Tecnologia Adaptativa Aplicada à Otimização de Código em Compiladores

Júlio C. Luz
Universidade de São Paulo, Depto. de Eng. de Computação e Sistemas Digitais
São Paulo, SP, Brasil, 05508-900
julio.luz@poli.usp.br

and

João J. Neto
Universidade de São Paulo, Depto. de Eng. de Computação e Sistemas Digitais
São Paulo, SP, Brasil, 05508-900
joao.jose@poli.usp.br

Abstract

The programming memory space of embedded microcontrolled systems is usually limited. Although, compilers nowadays apply optimizing transformations to the embedded software, the lack of memory space can become a critical problem to the designer with the introduction of new features and corrections in the original software. In contrast, workstations hosting development systems for embedded applications are faster and have much more memory. Given this scenario, we have developed a peephole optimizer exploring an adaptive technique that requires more memory and execution time, but is capable to achieve a better compression ratio of the object code than a conventional peephole optimizer. The introduction of an adaptive action enables the algorithm to self-modify its behavior in response to a specific input condition and to search the sequence of optimization rules that best optimizes the object code among the many possible sequences resulted from the superposition of two or more equally applicable optimization rules.

Keywords: compilers, peephole optimization, code size reduction, adaptive technology, self-modifying machines.

Resumo

O espaço de memória de programação de sistemas microcontrolados embutidos é normalmente limitado. Embora os compiladores atuais apliquem transformações otimizantes ao software embutido, a falta de espaço de memória pode se tornar um problema crítico para o projetista com a introdução de novas facilidades e correções no software original. Por outro lado, as estações de trabalho hospedando os sistemas de desenvolvimento para aplicações embutidas são mais rápidas e dispõem de mais memória. Diante deste panorama, desenvolvemos um otimizador *peephole* explorando uma técnica adaptativa que requer mais memória e tempo de execução, mas é capaz de obter uma melhor taxa de compressão do código objeto do que um otimizador *peephole* convencional. A introdução de uma ação adaptativa permite que o algoritmo auto modifique o seu comportamento em resposta a uma condição de entrada específica e procure a seqüência de regras de otimização que melhor otimiza o código objeto entre as muitas seqüências possíveis resultantes da superposição de duas ou mais regras de otimização igualmente aplicáveis.

Palavras chaves: compiladores, otimização *peephole*, redução do tamanho do código, tecnologia adaptativa, máquinas auto modificáveis.

Introdução

O estudo de trabalhos recentes sobre redução de código sugerem a existência de duas grandes frentes de pesquisa nesta área do conhecimento: uma delas explora a aplicação de algoritmos de compressão de código e a outra estuda a utilização mais eficiente das técnicas de redução de código clássicas.

Trabalhos na primeira categoria produzem código comprimido que pode ser descomprimido antes da execução [9, 10, 11], ou, alternativamente, pode ser executado sem descompressão [2, 13], ou, ainda, pode ser interpretado sem descompressão [27]. Contudo, estas técnicas requerem modificações no ambiente de execução, ou *run-time system*, para descomprimir o código, ou no hardware do processador, para a execução direta do código comprimido [8], acarretando assim o encarecimento do produto final.

Trabalhos na segunda categoria produzem código diretamente executável pelo processador, porém requerem compiladores otimizadores que utilizem mais eficientemente as técnicas clássicas de redução de código. Entre tais técnicas, a abstração de procedimento, ou *procedural abstraction*, é uma das mais estudadas [8, 19, 26, 28]. Esta técnica procura seqüências repetidas de instruções no código objeto e as substitui por chamadas de um único procedimento. Pode ser aplicada diretamente ao código objeto [8, 19] ou ao código intermediário [26, 28]. A obtenção de taxas de redução de código cada vez mais expressivas se deve ao contínuo aprimoramento da técnica e/ou do uso combinado da abstração de procedimento com técnicas tradicionais de redução de código. Um outro exemplo de utilização mais eficiente das técnicas de redução de código clássicas é dado por um algoritmo genético que coordena a aplicação das transformações otimizantes de um compilador otimizador e utiliza o tamanho do código produzido como realimentação na busca de uma seqüência de transformações otimizantes que melhor reduza o espaço ocupado pelo código [3]. Van de Wiel [32] descreve uma extensa bibliografia de trabalhos dedicados ao tema da redução de código.

