UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Luiz Fernando Penkal Santos

Um Escalonador de Código Redirecionável para Processadores Embarcados

Luiz Cláudio Villar dos Santos

Florianópolis
2006/2

Um Escalonador de Código Redirecionável para Processadores Embarcados

Luiz Fernando Penkal Santos

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

Orientador

Luiz Cláudio Villar dos Santos

Banca Examinadora

Olinto José Varela Furtado

Luís Fernando Friedrich

Sumário

Lista de Figuras

List of Algorithms

Resumo

O número cada vez maior de transistores que podem ser colocados em uma mesma área significa um poder computacional crescente. Mas toda esta capacidade também implica em uma maior complexidade no projeto de tais sistemas, já que existem vários fatores que o influenciam, como potência, área e desempenho.

Quando vamos avaliar uma alternativa de projeto que use um processador, seja ele de uso geral ou específico, temos que usar uma série de ferramentas (compilador, montador e ligador) para gerar o código da aplicação para o processador alvo.

Ocorrem casos onde as ferramentas de geração de código não existem ainda, se o processador for desenvolvido só para aquele projeto, e não for uma alternativa comercial. Nestes casos, construir toda a cadeia de ferramentas manualmente teria um custo proibitivo, e teria um grande impacto no time-to-market.

Neste contexto se insere a geração automática de ferramentas, através de descrições em alto nível dos processadores alvo em uma linguagem de descrição de arquitetura (ADL) [4].

Além dos problemas acima, muitas das ferramentas disponíveis para processadores comerciais não levam em conta restrições de tempo real, requisito comum em sistemas embarcados.

Este trabalho descreve uma ferramenta redirecionável que gera automaticamente um escalonador de código assembly com restrições de tempo real para uma arquitetura alvo, e foi realizado em cooperação com o aluno de mestrado José Otávio Carlomagno Filho.

Capítulo 1

Introdução

A tecnologia de semicondutores, com os quais são fabricados os sistemas computacionais, avança cada vez mais. Essa evolução é prevista pela Lei de Moore, que diz que a cada 18 meses o tamanho de um transistor é reduzido pela metade.

O número cada vez maior de transistores que podem ser colocados em uma mesma área significa um poder computacional crescente. Mas toda esta capacidade também implica em uma maior complexidade no projeto de tais sistemas.

Novos desafios foram introduzidos no projeto de sistemas embarcados. Um número maior de transistores costuma implicar em uma maior dissipação de potência pelo sistema, e como a tecnologia das baterias não teve um avanço tão expressivo quanto a microeletrônica, o tempo de vida das baterias é cada vez menor. Uma dificuldade adicional é o tamanho crescente do código das aplicações embarcadas, devido a maior complexidade dos sistemas, já que memória de acesso rápido tem um custo significativo.

Para enfrentar estes e outros desafios, o projeto de um sistema embarcado leva em conta o número de transistores necessários para implementá-lo, o consumo de potência do sistema, o desempenho da solução adotada, o tamanho do código do software embarcado, etc.

De acordo com o uso do sistema, uma ou outra variável é de maior importância. Existem diversas soluções de implementação para um mesmo sistema, cada uma com suas vantagens e desvantagens.

Uma primeira alternativa de projeto são os circuitos de aplicação específica (ASICs). Como são criados para realizar somente as operações desejadas, ASICs têm uma maior eficiência, mas não têm flexibilidade alguma.

Os processadores de aplicação específica (ASIPs), por sua vez, são mais flexíveis, pois são capazes de executar todo tipo de aplicação de um certo domínio. Apesar da maior flexibilidade, são menos eficientes que um ASIC e consomem mais potência.

Por fim, temos os processadores de uso geral (GPPs). Um processador de uso geral é capaz de executar qualquer tipo de aplicação. São os mais flexíveis, mas têm a menor eficiência e o maior gasto de potência.

Levantados os requisitos de um projeto, temos então várias alternativas de implementação, cada uma com sua vantagem. O processo de se escolher uma alternativa de projeto que melhor se encaixe aos requisitos do sistema é chamado de exploração do espaço de projeto.

A metodologia de projeto orientado a plataforma [8] consiste numa plataforma onde temos diversos componentes, como IPs, ASIPs e processadores de uso geral. IPs são blocos de propriedade intelectual que realizam alguma função específica, como por exemplo, decodificação de arquivos em formato mp3 ou realização de operações criptográficas custosas.

A partir desta plataforma podemos construir as diversas alternativas a serem avaliadas para um projeto. Esta noção de plataforma provê também a possibilidade do reuso de blocos de propriedade intelectual já existentes. Se já existe um IP que realiza uma função desejada, pode-se simplesmente plugá-lo ao projeto, sem gastar tempo projetando um novo. Quando vamos avaliar uma alternativa que use um processador, seja ele de uso geral ou específico, temos que usar uma série de ferramentas (compilador, montador e ligador) para gerar o código da aplicação para o processador alvo.

Ocorrem casos onde as ferramentas de geração de código não existem ainda, se o processador for desenvolvido só para aquele projeto, e não for uma alternativa comercial. Nestes casos, construir toda a cadeia de ferramentas manualmente teria um custo proibitivo, e teria um grande impacto no time-to-market. O sistema correria o risco de entrar no mercado quase ”obsoleto”, pois o projeto foi demorado demais.

Neste contexto se insere a geração automática de ferramentas, através de descrições em alto nível dos processadores alvo em uma linguagem de descrição de arquitetura (ADL) [4]. Com todos os dados sobre o processador em mãos, no pior caso são necessários alguns dias para se escrever um modelo em uma ADL.

