Otimização de Código em Ambiente de Semântica Formal Executável Baseado em ASM.

Fabíola F. de Oliveira[1], Roberto da Silva Bigonha[1], Mariza A. Silva Bigonha[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Belo Horizonte – MG – Brasil

{fabiola,bigonha,mariza}@dcc.ufmg.br

Abstract. *This paper presents the main problems to generate efficient code for Abstract State Machines specifications and proposes an instruction scheduling algorithm based on conflict graph to reduce the occurrences of updating list insertions in the compiled sequential code for the ASM transition rules.*

Resumo. *Apresentam-se os problemas de geração de código eficiente para linguagens de especificação do modelo de Máquinas de Estado Abstratas e propõe-se um método de escalonamento de instruções baseado em grafos de conflito para minimizar as ocorrências de inserção em lista de atualização de função em cada passo da regra de transição.*

1. Introdução

Métodos informais de definição de linguagens de programação carregam uma imprecisão semântica que pode resultar em descrição inconsistente ou ambígua, sendo, por isso, inadequados para descrever precisamente linguagens de programação. Métodos formais são fortes onde os métodos informais apresentam problemas, garantindo precisão e diminuindo as possibilidades de inconsistências em definições de linguagens. Porém, eles também apresentam problemas: podem tornar a leitura e escrita tarefas complicadas, e, além disto, são restritos pela falta de ferramentas que sejam fáceis de utilizar e, ao mesmo tempo, gerem código eficiente.

Máquinas de estado abstratas (ASM, *Abstract State Machines*), introduzidas por Yuri Gurevich [13,14] como um novo modelo computacional, utiliza conceitos simples e bem conhecidos, tornando a leitura e a escrita da especificação bastante direta. Foi originalmente proposta com o objetivo de prover semântica operacional para algoritmos de uma forma mais natural que a máquina de Turing. Este modelo tem sido usado com sucesso para formalizar sistemas de tempo real, seqüenciais, paralelos e distribuídos, além de arquitetura de computadores [5,6,15,20,28].

A idéia por trás deste modelo é realizar simulação de algoritmos via transições de estado. Inicialmente definem-se um estado inicial e uma regra de transição. Estado é um conjunto com nomes de funções e relações e suas interpretações. O conjunto dos nomes de funções e relações é o *vocabulário* do estado. A interpretação de um nome de função ou relação é um mapeamento deste nome do *vocabulário* na respectiva função ou relação. As principais regras são: atualização, regras guardadas e regras bloco.

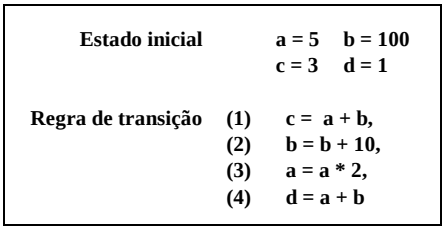
Todas as funções e relações são definidas em um conjunto denominado *superuniverso* do estado, e podem ser dinâmicas ou estáticas. Funções estáticas nunca

são modificadas, mantendo-se com a mesma interpretação em todos os seus pontos durante uma execução. Funções dinâmicas, por outro lado, podem sofrer mudanças na sua interpretação como conseqüência de atualizações durante a execução da regra de transição ou mesmo através do ambiente.

A atualização de funções acontece através da mudança de interpretação da assinatura da função de um estado para outro. Uma função dinâmica sem parâmetros se parece com variáveis de um programa imperativo, pois seu significado é um nome que possui uma única interpretação de valor e que pode ser modificado. A atualização de uma função dinâmica de um ou mais argumentos $f(a_1, ..., a_n) = v$ significa que a interpretação da função, *f*, nos argumentos, $(a_1, ..., a_n)$, passa a ter novo valor, *v*.

Uma computação em ASM causa uma mudança de estado, sendo descrita por uma regra de transição que modifica a interpretação de nomes de função do *vocabulário* do estado. Esse mecanismo de transição de estados por regras de transição assemelha-se a comandos de programação imperativa. Como resultado uma especificação ASM tem a aparência de um programa imperativo, sendo fácil de entender e de executar. A principal diferença entre as duas formas de programação é a ausência de iteração explícita em ASM, que é obtida usando o conceito implícito de execução repetitiva da regra de transição. Para ilustrar um programa no modelo ASM, o Exemplo 1.1 mostra um código em uma linguagem baseada em ASM. No estado inicial, ***a*** tem valor 5, ***b*** tem valor 100, ***c*** tem valor 3 e ***d*** tem valor 1.

As funções ***a***, ***b***, ***c*** e ***d*** passam, após a execução da primeira transição, a ter os valores 10, 110, 105 e 105 respectivamente. Como pode ser visto, elas recebem novos valores durante a execução do código. Também é importante notar que as ocorrências do lado direito das regras referem-se a valores do estado anterior.



Exemplo 1.1

Uma das principais características de ASM é a execução paralela de regras, as quais são executadas em relação a um mesmo estado global. A execução de regras em um dado estado produz um conjunto de atualizações. Para o caso de um conjunto de atualizações consistente, que não apresenta contradições na atualização de alguma localização, aplica-se o conjunto de atualizações ao estado corrente no fim da transição para se obter o próximo estado.

O estudo de otimização aqui realizado é parte de um projeto de implementação de um compilador para um ambiente de execução baseado no modelo ASM [25, 26,32].

Ainda nesta seção serão apresentados alguns problemas identificados na busca por otimizações em compiladores para linguagens baseadas no modelo ASM. Este texto segue mostrando na Seção 2 informações sobre a compilação de linguagens baseadas em ASM, focando na etapa de otimização. Seção 3 apresenta a proposta de solução adotada para a atualização de funções via escalonamento de instruções, destacando o uso de grafos. Na Seção 4 está uma aplicação da solução apresentada juntamente com sua avaliação, e a Seção 5 conclui este texto.