O presente trabalho se inscreve na segunda categoria, porém sugere um tratamento mais sistemático para a obtenção de algoritmos de otimização em função do emprego da tecnologia adaptativa, e escolheu a otimização *peephole* porque: (a) é uma técnica bem conhecida e, portanto, adequada para exemplificar o uso de técnicas adaptativas na obtenção de algoritmos poderosos a partir de algoritmos convencionais; (b) pode ser empregada em qualquer etapa da síntese do código objeto, ou seja, na otimização do código intermediário ou do código objeto, [3, 25] e (c) pode acelerar o processo de compilação como um todo em virtude da redução do número de instruções a serem processadas nos passos subseqüentes [8].

O restante do trabalho é organizado como segue. A seção 1 revisa a otimização *peephole* e algumas técnicas usadas na obtenção da melhor seqüência de regras de otimização. A seção 2 conceitua a tecnologia adaptativa e o paradigma de projetos de algoritmos adaptativos através de um exemplo simples de projeto de autômato adaptativo a partir de um autômato finito determinístico. A seção 3 descreve um otimizador *peephole* adaptativo que procura a seqüência de regras de otimização que melhor otimiza o código objeto e a seção 4 apresenta uma implementação preliminar do algoritmo esboçado na seção anterior.

1 Otimização *Peephole*

1.1 Princípio Básico e Regras de Otimização

O resultado da tradução de uma forma de representação intermediária efetuada pelo passo de geração de código de um compilador é o código objeto. Devido à justaposição direta de instruções e blocos de instruções traduzidos da forma de representação intermediária para o código objeto, o passo de geração de código freqüentemente cria seqüências de instruções ineficientes. A elaboração de um passo de geração de código que analise todas as combinações possíveis de justaposições de

blocos de instruções se torna difícil devido à existência de infinitas sentenças válidas no texto de entrada. É mais fácil manter a geração de código a cargo de um programa simples e aprimorar o código objeto através de um novo programa, o otimizador *peephole* [4, 5].

A otimização *peephole* tem uma longa história na literatura, pois a publicação do primeiro trabalho foi feita há quase quarenta anos atrás [23]. O otimizador move uma pequena fresta, ou *peephole*, sobre o código objeto [1, 23, 33], ou sobre o código intermediário [31], e compara as seqüências de instruções lidas através da fresta com as seqüências de instruções que podem ser eliminadas ou substituídas, as regras de otimização. Uma regra de otimização se divide em duas partes: uma seqüência de busca e uma seqüência de substituição. A regra de otimização seguinte:

```
jeq %00
jmp %01
%00:
=
jne %01
%00:
```

substitui uma seqüência de salto em igualdade e salto incondicional por uma seqüência de salto em desigualdade. O argumento %dd, onde d é um dígito, substitui a cadeia de caracteres encontrada até a ocorrência de um novo átomo, ou *token*, da regra de otimização. Esta notação é bastante empregada na codificação das regras de otimização [5, 6, 7, 21, 24, 31].

1.2 Otimização *Peephole* em um Único Passo

A eliminação ou a substituição de uma seqüência de instruções possibilita, em geral, a realização de uma nova otimização. Por essa razão, os primeiros otimizadores *peephole* efetuavam vários passos de otimização sobre o código gerado a fim de obter o código mais otimizado possível [1, 23, 31, 33].

O acréscimo do otimizador diminuía sensivelmente o desempenho do compilador, pois o otimizador precisava efetuar vários passos de otimização sobre o código gerado. Contudo, o desenvolvimento de novos algoritmos de busca e substituição minimizou a queda de desempenho do compilador otimizador por meio da eliminação, ou redução, dos passos de otimização sobre todo o código gerado [20, 21, 24, 30].

