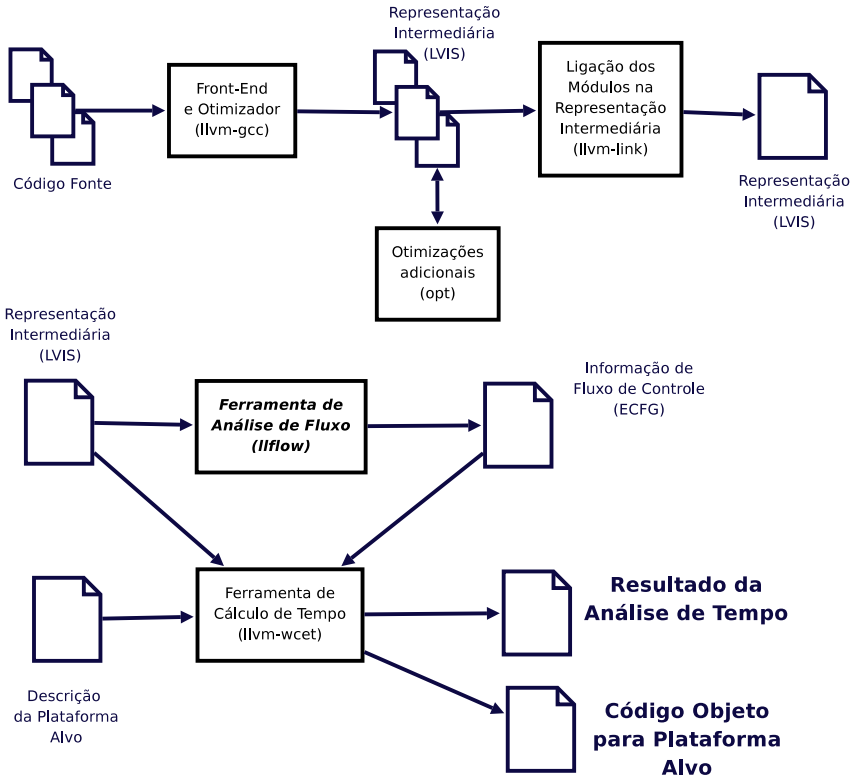


Figura 6.1: Arquitetura proposta para uma ferramenta de cálculo de tempo usando LLVM

- Grafo de fluxo de controle estendido, com indicação dos escopos de análise.
- Anotações em cada escopo, indicando:
  - número de execuções de cada loop ou recursão
  - intervalos válidos para as variáveis envolvidas nas decisões do fluxo de controle
- Caminhos possíveis e impossíveis no programa, o que pode ser feitor através de enumeração exaustiva dos caminhos possíveis ou das anotações no grafo de fluxo de controle estendido para que os caminhos sejam identificados pelo cálculo de tempo utilizando-se o algoritmo IPET.

Para isto, utilizamos como entrada o código do programa (em representação intermediária LVIS) e um conjunto de informações sobre as tarefas de tempo real:

- faixas de valores que podem ser retornados por chamadas de funções externas (funções cujo código não está disponível para análise).
- faixas de valores que podem ser retornados em parâmetros passados por referência a funções externas

- funções do programa que devem ser analisadas (pontos de entrada, caso apenas uma parte do programa for tarefa de tempo real ou quando o ponto de entrada não for a função `main`), com informações sobre as faixas de valores que podem ser aceitos como parâmetros de entrada.

Nesta implementação foi criada uma nova ferramenta, utilizando a infra-estrutura provida pela LLVM, composta dos seguintes passos:

- Criação do grafo de escopos, identificando escopos para componentes SCC do grafo de invocação, para cada função e para cada loop natural[4] dentro das funções.
- Análise de variáveis globais para identificação dos limites dos loops e recursões, percorrendo o grafo de chamadas de funções e o CFG de cada função para cada escopo criados na análise do grafo de invocação, e identificando os intervalos de valores destas variáveis.

6.3.1 Criação dos escopos para o grafo de invocação

A menos que o usuário informe funções específicas no arquivo de configuração, a ferramenta de análise assume que cada função no programa pode ser considerada uma tarefa de tempo real para a qual é necessário o cálculo do WCET. Desta forma, a hierarquia de escopos considera que qualquer uma das funções pode ser ponto de entrada, através da criação de um escopo global (`program`) e inserção de cada componente SCC do grafo de invocação como um sub-escopo do programa.

O resultado desta análise é um grafo de fluxo de controle estendido, com os escopos agrupados, como o mostrado na figura 6.2, na qual as linhas mais finas indicam fluxo de controle dentro das funções, as linhas mais grossas representam as chamadas das funções, as elipses representam blocos básicos e os retângulos representam os escopos (loops, funções e componentes SCC no grafo de invocação).

O algoritmo usado para construir este grafo de escopos é mostrado na figura 6.3, e são criados escopos para todos os componentes SCC do grafo de invocação, para cada função no programa e para cada loop.

[4] Nossa implementação não identifica loops em código não estruturado

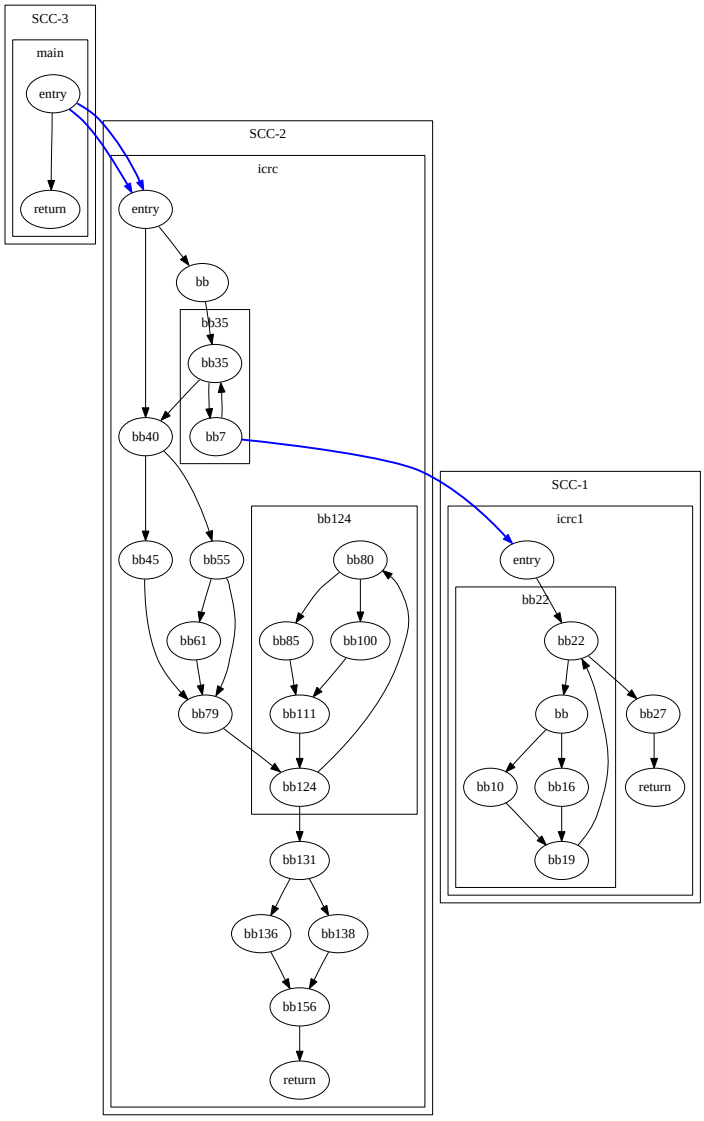


Figura 6.2: Grafo de Fluxo de controle com escopos (ECFG) para programa de cálculo de CRC

6.3.2 Descrição da implementação

A ferramenta desenvolvida foi organizada de acordo com a recomendação para projetos da documentação da LLVM (LATTNER, 2008), utilizando-se da estrutura de diretórios e dos scripts de compilação/montagem fornecidos, garantindo portabilidade do projeto para os ambientes suportados pela LLVM.