Linguagens de descrição de arquiteturas são linguagens que nos permitem descrever o conjunto de instruções de um processador, assim como (dependendo da linguagem) sua estrutura, como cache, memória, pipeline, etc. Entre as principais ADLs estão LISA, EXPRESSION, ISDL, nML e ArchC. Esta abordagem representa um grande ganho de produtividade, já que a partir do modelo seriam geradas todas as ferramentas necessárias, quase instantaneamente.

Com o uso destas ferramentas redirecionáveis, torna-se possível a exploração de diversas alternativas de CPUs sem grande prejuízo ao tempo de projeto, pois o tempo que antes era gasto desenvolvendo-se todas as ferramentas manualmente não é mais necessário. Além dos problemas acima, muitas das ferramentas disponíveis para processadores comerciais não levam em conta restrições de tempo real, requisito comum em sistemas embarcados.

Este trabalho descreve uma ferramenta redirecionável que gera automaticamente um escalonador de código assembly com restrições de tempo real para uma arquitetura alvo, e foi realizado em cooperação com o aluno de mestrado José Otávio Carlomagno Filho.

A adição da análise de restrições de tempo real ao escalonador poupa tempo no sentido de já conseguirmos saber se uma restrição será respeitada antes mesmo de executar o código escalonado, preservando a infraestrutura existente de geração de código do processador alvo.

Capítulo 2

O Escalonador

Ao avaliar o impacto de um processador no sistema, precisamos de ferramentas que gerem código para ele. Se as ferramentas gerarem código otimizado para o processador, poderemos ter uma idéia correta do desempenho deste.

Como um compilador otimizador é orientado ao desempenho médio, algumas otimizações podem ser negligenciadas. No caso de um segmento de código estar sujeito a restrições de tempo real, estas otimizações negligenciadas podem ser a diferença entre satisfazer ou não estas restrições. Teríamos então de modificar o compilador existente ou criar outro a partir do zero. Em ambos os casos, o trabalho é complexo e pode demorar muito.

O escalonador descrito neste trabalho trabalha em nível de código assembly, ou seja, após a compilação. Logo, esta ferramenta pode ser adicionada ao fluxo de geração de código sem modificações no compilador ou no montador, pois sua saída é o código de entrada reescalonado.

As otimizações realizadas não levam em conta a estrutura do programa em alto-nível. No nível em que o escalonador trabalha, as instruções são reordenadas de forma a evitar conflitos de recursos e hazards, tentando fazer com que o processador esteja sempre executando o máximo de instruções possível ininterruptamente.

Para que o escalonador saiba como reordenar as instruções, são necessárias informações como a latência entre instruções (que é o tempo necessário para os resultados poderem

ser consumidos pela próxima instrução), a sintaxe do código assembly e o conjunto de registradores.

Estas informações estão contidas na descrição do processador em uma ADL. A ADL adotada neste trabalho é ArchC [2]. Para tornar possível a geração do escalonador, foram adicionadas à linguagem construções para se especificar a latência entre instruções.

A descrição é passada então para a ferramenta que gera o escalonador para o processador alvo, permitindo a geração de código otimizado apenas adicionando o escalonador ao fluxo de geração de código, o que faz com que as estimativas de desempenho sejam mais precisas.

2.1 Modelagem

O escalonador apresentado neste trabalho se propõe a alcançar dois objetivos: redirecionabilidade e análise de restrições de tempo real. Para atingir o objetivo da redirecionabilidade, é necessário que a representação intermediária do código que vai ser escalonado seja independente de arquitetura. Já para tornar possível a análise de restrições de tempo real, é necessário que esta representação consiga modelar elegantemente estas restrições.

Tendo em vista estes dois pontos, foi então adotada a modelagem descrita em [6] e expandida em [5]. Esta modelagem consiste em um grafo de fluxo de dados ponderado, que é usado como representação intermediária para os blocos básicos do código. Este grafo além de capturar as restrições de precedência entre instruções também consegue codificar as restrições de tempo real.

Para realizar um escalonamento hierárquico do código foi utilizado um grafo de controle de fluxo em que cada vértice corresponde a um bloco básico. Estes são ligados por arestas que representam os fluxos possíveis de execução.

2.1.1 Grafo de fluxo de dados

Neste grafo cada vértice V_i corresponde a uma instrução do bloco básico representado (exceto por v0 e vn) e as arestas (V_i, V_j) representam restrições de precedência (dependência de dados entre as instruções). O peso de uma aresta representa a latência entre duas instruções ou uma restrição de tempo real. O grafo é orientado, polar e seus pólos v0 e vn são chamados de source e sink respectivamente. Um exemplo mostrado na figura 2.1.

Para representar uma restrição de tempo mínimo de execução entre V_i e V_j é inserida uma aresta (V_i, V_j) com peso +K, o que faz com que a instrução V_j só possa iniciar K ciclos após V_i. Já a restrição de tempo máximo é representada por uma aresta (V_j, V_i) com peso -K, ou seja, entre V_j e V_i deve haver um atraso de no máximo K ciclos. Uma restrição de tempo exato é representada com uma aresta (V_i, V_j) com peso +K e outra (V_j, V_i) com peso -K, fazendo com que V_j só possa ser executada exatamente K ciclos após V_i.

Se qualquer uma das restrições acima não puder ser respeitada, ou seja, gerar um ciclo no grafo cujo caminho tenha um peso resultante maior que zero, o algoritmo de Bellman-Ford nunca convergirá[1], pois sempre haverá um caminho mais longo. Já se as restrições puderem ser respeitadas, o algoritmo converge em no máximo n iterações, onde n é o número de vértices do grafo.