O algoritmo desenvolvido por Lamb [21] reduziu o número de passos a apenas um e se tornou um algoritmo típico de otimização *peephole*. A figura 1 ilustra o funcionamento do algoritmo sobre uma seqüência de instruções A;A;B;C com a aplicação da regra de otimização A;B→B.

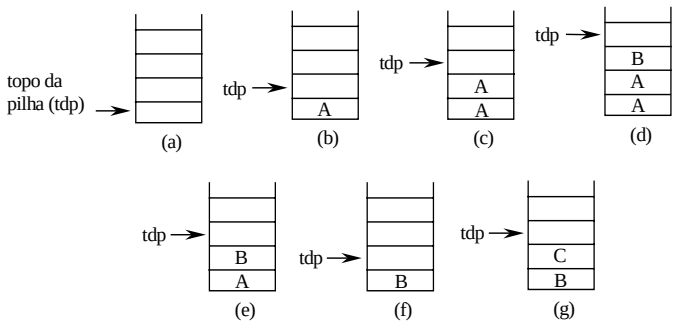

Figura 1 - Funcionamento do otimizador sobre a seqüência de instruções A;A;B;C: (a) no início a pilha se encontra vazia; (b) pilha após a inclusão da primeira instrução; (c) pilha após a inclusão da segunda instrução; (d) pilha após a inclusão da terceira instrução; (e) pilha após a aplicação da regra de otimização A;B→B; (f) pilha após nova aplicação da regra de otimização A;B→B e (g) pilha após a inclusão da última instrução.

O otimizador emprega uma pilha para o armazenamento das instruções do código objeto e efetua a

seguinte seqüência de ações após a leitura de uma nova instrução: (a) insere a instrução no topo da pilha; (b) verifica se existe uma seqüência de instruções a ser eliminada ou substituída do topo da pilha para baixo; (c) efetua a otimização, caso exista, e volta para o passo (b); do contrário (d) repete o mesmo procedimento enquanto houver instruções a serem processadas. Quando as instruções terminarem, o otimizador desempilha o conteúdo da pilha, obtendo o código otimizado.

O primeiro aspecto que se destaca neste otimizador é a realização de vários passos de otimização sobre um trecho localizado do código, em lugar de vários passos de otimização sobre todo o código. Isto torna o otimizador mais rápido que seu antecessor direto [33]. Outro aspecto que se destaca é a exploração de apenas uma das possibilidades de eliminação ou substituição quando ocorre a superposição de duas ou mais regras de otimização. O algoritmo pára a análise das instruções ao encontrar uma seqüência que possa ser eliminada ou substituída, e efetua uma otimização. Desta forma, o otimizador ignora outras possibilidades de otimização caso existam.

Para reduzir o número de passos de otimização *peephole* necessários McKenzie [24] empregou um autômato finito determinístico, AFD, sobre uma fresta de tamanho fixo que se desloca de um certo número de instruções após a análise do AFD. Partindo do seu estado inicial, o AFD percorre as instruções da fresta a partir de seu início até alcançar um estado final, que equivale a uma seqüência de instruções a ser eliminada ou substituída, ou até um estado não-final. Caso o AFD chegue a um estado final, efetua-se a eliminação ou substituição da instrução, ou seqüência de instruções, correspondente a um deslocamento da fresta, para que o AFD execute um novo ciclo de busca. Caso o AFD pare em um estado não-final, efetua-se apenas um deslocamento da fresta. Em ambos os casos, procura-se manter no interior da fresta um certo número de instruções já analisadas, que aumente a possibilidade da ocorrência de uma otimização no próximo ciclo de busca do AFD e reduza o número de passos de otimização sobre todo o código. O valor de deslocamento da fresta varia em função dos estados de parada do AFD. A figura 2 ilustra esse modelo do otimizador.