Construiu-se um módulo de controle (driver) que executa a inicialização dos componentes da LLVM (`Module` e `PassManager`), cria os passos necessários para a análise e registra estes passos no PassManager, fazendo com que as dependências sejam resolvidas automaticamente e os passos executados na ordem correta.

A análise foi dividida em três passos:

```
Para cada função no programa
    criar um escopo para função e armazená-lo em uma lista
    identificar loops naturais no código da função
    para cada loop na função
        criar um escopo para o loop
        para cada loop interno encontrado (recursivamente)
            criar um escopo para o loop
            inserir o escopo como ”sub-escopo”do loop anterior
        inserir o loop como ”sub-escopo”da função
Criar o escopo ”program”
Para cada nodo ”SCC”no grafo de invocacação
    criar um escopo para o SCC
    inserir o escopo criado como subescopo de ”program”
    Para cada função no SCC
        buscar na lista o escopo que foi criado para esta função
        inserir o escopo como subescopo do SCC criado
```

Figura 6.3: Algoritmo para identificação dos escopos globais

ScopeSCCPass : Um passo derivado do `CallGraphSCCPass` da LLVM, executado para cada componente SCC do grafo de invocação, e que identifica os escopos para as funções e as recursões (escopos SCC no grafo de invocação).

ScopePass : Passo derivado do `ModulePass` da LLVM, utilizando as informações de análise de loops (`LoopInfo`) e de análise de fluxo de dados (`AliasAnalysis`) para criar o grafo de escopos, incluindo nos escopos criados no passo anterior os escopos de loops dentro das funções.

ExecutionPass : Análise prorpiamente dita, derivando de um `FunctionPass` e utilizando-se dos dados identificados no passo anterior, e realizando a execução abstrata do código para determinar o maior caminho e o número máximo de recursões e iteraçãoes de loops.

Esta abordagem foi escolhida pelas seguintes razões:

- Com o uso de um `CallGraphSCCPass`, a LLVM identifica automaticamente as recursões diretas e indiretas (SCCs no grafo de invocação), tornando a criação de escopos por SCC trivial.
- O `CallGraphSCCPass` é executado pelo PassManager de forma independente dos passos que inspecionam o conteúdo das funções.

  Desta forma, não é possível vincular a execução deste com um passo de análise de loops e manter as informações dos loops entre as diversas invocações do passo de análise do grafo (ao analisar uma função, os loops detectados em outras funções são liberados da memória).
- Um `ModulePass` pode utilizar resultados de qualquer análise em qualquer parte do programa (ou seja, pode manter os resultados de todas as análises que requisitar), permitindo

a análise global do grafo de fluxo de controle. Ao colocar este passo como pré-requisito do passo de análise do grafo de invocação, as funções serão analisadas primeiro, mantendo este resultado durante toda a execução dos dois passos.

6.3.3 Estruturas de Dados

O grafo de escopos gerado pelo programa é um ECFG, ou seja, contém os dados dos escopos de análise e o grafo de fluxo de controle estendido (grafo de invocação e fluxo de controle juntos). Desta forma, foram utilizadas as informações constantes nas estruturas de dados da LLVM que representam blocos básicos (`BasicBlock`) e funções (`Function`). A estas estruturas foram acrescentadas as classes indicadas na figura 6.4.

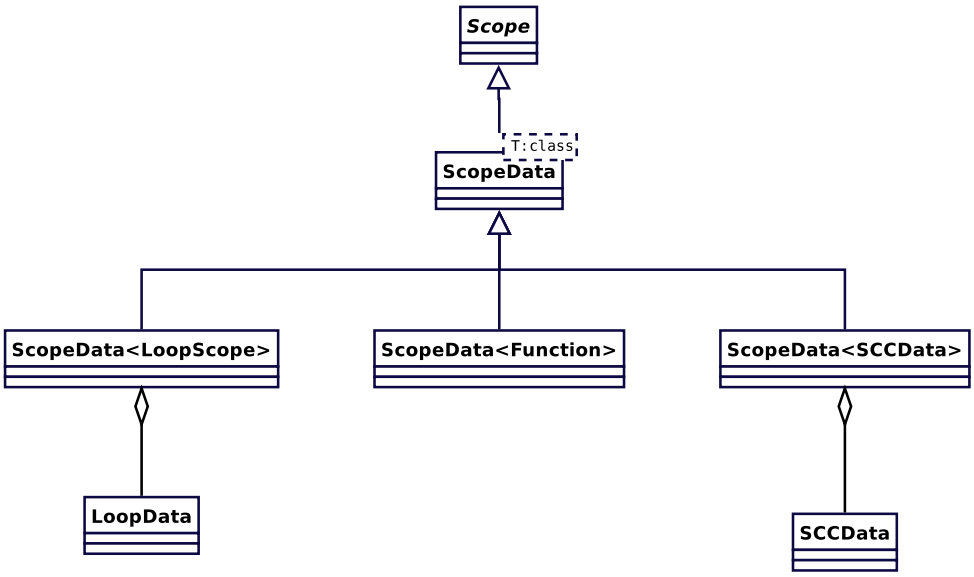


Figura 6.4: Hierarquia de classes de nodo do grafo de escopos

Estas classes armazenam e manipulam as informações do grafo e escopos:

Scope: classe que especifica e implementa a funcionalidade básica de um escopo:

- Conter e manipular a lista de escopos aninhados.
- Conter a manipular as anotações no grafo de fluxo de controle
- Prover iteradores para navegação na lista dos escopos aninhados.

ScopeData: template para criação de classes de escopo contendo dados do nodo do CFG ou IG que o escopo representa. Além do comportamento definido por `Scope`, a classe ScopeData define apenas o armazenamento e recuperação de um elemento abstrato de

dados. Este template é instanciado em três implementações concretas, que representam os escopos para *Funções*, *Componentes SCC no IG* e *Loops*

SCCData: consiste em uma lista de referências aos objetos `Function` que fazem parte deste SCC, ou seja, um grupo de funções que tem um comportamento recursivo.

LoopData: constiste em uma lista de referências aos blocos básicos (`BasicBlock`) que compõem um loop, e uma referência ao bloco básico que é a *cabeça* do loop, ou seja, o bloco básico que *domina*[5] todos os demais blocos.

6.3.4 Formato do arquivo de configuração

Para cada programa de entrada na ferramenta de análise de tempo deve ser escrito um arquivo de configuração que indique como esta análise deverá ser realizada, e como funções externas ao programa devem ser tratadas.