Se o escalonamento de uma instrução não resultar em violação alguma de restrição de tempo real, é adicionado um par de arestas entre ela e o source, assim como em uma restrição de tempo exato, para indicar que ela deve ser executada exatamente K ciclos após o início do código.

2.1.2 Grafo de fluxo de controle

Todo código assembly pode ser dividido em blocos básicos, que são pedaços do código onde não há ramificação no fluxo de execução. Esses blocos começam em instruções que são alvo ou vem depois de desvios ou se a instrução é a primeira do código, e terminam em instruções de desvios ou quando o código termina.

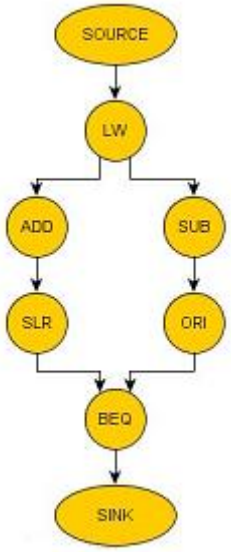

Figura 2.1: Exemplo de grafo de fluxo de dados.

No grafo de fluxo de controle, mostrado na figura 2.2, cada vértice b_i corresponde a um bloco básico do código e tem uma referência para seu grafo de fluxo de dados respectivo. As arestas (b_i, b_j) representam a ordem de precedência dos blocos no fluxo de execução.

Se o bloco termina em um desvio condicional, ele terá duas arestas, uma para cada alvo possível. Se terminar em desvio incondicional, aponta apenas para o bloco que é alvo deste desvio. Se não terminar em desvio, o bloco aponta para o bloco sucessor.

2.2 Estrutura

A figura 2.3 mostra a estrutura completa do escalonador. Primeiramente, o código assembly gerado por um compilador pode passar por um editor de restrições de tempo real, onde o usuário pode especificar restrições no código assembly.

Para se anotar uma restrição em um pedaço do código, usam-se as pseudo-instruções `[MIN K, LABEL]` e `[/MIN LABEL]`, `[MAX K, LABEL]` e `[/MAX LABEL]` ou `[EXACT K, LABEL]` e `[/EXACT LABEL]`. Se usarmos `[MIN K, LABEL]`, te-

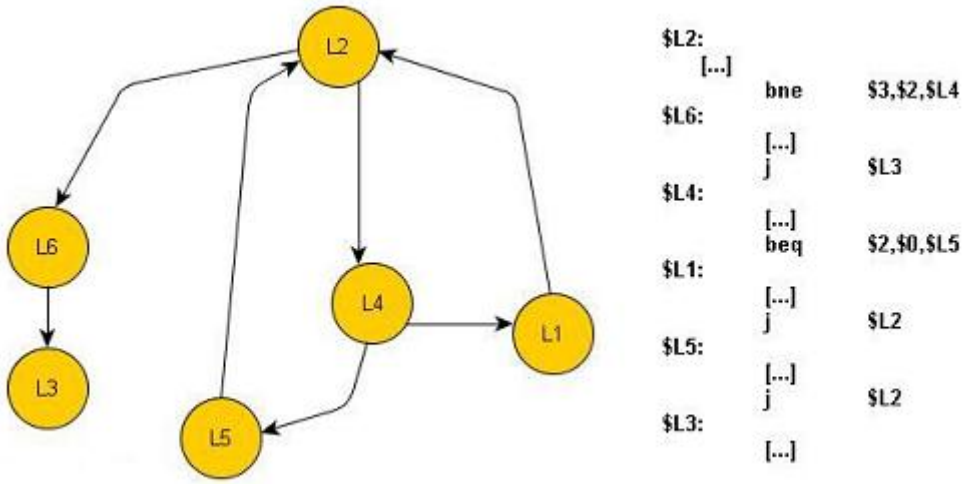

Figura 2.2: Exemplo de grafo de fluxo de controle.

remos um número de ciclos mínimo K em que o código englobado pelo par de instruções seja executado. Se usarmos `[MAX K, LABEL]`, impomos um número de ciclos máximo K para rodar o trecho de código. Já `[EXACT K, LABEL]` impõe uma restrição de exatamente K ciclos ao código desejado.

Ao código resultante da anotação de restrições é dado o nome de assembly instrumentado. Vale ressaltar que as pseudo instruções inseridas no código serão removidas quando da geração do código escalonado.

O assembly instrumentado é então usado como entrada para o code parser, ferramenta que lê o código e cria um grafo de fluxo de controle. Cada um dos vértices deste grafo aponta para o grafo de fluxo de dados do bloco básico que é representado por ele. O code parser é gerado a partir de informações extraídas do modelo ArchC pelo model parser, no caso a sintaxe do assembly e as latências entre instruções, que são necessárias para se gerar o grafo.

Após a geração do grafo, este é então escalonado levando-se em conta as restrições de tempo real anotadas anteriormente. O desrespeito de uma restrição qualquer resulta em um erro.

