UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Emílio Wuerges

Um analisador de restrições de tempo real para compiladores redirecionáveis automaticamente

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Olinto José Varela Furtado, Dr.
(orientador)

Florianópolis, dezembro de 2008

Um analisador de restrições de tempo real para compiladores redirecionáveis automaticamente

Emílio Wuerges

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Frank Siqueira

Banca Examinadora

Olinto José Varela Furtado, Dr. (orientador)

Luiz Cláudio Villar dos Santos, Dr. (co-orientador)

Rômulo Silva de Oliveira, Dr.

Sandro Rigo, Dr.

Work it harder, make it better
Do it faster, makes us stronger
more than ever, Hour after
Our work is never over
- Daft Punk

À minha família.

Agradecimentos

Agradeço à todos que torceram e acreditaram em mim, em especial aos meus orientadores. Agradeço à minha família, principalmente meus pais, meus irmãos e meus avós, que me deram apoio nas horas em que eu mais precisei. Agradeço à CAPES, no âmbito do Programa Nacional de Cooperação Acadêmica, pelo custeio parcial da execução deste trabalho (Processo nº 0326054).

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

ADL	Linguagem de descrição de Arquitetura
AI	Interpretação Abstrata
BCET	Tempo de execução do melhor caso
DSE	Exploração do espaço de projeto
ERTC	Checador de restrições de tempo precoce
ILP	Programação Linear Inteira
IPET	Técnica de enumeração implícita de caminhos
ISA	Arquitetura do conjunto de instruções
ISS	Simulador do conjunto de instruções
WCET	Tempo de execução do pior caso

Resumo

A crescente demanda por sistemas embarcados com requisitos de tempo e de uso eficiente de energia, juntamente com o crescimento de arquiteturas heterogêneas, tem apontado para a necessidade de técnicas de compilação automaticamente redirecionáveis. Entretanto, compiladores redirecionáveis não tratam restrições de tempo real e as técnicas convencionais de análise do tempo de execução do pior caso (WCET) não são automaticamente redirecionáveis: métodos baseados em medições necessitam de penosa caracterização dinâmica dos processadores alvo, já técnicas estáticas e interpretação abstrata são executadas após a compilação, restringindo o conjunto de alvos suportados.

Este trabalho propõe uma técnica redirecionável para dotar o compilador de capacidade para checar antecipadamente as restrições de tempo real (ERTC) com o propósito de exploração do espaço de projeto. Esta técnica permite que a análise de restrições de tempo (mínimas, máximas e exatas) seja feita em tempo de compilação. Ela permite a detecção antecipada de combinações de restrições inconsistentes antes da geração dos executáveis, possibilitando um aumento na produtividade do projeto. ERTC é um complemento aos fluxos de projeto no estado da arte, que podem se beneficiar da detecção antecipada de infactibilidade e a exploração de processadores alvo alternativos, antes de os executáveis binários serem submetidos à análise justa do BCET e do WCET para o processador selecionado.

Abstract

The growing demand for time-constrained energy-efficient embedded systems and the rise of heterogeneous architectures are showing the need for automatically retargetable compilation techniques. On the one hand, retargetable compilers do not handle real-time constraints properly. On the other hand, conventional worst-case execution time (WCET) approaches are not automatically retargetable: measurement-based methods require time-consuming dynamic characterization of target processors, whereas static program analysis and abstract interpretation are performed in a post-compiling phase, being therefore restricted to the set of supported targets.

This paper proposes a retargetable technique to grant early real-time checking (ERTC) capabilities for design space exploration. The technique provides a general (minimum, maximum and exact-delay) timing analysis at compile time. It allows the early detection of inconsistent time-constraint combinations prior to the generation of binary executables, thereby promising higher design productivity. ERTC is a complement to state-of-the-art design flows, which could benefit from early infeasiblity detection and exploration of alternative target processors, before the binary executables are submitted to tight-bound BCET and WCET analyses for the selected target processor.

Capítulo 1

Introdução

O enorme esforço de pesquisa para obtenção de estimativas justas (do inglês *tight-bound*) do tempo de execução do pior caso (WCET) para processadores com características dinâmicas trouxe a ascensão de técnicas eficientes (Cousot e Cousot 1977), que já estão disponíveis para uso industrial, talhadas de acordo com um portfólio que contém os processadores mais populares, cujo o foco é maximizar a desempenho média. Tais esforços resolveram o problema importantíssimo de se obter estimativas WCET justas para os processadores escolhidos.

Até onde os processadores suportados puderem lidar com os requisitos de um dado domínio de aplicações, esta continuará sendo uma maneira eficiente e pragmática de resolver o problema, já que os fornecedores dessas ferramentas se comprometem a manter portfólios atualizados dos processadores mais populares e (com sorte) mais adequados.

Entretanto, não é uma tarefa trivial selecionar o melhor processador (e seu subsistema de memória) para uma dada aplicação. Requisitos além de desempenho e previsibilidade podem se tornar aspectos importantes. Por exemplo, pode acontecer que as restrições de tempo real possam ser satisfeitas rodando a aplicação em um processador mais simples e que usa energia de maneira mais eficiente. Também pode ser que seja mais adequado utilizar *scratchpads* em vez de caches. Então, para aplicações em que existem diversos caminhos de otimização, a exploração do espaço de projeto (DSE) é mandatória e o projetista do sistema embarcado deve poder experimentar vários processadores candidatos nas etapas mais prematuras do projeto, com objetivo posterior de selecionar um processador e, posteriormente, talhar uma análise WCET justa mais elaborada.

Por outro lado, mesmo em um cenário em que inicialmente a análise das restrições de tempo não é rígida, a exploração do espaço de projeto pode se beneficiar da análise WCET para guiar o seu desenvolvimento e evitar que este sucumba a um estágio em que as restrições de tempo possam se tornar um problema. Nestes cenários, um comprometimento entre a velocidade e a qualidade da estimativa pode tornar viável a checagem antecipada das restrições de tempo. Afinal, não é prático explorar o espaço de projeto sem suporte de compilação para os processadores candidatos.

Embora alguns compiladores convencionais já possuam portes para a maioria dos processadores de uso geral e de processamento de sinal (GPPs e DSPs), a exploração de alternativas de projeto é inerentemente limitada ao conjunto de processadores suportados pela infra-estrutura de compilação utilizada, já que o redirecionamento manual para um processador candidato seria uma tarefa inaceitavelmente penosa.

Além disso, em aplicações onde a flexibilidade do projeto é tão importante quanto a economia de energia (*low-power*), a necessidade de se ter processadores customizados não pode ser ignorada. Os processadores específicos para domínios de aplicações (al. 2008) exigidos pela computação embarcada dificilmente se encaixariam num paradigma convencional baseado em portfólio. Assim, para não se limitar a um portfólio, e tomar vantagem de processadores customizados com maior eficiência energética ou para lidar com arquiteturas heterogêneas com vários processadores, ferramentas automaticamente redirecionáveis crescem em importância.

Elaborar um compilador completamente redirecionável é uma tarefa formidável. Apesar dos grandes avanços das pesquisas (Leupers e Marwedel 2001) (Mishra e Dutt 2008) e de algumas ferramentas comerciais maduras (Hoffmann et al. 2005) (Gonzalez 2000), ainda não ficou claro até que ponto compiladores automaticamente redirecionáveis podem competir com a qualidade do código dos compiladores redirecionados manualmente. Neste contexto, propomos uma técnica de análise redirecionável que se encaixa dentro de um compilador, analisando sua representação intermediária, já otimizada.

Os compiladores redirecionáveis convencionais não oferecem garantias de tempo real. Do lado oposto, técnicas WCET (Wilhelm et al. 2008) tratam as restrições de tempo real, mas não são automaticamente redirecionáveis: métodos baseados em medições requerem a (penosa) caracterização dinâmica do processador candidato, enquanto a análise

estática e a interpretação abstrata são efetuadas em uma etapa após a compilação e, portanto, ficam restritas ao conjunto de processadores suportados. Algumas técnicas WCET tentaram ser independentes do alvo (Wilhelm et al. 2008) (Ermedahl 2003), mas tiveram que se apoiar em simuladores de conjuntos de instruções (ISSs), cujo uso é tanto inseguro quanto penoso.

Por outro lado, mesmo que os requisitos de tempo real de várias aplicações importantes possam ser capturados por atrasos máximos, não podemos desconsiderar que várias aplicações em sistemas embarcados também possuem especificações de requisitos com atrasos mínimos e exatos (por exemplo, protocolos de tratamento de I/O em sistemas reativos).

Estas razões nos motivaram a desenvolver uma técnica de análise de restrições de tempo que permite a detecção antecipada de combinações inconsistentes de restrições antes mesmo da geração dos binários executáveis, com o objetivo de aumentar a produtividade e explorar mais amplamente o espaço de projeto.

A técnica proposta redireciona automaticamente os aspectos dependentes da microarquitetura do *back-end* de um compilador para fornecer a checagem antecipada das restrições de tempo real (ERTC). São tratadas múltiplas restrições de tempo, impostas simultaneamente e de tipos distintos (requisitos de atraso máximo, mínimo e exato são capturados). Além disso, com o temor de sacrificar a velocidade da compilação, não houve tentativas de integrar técnicas estáticas baseadas em programação linear (ILP). Assim, o ERTC proposto foi projetado como um complemento aos fluxos de projeto no estado da arte, que podem se beneficiar da detecção antecipada da infactibilidade e da possibilidade de se utilizar processadores alvos alternativos, mesmo que após esta etapa os binários executáveis sejam submetidos à análise de BCET e WCET que resulte em estimativas mas justas para o processador alvo escolhido.

O restante desta dissertação é organizado da seguinte maneira. No capítulo 2, são revisados os compiladores convencionais e os redirecionáveis, bem como a análise WCET. Uma prévia da técnica proposta é mostrada, no capítulo 3, através de um exemplo simples. Nos capítulos seguintes, a técnica é formalizada. O capítulo 4 apresenta a representação adotada, formaliza a modelagem da temporização proposta e da especificação das restrições, e o capítulo 5 mostra os algoritmos base da técnica. Os resultados experimentais, reportados na seção 5.2, apresentam evidência da eficácia e eficiência da técnica proposta.

Extensões planejadas da implementação e tópicos para pesquisa futura são discutidos no capítulo 6, junto com as conclusões obtidas.