O arquivo de configuração deverá conter duas seções: uma que especifica as funções externas (`extern`) e uma que especifica as funções de tempo real a serem analisadas (`analyse`). A figura 6.5 ilustra um exemplo de arquivo de configuração, com as seguintes características:

- configuração para anális do arquivo `loop1.bc`
- duas funções externas (ligadas ao programa a partir de bibliotecas nativas às quais a ferramenta de análise não tem acesso no momento da análise.

  atoi recebe um único parâmetro de entrada (ou seja, mesmo que este parâmetro seja um ponteiro, seu conteúdo não é alteardo pela função), sinalizado pela expressão `in`, e retorna um valor no intervalo de -1000 a 1000.

  exf1 recebe três parâmetros, dois apenas de entrada (in), e um que altera o valor da variável passada por referência, atribuindo a esta um valor entre -10 e 0 ou um valor entre 30 e 100. Esta função retorna um valor no intervalo entre -1 e 0.

- Uma função que deve ser analisada como tarefa de tempo real (`f1`), com três parâmetros:
  - dois valores entre -1000 e 1000
  - um valor que pode ser entre -10 e 100 ou entre 500 e 2000 ou entre 5000 e 12000.

[5] nossa implementação limita-se à análise dos loops naturais, em que um nodo *domina* todos os demais nodos no loop.

```
config: loop1.bc

external {
   atoi (in) = [-1000,1000];
   exf1 (in,in,[-10,0][30,100]) = [-1,0];
}

analyse {
   f1 ( [-1000,1000], [-1000,1000],
        [-10,100] [500,2000] [5000,12000]);
}
```

Figura 6.5: Exemplo de arquivo de configuração

Para as funções externas, basta configurar os parâmetros que alteram algum valor passado por referência, e a faixa de valores de retorno, enquanto para as funções que representam tarefas de tempo real que precisam ser analisadas, é preciso especificar, também, as faixas de valores válidos para estas entradas (esta informação será usada para interpretação abstrata do código da função, permitindo determinar o número máximo de iterações dos loops).

6.3.5 Determinação do número de iterações dos loops

A análise dos loops realizada pelo `llflow` consiste na interpretação abstrata do código, avaliando as instruções que influenciam nas variáveis de controle dos loops e estimando o número máximo de iterações para cada loop.

Para realizar esta análise, fizemos as seguintes simplificações:

- todas as variáveis numéricas do programa são tratadas como inteiros de 64 bits com sinal
- os valores que vêm do ambiente externo são informados para o programa como listas de intervalos de valores, que são manipulados pelo interpretador.

Do arquivo de configuração conseguimos obter as faixas de variação de cada uma das entradas externas do programa, na forma de uma lista de intervalos. Esta informação vai ser utilizada pelo interpretador para associar valores às variáveis

O intepretador mantém uma lista das funções configuradas para análise (se nenhuma função for configurada, todas as funções do programa são analisadas). Nesta lista, são armazenadas as faixas de variação dos parâmetros.

Se a faixa de variação dos parâmetros não for configurada, assume-se uma das alternativas:

- quando tipo de dado do parâmetro é menor que um inteiro de 64 bits, utiliza-se a faixa de variação para este tipo (por exemplo, para um inteiro de 8 bits, é assumida a faixa [-128,127], ou para um inteiro sem sinal de 16 bits, assume-se [0,65535]);
- quando o tipo de dados é um inteiro de 64 bits ou ponto flutuante, assume-se a variação máxima de um inteiro de 64 bits.

Para cada função a analisar, o interpretador cria um contexto inicial de execução e simula cada instrução da função, de acordo com as seguintes regras:

- Cada vez que o fluxo de execução entra em um dos escopos definidos no ECFG, o contador de execuções deste escopo para este contexto é incrementado.
- Cada vez que uma instrução de alocação de memória é executada, é acrescentada à tabela de símbolos global um símbolo para este endereço, permitindo a análise dos valores que são armazenados em memória.
- As expressões aritméticas são interpretadas da seguinte forma:

  Soma ou subtração: Estende ou reduz os intervalos para abranger todos os resultados possíveis da operação:

$$\begin{aligned} [2,7]\,[10,15] + [1,3] &= [3,18] \\ [1,3] + [1,3]\,[10,15] &= [1,6]\,[11,18] \\ [-10,10] - [2,3] &= [-13,8] \end{aligned}$$

  Multiplicação, Divisão e Resto: Realiza o produto cartesiano dos intervalos e aplica a operação (multipliação, divisão inteira ou resto) aos elementos dos pares ordenados resultantes, seguida da união dos intervalos resultantes:

$$\begin{aligned} [2,7] \times [2,3] &= [4,6]\,[14,21] \\ [2,10] \div [2,3] &= [0,1]\,[3,5] \end{aligned}$$

- As operações booleanas causam uma divisão no contexto de execução: os valores das variáveis envolvidas são recalculados para o valor verdadeiro e para o valor falso, e segue-se os dois caminhos de execução a partir daí:

$$
\begin{array}{c}
a = [5,15] \\
b = [3,12] \\
a < b \rightarrow \left\{ \begin{array}{ll}
resultado = TRUE & \left\{ \begin{array}{l} a = [5,11] \\ b = [6,12] \\ \text{restrição: } a < b \end{array} \right. \\
resultado = FALSE & \left\{ \begin{array}{l} a = [3,12] \\ b = [3,12] \\ \text{restrição: } \neg(a < b) \end{array} \right.
\end{array} \right.
\end{array}
$$

  Cada vez que um contexto é criado, todo o estado do interpretador (valores das variáveis globais, memória e valores das variáveis locais) é copiado para o novo contexto, antes das restrições serem aplicadas e os valores de variáveis atualizados. Assim, pode-se armazenar um contexto temporariamente em uma pilha enquanto outro é avaliado e voltar para seguir o outro caminho recuperando o contexto.

  Este comportamento é a chave para identificação dos caminhos impossíveis no programa: a cada operação de divisão de contexto, restringe-se as faixas de valores das variáveis e, com isto, os caminhos futuros que podem ser tomados pelo programa. Cada vez que esta divisão de intervalos resulta em um intervalo vazio para uma das condições (verdadeira ou falsa), este caminho é abandonado, considerado infactível.

- As chamadas de função provocam a execução da função, com os parâmetros restritos pelos valores ativos no contexto de execução atual, e com as variáveis globais e áreas de memória com os seus valores atuais.

- As demais instruções mantém sua semântica usual, conforme definida na documentação da LLVM (LATTNER, 2008).

A cada vez que o fluxo de controle chega ao final da função a ser analisada, considera-se que chegou-se ao final daquele caminho de execução, e retorna-se à última expressão booleana avaliada, para percorrer o outro caminho, em um algoritmo de busca em profundidade. A cada vez que se sobe um nível em direção à raiz do grafo, a implementação atual avalia o comprimento dos subcaminhos a partir deste ponto na árvore (já avaliados com busca me profundidade), e mantém apenas o maior caminho. Quando a ferramenta for estendida para simular o comportamento do processador, mesmo com um custo maior de memória, é recomendável manter-se

todos os caminhos, pois pode-se encontrar situações em que o caminho com o maior número de instruções não é necessariamente o que leva mais tempo a ser executado. Esta decisão deve ser adiada para depois da simulação do tempo de cada caminho.

Ao final deste processo, resta apenas um caminho de execução para cada função analisada, que é registrado como anotação no ECFG obtido pelos passos de análise de escopos, para gravação no arquivo de saída do programa.

7 Resultados Alcançados

A partir do levantamento bibliográfico realizado para este trabalho, pudemos constatar que a área da análise de tempo conta com uma comunidade bastante ativa de pesquisadores, e pudemos realizar um apanhado geral das principais técnias e ferramentas que vêm sendo discutidas nesta comunidade.