1.1. Problemas na Otimização de Linguagens Baseadas em ASM

ASM define uma notação de especificação executável e provê uma base formal para uma notação que pode ser usada tanto como uma linguagem de especificação quanto de programação de alto nível. Porém, para se tornar uma linguagem de programação real, a notação precisa, além de outros fatores, apresentar um código executável eficiente, que leve em consideração aspectos como espaço de memória e tempo. O atual estado da arte sobre ferramentas de suporte para este modelo oferece implementações com pouca ou nenhuma capacidade de otimização. Assim sendo, com o objetivo de garantir a eficiência na execução das especificações, e, considerando implementação, qualidade do código gerado, assim como o desempenho de ferramentas existentes, o ponto central desta pesquisa é propor técnicas de otimização que considerem as características específicas de linguagens baseadas em ASM.

Uma das particularidades deste modelo, como apresentado na Seção 1, diz respeito à forma de atualização de funções. Nele uma função que tem seu valor alterado em um determinado ponto da regra de transição pode ter seu valor de início da regra acessado em um ponto subsequente na regra. Como o valor de uma função é sempre seu valor original do início da execução de um passo da regra, um comando de atualização de função somente pode ser efetivado no fim da execução de um passo da regra, a fim de preservar seu valor original para possíveis acessos. Uma solução para este problema é substituir os comandos de atualização por informações de atualização inseridas em uma lista a ser processada no fim de cada passo. Isto garante a corretude do modelo, mas pode gerar um número grande de inserções na lista de atualizações em cada passo da regra, encarecendo a execução. Para que isto não ocorra, uma solução é, via técnicas de escalonamento de instruções [8], evitar ao máximo gerar comandos de inserção na lista de atualizações. Este escalonamento consiste em encontrar uma ordenação para as instruções que permita executar as atualizações diretamente e que assim gere a menor lista de atualizações a ser executada no final de um passo da regra.

Outro ponto importante de otimização está relacionado com desvios. Ao terminar a execução de um passo da regra de transição, o controle é transferido para o início da regra para um novo passo ser executado. Com o início desse novo passo, as guardas existentes na regra de transição são reavaliadas para que se possa caminhar pelo algoritmo. Se for possível determinar neste momento se algumas das guardas que estão sendo avaliadas possuem os mesmos valores do passo anterior da execução, então é possível aproveitar tais avaliações na escolha do caminho a seguir no novo passo. Uma solução para este problema é a aplicação de técnicas específicas para **otimização de desvios** [26] que determinam o ponto para o início do próximo estado de forma a evitar o maior número possível de avaliações desnecessárias de guardas boleanas.

Uma terceira característica de ASM oferece importante oportunidade de otimização: o armazenamento eficiente de funções dinâmicas. Como essas funções sofrem mudanças de interpretação durante a execução da regra de transição, elas precisam ser tratadas como uma estrutura de armazenamento de valores, que permita atualização durante a execução, e não como um algoritmo a ser executado. Neste caso, aplicam-se técnicas de otimização específicas para **implementação de funções dinâmicas** [25].

Este artigo está centrado na solução do problema de atualização de funções. Os demais problemas levantados e suas soluções serão abordados no futuro.

2. Ambientes de Execução ASM

Por tratar-se de área nova, ainda há muito a pesquisar sobre otimização em compiladores de linguagens baseadas em ASM, muito embora já existam vários interpretadores e compiladores implementados para executar especificações ASM. Alguns exemplos são o interpretador baseado em C [16], o interpretador baseado em Scheme [11], o compilador baseado em Prolog [18], o interpretador baseado em Prolog [4], EvADE [27] e o interpretador MAXEA [21]. Além desses está disponível GEM, uma ferramenta de edição gráfica [1], para ASM Montages [19]. Montages é uma extensão de ASM para descrever a parte estática de uma especificação de linguagem. Montages já foi usado para linguagens tais como Oberon [20] e Java [28]. O restante desta seção mostra alguns desses sistemas.

O sistema ASM-Workbench [9,10] introduz um esquema de compilação para transformar uma especificação ASM em C++. Seu objetivo é preservar a estrutura de especificação no código gerado sem gerar código ineficiente. Nele, regras podem ser executadas seqüencialmente sem coletar atualizações, ou seja, a execução seqüencial é equivalente à execução paralela. A técnica é baseada em *double buffering*, técnica muito conhecida para aplicações onde imagens precisam ser apresentadas. A técnica introduzida assegura que a execução seqüencial de regras é equivalente semanticamente à sua execução paralela. Esta é uma implementação importante, mas que não teve por objetivo alcançar eficiência de execução. Uma melhoria que ASM-Workbench procurou desenvolver foi investir em uma melhor forma de implementar a lista de atualização.

AsmGofer [22,23] é um sistema de programação ASM que é uma extensão de Gofer [17], uma linguagem funcional que introduz a noção de estado e atualização paralela. Ele foi usado com sucesso para modelos ASM em *Java and the Java Virtual Machine* [24] onde as semânticas estáticas e dinâmicas de Java foram definidas em AsmGofer. Isto inclui analisador sintático e verificação de tipo de programas Java tanto quanto GUI que mostra a avaliação de programas Java passo a passo. Assim como ASM-Workbench e a maioria das implementações para linguagens baseadas em ASM, AsmGofer não se preocupou com eficiência de execução, não investindo em otimizações do código gerado.