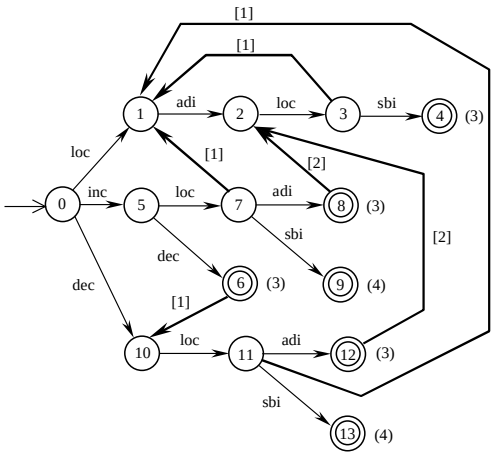

Figura 2 - AFD construído a partir de cinco regras de otimização. Valores entre parênteses (n) indicam o deslocamento da fresta em caso de substituição. Valores entre colchetes [n] indicam o deslocamento da fresta em caso de falha

Da mesma forma que no algoritmo desenvolvido por Lamb [21], a realização de otimizações adicionais sobre trechos localizados do código reduz a necessidade de um passo adicional de otimização sobre todo o código e resulta em um otimizador mais rápido que o seu antecessor direto [31]. Por outro lado, este algoritmo não dispensa a realização de passos adicionais de otimização sobre todo o código, mas explora melhor as possibilidades de eliminação ou substituição por escolher a seqüência de instruções mais longa quando ocorre uma superposição de regras de otimização igualmente aplicáveis. Seguindo esta mesma filosofia, desenvolveu-se um otimizador *peephole* usando busca e substituição de padrões em árvores, o que elimina a necessidade de passos adicionais de otimização sobre todo o código, mas com o acréscimo de fases adicionais de tradução do código para a forma de árvore e vice-versa após a otimização [20]. Com o intuito de aumentar

ainda mais a eficiência do otimizador desenvolveu-se recentemente um protótipo de otimizador *peephole* usando busca e substituição de expressões regulares ao invés da busca e substituição de cadeia de caracteres, porém o protótipo efetua apenas um passo de otimização por tratar apenas uma regra de otimização [30].

1.3 Aplicação Simultânea de Duas ou Mais Regras

Embora os otimizadores *peephole* tenham evoluído significativamente, a superposição de duas ou mais regras de otimização igualmente aplicáveis permaneceu pouco estudada. Um trabalho que visava estudar o número de regras de otimização necessárias para um compilador emitir código de qualidade comparável ao de compiladores mais sofisticados descobriu que a superposição de algumas regras de otimização é capaz de produzir um código objeto de baixa qualidade [7]. Sejam as seguintes regras de otimização:

```
mov $0, %00
=
clr %00
```

e

```
mov %00, r%01
mov r%01, %02
=
mov %00, %02
```

Se o compilador emite a seguinte seqüência de instruções:

```
mov $0, r3
mov r3,i.(fp)
```

O otimizador substitui a instrução de armazenamento de zero no operando de destino por uma instrução de reposicionamento do operando de destino, o que impossibilita a aplicação da regra de otimização mais vantajosa no lugar da regra aplicada. O acréscimo de novas regras de otimização que manipulem cada caso especial contorna o problema, conforme ilustra a seguinte regra de otimização que soluciona o impasse anterior:

```
clr r%00
mov r%00, %01
=
clr %01
```

Contudo, Davidson e Whalley [7] descobriram uma solução mais elegante: atrasa-se a aplicação de algumas regras de otimização até se dispor de mais informações de contexto que permitam a aplicação de regras mais específicas. Por exemplo, reformula-se a regra de otimização que substitui uma instrução de armazenamento de zero no operando de destino por uma instrução de reposicionamento do operando de destino para:

```
mov $0, %00
%01
=
clr %00
%01
```

e altera-se a ordem de análise das regras, de modo que a regra reformulada seja analisada após a regra que otimiza instruções de armazenamento de valores dos operandos de origem nos operandos de destino. A regra reformulada não pode ser aplicada até que a segunda instrução seja examinada,

o que permite a aplicação da regra que otimiza instruções de armazenamento de valores dos operandos de origem nos operandos de destino. Desta forma, quando ocorre uma superposição de regras de otimização igualmente aplicáveis, este algoritmo dá preferência à substituição das seqüências mais longas de instruções.