A abordagem que foi proposta neste trabalho para o desenvolvimento de uma ferramenta de análise de tempo recomenda a implementação da análise de tempo integrada a um compilador, como proposto em Engblom, Ermedahl e Altenbernd (1998), e valida o uso da LLVM como plataforma para o desenvolvimento deste compilador/ferramenta de análise.

Para validar o uso da LLVM para análise de tempo, foi desenvolvido um protótipo de uma ferramenta de análise de fluxo de controle do programa, descrita no capítulo 6.

7.1 Resultados da Ferramenta de Análise

Para ilustrar o resultado da análise realizada pela ferramenta `llflow`, partimos de um pequeno exemplo em linguagem C, ilustrado na figura 7.1

Este exemplo foi compilado com o `llvm-gcc` com duas opções: sem otimização nenhuma e com o conjunto padrão de otimizações (`-O3`). Estas duas opções são mostradas na representação textual da LVIS, nas figuras 7.2 e 7.3, respectivamente.

A diferença de entre as duas versões do código ilustra a principal motivação para o desenvolvimento da ferramenta de análise associada ao compilador ou da análise realizada diretamente em código objeto: as transformações realizadas pela otimização dificultam a identificação das estruturas do código fonte no código analisado.

A versão otimizada deste código foi passada para a ferramenta `llflow` desenvolvida neste trabalho, resultando no grafo de escopos da figura 7.5, com três escopos criados: o escopo `SSC1` (obtido do grafo de invocação), e, dentro deste, o escopo `inverteArray`, correspondente

à função ilustrada na figura 7.1. A configuração para execução é apresentada na figura 7.4, indicando que a função pode ser chamada com qualquer valor para o array de entrada, mas que o parâmetro `tamanho` tem que ser um valor entre 1 e 100.

Na figura 7.5 o resultado para número de iterações (valor `count`) calculado é mostrado para a função (uma única chamada) e para o loop.

```
void inverteArray (int *array, int tamanho)
{
   int tmp;
   int p,q;

   for (p=0,q=tamanho-1;p<q;++p,--q)
   {
      tmp =array[p];
      array[p] = array[q];
      array[q] = tmp;
   }
}
```

Figura 7.1: Código fonte

```
; ModuleID = 'caso1.bc'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:\
32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v128:128:128-a0:0:64-f80:32:32"
target triple = "i686-pc-linux-gnu"

define void @inverteArray(i32* %array, i32 %tamanho) {
entry:
        %array_addr = alloca i32*               ; <i32**> [#uses=5]
        %tamanho_addr = alloca i32              ; <i32*> [#uses=2]
        %q = alloca i32         ; <i32*> [#uses=6]
        %p = alloca i32         ; <i32*> [#uses=6]
        %tmp = alloca i32               ; <i32*> [#uses=2]
        %"alloca point" = bitcast i32 0 to i32          ; <i32> [#uses=0]
        store i32* %array, i32** %array_addr
        store i32 %tamanho, i32* %tamanho_addr
        store i32 0, i32* %p, align 4
        %tmp1 = load i32* %tamanho_addr, align 4                ; <i32> [#uses=1]
        %tmp2 = sub i32 %tmp1, 1                ; <i32> [#uses=1]
        store i32 %tmp2, i32* %q, align 4
        br label %bb22

bb:             ; preds = %bb22
        %tmp3 = load i32** %array_addr, align 4         ; <i32*> [#uses=1]
        %tmp4 = load i32* %p, align 4           ; <i32> [#uses=1]
        %tmp5 = getelementptr i32* %tmp3, i32 %tmp4             ; <i32*> [#uses=1]
        %tmp6 = load i32* %tmp5, align 4                ; <i32> [#uses=1]
        store i32 %tmp6, i32* %tmp, align 4
        %tmp7 = load i32** %array_addr, align 4         ; <i32*> [#uses=1]
        %tmp8 = load i32* %q, align 4           ; <i32> [#uses=1]
        %tmp9 = getelementptr i32* %tmp7, i32 %tmp8             ; <i32*> [#uses=1]
        %tmp10 = load i32* %tmp9, align 4               ; <i32> [#uses=1]
        %tmp11 = load i32** %array_addr, align 4                ; <i32*> [#uses=1]
        %tmp12 = load i32* %p, align 4          ; <i32> [#uses=1]
        %tmp13 = getelementptr i32* %tmp11, i32 %tmp12          ; <i32*> [#uses=1]
        store i32 %tmp10, i32* %tmp13, align 4
        %tmp14 = load i32** %array_addr, align 4                ; <i32*> [#uses=1]
        %tmp15 = load i32* %q, align 4          ; <i32> [#uses=1]
        %tmp16 = getelementptr i32* %tmp14, i32 %tmp15          ; <i32*> [#uses=1]
        %tmp17 = load i32* %tmp, align 4                ; <i32> [#uses=1]
        store i32 %tmp17, i32* %tmp16, align 4
        %tmp18 = load i32* %p, align 4          ; <i32> [#uses=1]
        %tmp19 = add i32 %tmp18, 1              ; <i32> [#uses=1]
        store i32 %tmp19, i32* %p, align 4
        %tmp20 = load i32* %q, align 4          ; <i32> [#uses=1]
        %tmp21 = sub i32 %tmp20, 1              ; <i32> [#uses=1]
        store i32 %tmp21, i32* %q, align 4
        br label %bb22

bb22:           ; preds = %bb, %entry
        %tmp23 = load i32* %p, align 4          ; <i32> [#uses=1]
        %tmp24 = load i32* %q, align 4          ; <i32> [#uses=1]
        %tmp25 = icmp slt i32 %tmp23, %tmp24            ; <i1> [#uses=1]
        %tmp2526 = zext i1 %tmp25 to i8         ; <i8> [#uses=1]
        %toBool = icmp ne i8 %tmp2526, 0                ; <i1> [#uses=1]
        br i1 %toBool, label %bb, label %bb27

bb27:           ; preds = %bb22
        br label %return

return:         ; preds = %bb27
        ret void
}
```

Figura 7.2: Código em representação LVIS, não otimizado

```
; ModuleID = 'caso1-opt.bc'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:\
32-i64:32:64-f32:32:32-f64:32:64-v64:64:64-v128:128:128-a0:0:64-f80:32:32"
target triple = "i686-pc-linux-gnu"

define void @inverteArray(i32* %array, i32 %tamanho) nounwind  {
entry:
        %q.034 = add i32 %tamanho, -1           ; <i32> [#uses=2]
        %tmp2537 = icmp sgt i32 %q.034, 0               ; <i1> [#uses=1]
        br i1 %tmp2537, label %bb, label %return

bb:             ; preds = %bb, %entry
        %p.0.reg2mem.0 = phi i32 [ 0, %entry ], [ %tmp19, %bb ]         ; <i32> [#uses=3]
        %q.0.reg2mem.0 = sub i32 %q.034, %p.0.reg2mem.0         ; <i32> [#uses=2]
        %tmp5 = getelementptr i32* %array, i32 %p.0.reg2mem.0           ; <i32*> [#uses=2]
        %tmp6 = load i32* %tmp5, align 4                ; <i32> [#uses=1]
        %tmp9 = getelementptr i32* %array, i32 %q.0.reg2mem.0           ; <i32*> [#uses=2]
        %tmp10 = load i32* %tmp9, align 4               ; <i32> [#uses=1]
        store i32 %tmp10, i32* %tmp5, align 4
        store i32 %tmp6, i32* %tmp9, align 4
        %tmp19 = add i32 %p.0.reg2mem.0, 1              ; <i32> [#uses=2]
        %q.0 = add i32 %q.0.reg2mem.0, -1               ; <i32> [#uses=1]
        %tmp25 = icmp slt i32 %tmp19, %q.0              ; <i1> [#uses=1]
        br i1 %tmp25, label %bb, label %return

return:         ; preds = %bb, %entry
        ret void
}
```