A saída deste passo é o grafo escalonado, que é então submetido ao code ge-

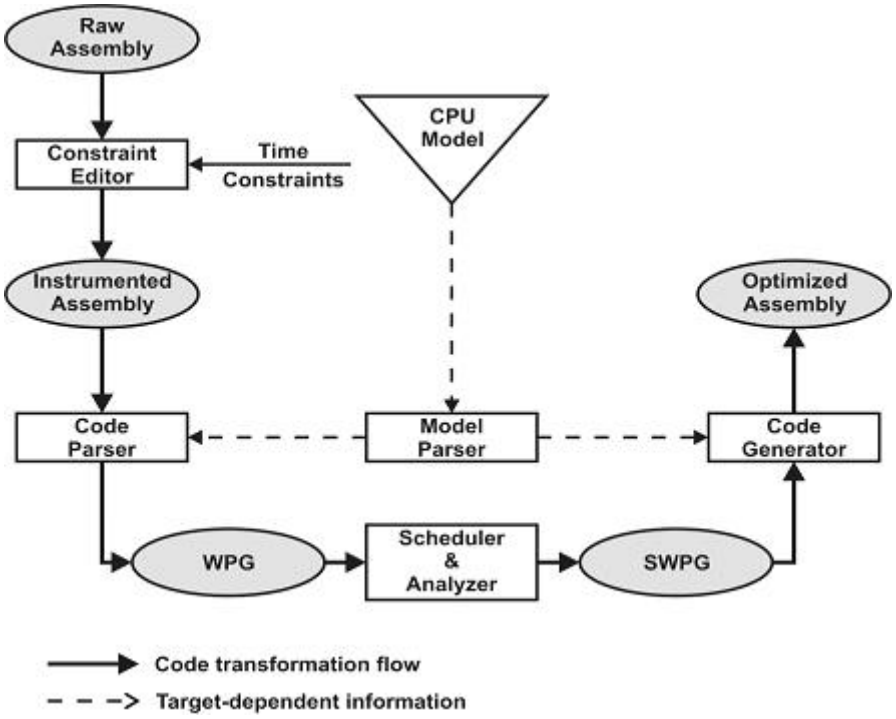

Figura 2.3: Estrutura do escalonador redirecionável.

nerator, que também usando as informações extraídas pelo model parser, gera o código otimizado resultante, completando o fluxo de otimização da ferramenta.

Capítulo 3

Otimização do algoritmo de Bellman-Ford

Este capítulo descreve a contribuição individual desta monografia ao trabalho conjunto de implementação do escalonador. Nele é abordada a otimização do algoritmo de Bellman-Ford, que é responsável pela maior parte do tempo de execução do escalonador.

Com esta otimização, busca-se diminuir o impacto da análise de restrições de tempo real (diferencial do escalonador), tornando-o mínimo.

3.1 Idéia básica

O uso do algoritmo de Bellman-Ford em conjunto com a modelagem utilizada [5] nos permite verificar restrições de tempo real a medida que arestas são introduzidas no grafo para indicar o escalonamento de uma instrução.

Porém, este algoritmo tem complexidade O(|V||E|), ou seja, o número de iterações realizado é igual ao número de vértices vezes o número de arestas do grafo. Dependendo da complexidade do código a ser escalonado e das restrições impostas a ele, isto pode resultar em um tempo de escalonamento excessivo.

Felizmente, existem otimizações possíveis no caso do escalonador, tornando o

tempo de execução visivelmente menor. Por exemplo, se uma instrução já foi escalonada, o caminho mais longo dela é o peso da aresta que vai do source até ela, e este caminho continuará a ser o mais longo até ela, não interessando outras arestas que sejam introduzidas em instruções posteriores.

Logo, vértices já escalonados só precisam ser visitados se houver um caminho no grafo partindo do vértice recém escalonado até eles, o que acontece se ambos os vértices fizerem parte de um bloco de código com restrições de tempo real do tipo MAX ou EXACT.

Também não é necessário visitar todos os vértices não escalonados, pois se não houver um caminho entre o vértice recém escalonado e um não escalonado este último não terá seu caminho mais longo modificado.

Como para o caso do grafo utilizado pelo escalonador as duas proposições acima são verdadeiras, só precisamos atualizar os caminhos mais longos dos sucessores do vértice recém escalonado.

3.2 O algoritmo e sua otimização

O algoritmo 1 é a versão original do algoritmo de Bellman-Ford. O primeiro **for** serve para inicializar os caminhos mais longos entre o source e os outros vértices. Se houver uma aresta entre os dois vértices, o caminho mais longo é igual ao peso desta aresta, caso contrário este caminho é inicializado como infinito negativo.

Em seguida são realizadas $|V|$ iterações, nas quais os caminhos mais longos são atualizados. Estas $|V|$ iterações garantem que os caminhos obtidos serão os mais longos possíveis, a não ser que haja um ciclo positivo no grafo, o que caracterizaria uma restrição de tempo real que não pode ser respeitada pelo escalonamento obtido.

O algoritmo 2 mostra a otimização realizada. A função *reachableVertexes* retorna os números dos vértices do grafo que são alcançáveis a partir do vértice recém escalonado. Ao invés de se realizarem $|V|$ iterações sobre todos os vértices, elas são realizadas apenas levando-se em conta os sucessores do último vértice escalonado.

Como no início do escalonamento de um bloco de código todas as instruções

podem ser alcançáveis a partir do vértice recém escalonado, o pior caso do algoritmo manteve-se o mesmo, ou seja O(|V||E|). No entanto, a medida que mais vértices vão sendo escalonados, a complexidade a cada execução do algoritmo é menor.