Montages [19,3] é um formalismo semi-visual que permite uma especificação unificada e coerente da sintaxe, análise, semântica estática e semântica dinâmica, sendo de fácil entendimento. O aspecto estático de suas descrições assemelha-se a gráficos de fluxo de controle e de fluxo de dados. Assim como AsmGofer, Montages também é uma implementação que não se preocupou com a otimização.

Xasm [2] é uma implementação de ASM seqüencial com foco na geração de programas executáveis eficientes, simulando a execução da especificação. Xasm está entre os compiladores que utilizam técnicas de otimização, sendo que essa técnica é a representação eficiente de funções por meio de tabelas hash.

Para o projeto EvADE [27] foram desenvolvidos um compilador, DASL-ALMA, e um interpretador, leanEA [4]. O compilador aplica uma otimização sobre o programa de álgebra evolutiva, transformando-o em uma árvore de decisão. Os nodos internos desta árvore são condições que são extraídas das guardas das regras de transição. As folhas da árvore de decisão são conjuntos de atualizações. EvADE também implementou uma otimização focada na eliminação de subexpressões comuns, que não

é um problema característico de linguagens baseadas no formalismo ASM, mas sim um problema clássico em otimização de código.

AsmL [29,30] é uma linguagem de especificação executável, baseada no modelo ASM, útil em qualquer situação onde é preciso uma maneira precisa e não ambígua para especificar um sistema, tanto de software quanto de hardware. Uma especificação AsmL é uma forma ideal para comunicações de decisões de projeto para grupos de trabalho.

3. Escalonamento de Instruções

Um dos pontos mais importantes para gerar um código de boa qualidade em programas escritos em uma linguagem baseada em ASM é conseguir atualizações que possam ser feitas diretamente, em vez de precisar gerar comandos para inseri-las em uma lista de atualizações, adiando sua execução até o fim de cada passo. O compilador precisa estar alerta a este fato, identificando as atualizações cujos alvos não são usados em instruções subseqüentes, e, portanto, podem ser executadas diretamente, trazendo melhorias no tempo de execução.

Contudo, como nem todas as atualizações estão nessa categoria, é importante o compilador realizar um escalonamento de instruções com o objetivo de aumentar o número de atualizações que podem ser feitas diretamente no ponto em que são encontradas no código. O escalonamento consiste em uma ordenação das instruções do código de forma a minimizar o tamanho da lista de atualizações gerada. O Exemplo 3.1 (b) ilustra a compilação para a linguagem C das instruções do trecho de código mostrado em (a) em uma linguagem baseada em ASM. O Exemplo 3.1 (c) ilustra a compilação do mesmo trecho de código realizando o escalonamento de tais instruções.

1	**a := 10,**	**1**	**Insert(a,10);**	**3**	**y = 2 + x;**
2	**b := a + 1,**	**2**	**Insert(b,a+1);**	**4**	**c = b;**
3	**y := 2 + x,**	**3**	**Insert(y,2+x);**	**5**	**if (a > 0) x = a;**
4	**c := b,**	**4**	**Insert(c,b);**	**2**	**b = a + 1;**
5	**if (a > 0) then x := a**	**5**	**if (a > 0) Insert(x,a)**	**1**	**a = 10**
	(a)		**(b)**		**(c)**

Exemplo 3.1 Escalonamento de instruções.(a) Código original. (b) Código C seqüencial (c) Código C escalonado seqüencial..

No Exemplo 3.1 (c), a instrução **y = 2 + x** é escalonada primeiro porque ***y*** é uma função que não é consultada depois deste comando ser executado. Por motivo semelhante a instrução **c = b** é escalonada logo a seguir, pois ***c*** não é consultada depois deste comando ser executado. A partir do momento que a instrução 3 foi selecionada, é possível selecionar a instrução **if (a > 0) then x = a** pois, com isso, a função ***x*** não será mais consultada. Também por motivo semelhante a instrução **b = a + 1** é escalonada a seguir, pois, sendo colocada após a instrução 4, a função ***b*** não será mais consultada. Por fim, a instrução **a = 10** é escalonada. Assim, neste exemplo, nenhuma inserção na lista de atualização precisa ser feita no código escalonado.

3.1. Algoritmo de Escalonamento Proposto

Seja **B** uma **regra bloco** $R_1 \ldots R_k$, onde R_i é uma **regra de atualização** ou uma **regra guardada**. Fazer um escalonamento da **regra bloco B** consiste em rearranjar as instruções R_i, de forma a minimizar inserções na lista de atualizações. Suponha que B seja a **regra bloco**

***a* := 10,**
***b* := *a* + 1**

Uma compilação para código seqüencial que preserve a semântica da regra acima deve guardar o valor de ***a*** antes de executar a atualização ***a* := 10**, pois ele é necessário no segundo comando. Mas, ao inverter a ordem dos comandos, o valor de ***a*** estará sendo modificado somente após seus usos. Por isso poderá ser atualizado no local de sua ocorrência.

Esta otimização pode ser modelada como um problema de grafo. A estrutura de dados mais importante no algoritmo de escalonamento de instruções pode ser representada por um grafo dirigido [7]. Estão representadas nessa estrutura de dados as instruções encontradas em uma **regra** ASM. Um **grafo de escalonamento ASM**, GEASM, G_s, = (V_s, E_s) é um grafo dirigido, dito **grafo de conflito**, construído da seguinte forma:

- Todo vértice $v \in V_s$ corresponde a uma instrução da **regra**, sendo que cada vértice tem como peso o benefício potencial pela retirada desse vértice durante o escalonamento.
- Existe uma **aresta dirigida** $(v_1, v_2) \in E_s$, de v_1 para v_2, **se e somente se** a regra v_1 deve ser executada antes de v_2, ou seja, se uma função consultada na instrução do vértice v_1 estiver sendo alterada na instrução do vértice v_2. Diz-se que v_1 **deve preceder** v_2, ou que v_2 **deve ser precedido** v_1;
- As arestas de G_s são rotuladas com um mesmo peso.