Figura 7.3: Código em representação LVIS, otimizado

```
config: caso1.bc;

analyse {
   inverteArray ([],[1,100]);
}
```

Figura 7.4: Arquivo de configuração da ferramenta de análise

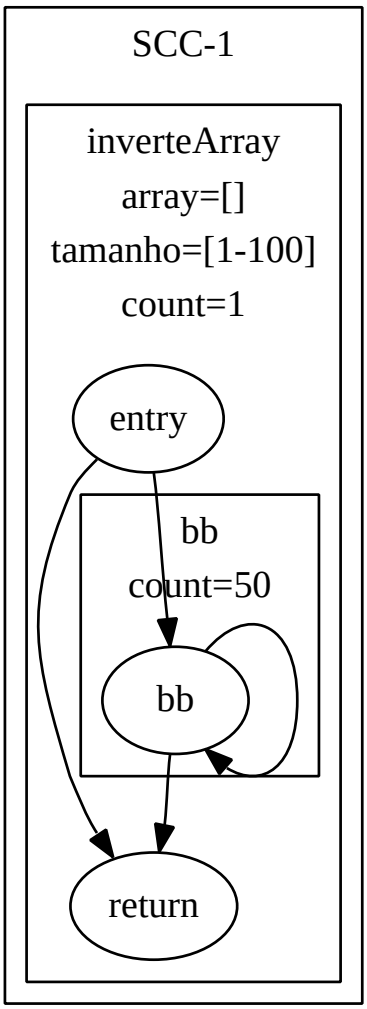


Figura 7.5: Código em representação LVIS, otimizado

7.2 Um caso mais complexo: cálculo de CRC

Para ilustrar a viabilidade do uso da ferramenta desenvolvida em problemas do mundo real, executamos a análise de uma implementação do algoritmo de cálculo de CRC (código de redundância cíclica)

O código fonte analisado é o demonstrado na seção 7.2.1, e os resultados de análise são mostrados em 7.6.

Destaca-se, neste caso, o número de invocações à função crc1, que, a pesar de não ser uma função recursiva, tem a anotação de quantidade de vezes que foi invocada de dentro do loop da função crc.

7.2.1 Código fonte do programa CRC

```
/* $Id: crc.c,v 1.1 2008-05-11 17:59:08 xande Exp $ */

/*************************************************************************/
/*                                                                       */
/*   SNU-RT Benchmark Suite for Worst Case Timing Analysis               */
/*   =====================================================               */
/*                              Collected and Modified by S.-S. Lim       */
/*                                        sslim@archi.snu.ac.kr          */
/*                                      Real-Time Research Group         */
/*                                     Seoul National University         */
/*                                                                       */
/*                                                                       */
/*        < Features > - restrictions for our experimental environment   */
/*                                                                       */
/*          1. Completely structured.                                    */
/*               - There are no unconditional jumps.                     */
/*               - There are no exit from loop bodies.                   */
/*                 (There are no 'break' or 'return' in loop bodies)     */
/*          2. No 'switch' statements.                                   */
/*          3. No 'do..while' statements.                                */
/*          4. Expressions are restricted.                               */
/*               - There are no multiple expressions joined by 'or',     */
/*                'and' operations.                                      */
/*          5. No library calls.                                         */
/*               - All the functions needed are implemented in the       */
/*                 source file.                                          */
/*                                                                       */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
/*  FILE: crc.c                                                          */
/*  SOURCE : Numerical Recipes in C - The Second Edition                 */
/*                                                                       */
/*  DESCRIPTION :                                                        */
```

```
/*                                                                      */
/*     A demonstration for CRC (Cyclic Redundancy Check) operation.     */
/*     The CRC is manipulated as two functions, icrc1 and icrc.         */
/*     icrc1 is for one character and icrc uses icrc1 for a string.     */
/*     The input string is stored in array lin[].                       */
/*     icrc is called two times, one for X-Modem string CRC and the     */
/*     other for X-Modem packet CRC.                                    */
/*                                                                      */
/*  REMARK :                                                            */
/*                                                                      */
/*  EXECUTION TIME :                                                    */
/*                                                                      */
/*                                                                      */
/************************************************************************/


typedef unsigned char uchar;
#define LOBYTE(x) ((uchar)((x) & 0xFF))
#define HIBYTE(x) ((uchar)((x) >> 8))

unsigned char lin[256] = "asdffeagewaHAFEFaeDsFEawFdsFaefaeerdjgp";

unsigned short icrc1(unsigned short crc, unsigned char onech)
{
int i;
unsigned short ans=(crc^onech << 8);

for (i=0;i<8;i++) {
if (ans & 0x8000)
ans = (ans <<= 1) ^ 4129;
else
ans <<= 1;
}
return ans;
}

unsigned short icrc(unsigned short crc, unsigned long len,
    short jinit, int jrev)
{
  unsigned short icrc1(unsigned short crc, unsigned char onech);
  static unsigned short icrctb[256],init=0;
  static uchar rchr[256];
  unsigned short tmp1, tmp2, j,cword=crc;
  static uchar it[16]={0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15};

  if (!init) {
    init=1;
    for (j=0;j<=255;j++) {
      icrctb[j]=icrc1(j << 8,(uchar)0);
      rchr[j]=(uchar)(it[j & 0xF] << 4 | it[j >> 4]);
    }
  }
  if (jinit >= 0) cword=((uchar) jinit) | (((uchar) jinit) << 8);
  else if (jrev < 0)
```

```
    cword=rchr[HIBYTE(cword)] | rchr[LOBYTE(cword)] << 8;
#ifdef DEBUG
  printf("len = %d\n", len);
#endif
  for (j=1;j<=len;j++) {
    if (jrev < 0) {
      tmp1 = rchr[lin[j]]^ HIBYTE(cword);
    }
    else {
      tmp1 = lin[j]^ HIBYTE(cword);
    }
    cword = icrctb[tmp1] ^ LOBYTE(cword) << 8;
  }
  if (jrev >= 0) {
    tmp2 = cword;
  }
  else {
    tmp2 = rchr[HIBYTE(cword)] | rchr[LOBYTE(cword)] << 8;
  }
  return (tmp2 );
}


int main(void)
{

  unsigned short i1,i2;
  unsigned long n;

  n=40;
  lin[n+1]=0;
  i1=icrc(0,n,(short)0,1);
  lin[n+1]=HIBYTE(i1);
  lin[n+2]=LOBYTE(i1);
  i2=icrc(i1,n+2,(short)0,1);
  return 0;
}
```