Algorithm 1 Cálculo dos caminhos mais longos

```
1:  s_{0,1} = 0;
2:  for i = 1 to n do
3:      if ∃(v_0, v_i) then
4:          s_{i,1} = w_{0i};
5:      else
6:          s_{i,1} = -∞;
7:      end if
8:  end for
9:  for j = 1 to n do
10:     for i = 1 to n do
11:         for k = 1 to n do
12:             if k ≠ i then
13:                 s_{i,j+1} = MAX{s_{i,j}, (s_{i,k} + w_{ki})};
14:             end if
15:         end for
16:     end for
17:     if s_{i,j+1} == s_{i,j} ∀i then
18:         return TRUE;
19:     end if
20: end for
21: return FALSE;
```

Algorithm 2 Cálculo dos caminhos mais longos - Otimizado

1: $s_{0,1} = 0$;
2: **for** i = 1 to n **do**
3: **if** $\exists(v_0, v\_i)$ **then**
4: $s_{i,1} = w_{0i}$;
5: **else**
6: $s_{i,1} = -\infty$;
7: **end if**
8: **end for**
9: $modifiedVertexes \leftarrow reachableVertexes(lastModifiedVertex)$
10: **for** j = 1 to n **do**
11: **for each** v **in** modifiedVertexes **do**
12: **for** k = 1 to n **do**
13: **if** $k \neq v$ **then**
14: $s_{i,j+1} = \text{MAX}\{s_{i,j}, (s_{i,k} + w_{ki})\}$;
15: **end if**
16: **end for**
17: **end for**
18: **if** $s_{i,j+1} == s_{i,j}$ $\forall$i **then**
19: return TRUE;
20: **end if**
21: **end for**
22: return FALSE;

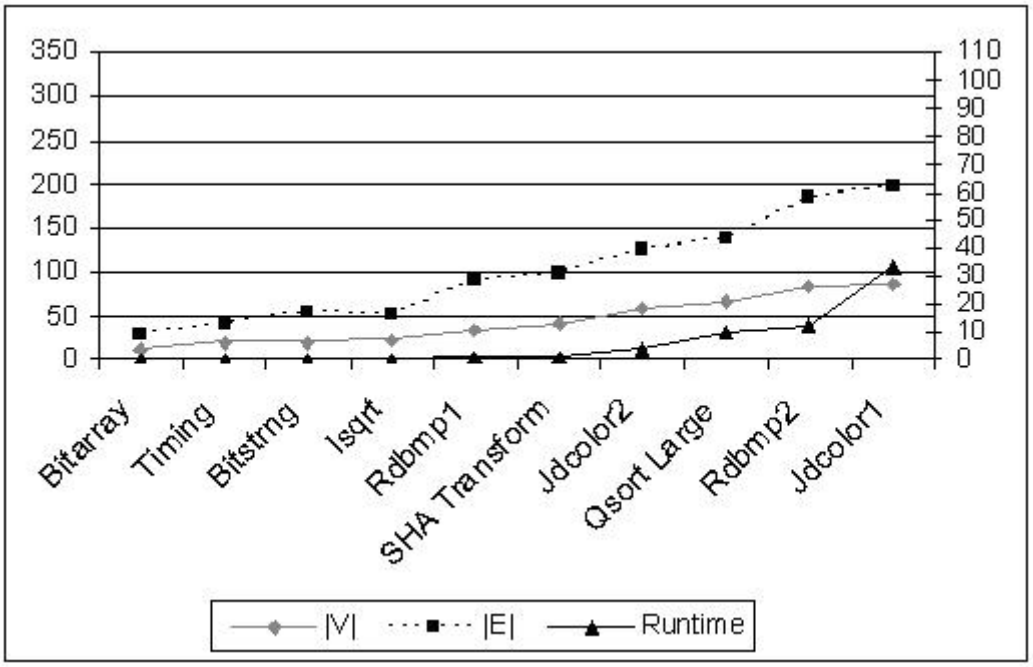

Figura 3.1: Tempos de execução antes da otimização, correlacionados com a complexidade do grafo (MIPS).

3.3 O impacto da otimização proposta

O resultado da otimização pode ser percebido observando-se a diminuição dos tempos de execução para uma série de benchmarks descritos no apêndice A. Na figura 3.1, temos os tempos de execução do escalonamento para o processador MIPS antes da otimização, e na figura 3.2, após a modificação.

Pode-se notar que a otimização realizada reduziu o tempo de execução do escalonador visivelmente. Em alguns casos o tempo de execução é quase três vezes menor que usando-se o algoritmo original.

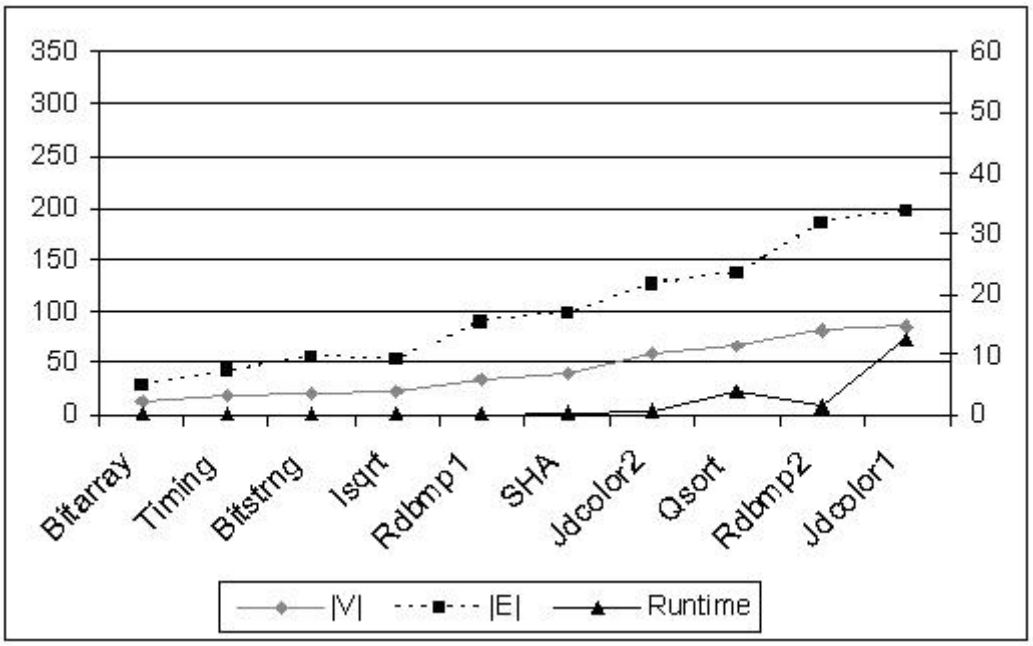

Figura 3.2: Tempos de execução após a otimização, correlacionados com a complexidade do grafo (MIPS).