Definição 3.1.1:
Um caminho $v_i \rightarrow v_j$ em um grafo GEASM é uma seqüência de vértices distintos (v_i, v_{i+1},, ..., v_j), que representam um conjunto de vértices tal que (v_k, v_{k+1}), i ≤ k ≤ j, é uma aresta em G_s.

Definição 3.1.2:
O **grau de entrada** de um vértice $v_i \in V_s$ é o número de arestas $(v_k, v_i) \in E_s$.

Definição 3.1.3:
Um **ciclo** em G_s é uma seqüência de vértices na forma (v_i, v_{i+1},, ..., v_j, v_i) tal que (v_i, v_{i+1},, ..., v_j) forma um caminho em G_s, para i, j > 0 e (v_j, v_i) é uma aresta em G_s.

Definição 3.1.4:
Um **ciclo mínimo** em G_s é um ciclo em que todas as arestas entre dois vértices do ciclo sejam arestas pertencentes ao ciclo.

Definição 3.1.5:
Um **grupo** {v_j,.., v_i} de G_s, que representa alguma alocação de comandos, é um subconjunto de vértices com grau de entrada zero.

Definição 3.1.6:
Uma **cobertura por vértice de um grafo** GEASM é um subconjunto de V_s que minimiza o número total de funções que participarão da lista de atualizações de um passo do programa ASM.

O algoritmo proposto é constituído de três passos. O primeiro deles é responsável por montar as dependências entre as funções. No segundo passo é construído o grafo de

conflitos a partir das dependências detectadas anteriormente. O último passo é responsável pelo escalonamento das instruções.

No primeiro passo do algoritmo de escalonamento, a **regra bloco** é analisada para verificar quais funções são consultadas e quais são alteradas (vide Figura 1).

	Instrução	Altera	Consulta
1	a := 10,	a	–
2	b := a + 1,	b	a
3	y := 2 + x,	y	x
4	c := b,	c	b
5	x := a	x	a

Figura 1 Coleta das informações de alterações e consultas

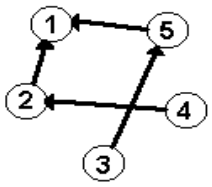


Figura 2 Grafo de conflitos

No segundo passo é construído o grafo de conflitos. Com a **regra bloco** da Figura 1 é possível montar a grafo da Figura 2. Nele existe uma aresta da instrução 2 para a 1, pois ***a*** é atualizada em 1 e consultada em 2. Da mesma forma existe uma aresta da instrução 5 para a 1, pois ***a*** é atualizada em 1 e consultada em 5, e assim sucessivamente.

No último passo desse algoritmo o escalonamento é realizado. O escalonamento é um problema NP-Completo [7], de modo que foi utilizada uma heurística para resolvê-lo. A idéia principal desse algoritmo é retirar vértices do grafo de conflitos até ele não ter mais vértice para retirar. Retirar um vértice significa gerar código na compilação. Esse código pode ser uma atualização direta na função ou um código para inserção da instrução na lista de atualizações executada no final do código gerado pela **regra bloco**. A escolha do próximo vértice a ser removido do grafo obedece ao seguinte algoritmo:

1. Se não existem vértices no grafo, o escalonamento das instruções está completo e o algoritmo termina.

2. Se ainda existem vértices no grafo, é feita uma pesquisa para montar o **grupo** atual de G_s, ou seja, vértices com grau de entrada igual a zero são escolhidos.

   2.1 Se um **grupo** puder ser montado, os vértices pertencentes a esse **grupo** saem do grafo para ocupar uma posição na lista de comandos.

   2.2 Se nenhum **grupo** puder ser montado, ou seja, se em um determinado momento todos os vértices possuem grau de entrada diferente de zero, é preciso escolher um vértice, que necessariamente provoque inserção da instrução na lista de atualizações.

   2.2.1 Para escolher um vértice com grau de entrada diferente de zero procuram-se os **ciclos mínimos** do grafo atual. Dentre os vértices que participam dos ciclos mínimos encontrados, aquele que tiver maior relação de benefício devido à sua retirada do grafo é o vértice escolhido para sair do grafo e gerar inserção na lista de atualizações.

3. Volta-se ao Passo 1 do algoritmo para continuar o escalonamento.

Vértices selecionados no Passo 2.1 são traduzidos para código que realiza a atualização imediatamente, sendo que vértices selecionados no Passo 2.2 são traduzidos

para instruções que inserem informações na lista de atualizações. Por essa razão, quanto mais vértices puderem ser escolhidos no Passo 2.1 melhor é o código produzido.

A idéia subjacente nesta heurística é a seguinte: o Passo 2.1 tem precedência sobre o Passo 2.2, pois vértices, ou seja, atualizações, que não afetam outras regras podem ser escalonadas para atualização direta. O Passo 2.2 está baseado na tentativa de quebrar ciclos, além de procurar ter um grafo resultante com maior relação de benefício devido a retirar o vértice escolhido do grafo.

Para encontrar ciclos mínimos no grafo G_s é necessário encontrar caminhos mais curtos (v_i, v_{i+1},, ..., v_j) entre todos os vértices do grafo, verificando então se (v_j, v_i) é uma aresta em G_s, formando um ciclo. Para resolver esse problema é necessário executar um algoritmo de caminhos mais curtos de origem única $|V_s|$ vezes, uma para cada vértice sendo usado como origem. Para tanto, utiliza-se o algoritmo de Dijkstra [7].