Capítulo 2

Revisão Bibliográfica

2.1 Infra-estrutura clássica de compilação

O GCC (Stallman 2002) é a infra-estrutura de compilação referência do mercado. A sua natureza livre possibilita sua extensão e seu uso em larga escala. O GCC possui *frontends* para diversas linguagens de programação, incluindo C e C++, e *back-end* para as mais diversas arquiteturas. É utilizado para compilar vários programas dos quais dependemos todos os dias, como o Linux, o Apache, os BSDs e a si próprio, bem como outras ferramentas da GNU. Apesar de sua eficiência e aceitação, o GCC apóia-se numa infra-estrutura complexa, projetada para arquiteturas de 32 bits com registradores de propósito geral e palavra de 8 bits (Stallman et al. 1999).

Uma infra-estrutura de compilação alternativa, muito promissora, é fornecida pela Low Level Virtual Machine (Lattner e Adve 2004). A LLVM prima por ter um projeto mais simples e mais fácil de estender e adaptar. Por ter um *framework* comum para a representação intermediária do programa em diferentes níveis de abstração e combinando este com um otimizador poderoso, a LLVM se destaca como uma alternativa interessante para se construir compiladores redirecionáveis automaticamente, e *livres*.

Ambos o GCC e a LLVM são compatíveis com o pacote GNU Binutils (Pesch e Osier 1993), para o qual estão disponíveis utilitários binários redirecionáveis automaticamente, e *livres* (Baldassin et al. 2007).

2.2 Compiladores redirecionáveis

Para a verificação das restrições de tempo, é necessário que a arquitetura do conjunto de instruções (ISA) e a temporização da microarquitetura sejam descritas pela linguagem (Li et al. 2007). A maneira com que estas informações são descritas determina como a análise WCET pode ser realizada.

Algumas linguagens aproveitam estas informações para a geração automática de simuladores de ISA com precisão de ciclo. Os pacotes baseados em ADLs como nML (Fauth, Praet e Freericks 1995), LISA (Ceng et al. 2005) e ISDL (Hanono e Devadas 1998), bem como os pacotes de projeto de ASIPs como Mescal (Qin e Malik 2003) e PEAS-III (Kobayashi et al. 2001) provêm montadores redirecionáveis, ligadores redirecionáveis e algumas vezes suporte parcial para seleção de instruções de maneira redirecionável. Desta maneira, torna-se possível a análise WCET (como em (Li et al. 2007)), mas ao custo de executar o programa no simulador gerado, tornando a tarefa penosa, além de esta análise não ser exaustiva.

Para verificar restrições temporais durante a compilação, sem a necessidade de um simulador auxiliar, latências devem ser extraídas automaticamente do modelo do processador e anotadas na representação do programa. Entretanto, algumas ADLs (como LISA) não descrevem instruções individualmente, mas como uma composição de operações. Isto torna difícil inferir as latências a partir de uma descrição da arquitetura alvo nesta ADL.

A ADL EXPRESSION (Halambi et al. 1999) modela o *pipeline* através de um grafo, chamado grafo de execução, onde os nodos são os recursos do hardware e as arestas são as dependências para a execução. Este grafo é adequado para formular o problema ILP para utilizar a IPET(Li et al. 2007), os quais serão esclarecidos na seção seguinte.

A técnica proposta neste trabalho, como descrito na sessão 5, necessita de uma forma de descobrir a latência relativa entre as instruções. Para isto, foi adotado o modelo descrito em (Carlomagno, Santos e Santos 2007), que usa a ADL
ArchC (Rigo et al. 2004) para descrever as latências entre pares de instruções. Neste trabalho, as latências são descritas através de um conjunto onde cada elemento é uma estrutura contendo identificadores do par de instruções e a latência relativa entre esse par para cada operando comum. Um exemplo de como isto é feito pode ser visto na figura 2.1.

```
ldub_reg.set_property(latency,(sll_imm,rs1,1));
ldub_reg.set_property(latency,(and_imm,rs1,1));
ldub_reg.set_property(latency,(ld_imm,rs1,1));
ldub_reg.set_property(latency,(stb_reg,rd,1));
ldub_reg.set_property(latency,(stb_reg,rs1,1));
```

Figura 2.1: Fragmento de código da descrição ArchC para o SPARC.

2.3 Análise WCET

O objetivo principal da análise de tempo de execução é verificar se uma tarefa satisfaz as restrições de tempo real de uma aplicação. Tal análise pode exibir diferentes níveis de complexidade, dependendo das simplificações aceitas no modelo de abstração do hardware adotado (Kirner e Puschner 2003) (Petters, Zadarnowski e Heiser 2007).

Existem duas maneiras principais de executar tal análise: estaticamente e dinamicamente (Engblom 2002) (Ermedahl 2003) (Wilhelm et al. 2008). Na maneira estática, o tempo de execução é calculado a partir da análise do código (possivelmente anotado) sem precisar ser executado ou simulado. Esta maneira envolve análise do fluxo de controle (alto nível), análise do hardware (baixo nível), e o cálculo do WCET em si, que leva em conta o resultado das análises anteriores. Por outro lado, a maneira dinâmica é baseada na medição do tempo gasto executando o código (ou um segmento do mesmo), para um conjunto de estímulos de entrada rigidamente escolhido, seja rodando o código no próprio processador alvo ou simulando sua execução em um modelo do hardware. Um *survey* recente (Wilhelm et al. 2008) esclarece este ponto e fala das principais ferramentas disponíveis (tanto comerciais quanto acadêmicas).

A maioria das técnicas de análise WCET *tight-bound* se baseiam principalmente em dois conceitos: A enumeração implícita dos caminhos (IPET) (Li, Malik e Inc 1997) e a teoria de interpretação abstrata (AI) (Cousot e Cousot 1977). A IPET trata da formulação do problema em programação linear (ILP) onde o custo do percurso pelos blocos básicos na execução é expresso por um sistema de polinômios. É possível com IPET modelar caches e *pipelines*, mas tal extensão leva a tempos de análise muito altos.

Uma execução de um programa pode ser representada por uma seqüência

de contextos, que representam os estados das variáveis do programa. Normalmente, para cada conjunto de entrada existe um fluxo de execução diferente, e por esse motivo, uma seqüência de contextos diferentes. Já utilizando AI, as variáveis têm seu valor descrito de forma abstrata, permitindo que todas as execuções do programa sejam representadas por uma única seqüência de contextos. De posse dos intervalos de valores que as variáveis de controle podem assumir, pode-se inferir o número máximo e mínimo de iterações nos laços. Da mesma maneira, os endereços das variáveis podem ser descritos de maneira abstrata e seus intervalos podem ser utilizado para traçar restrições para o comportamento da hierarquia de memória.

Com o objetivo de reduzir a complexidade da IPET, AI pode ser utilizada para análises de valor, cache e *pipeline*, deixando a IPET apenas para a análise dos caminhos (Theiling e Ferdinand 1998). Uma combinação de ILP e AI é utilizada em uma ferramenta para uso industrial (Wilhelm et al. 2008).

Em (Ermedahl 2003), uma arquitetura foi proposta para uma ferramenta de análise WCET, que depende de estimativas extraídas de um modelo do hardware. Ela propõe três métodos para computar o fluxo de execução mais longo. O primeiro é baseado em ILP; o segundo depende de informação do fluxo do programa para quebrar os caminhos em sub-caminhos com o objetivo de reduzir a complexidade do problema; o último método primeiramente identifica o caminho mais longo para posteriormente permitir análise em tempo quase linear. Apesar de o autor enumerar vários aspectos do hardware que influenciam a temporização e dizer que a arquitetura proposta facilita a customização da ferramenta para processadores distintos, a técnica não é capaz de determinar diretamente o WCET sem utilizar um ISS com precisão de ciclos (ela usa simuladores dirigidos por *trace* previamente existentes para capturar a temporização do programa).

O importante problema de computar os caminhos falsos antes da análise WCET é tratado em (Gustafsson, Ermedahl e Lisper 2006), por exemplo, onde AI é empregada para evitar blocos básicos que são inalcançáveis durante a execução do programa.

O trabalho em (Leung, Palem e Pnueli 2002) define uma linguagem para anotar o programa fonte com restrições de tempo e propõe uma técnica para manter tal anotação em uma representação intermediária. Apesar de não serem apresentados resultados experimentais, estre trabalho mostra como restrições podem ser exploradas por algoritmos

de escalonamento de código convencionais para aumentar a factibilidade. Uma conclusão importante retirada deste trabalho é que as restrições anotadas podem ser usadas não somente para avaliar a factibilidade do código pré-otimizado, mas também podem ser exploradas para guiar a otimização em direção da factibilidade. Apesar da técnica proposta neste trabalho ser apresentada analisando código já otimizado, nada impede que seja utilizada em fases anteriores do processo de compilação. Desta maneira, poderia ser utilizada como métrica para uma fase de otimização.

2.4 Discussão

Sumarizando, compiladores redirecionáveis podem aumentar a produtividade de projeto, mas não tratam BCET e WCET. Por outro lado as técnicas de análise BCET/WCET justas não são adequadas para exploração do espaço de projeto (apesar da eficiência da IPET e AI) porque os modelos que as suportam ainda são muito complexos para serem invocados em tempo de compilação. Por este motivo, desenvolvemos uma técnica com as seguintes características:

- Extração das latências de uma microarquitetura e do subsistema de memória a partir de uma descrição em uma ADL.
- Especificação independente de linguagem de diversos tipos de requisitos de tempo real (restrições de atraso mínimo, máximo e exato).
- Pré-verificação da factibilidade das restrições de tempo durante a compilação.
- Modelagem da temporização voltada para DSE visando disponibilizar garantias de tempo real preliminares.
- Ser complementar às ferramentas de análise de tempo de execução que fazem estimativas justas de BCET/WCET no estado da arte.

Capítulo 3

Um exemplo ilustrativo

Para ilustrar o problema alvo e a solução proposta nesta dissertação, apresentaremos nesta seção um exemplo simples.

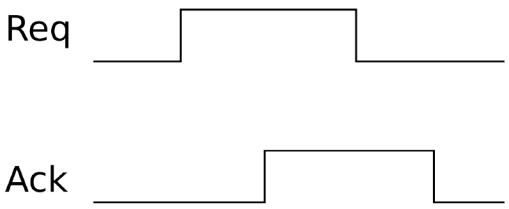

Figura 3.1: Operação de *handshake*

A figura 3.1 mostra o exemplo considerado, uma aplicação onde operações de I/O precisam de um mecanismo simples de *handshake* para sincronizar a leitura (ou escrita) de um registrador de I/O mapeado em memória que armazena dados capturados por um sensor (ou dados para serem tratados por um atuador).