Capítulo 4

Alocação de Registradores

A alocação de registradores é um passo importante na geração de código, pois é quando os valores simbólicos são mapeados para registradores ou posições da memória. Uma alocação ineficiente deixa o código lento, pois tende a usar os registradores muito depressa, levando ao uso de posições de memória para guardar valores usados no programa.

O assunto da alocação de registradores já foi estudado extensivamente na literatura, sendo que a abordagem predominante é a descrita em [3], que abstrai a alocação como um problema de coloração de grafos. Esta será a abordagem utilizada por este trabalho. Um dos problemas desta abordagem é o seu desempenho no pior caso possível, devido a complexidade da coloração de grafos.

Existem abordagens mais eficientes, mas que em geral não produzem um resultado tão bom quanto o da coloração. Entre estas abordagens, temos algoritmos simples como alocar os k registradores disponíveis para as k variáveis mais usadas, o que resulta em código ineficiente já que todas as outras variáveis tem de ser guardadas em memória.

Em [7] é descrito um algoritmo eficiente que passa uma vez pelos intervalos de vida das variáveis e faz uma alocação gulosa de registradores que em geral resulta em código um pouco menos eficiente do que se fosse usada coloração de grafos. Este algoritmo é muitas vezes mais rápido do que a coloração de grafos, o que o torna uma boa opção quando alocação rápida e código eficiente são desejados, como em compiladores

just-in-time ou ambiente interativos de desenvolvimento.

4.1 Modelagem

A abordagem descrita em [3] consiste em criar um grafo de conflito dos intervalos de vida das variáveis simbólicas do programa. Os vértices deste grafo são as variáveis do programa, e as arestas representam conflitos dos intervalos de vida entre os vértices ligados por elas, ou seja, dois vértices são adjacentes se uma das váriaveis representadas por eles está viva no momento em que a outra é definida.

A coloração deste grafo nos dá o nome dos registradores que vão substituir as váriaveis simbólicas. Se o número cromático deste grafo for menor ou igual ao número de registradores disponíveis, a alocação é válida. Caso contrário, se o número cromático for maior, é necessário realizar o spilling de variáveis para a memória, com o fim de reduzir o número cromático do grafo até que uma alocação válida seja alcançada.

A figura 4.1 mostra um exemplo desta abordagem. Cada grupo de valores com a mesma cor utilizará o mesmo registrador.

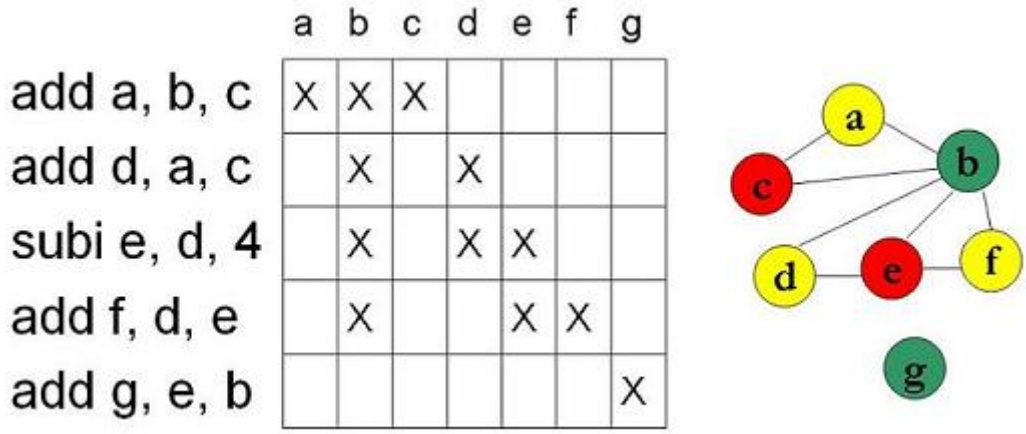

Figura 4.1: Código, intervalos de vida dos valores e grafo de conflito colorido.

4.2 Algoritmos

O algoritmo 3 mostra como o grafo de conflito entre os intervalos de vida dos valores é construído. Os vértices do grafo são os valores presentes no código. Primeiro determinam-se os intervalos de vida de cada valor, em termos de caminhos no CFG, o que é realizado pelo procedimento *getLifeIntervals*. A seguir, para cada par de valores adiciona-se uma aresta entre eles se houver intersecção entre seus intervalos de vida.

Este grafo de conflito é então colorido usando-se o algoritmo 4. Este algoritmo foi proposto em [3] e é recursivo. Ele baseia-se na idéia de que para achar uma k-coloração de um grafo que tenha um nodo com grau menor que k, é preciso que o subgrafo resultante da sua remoção tenha uma k-coloração.