Os vértices que participam dos ciclos mínimos encontrados são candidatos a serem retirados do grafo. Nesse momento é preciso calcular a relação de benefício de cada um dos vértices candidatos, para, então, escolher um dentre esses vértices.

Para a relação de benefício é importante o benefício potencial de um vértice, que leva em consideração a relação de custo (***e***/***u***) entre o grau de entrada de um vértice (***e***) e seu peso no caso de sua inserção na lista de atualização (***u***). O objetivo é encontrar um vértice (v_i) que provoque transformações no grafo no sentido de se obter um maior impacto no custo dos vértices (v_{ij}, $1 \leq j \leq k$) que devem ser precedidos pelo vértice a ser retirado.

Assim propõe-se calcular o benefício potencial (***B***) da retirada do vértice em função do custo dos vértices que este precede. A base de calculo desse benefício é a fórmula ao lado.

$$B_i = \sum_{j=1}^{K} \frac{u_j}{e_j}$$

Nessa fórmula, **i** representa o vértice que está tendo seu benefício em potencial (B_i) analisado, e **j** representa o conjunto de vértices que tenham como precedente o vértice **i**. O benefício final ($Bfinal_i$) do vértice **i** é o seu benefício em potencial subtraído seu peso no caso de uma inserção na lista de atualização, que é o benefício devido à retirada do vértice **i** considerando seu impacto em evitar inserção na lista de atualização de seus sucessores ($Bfinal_i = B_i - u_i$).

A Figura 3 ilustra uma instância do grafo no momento em que o benefício potencial de um vértice v_i está sendo analisado. Para essa análise são importantes os vértices sucessores v_{ij}, $1 < j < k$.

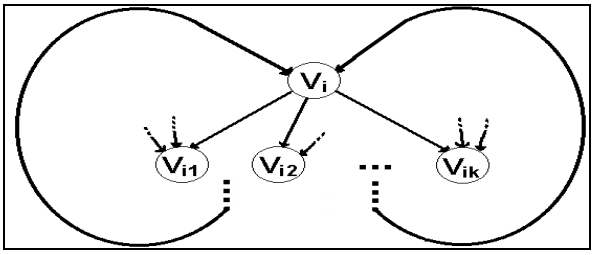


Figura 3 Relação no grafo entre v_i e seus sucessores v_j

A complexidade do algoritmo proposto é $O(n^3)$, cujo cálculo foi omitido por falta de espaço. Embora a complexidade possa ser considerada alta do ponto de vista teórico, na prática o algoritmo é efetivo porque o tamanho **n** da entrada refere-se ao número de instruções contidas em blocos básicos, tipicamente não superando a poucas dezenas. Por outro lado algoritmos de otimização encontrados na literatura têm custos equivalentes, demonstrando a inerente complexidade do problema. Por exemplo, a construção de um PDG, técnica recente muito utilizada para otimização em compiladores e apresentada a

seguir, tem também custo com ordem de crescimento elevada em termos de tempo de computação, $O(n^3)$ [31], onde **n** é o número de comandos do programa.

3.2. Grafo de Dependência de Programa

O grafo de dependência de programa (PDG – *program dependence graph*) [12] é uma representação de código projetada para ser usada em otimização e que visa expor o paralelismo potencial do código fonte. Essa representação procura compilar programas com paralelismo para código a ser executado em uma máquina seqüencial. Como a técnica proposta pelo PDG produz excelentes resultados [12] e a implementação de ASM também baseia-se no mapeamento de paralelismo em máquinas seqüenciais, foram realizados estudos para verificar a viabilidade de utilizar PDG na otimização que está sendo apresentada.

PDG representa explicitamente dependências de controle e de dados, por meio de um grafo de dependência de controle (CDG) e de dependência de dados, respectivamente. O PDG somente força uma ordem de execução nas operações que dependem umas das outras. Assim, ele mostra somente a seqüência mínima necessária das operações. Um vértice no PDG representa um bloco básico, comados ou operações e uma aresta representa dependência de controle e/ou dado.

Grafo de dependência de dados é usado por compiladores otimizantes como uma representação explícita das relações de uso e definição implicitamente presentes em um programa fonte. Um grafo de fluxo de controle é a representação usual para a relação de fluxo de controle de um programa. Essas relações de dependências determinam a seqüência necessária entre operações.

Ao analisar a estrutura de um programa ASM é possível verificar que uma das vantagens de ASM é a execução paralela de regras que é realizada em relação a um mesmo estado global. A execução de regras em um dado estado produz um conjunto de atualizações que é aplicado ao estado corrente para se obter o próximo estado. Ou seja, funções usadas em um estado da máquina são definidas em estados anteriores, bem como as funções que são definidas no estado atual da execução só são usadas em estados seguintes. Dada essa característica, os comandos de um programa em ASM não apresentam dependência de dados. Os comandos existentes em ASM são regras para atribuição e regras condicionais. Nenhuma função que tem seu valor modificado por uma regra de atribuição em um estado E_i tem seu novo valor utilizado em uma outra regra durante a execução do mesmo estado E_i. O que é importante para a otimização em um programa ASM é o conflito entre as regras, e não a dependência de dados que é representada no PDG.

Outra característica de ASM é que a dependência de controle se restringe às regras guardadas, e, por isso, é possível verificar que o grafo de conflitos necessário em ASM não é influenciado pelas informações de controle. Para ilustrar este fato, a Figura 4 mostra um programa ASM, e a Figura 5, o PDG para os comandos do programa.