Suponha que um processo implementa a requisição de leitura por meio de um procedimento chamado `block`, como mostrado na figura 3.2 (onde a leitura e a escrita do registrador foram omitidas por simplicidade).

Agora vamos assumir que a especificação do sistema impõe as seguintes restrições de tempo no mecanismo de *handshake*:

- Restrição 1: Após o programa enviar uma requisição de escrita, o periférico deve responder com uma confirmação em no máximo 12 ciclos mais tarde.

```
1 void block(unsigned char * req,
2   unsigned char * ack)
3 {
4   /* request activation */
5   *req = 0x01;
6   /* some code would be here (omitted) */
7   while(!(*ack));
8  /* acknowledgement activated */
9 }
```

Figura 3.2: Código fonte original

- Restrição 2: O programa deve receber a *flag* de confirmação do periférico pelo menos 3 ciclos depois de enviar a requisição de leitura.

A temporização das instruções determinará se tais restrições de tempo poderão ser satisfeitas. Para checar se o código compilado para uma determinada microarquitetura de fato as satisfaz, devemos primeiramente instrumentar o código com restrições de tempo.

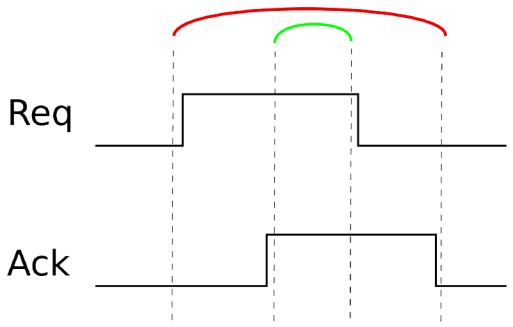

Figura 3.3: Operação de *handshake* com as restrições

A figura 3.3 ilustra o código instrumentado, apresentado na figura 3.4. As linhas 4, 7, 9 e 12 possuem diretivas de compilação que permitem a inserção de um conjunto de restrições equivalentes às especificadas. O primeiro elemento dentro da diretiva especifica se ela representa um ponto inicial ou final de uma restrição dada, o segundo associa uma etiqueta a ela. Uma diretiva que representa um ponto inicial possui dois elementos adicionais representando, respectivamente, o tipo da restrição (máximo ou mínimo) e o seu valor

(expresso em número de ciclos).

A restrição 1 é capturada inserindo um par de marcadores (com a mesma etiqueta `constr1`) no código fonte: um imediatamente antes da ativação do sinal da requisição (`req`, linha 4), e outro logo após a ativação do sinal de reconhecimento (`ack`, linha 12). De maneira semelhante, a restrição 2 é codificada por outro par de marcadores (etiquetados como `constr2`): o marcador do começo fica imediatamente após a ativação do `req` (linha 7) e o marcador do final fica imediatamente antes da ativação do `ack` (linha 9).

```
1  void block(unsigned char * req,
2    unsigned char * ack)
3  {
4    asm volatile("begin,constr1,MAX,12;");
5    /* request activation */
6    *req = 0x01;
7    asm volatile("begin,constr2,MIN,3;");
8    /* some code would be here (omitted)*/
9    asm volatile("end,constr2;");
10   while(!(*ack));
11   /* acknowledgement activated */
12   asm volatile("end,constr1;");
13 }
```

Figura 3.4: O código fonte instrumentado

Depois da instrumentação, o código é compilado para uma representação intermediária, na qual são anotadas as latências das instruções de uma dada microarquitetura (por exemplo, o PowerPC 405 ou o SPARC V8).

A figura 3.5 mostra esquematicamente a representação intermediária do procedimento `block` da figura 3.4 quando redirecionada para o PowerPC 405. Cada retângulo exterior representa um bloco básico, cada retângulo interior (arredondados) representa uma instrução. Os círculos denotam os pólos dos grafos ou subgrafos: O preto representa as fontes e o preto e branco os sumidouros. O losango representa o efeito de um desvio condicional (`branch_equals ack, 0x01`).

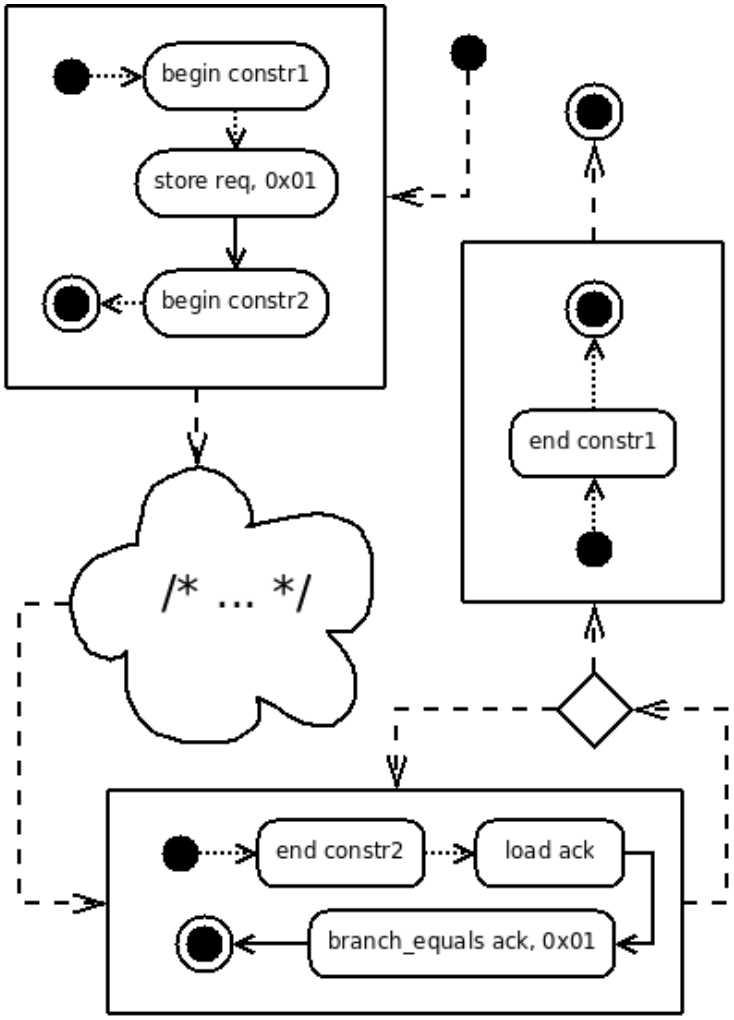

Figura 3.5: Representação intermediária para o PowerPC 405

Para simplificação (mas sem perda de generalidade), vamos assumir que todas as instruções tem latências de um ciclo. As setas com traço sólido na figura 3.5 denotam as latências das instruções, já as setas tracejadas denotam apenas a seqüência das operações (com latência zero).

Há dois pares de pseudo-instruções que denotam o começo e o fim das restrições `constr1` e `constr2`, que correspondem as diretivas de compilação inseridas no código fonte. Estas pseudo-instruções servirão como âncoras para a checagem das restrições temporais; elas serão os vértices iniciais e finais cujos caminhos mínimos ou máximos serão calculados.

Para checar por restrições do tipo MIN e MAX, usamos algoritmos de caminhos máximos (adaptado) e mínimos (veja a seção 5.2). Restrições exatas são checadas

como uma combinação de restrições do MIN e MAX (veja seção 4.2).

Ao analisar o grafo, nossa ferramenta protótipo emite um relatório das restrições violadas. Para cada restrição violada, tal relatório discrimina seu tipo (MAX ou MIN), o número de ciclos alocados, o número de ciclos gastos e a folga (diferença entre o valor especificado e o número de ciclos gastos) existente.

A importância deste relatório é que não é mais necessário testar o software para ver se ele extrapola as restrições de tempo. Agora, o teste do software deve apenas verificar se as restrições foram bem especificadas. O relatório também permite ajustar o software pra satisfazer as restrições que previamente seriam infactíveis. Além disto, o tamanho das folgas pode dar uma idéia da portabilidade do software. Restrições muito justas podem não ser satisfeitas se o programa for compilado para uma outra arquitetura.

Capítulo 4

Representação do programa e Modelagem da temporização e das restrições

Este capítulo propõe uma modelagem para a temporização das instruções e a especificação das restrições de tempo. As noções principais formalizadas abaixo nos permitirão apresentar um algoritmo compacto, porém detalhado da técnica proposta (veja o capítulo 5). O modelo de temporização integra as propriedades do processador alvo (latências de instruções relativas aos *hazards* do *pipeline*), o efeito do escalonamento das instruções e os efeitos da execução de desvios condicionais e de iterações em laços.

4.1 Representação do programa

Esta seção especifica formalmente a estrutura dos programas que será considerada na entrada do fluxo de compilação e da representação intermediária suportada pelas técnicas e ferramentas propostas nesta dissertação [1].

Um programa consiste de um conjunto de procedimentos ou funções $P = \{p_1, p_2, \ldots, p_n\}$. Cada procedimento p_i é modelado por um *grafo de fluxo de controle* $CFG = (B, F)$ (Aho, Sethi e Ullman 1986), no qual cada vértice $b \in B$ é um bloco básico e cada

[1] A qual é deliberadamente similar a da LLVM.

aresta $f \in F$ representa o fluxo de execução entre os blocos básicos. Cada bloco básico é modelado por um *grafo de fluxo de dados* $DFG = (V, E)$, onde cada vértice $v \in V$ representa uma instrução e onde cada aresta $(u, v) \in E$ representa uma dependência de dados.

Como o código passa por diferentes níveis de otimização, a representação do programa é capturada em formatos intermediários distintos. Uma *representação intermediária* (IR) é uma representação equivalente do programa onde alguns dos elementos da representação original podem ser descartados desde que eles já tenham sido utilizados pelas otimizações anteriores. Por exemplo, depois do escalonamento das instruções, uma IR pode representar as instruções como um conjunto ordenado, já que o escalonamento garante que esta ordem satisfaz a relação de precedência no DFG.

Dado um bloco básico $b_i \in B$ e seu $DFG = (V_i, E_i)$ associado, seja I_i um conjunto linear ordenado obtido do conjunto V_i tal que a ordem parcial E_i é obedecida. A IR adotada na entrada do *back-end* (a partir de agora chamada somente de IR) consiste dos CFGs que representam cada procedimento e as instruções (ordenadas) de cada bloco básico, como formalizado abaixo.