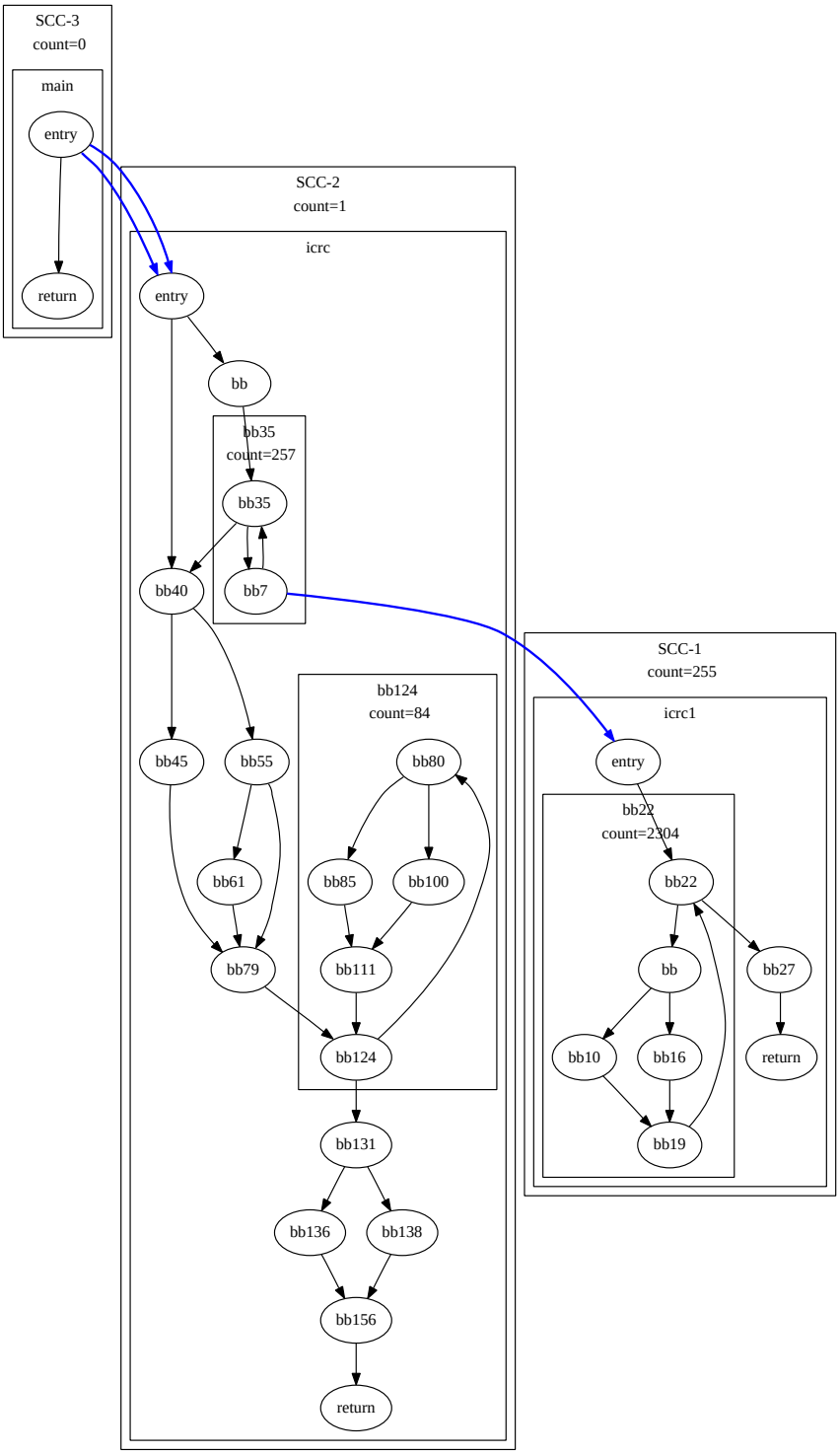


Figura 7.6: Resultados obtidos na análise de CRC

7.3 Recursões

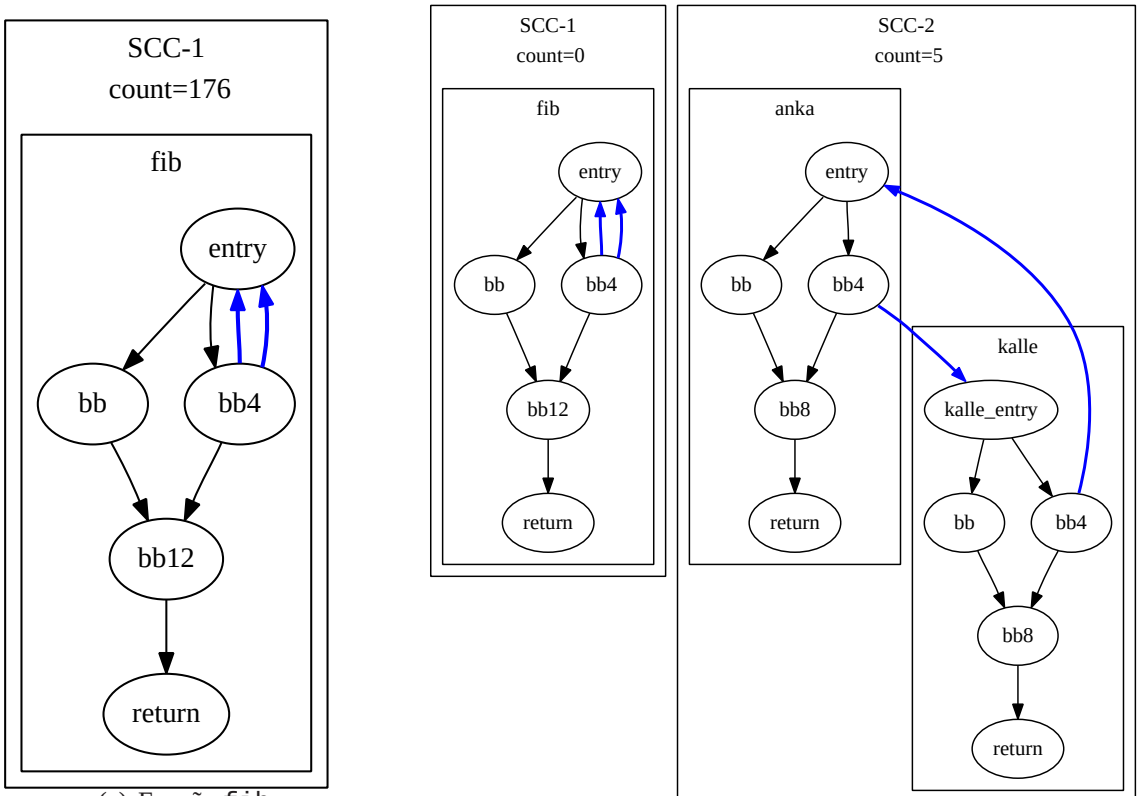
Para verificar a análise de chamadas de funções recursivas e de caminhos impossíveis no código, foi utilizado o programa da figura 7.7. Neste programa foram efetuadas análises das funções `fib` e `kalee`, ambas parametrizadas para entradas no intervalo de zero a 10.

O resultado desta análise é um conjunto de dois grafos, um para cada função analisada, conforme ilustrado na figura 7.8.