Ao analisar a construção do grafo de conflitos entre as regras de um programa ASM verifica-se que o escalonamento das instruções tem por objetivo minimizar o tamanho da lista de atualizações que são executadas no fim da regra. Portanto o PDG não é exatamente a melhor representação para um programa ASM, pois ele reflete informações que não são as mais relevantes na programação ASM, bem como não considera as características mais importantes deste modelo de programação.

01	a = 2
02	b = a + 4
03	c = 10 + f
04	d = c + b
05	f = d + 2
06	if (e and g) then
07	h = 2 x f
08	i = h + 7
09	if (c < 100) then
10	j = a
11	else j = b endif
12	else h = d + f
13	i = f + h
14	if (b < 50) then
15	j = c
16	else j = a endif
	endif
17	k = i + d
18	l = h + j

Figura 4 Programa em ASM

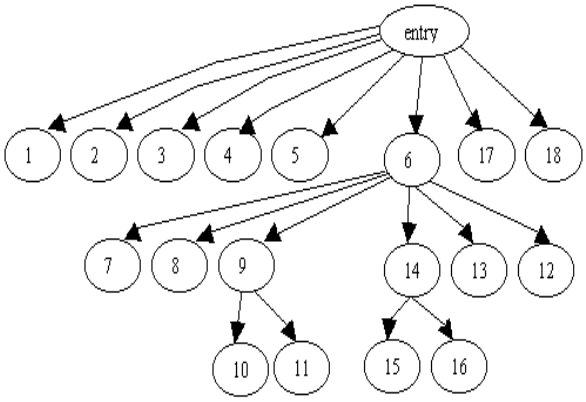


Figura 5 Grafo PDG referente à Figura 4

4. Aplicação e Avaliação do Algoritmo de Escalonamento

Nesta seção apresenta-se um exemplo de otimização em um programa muito simples no modelo ASM, bem como traduções desse código para a linguagem C. Neste exemplo de tradução para a linguagem C estão apresentados comandos que necessariamente vão gerar inclusão de atribuições na lista de atualizações para serem executadas somente no fim da regra de transição. Figura 6 apresenta um programa no modelo ASM. Nele os comandos estão separados por vírgula, indicando o paralelismo entre eles.

1	if (b1 < b2) then	a1 = b1,	a2 = b1,+b2,	a3 = b2	endif,		
2	if (a1 < d) then	c1 = a1,	c2 = a2,	c3 = a3,	c4 = d,	c5 = f1 + f2	endif,
3	if (c1 = c2) then	b1 = c2,	b2 = c1	endif,			
4	d = e2 + e3,						
5	if (c1 < c2) then	e1 = c3 + c1,	e2 = c4,	e3 = c5 + c2	endif,		
6	if (b1 < 100) then	f1 = b2,	f2 = b1	endif,			
7	g = e1 + e2 + f1,						
8	h = c1 + d,						
9	i = h						

Figura 6 - Programa no modelo ASM.

A Figura 7 mostra a tradução do código da Figura 6 para a linguagem C sem a etapa de otimização, com a geração da lista de atualizações feita pela função **Insert**. Com isso a lista de atualizações contém quinze comandos que são executados no fim da regra, pela função **ExecuteUpdate**.

1	if (b1 < b2) {	Insert(a1,b1);	Insert(a2,b1+b2);	Insert(a3,b2);	};		
2	if (a1 < d) {	Insert(c1,a1);	Insert(c2,a2);	Insert(c3,a3);	Insert(c4,d);	Insert(c5,f1+f2);	};
3	if (c1 = c2) {	Insert(b1,c2);	Insert(b2,c1);	};			
4	Insert(d,e2 + e3);						
5	if (c1 < c2) {	Insert(e1,c3+c1);	Insert(e2,c4);	e3 = c5 + c2	};		
6	if (b1 < 100) {	Insert(f1,b2);	f2 = b1;	};			
7	g = e1 + e2 + f1;						
8	Insert(h,c1 + d);						
9	i = h;						
10	ExecuteUpdate;						

Figura 7 - Tradução do programa da Figura 6 para a linguagem C, sem otimização.

Com a utilização da heurística proposta neste artigo, durante a etapa de otimização, a lista de atualizações diminui significativamente, trazendo um ganho na execução de cada passo da regra de transição. A lista de atualizações utilizando a heurística proposta contém cinco atribuições, como pode ser visto na Figura 8.

```
7   g = e1 + e2 + f1;
9   i = h;
8   h = c1 + d;
2   if (a1 < d) {       Insert(c1,a1);   Insert(c2,a2);   Insert(c3,a3);   Insert(c4,d);   Insert(c5,f1+f2);   };
1   if (b1 < b2) {      a1 = b1;         a2 = b1,+b2;     a3 = b2;         };
4   d = e2 + e3;
6   if (b1 < 100) {     f1 = b2;         f2 = b1;         };
3   if (c1 = c2) {      b1 = c2;         b2 = c1;         };
5   if (c1 < c2) {      e1 = c3 + c1;    e2 = c4;         e3 = c5 + c2     };
10  ExecuteUpdate;
```

Figura 8 - Tradução do programa da Figura 6 para C, com otimização da heurística proposta.