A *representação intermediária* de um programa é definida por uma tupla $IR = (P, C, I)$, onde:

- $P = \{p_k : p_k$ é um procedimento ou função $\}$;
- $C = \{\forall p_k \in P : (B_k, F_k)\}$;
- $I = \{\forall b_i \in B \wedge \forall (B, F) \in C : I_i\}$.

4.2 Especificação das restrições de tempo real

Restrições de tempo real são especificadas por meio de um par de delimitadores de escopo, chamados *fonte* e *sumidouro* dentro de um BB. Como as restrições podem ser impostas através dos BBs, um delimitador deve identificar não somente a instrução para a qual ela está apontando, mas também o BB dono daquela instrução. Então, esta noção pode ser formalizada como segue.

Uma *restrição de tempo real* será representada por uma tupla

$tc = (b_i, d_i, b_j, d_j, t, \tau)$, onde:

- $b_i \in B$ é o BB que contém a instrução apontada pelo delimitador fonte;
- $d_i \in Z^+$ é a posição em I_i para a qual o delimitador fonte está apontando;
- $b_j \in B$ é o BB que contém a instrução apontada pelo delimitador sumidouro;
- $d_j \in Z^+$ é a posição em I_j para a qual o delimitador sumidouro está apontando;
- $t \in Z$ é o número de ciclos especificados entre a fonte e o sumidouro.
- $\tau \in \{\text{MIN, MAX}\}$ é o tipo da restrição que especifica t como um valor mínimo ou máximo.

É importante notar que um requisito de tempo real pode especificar um número exato de ciclos entre dois eventos. Isto pode ser capturado na nossa modelagem pela atribuição de um par de restrições de tempo $tc1 = (b_i, d_i, b_j, d_j, t, MIN)$ e $tc2 = (b_i, d_i, b_j, d_j, t, MAX)$, desta maneira o modelo apresentado fica habilitado a tratar a especificação de intervalos de tempo exatos.

4.3 Especificação da temporização da microarquitetura

Para garantir a redirecionabilidade automática, temos que capturar a temporização das instruções de uma descrição da microarquitetura alvo (por exemplo de um modelo numa ADL). A propriedade de temporização mais importante é formalizada abaixo.

A *latência entre duas instruções* u e v, denominada $\lambda(u, v)$, é o tamanho do intervalo, em número de ciclos, necessário para garantir que o dado produzido por u ficará disponível antes que possa ser consumido por v.

Adicionalmente o *pipeline* e o subsistema de memória podem contribuir para a latência, como descrito nos parágrafos seguintes.

A *latência do pipeline entre duas instruções* u e v, denominada $\lambda_p(u, v)$, é o número mínimo de ciclos entre o momento da produção de um valor por u na saída de um estágio do *pipeline* e o momento em que o valor é disponibilizado na entrada do estágio onde é consumido por v.

Para modelar a contribuição do subsistema de memória para a latência, precisamos distinguir as classes de instruções e então quantificar o impacto dos acessos de memória. Seja $\tau : I \rightarrow \{store, load, reg\}$ a função *tipo*, que mapeia cada instrução $v \in I$ em uma instrução tipo $\tau(v)$, onde *load* e *store* denotam instruções lendo e escrevendo operandos na memória, respectivamente, e reg denota instruções cujos operandos estão sempre em registradores.

Tempos de acesso na memória podem depender do endereço acessado e do tipo de acesso, que pode ser distinguido como segue. Seja ϵ_v o endereço efetivo do dado referenciado pela instrução v tal qual $\tau(v) \neq reg$. Sejam $R(\epsilon)$ e $W(\epsilon)$ os números de ciclos gastos lendo e escrevendo, respectivamente, em um dado endereço ϵ. A latência entre as instruções u e v pode ser obtida da seguinte forma:

$$\lambda(u,v) = \begin{cases} \lambda_p(u,v) & \text{if } \tau(u) = reg \\ R(\epsilon_u) + \lambda_p(u,v) & \text{if } \tau(u) = load \\ W(\epsilon_u) + \lambda_p(u,v) & \text{if } \tau(u) = store \end{cases}$$

Onde $\lambda_p(u,v)$ é constante para dado *pipeline*, $R(\epsilon_u)$ e $W(\epsilon_u)$ variam dependendo se o endereço ϵ_u é mapeado para um *scratchpad*, um registrador de I/O mapeado em memória, ou a *cache*. Já que no último caso, $R(\epsilon_u)$ e $W(\epsilon_u)$ variam dinamicamente, eles não podem ser sempre inferidos a partir da descrição do sistema e, portanto, requerem um modelo do comportamento dinâmico da *cache*. Já que tal modelo é esperado para retornar uma estimativa precisa do número de ciclos gastos no acesso à memória, ele pode ser chamado de um *oráculo da memória*. Como o objetivo é a exploração do espaço de projeto, um oráculo de memória não pode ser complexo demais; uma abordagem pragmática seria fornecer oráculos de memória selecionáveis, com complexidades distintas para permitir um compromisso entre precisão e tempo de análise.

Seja $\phi : I \rightarrow Z^+$ a função de *emissão* que mapeia cada instrução $v \in I$ para um passo de emissão $\varphi(v)$. O *número de ciclos do intervalo* u e v, como resultado da emissão dinâmica, denominado $\varphi_{(u,v)}$, é obtido por $(\varphi(u) - \varphi(v)) - 1$.

Cada instrução pode contribuir para o tempo de execução com um número de ciclos de atraso no intervalo $[0, \lambda(u,v)]$, como resultado de um *hazard* de dados, dependendo de quão longe as instruções u e v forem emitidas, como formalizado abaixo.

Dada uma instrução v que é dependente de dados em u, o *número de ciclos de atraso* entre u e v, denominado $\sigma(u, v)$, é dado por:

$$\sigma(u,v) = \begin{cases} 0 & \text{if } \varphi_{(u,v)} \geq \lambda(u,v) \\ \lambda(u,v) - \varphi_{(u,v)} & \text{if } \varphi_{(u,v)} < \lambda(u,v) \end{cases}$$

Para obter estimativas rápidas para BCET e WCET, assumimos que a estrutura do fluxo de controle é preservada em tempo de execução. Esta premissa pessimista tem duas conseqüências pragmáticas: 1) WCETs podem ser calculados sem a necessidade de enumerar os cenários de execução; 2) Tempos de execução podem ser medidos relativamente às fronteiras dos BBs, como segue.

Para computar o número de ciclos gastos por uma seqüência de instruções (sobrepostas) contidas em um BB, precisamos considerar o resultado do escalonamento do código. Seja $\phi : V \rightarrow Z^+$ a função *escalonamento* que mapeia cada instrução $v \in V$ em um único momento de tempo $\phi(v)$ dentro do escopo de um BB, que determina o ciclo em que a busca da instrução é iniciada. Seja α_v o endereço da instrução v e seja $R(\alpha_v)$ o atraso da busca.

A contribuição incremental de uma nova instrução emitida v para o tempo de execução do programa é o número de ciclos de instruções que não se sobrepõe, o que compreende os ciclos gastos buscando v e, possivelmente, os ciclos gastos esperando por dados produzidos por alguma instrução precedente u. Esta noção pode ser formalizada como segue.

Dado um BB e seu DFG (V, E), a *latência relativa* até a instrução x contida no BB, denominada λ_x, é obtida por:

$$\lambda_x = \sum_{v=\phi^{-1}(0)}^{y} (R(\alpha_v) + \max_{(u,v)\in E} \sigma(u,v))$$

4.3.1 Combinando a temporização com a representação do programa

Até este ponto nossa modelagem captura propriedades do processador e coloca-as na representação do programa. Para realizar as análises BCET/WCET, precisamos considerar os efeitos das iterações em laços, como segue.

Assumimos que os passos de compilação que levam até a IR foram restringidos para forçar uma topologia do CFG exibindo uma *estrutura de aninhamento*, i.e., dados dois laços I_i e I_j, somente uma das condições permanece:

- l_i e l_j são inteiramente disjuntos;
- l_i é inteiramente contido em l_j;
- l_j é inteiramente contido em l_i.

Como resultado, o comportamento dos laços pode ser capturado como segue:

Dado um BB $b_i \in B$, seja L_i o conjunto de laços aninhados contendo b_i. Seja n_l o número de iterações de um laço $l \in L_i$. Assim o número de invocações N_i de b_i é dado por:

$$N_i = \begin{cases} 1 & \text{if } L_i = \emptyset \\ \prod_{l \in L_i} n_l & \text{if } L_i \neq \emptyset \end{cases}$$

Agora podemos combinar as propriedades do processador e as propriedades dos laços. A média do tempo gasto na execução das instruções de um BB (possivelmente contido num aninhamento de laços) é o tempo gasto na execução das instruções do BB (tomando as latências e os ciclos de atraso em consideração como discutido na seção 4.3) multiplicado pelo seu número de invocações, como formalizado a seguir:

Dado um BB b_i e seu DFG (V_i, E_i), a latência de b_i, denominada $\lambda(b_i)$, é computada como:

$$\lambda(b_i) = N_i \times \sum_{v \in V_i} (R(\alpha_v) + \max_{(u,v) \in E_i} \sigma(u, v))$$

Agora podemos generalizar esta noção para um caminho no CFG.

Dado um caminho $\pi = (b_0, b_1, b_2, ..., b_k, ..., b_n, b_{n+1})$ em um CFG, a *latência entre* b_0 e b_{n+1} através de π é $\lambda(\pi) = \sum_{k=1}^{n} \lambda(b_k)$. Os vértices iniciais e finais são deliberadamente excluídos do caminho π no cálculo da latência por razões que ficam claras na seção 4.3.2.

4.3.2 Cálculo dos valores dos atrasos

Uma vez que estas latências são capturadas na IR, podemos agora definir a noção de atraso máximo, que é utilizada para checar se uma restrição de tempo é satisfeita pelo programa em sua execução no processador alvo.

Antes de formalizar estas noções, serão ilustrados todos os cenários do fluxo de controle possíveis nas figuras 4.1 e 4.2. Assuma que os delimitadores fonte e sumidouro de uma restrição de tempo apontam para instruções x e y, respectivamente. Lembre que λ_x e λ_y denotam o número de ciclos gastos desde a primeira instrução do BB até as instruções x e y, respectivamente. Lembre que $\lambda(b_i)$ denota o número de ciclos gastos no BB b_i.