Inicialmente, removem-se os vértices com grau menor que k um a um, até que o grafo esteja vazio. Caso em algum momento só restem vértices com grau maior ou igual a k, é necessário fazer o *spilling* do valor, armazenando em uma posição de memória ao invés de em um registrador. A coloração então continua se o subgrafo resultante tem algum vértice com grau menor que k.

Após esvaziar-se o grafo, os vértices e suas arestas são recolocados no grafo um a um, na ordem inversa à da remoção, e escolhe-se uma cor qualquer para cada vértice, desde que não seja usada por nenhum vértice adjacente.

Algorithm 3 Construção do grafo de conflito

1: $lifeIntervals \leftarrow getLifeIntervals(G, V)$
2: $E \leftarrow \emptyset$
3: **for each pair** (V_i, V_j) **do**
4: **if** $lifeIntervals(i) \cap lifeIntervals(j)$ **then**
5: $E \leftarrow E \cup (V_i, V_j)$
6: **end if**
7: **end for**
8: **return** $C(V, E)$

Algorithm 4 Coloração do grafo de conflito

1: **if** $V = \emptyset$ **then**
2: **return**
3: **end if**
4: **if not** $\exists\, v \in V \mid neighbors(v) < availableColors$ **then**
5: $spilledNode \leftarrow spillNode(V)$
6: **return** $colorGraph(E - adjecentEdges(spilledNode), V - spilledNode)$
7: **end if**
8: $node \leftarrow select(V \mid neighbors(v) < availableColors)$
9: $coloring \leftarrow colorGraph(E - adjecentEdges(node), V - node)$
10: $coloring(node) \leftarrow select(availableColors - coloring(x) \mid x \in neighbors(node))$
11: **return** $coloring$

Capítulo 5

Validação experimental do escalonador

A ferramenta foi validada através do escalonamento de uma série de benchmarks, caracterizados no apêndice A, para três arquiteturas diferentes: MIPS, PowerPC e SPARC.

Os experimentos foram realizados em computador com processador Pentium 4, operando a freqência de 3 GHz com 1 GB de memória principal, sob o sistema Linux Debian.

Os tempos de execução, correlacionados com a complexidade do grafo são mostrados nas figuras 3.2, 5.1 e 5.2. Pode-se observar que enquanto para o MIPS o benchmark rdbmp2 roda em um tempo muito pequeno, para as arquiteturas PowerPC e SPARC o tempo é significativo. Isto ocorre porque para estas duas últimas a complexidade do grafo resultante é bem maior, devido a particularidades de cada arquitetura.

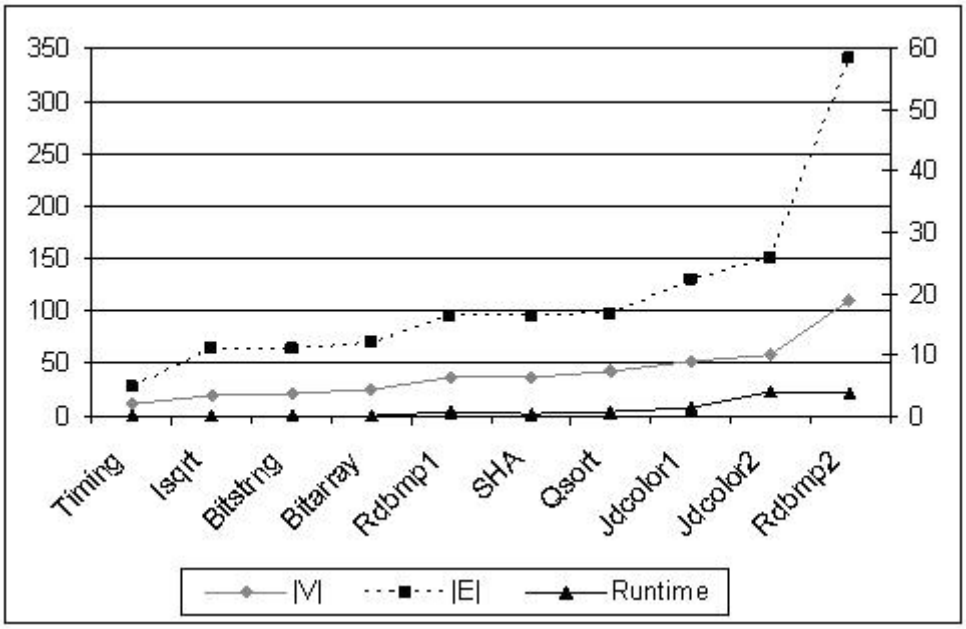

Figura 5.1: Tempos de execução após a otimização, correlacionados com a complexidade do grafo (PowerPC).

A otimização obtida pela ferramenta pode ser observada no gráfico da figura 5.3. A otimização média para os cinco menores segmentos de código foi de 1,18 enquanto para os 5 maiores foi de 1,23. Isso indica que em segmentos maiores há mais locais onde pode haver otimização, e que a otimização resultante do escalonamento é significativa.