Ainda a título de análise, a Figura 9 mostra o resultado obtido com a execução de um algoritmo de otimização de força bruta, cuja complexidade é exponencial. Ele calcula todas as possibilidades de escalonamento das instruções, verificando quantas atribuições são incluídas na lista de atualizações para cada escalonamento. Com isso escolhe o escalonamento que gera a menor quantidade de atribuições na lista de atualizações, que nesse caso é de três atribuições.

```
7   g = e1 + e2 + f1;
9   i = h;
8   h = c1 + d;
3   if (c1 = c2) {      Lista(b1,c2);    Lista(b2,c1);    };
4   Lista(d,e2 + e3);
5   if (c1 < c2) {      e1 = c3 + c1;    e2 = c4;         e3 = c5 + c2     };
2   if (a1 < d) {       c1 = a1;         c2 = a2;         c3 = a3;         c4 = d;   c5 = f1 + f2;   };
1   if (b1 < b2) {      a1 = b1;         a2 = b1,+b2;     a3 = b2;         };
6   if (b1 < 100) {     f1 = b2;         f2 = b1;         };
10  ExecuteUpdate;
```

Figura 9 - Tradução do programa da Figura 6 para C, com otimização de força bruta.

A partir do exemplo mencionado é possível acompanhar a idéia para obter uma otimização no tamanho da lista de atualizações gerada na tradução do programa no modelo ASM para um programa na linguagem C. Também é possível perceber o ganho na execução com a utilização da heurística proposta. Apesar de não alcançar o tamanho ótimo para a lista de atualizações, obtido pelo algoritmo de força bruta (3 inserções), esta heurística encontra uma solução próxima à solução ótima (5 inserções), bem melhor que a solução sem otimização (15 inserções).

A seguir, na Figura 10, está um outro exemplo de programa no modelo ASM. A configuração ótima para o tamanho da lista de atualizações neste novo exemplo, obtido pelo algoritmo de força bruta, é de 7 inserções, que foi encontrado em um tempo de execução aproximado de 14 segundos. Utilizando a heurística proposta foi possível encontrar uma solução próxima à ótima, com 8 inserções na lista de atualizações, solução esta que foi encontrada em um tempo de execução aproximado de 0,5 segundos. O tempo de execução reduziu-se aproximadamente 28 vezes.

1	if (e3) then	a1 = a2,	b1 = b2,	c1 = c3,	Endif,				
2	if (g3) then	a2 = a3,	b2 = c2,	c2 = d2,	d2 = e2,	e2 = g2,	f2 = not f2,	g2 = g2*10,	endif,
3	if (e4) then	a3 = a5,	b3 = b4,	c3 = c11,	d3 = f3*5,	e3 =not e3,	f3 = f3-10,	g3 = not g3,	endif,
4	if (c11) then	a4 = a1,	b4 = b11,	c4 = c1,	d4 = d4+5,	e4 = not e4,	endif,		
5	if (f2) then	a5 = a2,	b5 = b6,	c5 =not c5,	endif,				
6	if (b8) then	a6 = a8,	b6 = c6,	c6 = d6,	d6 = e6,	e6 = f6,	f6 = g6,	g6 = g6-1,	endif,
7	if (c5) then	a7 = a5,	b7 = d7,	c7 = not c7,	d7 = e7,	e7 = e7*2,	endif,		
8	if (c11) then	a8 = a7,	b8 = not b8,	endif,					
9	if (b8) then	a9 = a6,	b9 = b6,	c9 = c6,	endif,				
10	a10=a9+b9,								
11	if (a10>10) then	a11 = a10,	b11 = b11+2,	c11= not c11, endif,					

Figura 10 - Programa no modelo ASM.

A solução ótima encontrada pelo algoritmo de força bruta tem como escalonamento para as instruções a seqüência (1), (5), (2), (3), (4), (10), (9), (6), (8), (7), (11). A solução encontrada pela heurística tem como escalonamento para as instruções a seqüência (8), (7), (1), (5), (2), (3), (4), (11), (10), (9), (6). A Figura 11 apresenta o grafo de conflitos utilizado na heurística para encontrar uma solução para o problema de otimização do programa da Figura 10.

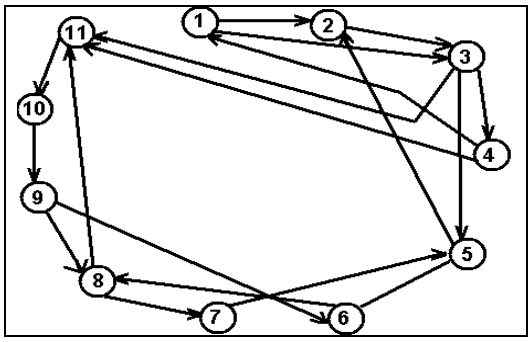


Figura 11 – Grafo de Conflitos relativo à Figura 10.

Considerando os exemplos analisados existe um ganho de tempo de execução da heurística em relação ao algoritmo de força bruta que varia de 20% a 350% (exemplo mostrado anteriormente). Pelos ganhos no tempo de execução e pelo fato das soluções encontradas pelo algoritmo com heurística serem próximas às soluções ótimas possíveis de serem encontradas pelo algoritmo com força bruta, é possível concluir que a estratégia heurística é mais interessante que uma implementação de força bruta.

5. Conclusão

Esse artigo apresenta uma revisão do modelo ASM e de compiladores existentes para linguagens nele baseadas. O modelo ASM possui características especiais que dificultam a geração de código eficiente.

Seção 3 mostra uma possível solução para uma questão inerente ao modelo ASM, via o escalonamento de instruções, que consiste na reordenação de regras de um bloco visando diminuir o número de inserções na lista de atualizações de um passo de uma regra. Ainda nessa seção é feita uma análise para uso do grafo PDG para auxiliar nessa otimização. O que ficou constatado é que o grafo PDG apresenta características distintas daquelas necessárias para a otimização que se deseja no modelo ASM, portanto sem benefício evidente para as otimizações desejadas.

Seção 4 mostra um exemplo de programa no modelo ASM, assim como traduções deste programa para a linguagem C, sem e com otimização. Outros testes mais complexos precisam ser implementados para apresentar dados estatísticos que validem efetivamente a eficácia da heurística proposta. Além disso, os demais problemas relacionados ao modelo ASM, levantados na Seção 1, precisam ser estudados.