A figura 4.1 mostra as duas possibilidades quando ambas as instruções residem no mesmo BB. A figura 4.1a corresponde a uma restrição imposta dentro do escopo de um BB, já a figura 4.1b captura uma restrição imposta através de um auto-laço.

A figura 4.2 distingue dois cenários, quando as instruções apontadas pelos delimitadores pertencem a BBs distintos, b_i e b_j. Enquanto a figura 4.2a reflete uma restrição imposta através de um caminho de acordo com o fluxo do programa, a figura 4.2b retrata uma restrição aplicada através de um ciclo no CFG, isto é, através do corpo de um laço no código. Na figura4.2, $\lambda(b_i, b_j)$ denota o valor de uma latência medida entre b_i e b_j, num caminho que começa com b_i e termina com b_j.

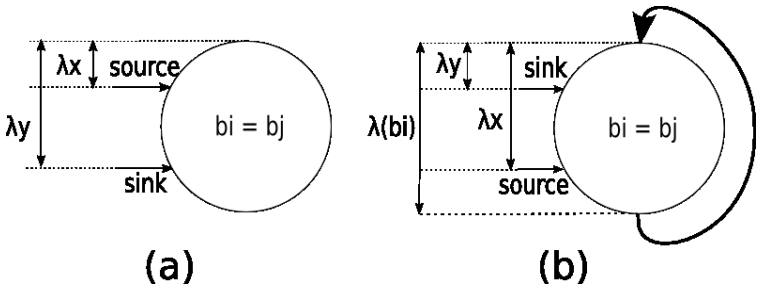

Figura 4.1: Delimitadores no mesmo BB

Com a ajuda das figuras 4.1 e 4.2, será agora formalizada uma noção crucial para a análise. Sejam b_i e b_j os BBs para os quais I_i e I_j denotam seus conjuntos ordenados de instruções. Dadas as instruções $x \in I_i$ e $y \in I_j$, as quais são apontadas respectivamente pelos delimitadores fonte e sumidouro de uma dada restrição de tempo, o *atraso calculado* entre x e y, denominado $\delta(x, y)$, é obtido da maneira seguinte:

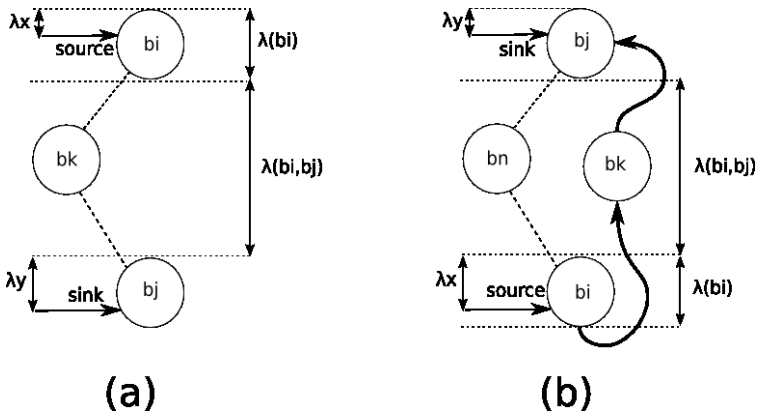

Figura 4.2: Delimitadores em BBs distintos

$$\delta(x,y) = \begin{cases} \lambda_y - \lambda_x & \text{if } b_i = b_j \wedge \lambda_x \leq \lambda_y \\ \lambda(b_i) + \lambda_y - \lambda_x & \text{if } b_i = b_j \wedge \lambda_x > \lambda_y \\ \lambda(b_i) - \lambda_x + \lambda(b_i, b_j) + \lambda_y & \text{if } b_i \neq b_j \end{cases}$$

Note que a primeira e segunda cláusulas representam, respectivamente, os cenários das figuras 4.1a e 4.1b. Observe que a terceira cláusula representa ambos os cenários da figura 4.2.

4.3.3 Estimando os limites do BCET e do WCET

Antes de poder avaliar os atrasos mínimos e máximos entre duas instruções, temos que considerar cada variável em $\delta(x, y)$. Primeiramente, consideramos os BB e as latências dos caminhos para posteriormente discutir como cercar a latência das instruções.

Seja ${n_l}^{max}$ (${n_l}^{min}$) o número máximo (ou mínimo) de iterações de um laço $l \in L_i$. Seja ${N_i}^{max}$ (${N_i}^{min}$) o número médio de invocações quando $n_l = {n_l}^{max}$ ($n_l = {n_l}^{min}$) para cada laço $l \in L_i$.

Agora podemos definir as latências máximas e mínimas em um BB B_i da maneira seguinte:

$$\lambda^{max}(b_i) = \lambda(b_i) \mid N_i = {N_i}^{max}$$
$$\lambda^{min}(b_i) = \lambda(b_i) \mid N_i = {N_i}^{min}$$

Assim podemos generalizar os limites de tempo de execução para um caminho no CFG. Seja $b_i \xrightarrow{\pi} b_j$ a expressão que denota que b_i alcança b_j através do caminho π. Então *as latências máximas* e *mínimas entre* eles são, respectivamente:

$$\lambda^{max}(b_i, b_j) = max\{\forall \pi \mid b_i \xrightarrow{\pi} b_j : \lambda(\pi)\}$$
$$\lambda^{min}(b_i, b_j) = min\{\forall \pi \mid b_i \xrightarrow{\pi} b_j : \lambda(\pi)\}$$

Finalmente, é possível definir os *atrasos mínimos e máximos calculados* entre x e y, respectivamente, da maneira seguinte:

$$m(x, y) = {}_{\delta(x,y)|((\lambda(b_i)=\lambda^{min}(b_i))\wedge(\lambda(b_i,b_j)=\lambda^{min}(b_i,b_j)))}$$
$$M(x, y) = {}_{\delta(x,y)|((\lambda(b_i)=\lambda^{max}(b_i))\wedge(\lambda(b_i,b_j)=\lambda^{max}(b_i,b_j)))}$$

Definidos os atrasos, é hora de discutir como as latências das instruções podem ser tratadas dentro dos BBs e dos caminhos.

Note que, para obter λ_x precisamos determinar o valor de $\sigma(u, v)$ para cada par de instruções dependentes. Entretanto, $\sigma(u, v)$ depende de $\varphi(u, v)$, que não pode ser completamente tratada em tempo de compilação sem enumerar (explicitamente ou implicitamente) todos os cenários distintos de execução induzidos por efeitos dinâmicos do *pipeline*. Já que para DSE não é conveniente o esforço computacional de tal enumeração, somos obrigados a depender de estimativas aproximadas. Finalmente, podemos discutir como as latências podem ser tratadas dentro dos BB e dos caminhos.

Dada uma instrução arbitrária u, para obter as estimativas de tempo superiores e inferiores para $R(\alpha_u)$, $R(\epsilon_u)$ e $W(\epsilon_u)$, assumimos a existência de um oráculo de memória implementando um modelo de cache baseado em AI, tal qual o presente em (Theiling e Ferdinand 1998).

Vamos denotar as estimativas dos limites superiores e inferiores de uma variável arbitrária f por $\hat{f}$ e $\breve{f}$, respectivamente. Para obter uma estimativa do limite superior para os ciclos de atraso do *pipeline*, assumimos de maneira pessimista que cada par de instruções dependentes são emitidas o mais perto possível ($\varphi(u, v) = 0$) de maneira que a latência completa entre eles fica explícita. Sob essa pressuposição, temos:

$$\hat{\sigma}(u, v) = \begin{cases} {}_{\lambda_p(u,v)} & \text{if } \tau(u) = reg \\ {}_{\hat{R}(\epsilon_u)+\lambda_p(u,v)} & \text{if } \tau(u) = load \\ {}_{\hat{W}(\epsilon_u)+\lambda_p(u,v)} & \text{if } \tau(u) = store \end{cases}$$

Para obter uma estimativa do limite inferior para os ciclos de atraso do *pipeline*, assumimos de maneira otimista que as instruções dependentes são emitidas suficientemente longe umas das outras para desconsiderar suas latências, i.e. $\breve{\sigma}(u, v) = 0$.

Observe que a velocidade da estimativa depende essencialmente do oráculo de memória, que pode depender da eficiência da técnica de AI empregada, enquanto a IPET necessita de ILP (que requer um esforço computacional mais alto, inadequado para DSE). Usar estimativas justas no oráculo de memória (e não para tratar o comportamento do *pipeline*) é uma estratégia para tentar manter a qualidade das estimativas na mesma ordem de magnitude que estimativas justas mais complexas.

Capítulo 5

O analisador de restrições proposto: Formulação, implementação e resultados experimentais

5.1 Formulação

Antes de apresentar os algoritmos que suportam a técnica apresentada nesta dissertação, serão explicitadas as premissas utilizadas. Foi assumido que:

- Há a disponibilidade de uma descrição formal (em uma ADL) do processador alvo a partir da qual as latências podem ser inferidas;
- O programa foi escrito em uma linguagem de alto nível estruturada e a topologia do seu CFG apresenta as propriedades discutidas na seção 4.3.1;
- O número de invocações N_i de um BB b_i é pré-determinado a partir do numero máximo e mínimo de iterações de um laço, como resultado da análise clássica de laços;
- Uma análise de caminhos falsos (como a descrita em (Gustafsson, Ermedahl e Lisper 2006), por exemplo) foi efetuada previamente, antes da execução dos algoritmos descritos nesta dissertação e por meio da marcação dos BBs, evita adicionar latências pertencentes a caminhos falsos.

Dado um modelo do processador `alvo` (onde a temporização da microarquitetura é descrita) e o conjunto de procedimentos `P` da IR de um programa, o Algoritmo 2 determina o conjunto de restrições impossíveis de serem satisfeitas (se houver alguma). Primeiramente, ele invoca o Algoritmo 1 para extrair as latências de cada par de instruções dependentes do modelo do processador (este algoritmo emprega uma função cujo o corpo é omitido, por razões de simplicidade). Depois desta anotação, o Algoritmo 2 checa se o atraso entre duas restrições obedece ou não o valor e o tipo especificados, para cada restrição de tempo imposta, no escopo de cada procedimento.

O conjunto das restrições não satisfeitas resultante deve ser então utilizado para definir a factibilidade (se `Infeasible` $= \emptyset$) ou para identificar os pares de instruções violando as restrições de tempo (se `Infeasible` $\neq \emptyset$).