Um outro estudo tratando especificamente esse problema [14] sugeriu o acréscimo de extensões ao gerador de código do compilador: uma denominada de dominó simples, ou *simple domino*, e outra, construção dinâmica de código, ou *dynamic code building*, que, juntas, eliminariam totalmente a necessidade de um estágio de otimização *peephole*. A técnica do dominó simples visa aprimorar a justaposição de instruções e blocos de instruções traduzidos do código intermediário para o código objeto e a técnica de construção dinâmica de código visa resolver o problema da superposição de uma ou mais regras de otimização. Como este trabalho enfoca especificamente o problema da superposição das regras de otimização, vamos analisar com mais detalhes a técnica de construção dinâmica de código e remeter o leitor interessado na técnica do dominó simples à publicação que relata o estudo realizado [14].

Na técnica de construção dinâmica de código considera-se que cada instrução é uma regra de otimização formada por uma única instrução de busca e uma única instrução de substituição, e atribui-se a cada instrução um peso, que representa o número de ciclos de máquina necessários para a sua execução. As regras de otimização com duas ou mais instruções recebem a soma dos pesos das instruções que compõem a seqüência de substituição. Considere-se, por exemplo, três instruções A, B e C e as regras de otimizações A;B e B;C:

$A \rightarrow (A,5)$
$B \rightarrow (B,10)$
$C \rightarrow (C,15)$
$A;B \rightarrow (X,10)$
$B;C \rightarrow (Y,10)$

às quais se associam pares ordenados cujos elementos representam respectivamente a seqüência de substituição associada à seqüência de busca e o número de ciclos necessários para a sua execução. Dadas as definições acima e na ausência de otimização, o custo de execução da seqüência de instruções A;B;C é:

$A;B;C \rightarrow (A,5);(B,10);(C,15) \rightarrow (A;B;C,30)$

Quando um otimizador *peephole* convencional encontra a mesma seqüência de instruções, o custo de execução da seqüência de instruções A;B;C diminui com a substituição da seqüência A;B por X e se obtém:

$A;B;C \rightarrow (X,10);(C,15) \rightarrow (X;C,25)$

O que resulta numa redução sub-ótima de 5 ciclos de acordo com as definições dadas. O custo de execução mínimo e, por conseqüência, a redução ótima, são atingidos ao se substituir B;C na seqüência de instruções A;B;C:

$A;B;C \rightarrow (A,5);(Y,10) \rightarrow (A;Y,15)$

A simples substituição das seqüências de instruções mais longas não é suficiente para garantir a geração da seqüência de instruções ótima. A fim de atingir esse objetivo, emprega-se um algoritmo de programação dinâmica que minimize o custo de execução. O algoritmo usa uma pilha para armazenar a tripla: seqüência de instruções, custo total e lista de instruções. A pilha se encontra vazia pilha no início da execução do algoritmo:

Seq. de Instruções	Custo	Lista de Instruções	Comentários
-	0	-	Pilha vazia

O algoritmo busca a próxima instrução, verifica as possibilidades de substituições possíveis em função dos valores armazenados na pilha e empilha o resultado para uso na próxima iteração do algoritmo. Assim, ao buscar A, a única possibilidade de substituição é A → (A,5) e a pilha fica:

Seq. de Instruções	Custo	Lista de Instruções	Comentários
A	5	A	Uma única possibilidade
-	0	-	Pilha vazia

Ao buscar B, existem duas possibilidades de substituição: B → (B,10) e A;B → (X,10). Como o custo de A;B é 5+10=15 e o custo de X é 10, empilha-se o menor custo total, obtendo-se a nova configuração da pilha:

Seq. de Instruções	Custo	Lista de Instruções	Comentários
B	10	X	Duas possibilidades consideradas
A	5	A	Uma única possibilidade
-	0	-	Pilha vazia

Ao buscar C, existem duas possibilidades de substituição: C → (C,15) e B;C → (Y,10). Como o custo de X;C é 10+15=25 e o custo de A;Y é 5+10=15, empilha-se o menor custo total, obtendo-se a nova configuração da pilha:

Seq. de Instruções	Custo	Lista de Instruções	Comentários
C	15	A;Y	Seqüência Ótima
B	10	X	Duas possibilidades consideradas
A	5	A	Uma única possibilidade
-	0	-	Pilha vazia

Esta é a forma com que o algoritmo obtém dinamicamente a seqüência de instruções ótima, uma vez que a seqüência ótima se encontra no topo da pilha. Para seqüências mais longas de instruções o algoritmo repetiria os passos de iteração descritos anteriormente e pesquisaria mais profundamente a pilha em busca das seqüências mais longas de instruções.