Para os dois casos, o processo de análise gerou as estatísticas ilustradas nas tabelas 7.1 e 7.2.

```
/* $Id: recursion.c,v 1.1 2008-05-11 17:59:08 xande Exp $ */

/* Generate an example of recursive code, to see  *
 * how it can be modeled in the scope graph.      */

/* self-recursion  */
int fib(int i)
{
   if(i <= 1)
      return 1;
   else
      return fib(i-1) + fib(i-2);
}

/* mutual recursion */
int anka(int);

int kalle(int i)
{
  if(i<=0)
    return 0;
  else
    return anka(i-1);
}

int anka(int i)
{
  if(i<=0)
    return 1;
  else
    return kalle(i-1);
}

int main(void)
{
   int In;
   In = fib(10);
   return 0;
}
```

Figura 7.7: Código fonte do programa utilizado para análise de funções recursivas



(a) Função `fib` (b) Função `kalee`

Figura 7.8: Resultados de análise de recursão

Estatística	**valor**
Escopos SCC criados	3
Escopos de função criados	4
Aninhamento máximo de chamadas de função	9
Número de chamadas de função analisadas	176
Número de instruções analisadas	3625
Número de caminhos impossíveis detectados	354

Tabela 7.1: Estatísticas para chamadas recursivas da função `fib`

Estatística	**valor**
Escopos SCC criados	3
Escopos de função criados	4
Aninhamento máximo de chamadas de função	10
Número de chamadas de função analisadas	10
Número de instruções analisadas	217
Número de caminhos impossíveis detectados	22

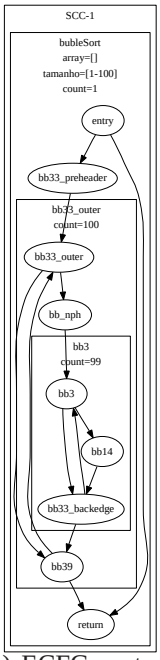
Tabela 7.2: Estatísticas para chamadas recursivas da função `kalee`

7.3.1 Limitações do protótipo

A figura 7.9 ilustra um exemplo um pouco mais complexo, demonstrando algumas limitações da abordagem proposta neste trabalho. Ao analisar o código deste exemplo, obtivemos os resultados indicados no grafo que aparece na mesma figura, com o loop mais interno podendo ser executado, no pior caso, 99 vezes. Porém, pode-se facilmente observar pelo código fonte do programa que o número de iterações do loop interno é decrescente para cada iteração do loop externo.

```
void bubleSort (int *array, int tamanho)
{
   int i,j, tmp;

   for ( i=0;i<tamanho;++i) {
      for (j=i+1;j<tamanho;++j) {
         if (array[j] < array[i]) {
            tmp = array[i];
            array[i] = array[j];
            array[j] = tmp;
         }
      }
   }
}
```

(a) Código Fonte



(b) ECFG anotado

Figura 7.9: Exemplo de análise 2: *buble-sort*

Pode-se contornar esta limitação através de duas abordagens:

- Ao invés de anotar os escopos com número de iterações, anotar cada bloco básico, e enumerar todos os caminhos no programa, mostrando como resultado o maior caminho, e não o grafo de fluxo de controle.

  Esta abordagem, para ser efetiva, exige que a análise de baixo nível (simulação do comportamento do processador) seja implementada em conjunto com a análise de fluxo de controle.

- Realizar anotações simbólicas, ou seja, deixar o número de iterações do loop interno como uma função da iteração atual do loop externo.

Esta abordagem tornaria mais complexa a análise final para identificação do caminho no grafo que representa o pior caso para o cálculo de tempo.

7.4 Sugestões para trabalhos futuros

Além da implementação da análise de baixo nível, para ter uma ferramenta completa de análise de tempo, alguns aspectos da análise de fluxo de controle devem ser melhor investigados:

- Uso mais eficiente das informações de análise providas pela LLVM
- Implementação das anotações de resultados de análise por bloco básico ao invés de anotações por escopo.

No contexto de sistemas embarcados de tempo real, é necessária uma proposta de geração de ferramentas redirecionáveis com a LLVM. Conseguimos, neste trabalho, identificar que boa parte do esforço para criação de um gerador de códigos com a LLVM (e, conseqüentemente, da geração de uma ferramenta de análise de baixo nível) é realizada através da definição de um conjunto de estruturas de dados, e da geração de código automática a partir destas estruturas utilizando-se a ferramenta tablegen. Acreditamos que este processo possa ser estendido ou adaptado para gerar estas estruturas a partir de uma linguagem de definição de arquitetura (*ADL*), possibilitando a geração automática de ferramentas redirecionáveis. Porém, este trabalho não aprofundou esta investigação o suficiente para identificar se este processo é realmente viável.

8 Conclusões

A análise do tempo de execução, e principalmente, a análise do tempo de execução no pior caso, é um tema de pesquisa bastante ativo, e observa-se uma forte tendência para o desenvolvimento de ferramentas integradas a compiladores e/ou ambientes de desenvolvimento, principalmente no contexto de sistemas embarcados de tempo real.

Da mesma forma, pode-se perceber uma tendência da comunidade científica em focar a discussão e a pesquisa em métodos de análise estática, usada sozinha, ou em conjunto com técnicas de medição (PETTERS; ZADARNOWSKI; HEISER, 2007).

O problema da análise estática de tempo de execução pode ser subdividido em fases, e cada uma destas fases tem seus próprios desafios:

- Análise de Fluxo de Controle do programa, para a qual os principais desafios são:
  - determinação do número de iterações em loops
  - determinação da profundidade das recursões
  - análise global de dados[1]
- Simulação do comportamento da máquina alvo, que, por sua vez, traz à tona o problema de simular os seguintes aspectos:
  - comportamento da memória cache (tanto de instruções como de dados)
  - comportamento dos preditores de desvios (*branch predictors*)
  - comportamento dos pipelines dos processadores

O presente trabalho concentrou-se em alguns aspectos da análise de fluxo de controle do programa, e, através de implementação de um protótipo de ferramenta de análise deste fluxo,

[1] variáveis de controle de loops e ou de instruções de desvio podem ser dependentes de dados globais que são alterados em outras funções, ou que são alterados indiretamente através ponteiros, e as chamadas de função podem ser realizadas indiretamente, a partir de ponteiros para função, dificultando a análise do fluxo de controle do programa

procurou indicar a viabilidade da construção deste tipo de ferramenta utilizando como base a LLVM.

Boa parte do trabalho de implementação de uma ferramenta de análise de tempo consiste na análise do código do programa e construção dos grafos de invocação de funções e do grafo de fluxo de controle de cada função. Estes dois aspectos são completamente resolvidos pela infra-estrutura fornecida pela LLVM, permitindo ao desenvolvedor concentrar-se nas questões específicas relativas à análise de tempo.

A escolha da LLVM como infra-estrutura de desenvolvimento mostrou-se acertada, e pode-se observar, nesta ferramentas, uma estrutura modular favorável à implementação de processos de análise: o processo de compilação é organizado em passos independentes, e um novo passo acrescentado é transparente para o compilador. Em outras palavras, ao acrescentar-se novos passos de análise, não é preciso realizar nenhuma alteração no compilador.

Outra vantagem importante da LLVM é a capacidade de um passo de análise reutilizar resultados de outros passos. Desta forma, pode-se utilizar, por exemplo, o passo *Alias Analysis*, e ter informações sobre o acesso a variáveis e funções através de ponteiros.

A implementação realizada neste trabalho utilizou como entrada a representação intermediária da LLVM e um conjunto de resultados de análise, como o grafo de invocação de funções, os grafos de fluxo de controle para cada função e a identificação de loops.