Algoritmo 1 `annotate-Latencies(P, target)`

1: **for** each $p \in P$ **do**
2: let (B, F) be the CFG representing p
3: **for** each BB $b_i \in B$ **do**
4: let (V, E) be the DFG representing b_i
5: **for** each $(u, v) \in E$ **do**
6: $\lambda(u, v) =$ `extract-Latency` $(u, v, target)$
7: **end for**
8: **end for**
9: **end for**

Após a execução da ferramenta, $infeasible$ conterá o resultado da verificação: Se $infeasible = \emptyset$ significará que todas as restrições são satisfeitas; caso contrário, $infeasible$ conterá a relação das restrições não satisfeitas.

5.2 Decisões de implementação

Como o foco desta dissertação é redirecionar automaticamente os aspectos do *back-end* do compilador dependentes da microarquitetura, por enquanto, a tarefa de redirecionar os aspectos dependentes do ISA (tais como a seleção de instruções) é deixada

Algoritmo 2 `RTC(P, target)`

```
1: annotate-Latencies(P, target)
2: Infeasible = ∅
3: for each p ∈ P do
4:    let (B, F) be the CFG representing p
5:    let TC be set of time constraints on procedure p
6:    for each tc = (b_i, d_i, b_j, d_j, t, τ) ∈ TC do
7:       let x be the instruction at position d_i in BB b_i ∈ B
8:       let y be the instruction at position d_j in BB b_j ∈ B
9:       if (τ= MIN ∧ m(x, y)< t)∨(τ = MAX ∧ M(x, y)> t) then
10:         Infeasible= Infeasible ∪ {tc}
11:      end if
12:   end for
13: end for
```

de lado. Por este motivo, seria conveniente reusar um compilador redirecionável para lidar com as tarefas dependentes do ISA. Entretanto, como não foi possível encontrar um compilador automaticamente redirecionável de código livre, foi necessário se apoiar em um compilador redirecionável manualmente para concluir os experimentos. Para este propósito, foi reusado o redirecionamento do ISA realizado pelo compilador `llc` do *LLVM Project* (Lattner e Adve 2004) (ao invés do gcc), já que ele é mais fácil de se customizar e parece uma infra-estrutura mais promissora para desenvolvimentos futuros. Infelizmente, o lado ruim desta escolha é que os experimentos tiveram que ser limitados aos alvos PowerPC 405 e SPARC V8 (os únicos utilizados em sistemas embarcados para os quais já existem portes disponíveis).

Para descrever a microarquitetura do processador alvo, foi adotada a ADL ArchC (Azevedo et al. 2005). Não somente o projeto ArchC é de código livre, mas também o gerador automático de ferramentas binárias (Baldassin et al. 2008), que pode ser reusado (Note que esta é somente uma escolha pragmática; qualquer ADL capaz de escrever a temporização das instruções serviria, desde que permita a inferência das latências).

O tratamento das restrições utilizado, que é incluído no programa fonte, é altamente independente de linguagem, mas necessita de um mecanismo para inserção de

instruções *assembly* no código (tal qual "asm inline" no C e no C++).

Para checar por restrições do tipo MIN, foi utilizado um algoritmo clássico para o cálculo de caminhos mínimos (Djikstra's). Pra verificação das restrições do tipo MAX, implementou-se uma busca em profundidade, na qual as arestas improdutivas são ignoradas, para o cálculo dos caminhos mais longos. Assim, iterações em laços e execução seqüencial podem ser tratados separadamente, refletindo a modelagem descrita na seção 4.3.1.

A figura 5.1 ilustra os passos principais pelos quais um programa passa quando é submetido ao *back-end* da LLVM, desde a representação intermediária (IR) até o código *assembly* (ASM). A ferramenta desenvolvida, destacada no retângulo preto, localiza-se imediatamente antes da emissão do código

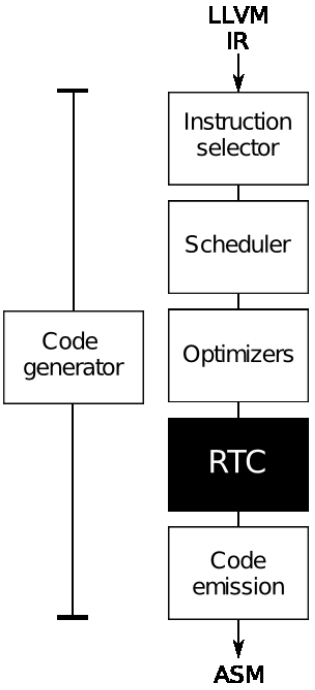

Figura 5.1: Os passos principais do compilador llc

Para validar os resultados do verificador de tempo real (ERTC) descrito nesta dissertação, eles foram comparados ao resultado de um ISS com precisão de ciclos. Foi obtido um ISS funcional para cada processador alvo, utilizando o gerador de simuladores (`acsim`) do pacote do ArchC (Azevedo et al. 2005). A ferramenta `acsim` gera automaticamente um ISS a partir da descrição ADL do processador alvo. Cada ISS funcional foi transformado em um ISS com precisão de ciclos, explorando a característica do ArchC de se anotar propriedades nas instruções para capturar suas latências.

Já que a integração do oráculo de memória não foi completada até o momento, escolhemos uma configuração experimental que assume acesso à memória em tempo constante, como se as instruções e os dados fossem alocados em memórias *scratch pad* (i. e., $\forall u \in I : R(\epsilon_u) = W(\epsilon_u) = R(\alpha_u) = 1$). Apesar da limitação dos experimentos, tal configuração possui a vantagem de determinar valores a serem usados como base para os tempos de execução para DSE.

5.3 Detecção da infactibilidade

A tabela 5.1 sumariza os programas e as restrições empregadas nos experimentos. A primeira coluna enumera os programas de benchmark (obtidos de (Gustafsson 2007)). As próximas três colunas mostram o número de linhas do código fonte e código *assembly*. As últimas duas colunas mostram o número e o tipo de restrições aplicadas.

Para um procedimento selecionado p_i de dado programa, foi simultaneamente imposta uma restrição do tipo MIN e MAX para p_i. Cada restrição foi especificada de maneira que sua fonte e sumidouro delimitam todo o corpo do procedimento. Para a maioria dos programas foi selecionado somente um único procedimento, mas 4 procedimentos foram selecionados no programa `adpcm` e 8 no programa `edn`. Para a coleta de dados do benchmark, os valores especificados para as restrições do tipo MAX eram sempre 0 e para o tipo MIN, um valor muito alto, para que nenhuma restrição seja satisfeita e conseqüentemente, todas fossem incluídas nos relatórios. O ERTC descrito nesta dissertação foi capaz de detectar a infactibilidade em todos os casos da maneira esperada.

5.4 Corretude e redirecionabilidade

Para cada processador alvo, todos os programas foram executados no ISS correspondente. As tabelas 5.2 e 5.3 sumarizam a análise do resultado para os alvos PowerPC e SPARC, respectivamente. Estas tabelas possuem a mesma estrutura. A segunda coluna denota um procedimento p_i selecionado do programa da primeira coluna. A terceira e a quarta colunas mostram os atrasos mínimos (m(x,y)) e máximos (M(x,y)) calculados pelo ERTC descrito nesta dissertação. A última coluna exibe os atrasos obtidos através da execução do

Tabela 5.1: Caracterização dos benchmarks

	Linhas de código			#TCs	
Programa	fonte	PowerPC	SPARC	MIN	MAX
adpcm	910	2187	2434	4	4
bs	115	127	132	1	1
crc	133	274	307	1	1
edn	302	1207	1266	8	8
fdct	240	598	609	1	1
fibcall	76	71	70	1	1
fir	277	1623	1636	1	1
insertsort	93	86	93	1	1
jannei_complex	67	96	99	1	1
jfdctint	378	453	500	1	1
ns	533	1261	1273	1	1
prime	50	177	194	1	1

ISS.

As figuras 5.2 e 5.3 mostram as tabelas 5.2 e 5.3 de forma mais ilustrativa. Nestas figuras, a linha superior é referente a coluna M(x,y) da respectiva tabela. A linha inferior corresponde a coluna m(x,y). A linha que fica entre a superior e a inferior é referente ao valor simulado pelo ISS.

O tempo de execução de um simulador que realiza o comportamento real dos *benchmarks* é em várias ordens de grandeza superior ao tempo de execução da técnica proposta neste trabalho. Para tentar comparar melhor os tempos de execução, o ISS foi alterado para que ao executar os *benchmarks*, cada laço fosse executado uma única vez ($N_i = 1, \forall b_i$). Obviamente, a mesma instrumentalização foi utilizada para ambos o ERTC e o ISS. O resultado dos benchmarks no simulador instrumentado não é equivalente ao resultado do simulador original, mas a temporização da travessia entre os blocos básicos é preservada, o que é suficiente para a comparação com a atual implementação da técnica.

Observe que, para cada programa e alvo, os valores obtidos pelo ISS sempre se encaixam entre os valores mínimos e máximos calculados pelo ERTC. Isto dá forte credibilidade para a corretude da técnica, para ambos os alvos.

Tabela 5.2: Resultado da análise para o PowerPC

		ERTC		
Program	p_i	m(x,y)	M(x,y)	ISS
bs	p_1	6	34	16
ns	p_1	6	53	11
crc	p_1	12	24	18
edn	p_1	11	14	13
edn	p_2	5	37	9
edn	p_3	5	88	9
edn	p_4	15	56	19
edn	p_5	11	249	14
edn	p_6	22	47	30
edn	p_7	12	33	20
edn	p_8	5	19	9
fir	p_1	27	91	33
fdct	p_1	17	483	31
adpcm	p_1	14	14	14
adpcm	p_2	19	36	26
adpcm	p_3	19	35	35
adpcm	p_4	88	159	90
prime	p_1	13	31	30
jfdctint	p_1	16	410	18
fibcall	p_1	11	26	23
insertsort	p_1	28	63	33
janne_complex	p_1	5	32	10

5.5 Eficiência

Para quantificar como ISSs são inadequados para DSE, as tabelas 5.4 e 5.5 relatam os tempos de execução da análise em tempo de compilação (ERTC), comparada com o tempo de execução do ISS para os alvos PowerPC e SPARC, respectivamente. Elas também mostram o aumento da velocidade obtido com o uso da técnica descrita nesta dissertação. Os tempos de execução são expressos em segundos e foram medidos em um computador com processador Core2 Duo T5250 (1.5GHz, 2MB L2) e 2GB (667 MHz DDR2) de memória principal.