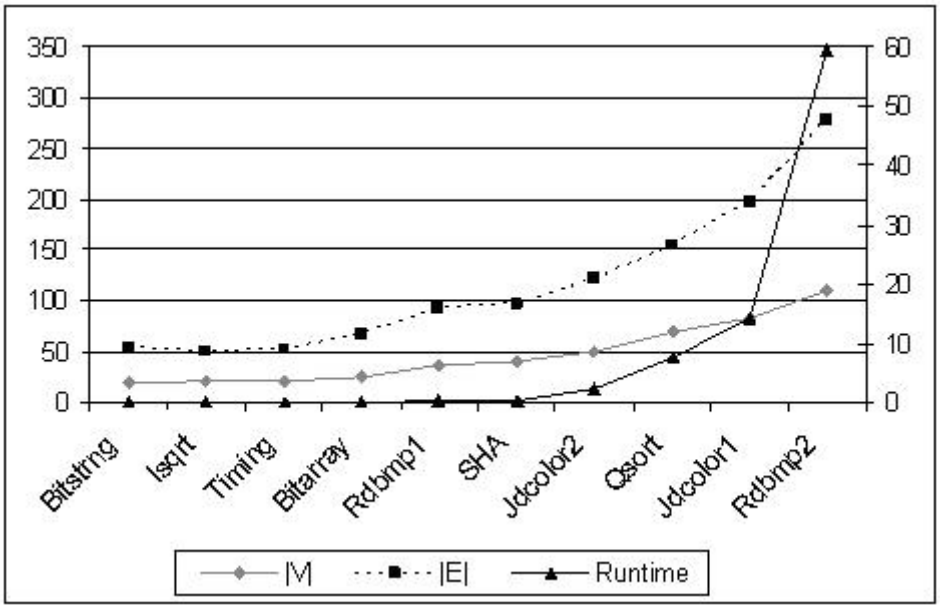

Figura 5.2: Tempos de execução após a otimização, correlacionados com a complexidade do grafo (Sparc).

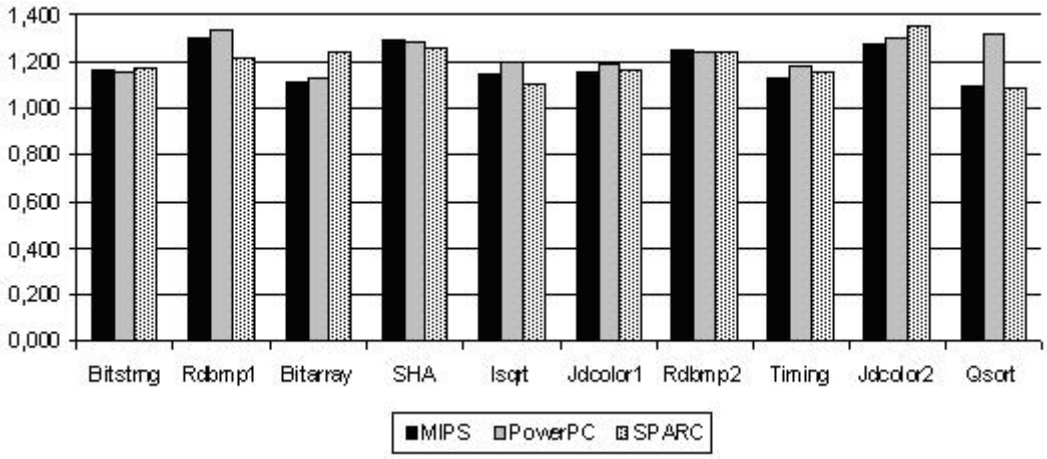

Figura 5.3: Otimização resultante do escalonamento.

Capítulo 6

Conclusões e Trabalhos Futuros

As ferramentas existentes geradas a partir de ADLs não lidam com restrições de tempo real, enquanto os trabalhos relacionados a tempo real normalmente são voltados para só uma arquitetura específica.

Neste trabalho foi descrita uma ferramenta para geração automática de escalonadores com restrições de tempo real a partir de uma descrição ADL. A modelagem realizada permite a análise de restrições de tempo real, e a extração das informações dependentes da arquitetura a partir da descrição torna a ferramenta redirecionável.

A ferramenta proposta trabalha entre o compilador e o montador, e não requer modificações em ferramentas já existentes. Os resultados experimentais mostram que otimizações significativas podem ser obtidas, e que o tempo necessário para a análise das restrições de tempo real não compromete a eficiência da ferramenta.

Os resultados também mostram que a otimização do algoritmo de Bellman-Ford, contribuição individual desta monografia ao trabalho conjunto, reduziu significamente o overhead da análise de restrições de tempo real.

Como trabalhos futuros, o escalonador deve ser aperfeiçoado para realizar otimizações globais e para lidar com modelos mais gerais de máquinas, como processadores superescalares. Novas otimizações na análise de restrições de tempo real também serão buscadas, reduzindo ainda mais o seu impacto no tempo de execução.

Referências Bibliográficas

[1] *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[2] ArchC. *The ArchC Architecture Description Language*. ArchC Team, v1.5 edition, JUL 2005.

[3] Gregory Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 39(4):66–74, 2004.

[4] Rainer Leupers and Peter Marwedel. *Retargetable compiler technology for embedded systems: tools and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[5] Bart Mesman, Adwin H. Timmer, Jef L. van Meerbergen, and Jochen A. G. Jess. Constraint analysis for dsp code generation. pages 485–498, 2002.

[6] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[7] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[8] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, NOV 2001.

Tabela A.1. Caracterização dos benchmarks

Segmento	**Benchmark**	*MIPS*		*Powerpc*		*SPARC*	
		|V|	|E|	|V|	|E|	|V|	|E|
Isqrt	basicmath	22	52	20	63	21	51
Bitarray	bitcount	12	29	26	70	25	67
Bitstrng	bitcount	20	54	22	64	19	55
Qsort	qsort	67	137	43	97	70	154
Jdcolor1	jpeg	86	197	52	130	83	198
Jdcolor2	jpeg	58	126	58	150	51	122
Rdbmp1	jpeg	35	90	36	95	36	93
Rdbmp2	jpeg	83	185	110	340	110	278
SHA	sha	41	98	37	94	41	96
Timing	adpcm	19	42	12	28	22	53