Mesmo que o protótipo desenvolvido para este trabalho não tenha abrangência sobre a questão da chamada “análise de baixo nível”, pudemos constatar, pelo estudo da LLVM, que a plataforma é adequada a este tipo de análise também, em vários aspectos:

Otimizações Globais: Como o LLVM permite a ligação de diversos módulos e até bibliotecas ainda utilizando a representação intermediária, as otimizações globais, como a expansão de funções *inline* ou mesmo alterações estruturais no código, podem ser realizadas antes do início da análise de tempo, e, ainda assim, a representação otimizada é de um nível alto o suficiente para que a análise de fluxo de controle seja facilitada.

Otimizações na fase de geração: As otimizações realizadas pelo gerador de código da LLVM na fase de geração final do código são isoladas ao contexto de uma função. Eventualmente, estas transformações podem alterar o grafo de fluxo de controle da função, invalidando alguns dos dados calculados em passos anteriores. Nesta situação, a abordagem recomendada é a apresentada por Engblom, Ermedahl e Altenbernd (1998), cuja implementação implica em alterações nos geradores atuais da LLVM.

Como a geração é feita uma função por vez, a simulação do comportamento do processador pode ser implementada como um passo do tipo `MachineFunctionPass`, que leve em conta os resultados da análise de alto nível para identificação dos possíveis contextos de execução de cada função e/ou bloco básico, resultando em um valor de tempo para a função para cada contexto de invocação. Um passo final, por conta da ferramenta de medição de tempo, pode totalizar estes valores, obtendo o WCET para o programa.

Gerador de código e análise redirecionáveis: A LLVM já suporta um conjunto de arquiteturas alvo, e o redirecionamento da mesma se dá através de dois passos: a definição de um conjunto de estruturas de dados que descreva o conjunto de instruções e outras características do processador, e a implementação de algumas rotinas para tratar de especificidades da plataforma.

O cálculo de tempo não é previsto pela plataforma como um requisito na especificação dos dados do processador, mas pode ser desenvolvido como um *plug-in*. Para suportar este tipo de extensão, deve-se realizar algumas alterações nos componentes do gerador de código da LLVM que são independentes da arquitetura alvo, especificando interfaces que devem ser implementadas pelo gerador específico de cada arquitetura para permitir a simulação do comportamento do processador.

Não foi possível, no escopo deste trabalho, analisar a viabilidade da geração automática do cálculo de tempo a partir da especificação do processador em uma ADL (*Architecture Definition Language*).

Com o desenvolvimento da ferramenta `llflow`, pudemos comprovar a viabilidade do uso da LLVM para o desenvolvimento de ferramentas de análise de código em geral e análise de tempo em particular, bem como apontar estratégias para a criação deste tipo de ferramenta.

Para o desenvolvimento da análise de baixo nível, pode-se recomendar a abordagem utilizada em Engblom, Ermedahl e Altenbernd (1998), com alterações nos geradores de código da LLVM para realizar a *co-transformação* das informações levantadas na análise de alto nível em cada transformação realizada no código por otimizações realizadas pelo gerador.

Referências Bibliográficas

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques and Tools*. Reading, Massachusetts, USA: Addison-Wesley, 1988. ISBN 0-201-10088-6.

BARR, M. *Michael Barr's Embedded Systems Glossary*. julho 2007. Disponível em: <http://www.netrino.com/Publications/Glossary/>. Acesso em: 14 de julho de 2007.

ENGBLOM, J. *Worst-Case Execution Time Analysis for Optimized Code*. Dissertação (Master of Science in Computer Science) — Departament of Computer Systems of the Upsala University, Upsala, Sweden, 1997.

ENGBLOM, J. Why SpecInt95 should not be used to benchmark embedded systems tools. In: *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems*. Atlanta, USA: [s.n.], 1999. p. 96–103.

ENGBLOM, J. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. Tese (Doctor in Philosophy in Computer Systems Thesis) — Faculty of Science and Technology of the Upsala University, Upsala, Sweden, April 2002.

ENGBLOM, J. *Worst Case Execution Time Analysis*. feb 2002. Disponível em: <http://user.it.uu.se/~jakob/presentations/kth-wcetfeb2002.pdf>. Acesso em: 18 de novembro de 2007.

ENGBLOM, J.; ERMEDAHL, A.; ALTENBERND, P. Facilitating worst-case execution times analysis for optimized code. In: *Proceedings of the* 10^{th} *EuroMicro Workshop on Real-Time Systems*. Berlin, Germany: [s.n.], 1998.

ENGBLOM, J. et al. Worst case execution time analysis for embedeed real time systems. *Springer International Journal of Software Tools for Technology Transfer*, n. 14, 2001. Disponível em: <http://citeseer.ist.psu.edu/engblom00worstcase.html>.

ERMEDAHL, A.; GUSTAFSSON, J. Deriving annotations for tight calculation of execution time. In: *European Conference on Parallel Processing*. [S.l.: s.n.], 1997. p. 1298–1307.

ERMENDAHL, A. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. Tese (Doctoral Thesis) — Uppsala University, Teknisk-naturvetenskapliga vetenskapsområdet, Mathematics and Computer Science, Department of Information Technology, Univesity Library, Box 510, 75120, Uppsala, Sweden, 2003.

Free Software Fondation. *GCC - GNU Compiler Collection*. Free Software Foundation, outubro 2007. Disponível em: <http://gcc.gnu.org/>. Acesso em: 14 de outubro de 2007.

GUSTAFSSON, J.; BERMUDO, N.; SJöBERG, L. *Flow Analysis for WCET calculation*. Västerås, Sweden, April 2002.

KIRNER, R.; PUSHNER, P. Discussion of misconceptions about WCET analysis. In: *Proceedings of the 3rd International Workshop on Worst Case Execution Time Analysis (WCET'2003)*. [S.l.: s.n.], 2003. p. 61 – 64.

LATTNER, C. *LLVM Home Page*. maio 2008. Disponível em: <http://www.llvm.org/>. Acesso em: 11 de maio de 2008.

LATTNER, C. A. *LLVM: an infrastructure for multi-stage optimization*. Dissertação (Master of Science in Computer Science) — Graduate College of the University of Illinois, Urbana, Illinois, 2002.

LI, Y.-T. S.; MALIK, S. Performance analysis of embedded software using implicit path enumeration. In: *Workshop on Languages, Compilers & Tools for Real-Time Systems*. [S.l.: s.n.], 1995. p. 88–98.

LOKUCIEJEWSKI, P. et al. Tighter WCET estimates by procedure cloning. In: *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis*. Pisa, Italy: [s.n.], 2007. p. 23 — 28.

PETTERS, S. M.; ZADARNOWSKI, P.; HEISER, G. Measurements or static analysis or both? In: *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis*. Pisa, Italy: [s.n.], 2007.

RAMALINGAM, G. On loops, dominators and dominance frontiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v. 24, n. 5, p. 455 – 490, 2002. ISSN 0164-0925.

SCHULTZ, M. R. de O. *Geração Automática de Ferramentas de Inspeção de Código Para Processadores Especificados em ADL*. Dissertação (Mestrado em Ciência da Computação) — Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, Florianópolis, 2007.

SINGER, J. *Static Program Analysis based on Virtual Register Renaming*. Tese (Phd Thesis) — University of Cambridge, 2005.

SREEDHAR, V. C.; GAO, G. R.; LEE, Y.-F. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, v. 18, n. 6, p. 649–658, 1996.

WILHELM, R. et al. *The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools*. [S.l.], March 2007.