Note que, mesmo se assumirmos que estímulos podem ser judiciosamente escolhidos para que um ISS possa estimar BCET/WCET, tal técnica seria no mínimo 5 vezes mais lenta que o ERTC. Enquanto o número de instruções executadas no ISS cresce com o número de iterações (possivelmente um crescimento exponencial devido a laços aninhados), os cálculos realizados pelo ERTC não são impactados pelo número de iterações, já que a

Figura 5.2: Gráfico de resultado da análise para o PowerPC

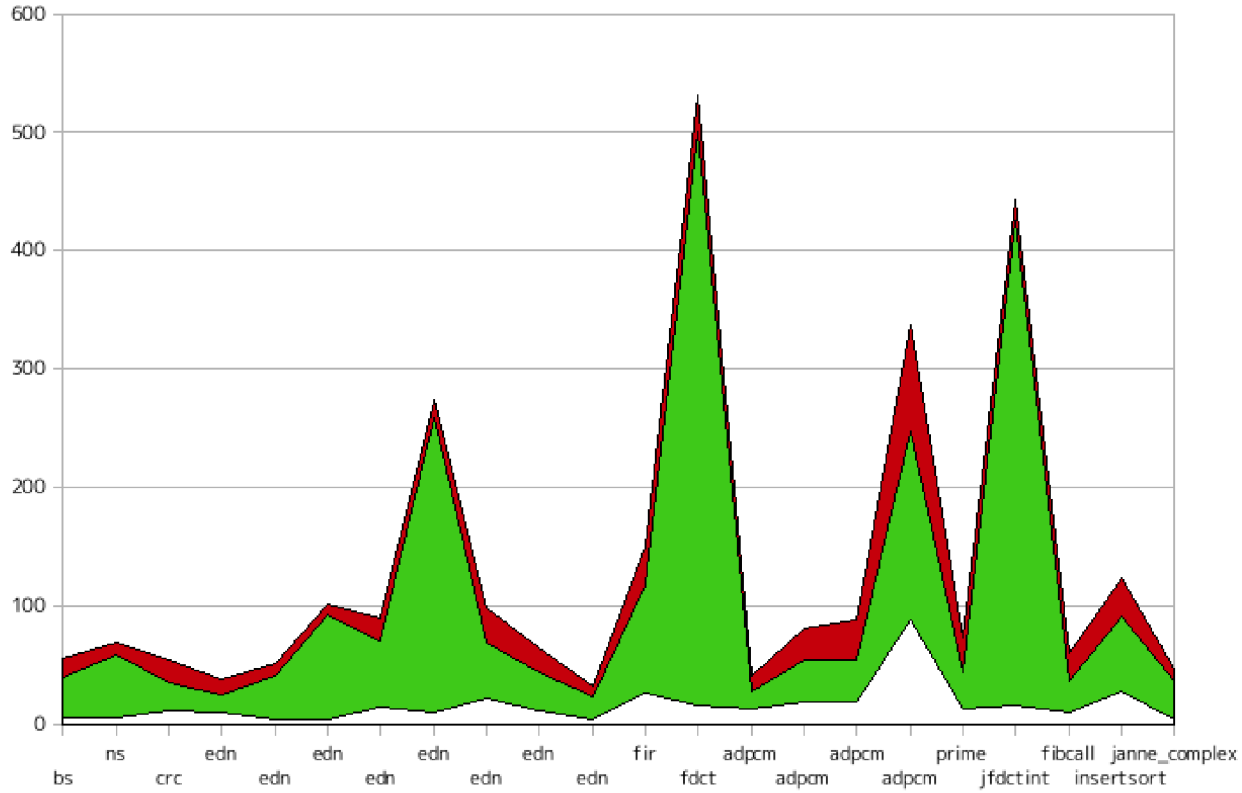

Figura 5.3: Gráfico de resultado da análise para o SPARC

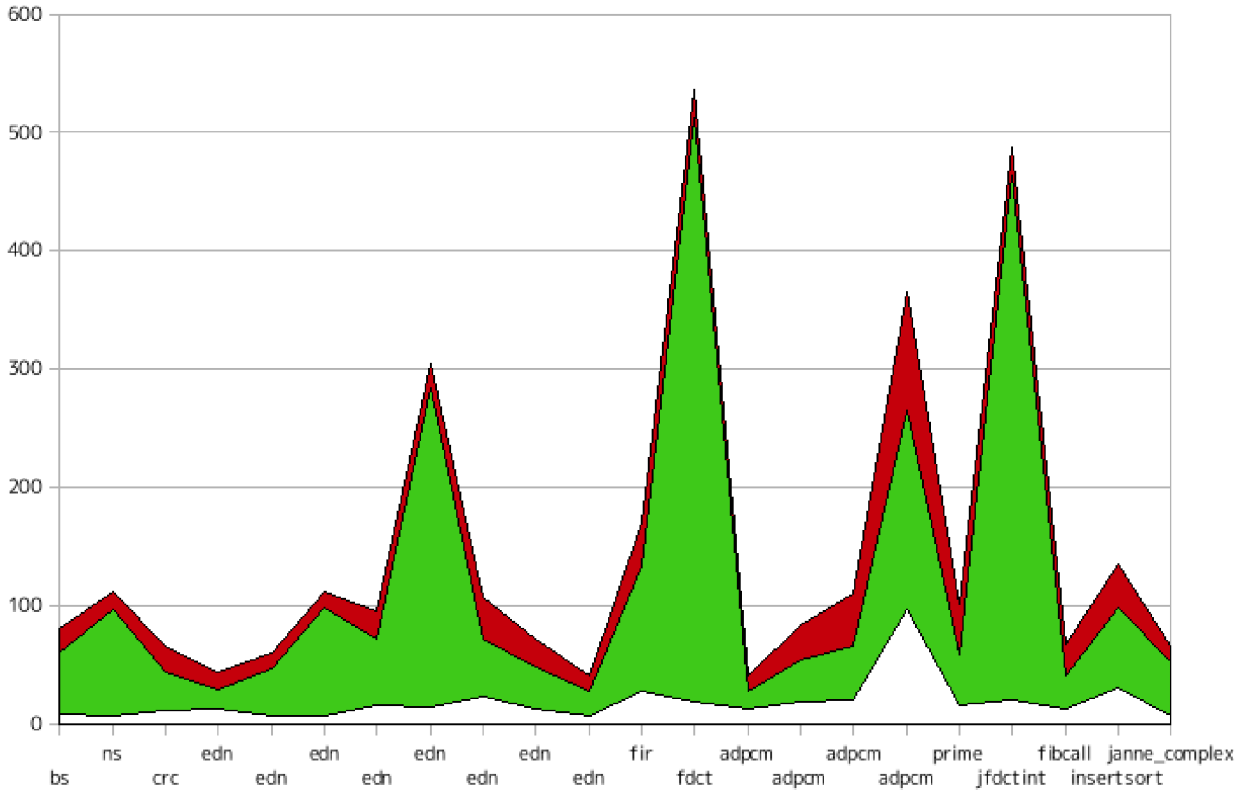

Tabela 5.3: Resultado da análise para o SPARC

		ERTC		
Programa	p_i	m(x,y)	M(x,y)	ISS
bs	p_1	9	52	20
ns	p_1	8	89	15
crc	p_1	12	33	21
edn	p_1	13	16	15
edn	p_2	7	41	13
edn	p_3	7	92	13
edn	p_4	16	57	23
edn	p_5	15	270	20
edn	p_6	24	49	34
edn	p_7	14	35	24
edn	p_8	7	21	13
fir	p_1	28	106	37
fdct	p_1	19	493	24
adpcm	p_1	14	14	14
adpcm	p_2	19	36	29
adpcm	p_3	21	45	45
adpcm	p_4	97	168	100
prime	p_1	16	43	43
jfdctint	p_1	21	443	24
fibcall	p_1	13	29	26
insertsort	p_1	31	68	37
janne_complex	p_1	7	46	14

Tabela 5.4: Aceleração do tempo de execução em segundos para o PowerPC

	runtime		
Programa	ISS	ERTC	Aceleração
fibcall	0.833	0.118	7.059
jfdctint	0.849	0.153	5.549
adpcm	1.471	0.648	2.270
janne_complex	0.827	0.138	5.993
prime	1.029	0.212	4.854
fir	0.935	0.166	5.633
edn	2.133	0.389	5.483
insertsort	0.847	0.093	9.108
ns	0.792	0.168	4.714
bs	0.775	0.12	6.458
fdct	0.911	0.148	6.155
crc	0.828	0.162	5.111
média			5.7

técnica descrita nesta dissertação quebra os laços para desacoplar a execução seqüencial e iteração em laços. Desta maneira, os valores reportados para a aceleração podem ser vistos,

quando múltiplas iterações entram em jogo, como a aceleração mínima em relação a técnicas de análise baseadas em ISS.

Tabela 5.5: Aceleração do tempo de execução em segundos para o SPARC

	runtime		
Programa	ISS	ERTC	Aceleração
fibcall	0.767	0.114	6.728
jfdctint	0.812	0.16	5.075
adpcm	1.484	0.672	2.208
janne_complex	0.736	0.152	4.842
prime	0.916	0.203	4.512
fir	0.793	0.158	5.019
edn	2.165	0.358	6.047
insertsort	0.748	0.098	7.633
ns	0.704	0.179	3.933
bs	0.729	0.118	6.178
fdct	0.748	0.15	4.987
crc	0.753	0.155	4.858
média			5.2

Não encontramos tempos de execução reportados para técnicas WCET *tight-bound* para comparar com nossos resultados. Apesar da falta de resultados, é amplamente conhecido que a formulação ILP é vulnerável a tempos de execução muito grandes mesmo quando aplicada a código pré-compilado para um único alvo. Por isto, fica claro que ILP é proibitivo para análise em tempo de compilação e para vários alvos. Por outro lado, apesar dos tempos de execução do ERTC serem maiores para modelos mais realísticos da hierarquia de memória, o incremento no tempo de execução seria aceitável desde que AI fosse empregado no oráculo de memória.

5.6 Impacto das limitações da implementação

Observando as figuras 5.2 e 5.3, a linha que representa o resultado da simulação fica muito perto da linha que representa o tempo estimado para o valor MAX. Isto se deve a uma implementação simplificada do comportamento da hierarquia de memória no simulador utilizado e no ERTC. A adição do suporte a hierarquias de memória mais complexas provavelmente alteraria estes resultados.