Referências

[1] M. Anlauff. *GEM – A graphical editor for montages.* Berkley, 1997.

[2] M. Anlauff. *Xasm –An Extensible, Component-Based Abstract State Machines Language.* In Proceedings of the ASM 2000 Workshop, pp 1-21, Monte Veritá, Switzerland, Março 2000.

[3] M. Anlauff, P. W. Kutter and A. Pierantonio. *Tool Suport for Language Design and Prototyping with Montages.* In Proceedings of Compiler Construction (CC´99). Springer, LNCS, 1999.

[4] B. Beckert and J. Posegga. *leanEA: A Lean Evolving Algebra Compiler.* In: Computer Science Logic, Selected papers from CSL'95, ed. H.K. Büning, Springer Lecture Notes in Computer Science 1092, 1996, pp 64-85.

[5] E. Borger and J. Huggings. *Abstract State Machine 1988-1998: Commented ASM Bibliography.* Bulletin of EATCS, 64: 105-127. Fevereiro, 1998.

[6] E. Borger and D. Rosenzweig. *A Mathematical Definition of Full Prolog.* In Science of Computer Programming, volume 24, pp 249-286, 1994.

[7] T. H. Cormem, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw Hill, Boston, 1990.

[8] J. W. Davidson and C. W. Fraser. *Code selection through object code optimization.* TOPLAS 6:4, 505-526, 1984.

[9] G. Del Castillo. *The ASM Workbench: na Open and Extensible Tool Enviroment for Abstract State Machine.* In Proceedings of the 28th Annual Conference of the German Society of Computer Science. Technical Report 1998.

[10] G. Del Castillo, I. Durdanovic, and U. Glasser. *The Evolving Algebra Interpreter Version 2.0*, 191-214, 1996.

[11] D. Diesen. Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages. Ph.D. Thesis,Dept. of Informatics, University of Oslo, Oslo, March 1995.

[12] J.Ferrante, Karl J. Ottenstein, and Joe D. Warren. *The program dependence graph and its use in optimization.* ACM Transactions on Programming Languages and Systems, 9(3):319-349, July 1987.

[13] Y. Gurevich. *Evolving algebras: An Attempt to discover semantics.* Bulletin of the European Association or Theoretical Computer Science, 43:264-284, 1991

[14] Y. Gurevich. *Evolving algebras 1993: Lipari guide.* In E. Borger, editor, Specification and Validation Methos, pages 9-36. Oxford University Press, 1995

[15] Y. Gurevich and J. Huggings. *The Semantics of the C Programming Language.* In E. Borger, H. Kleine , G. Jager, S. Martini, and M. M. Richter, editors, Computer Science Logic, volume 702 of LNCS, pp 274-309, Springer, 1993.

[16] J.Huggings & Raghu Mani. *The Evolving Algebra Interpreter Version 2.0*, Michigan, 1995.

[17] M. P. Jones. *Gofer – Functional Programming environment.* 1994.

[18] A. M. Kappel. *Implementation of algebras with an application to Prolog.* Universitat Dortmund, Dortmund, 1994

[19] P.W. Kutter and A. Pierantonio. *Montages specifications of realistic programming languages.* In Journal of universal computer science, Vol. 3, No 5 (1997), pp 416-442, Springer.

[20] P. W. Kutter and A. Pierantonio, *The Formal Specification of Oberon.* Journal of universal computer science, Vol. 3, No 5 (1997), pp 443-503, Springer.

[21] A. Poetzsch. *Developing Efficient Interpreters Based on Formal Language Specifications.* In P.Fitzson. Ed:Compiler Construction,1994, Springer LNCS 786.

[22] J. Schmid. *Introduction do AsmGofeer*. Siemens Corporate Technology. Munich, Março, 2001.

[23] J. Schmid. *Executing ASM specifications with AsmGofeer.* 1999.

[24] R. F. Stärk, J. Schmid and E. Borger. *Java and the Java Virtual Machine: Definition, Verification and Validation.* Springer-Verlag, 2001.

[25] F. Tirelo *Uma Ferramenta para Execução de um Sistema Dinâmico Discreto Baseado em Álgebras Evolutivas*; dissertação apresentada como requisito parcial para obtenção do grau de Mestre em Ciência da Computação. UFMG. 2000.

[26] F. Tirelo and R. S. Bigonha. *Técnicas de Otimização de Código Baseado em Máquinas de Estados Abstratas.* In Anais do IV Simpósio Brasileiro de Linguagens de Programação, Recife, PE, 2000.

[27] J. M. W. Visser. *Evolving algebras*. Delft University of Technology, Delt, 1996

[28] C. Wallace, *The Semantics of the Java Programming Language: Preliminary Version.* Univ. of Michigan EECS Department Technical Report CSE-TR-355-97

[29] *AsmL: The Abstract State Machine Language*, Foundations of Software Engineering – Microsoft Research © Microsoft Corporation, 2002.

[30] *Introducing AsmL: A Tutorial for the Abstract state Machine Language*, Foundations of Software Engineering – Microsoft Research © Microsoft Corporation, *may*, 2002.

[31] Fumiaki Ohata, Akira Nishimatsu, Katsuro Inoue; *Analyzing Dependence Locality for Efficient Construction of Program Dependence Graph.* Osaka Univ., Graduate School of Engineering Science. Japan. Dec. 2000.

[32] F. Tirelo, M. Maia, V. Dioro, R. S. Bigonha. *Máquinas de Estados Abstratas.* Tutorial do 3° SBLP, Porto Alegre, RS, 1999.