Nos *benchmarks* utilizados, observando a semântica dos programas, verifica-

se que existem laços de repetição que serão utilizados sempre no mínimo uma vez. Isto cria no CFG caminhos falsos menores que os caminhos mínimos reais. Como a análise dos caminhos falsos não foi implementada, o ERTC admite caminhos menores que os de fato existem nos *benchmarks*. Isto explica a grande distância entre as estimativas MIN e os valores simulados, que pode ser observada nas figuras 5.2 e 5.3.

Capítulo 6

Conclusões e trabalhos futuros

Nesta dissertação foi descrita uma maneira de se abordar a análise de restrições de tempo de forma complementar às técnicas no estado da arte. Estas têm como objetivo obter estimativas justas para a WCET e o BCET, que tradicionalmente operam sobre binário executável, enquanto que a técnica apresentada visa estimativas aproximadas e a exploração do espaço de projeto.

A técnica proposta nesta dissertação é voltada para a exploração do espaço de projeto e opera dentro do fluxo de um compilador tradicional. Ela captura um conjunto comum de restrições de tempo (atrasos mínimos, máximos e exatos). Embora os experimentos realizados tenham sido limitados a dois alvos, os resultados apresentados nos permitem concluir que a redirecionabilidade pode ser afetada por uma escolha inadequada da ADL, mas não por limitações da técnica proposta, já que o redirecionamento automático só é afetado se a temporização das instruções não puder ser inferida a partir da descrição da ADL.

A IR adotada para a representação do programa, é deliberadamente similar ao formato intermediário utilizado pela infra-estrutura subjacente de compilação, no caso a LLVM. Isto possibilita, por exemplo, que a técnica seja utilizada como métrica, para guiar a otimização.

A implementação foi baseada em ferramentas de distribuição livre, como o projeto ArchC e a LLVM, portanto não possui empecilhos legais para sua distribuição e aprimoramento.

A essência da técnica proposta é similar a IPET, que é a técnica atualmente

estabelecida para a obtenção de estimativas justas levando em conta os estágios do *pipeline*. Porém há indícios muito fortes, descritos na revisão bibliográfica, que ela não é adequada para DSE.

Os resultados obtidos nos experimentos evidenciam a corretude da técnica. Porém não há dados do tempo de execução da IPET sob o mesmo conjunto de benchmark para comparação direta. A comparação direta com a IPET poderia ser utilizada para evidenciar o valor da técnica desta dissertação para a DSE. Apesar de existirem resolutores de ILP de distribuição livre, as ferramentas do mercado que fazem IPET têm distribuição restrita. Assim, para a comparação direta será necessário a implementação não somente da IPET, mas também a elaboração dos modelos dos processadores utilizados.

Como discutido na seção 5.6, uma técnica complementar a este trabalho é a AI. Na versão atual da ferramenta, ainda não é possível tratar o comportamento dinâmico da hierarquia de memória, que pode ser resolvido com o uso da AI. Convenientemente, a modelagem das restrições de tempo se encaixa no CFG do programa. A disposição das restrições no CFG obedece o mecanismo de construção do CFG a partir do código fonte, possibilitando que informações do alto nível sejam utilizadas na análise, o que é interessante para a incorporação da AI na ferramenta.

Enfim, como trabalhos futuros, além da incorporação da AI, há a necessidade da comparação com outras técnicas de estimativa do WCET e BCET para quantificar os intervalos de confiança no qual a DSE possa se apoiar.

Referências Bibliográficas

Aho, Sethi e Ullman 1986 AHO, A.; SETHI, R.; ULLMAN, J. *Compilers: principles, techniques, and tools*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1986.

al. 2008 AL., W. J. D. et. Efficient Embedded Computing. *IEEE Computer Magazine*, v. 41, n. 7, p. 27–33, 2008.

Azevedo et al. 2005 AZEVEDO, R. et al. The ArchC Architecture Description Language. *International Journal of Parallel Programming*, v. 33, n. 5, p. 453–484, October 2005.

Baldassin et al. 2007 BALDASSIN, A. et al. Automatic Retargeting of Binary Utilities for Embedded Code Generation. In: *Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. [S.l.: s.n.], 2007. p. 253–258.

Baldassin et al. 2008 BALDASSIN, A. et al. An Open-Source Binary Utility Generator. *ACM Transactions on Design Automation of Electronic Systems*, ACM New York, NY, USA, v. 13, n. 2, April 2008.

Carlomagno, Santos e Santos 2007 CARLOMAGNO, J. O.; SANTOS, L. F. P.; SANTOS, L. C. V. A Retargetable Embedded Code Scheduler for SoC Design Space Exploration Under Real-Time Constraints. In: *Proc. IEEE Northeast Workshop on Circuits and Systems (NEWCAS)*. [S.l.: s.n.], 2007. p. 233–236.

Ceng et al. 2005 CENG, J. et al. C Compiler Retargeting Based on Instruction Semantics Models. In: *Proc. Design, Automation and Test in Europe Conference (DATE)*. [S.l.: s.n.], 2005.

Cousot e Cousot 1977 COUSOT, P.; COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.

In: *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, 1977. p. 238–252.

Engblom 2002 ENGBLOM, J. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. Tese (Doutorado) — Uppsala University, Department of Information Technology, 2002.

Ermedahl 2003 ERMEDAHL, A. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. Tese (Doutorado) — Uppsala University, Department of Information Technology, 2003.

Fauth, Praet e Freericks 1995 FAUTH, A.; PRAET, J. V.; FREERICKS, M. Describing Instruction Set Processors using nML. In: *Proc. European Design and Test Conference (EDTC)*. [s.n.], 1995. p. 503–507. Disponível em: <citeseer.nj.nec.com/fauth95describing.html>.

Gonzalez 2000 GONZALEZ, R. Xtensa: a configurable and extensible processor. *Micro, IEEE*, v. 20, n. 2, p. 60–70, 2000.

Gustafsson 2007 GUSTAFSSON, J. *The Mälardalen WCET benchmarks*. 2007. Http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

Gustafsson, Ermedahl e Lisper 2006 GUSTAFSSON, J.; ERMEDAHL, A.; LISPER, B. Algorithms for Infeasible Path Calculation. In: *Proc. Workshop on Worst-Case Execution Time Analysis (WCET)*. [S.l.: s.n.], 2006.

Halambi et al. 1999 HALAMBI, A. et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In: *Proc. Design, Automation and Test in Europe Conference (DATE)*. [S.l.: s.n.], 1999.

Hanono e Devadas 1998 HANONO, S.; DEVADAS, S. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In: *Proc. Design Automation Conference (DAC)*. [S.l.: s.n.], 1998. p. 510–515.

Hoffmann et al. 2005 HOFFMANN, A. et al. A methodology and tooling enabling application specific processor design. *VLSI Design, 2005. 18th International Conference on*, p. 399–404, 2005.

Kirner e Puschner 2003 KIRNER, R.; PUSCHNER, P. Discussion of Misconceptions about WCET Analysis. In: *Proc. Workshop on Worst-Case Execution Time Analysis (WCET).* [S.l.: s.n.], 2003. p. 61–64.

Kobayashi et al. 2001 KOBAYASHI, S. et al. Compiler Generation in PEAS-III: an ASIP Development System. In: *Proc. Workshop on Software and Compilers for Embedded Systems (SCOPES).* [S.l.: s.n.], 2001.

Lattner e Adve 2004 LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proc. Symposium on Code Generation and Optimization (CGO).* [S.l.: s.n.], 2004.

Leung, Palem e Pnueli 2002 LEUNG, A.; PALEM, K.; PNUELI, A. TimeC: A Time Constraint Language for ILP Processor Compilation. *Constraints*, Springer, v. 7, n. 2, p. 75–115, 2002.

Leupers e Marwedel 2001 LEUPERS, R.; MARWEDEL, P. *Retargetable Compiler Technology for Embedded Systems: Tools and Applications.* [S.l.]: Kluwer Academic Publishers, 2001.

Li et al. 2007 LI, X. et al. A Retargetable Software Timing Analyzer Using Architecture Description Language. In: *Proc. Asian and South Pacific Design Automation Conference.* [S.l.: s.n.], 2007. p. 396–401.

Li et al. 2007 LI, X. et al. A retargetable software timing analyzer using architecture description language. In: *ASP-DAC '07: Proceedings of the 2007 conference on Asia South Pacific design automation.* Washington, DC, USA: IEEE Computer Society, 2007. p. 396–401. ISBN 1-4244-0629-3.

Li, Malik e Inc 1997 LI, Y.; MALIK, S.; INC, M. Performance analysis of embedded software using implicit pathenumeration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, v. 16, n. 12, p. 1477–1487, 1997.

Mishra e Dutt 2008 MISHRA, P.; DUTT, N. *Processor Description Languages.* [S.l.]: San Francisco: Morgan Kaufmann, 2008. ISBN 978-0123742872.

Pesch e Osier 1993 PESCH, R.; OSIER, J. *The Gnu Binary Utilities.* [S.l.]: May, 1993. Http://www.gnu.org/software/binutils/manual/. Version 2.12.

Petters, Zadarnowski e Heiser 2007 PETTERS, S.; ZADARNOWSKI, P.; HEISER, G. Measurements or static analysis or both? In: *Proc. Workshop Worst-Case Execution-Time Analysis (WCET)*. [S.l.: s.n.], 2007.

Qin e Malik 2003 QIN, W.; MALIK, S. Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation. In: *Proc. Design, Automation and Test in Europe Conference (DATE)*. [S.l.: s.n.], 2003. p. 556–561.

Rigo et al. 2004 RIGO, S. et al. ArchC: a SystemC-based Architecture Description Language. In: *Proc. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. [S.l.: s.n.], 2004. p. 66–73.

Stallman 2002 STALLMAN, R. GNU Compiler Collection Internals. *Free Software Foundation, Reference Manual, Boston, Mass*, 2002.

Stallman et al. 1999 STALLMAN, R. et al. *Using and Porting the GNU Compiler Collection*. [S.l.]: Free Software Foundation, 1999.

Theiling e Ferdinand 1998 THEILING, H.; FERDINAND, C. Combining abstract interpretation and ILP for microarchitecturemodelling and program path analysis. *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, p. 144–153, 1998.

Wilhelm et al. 2008 WILHELM, R. et al. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, ACM, New York, NY, USA, v. 7, n. 3, p. 1–53, 2008. ISSN 1539-9087.