Dos algoritmos clássicos este é o que melhor resolve o problema da superposição de regras de otimizações igualmente aplicáveis. Embora o algoritmo procure minimizar o tempo de execução do código objeto, o mesmo pode ser facilmente adaptado para minimizar o espaço ocupado pelo código objeto. Na seção 4 vemos como um otimizador *peephole* adaptativo resolve esse mesmo problema por meio da busca em profundidade de seqüências de regras de otimização.

2 Tecnologia Adaptativa

A tecnologia adaptativa trata técnicas e dispositivos que modificam espontaneamente o seu comportamento em resposta a uma condição especial de entrada [17]. Os autômatos adaptativos são uma classe particular de dispositivos adaptativos resultantes da conjunção de autômatos de pilha estruturados e *ações adaptativas*. Uma ação adaptativa é um procedimento inerente ao formalismo que modifica o comportamento do autômato de pilha subjacente. As ações adaptativas estão associadas a determinadas transições do autômato. As transições assim associadas passam a se chamar *transições adaptativas*. Quando se reúnem as condições necessárias para a execução de uma transição adaptativa, o autômato executa a ação adaptativa associada a essa transição.

José Neto [15, 16] descreve em detalhes a teoria formal dos autômatos adaptativos e algumas das aplicações de autômatos adaptativos à teoria de linguagens. José Neto [17] descreve mais exemplos de aplicações de autômatos adaptativos que são conseqüência da propriedade de os autômatos adaptativos serem tão potentes quanto as máquinas de Turing. Pela Tese de Church [22], há uma identificação possível entre autômatos e algoritmos. Assim sendo, pode-se dizer que um algoritmo é

adaptativo quando modificar espontaneamente o seu comportamento em resposta a uma condição especial de entrada. Como conseqüência, pode-se formular algoritmos adaptativos pela introdução de ações adaptativas em algoritmos convencionais, da mesma forma como se formulam autômatos adaptativos pelo acoplamento de ações adaptativas a autômatos convencionais [18]. Este é o paradigma empregado no projeto do algoritmo de otimização *peephole* adaptativo adotado neste trabalho.

O exemplo abaixo mostra o projeto de um autômato adaptativo que aceita a linguagem $a^n b^n c^n$, com $n \geq 0$, a partir de um autômato finito determinístico, AFD, que aceita a linguagem abc. Da teoria das linguagens sabe-se que o reconhecimento da linguagem $a^n b^n c^n$ não pode ser feita por um autômato finito ou de pilha em função da mesma não ser regular e nem livre de contexto [22]. Seja, portanto, o AFD representado na figura 3, que aceita a linguagem abc:

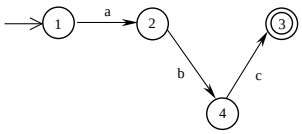

Figura 3 - AFD que aceita a cadeia abc.

Caso se insira uma transição com consumo de átomo 'a' no estado 2 do AFD da figura 3 e se acople à transição criada uma ação adaptativa que: (1) gere dois novos estados, por exemplo, p* e q*; (2) remova a transição com consumo de átomo 'b' que parte do estado 2; (3) insira uma nova transição com consumo de átomo 'b' do estado 2 para o estado p*; (4) insira uma nova transição com consumo de átomo 'b' do estado p* para o estado no qual terminava a transição removida em (1); (5) remova a transição com consumo de átomo 'c' que termina no estado 3; (6) insira uma nova transição com consumo de átomo 'c' para o estado q* a partir do estado em que se originava a transição removida em (5); (7) insira uma nova transição com consumo de átomo 'c' do estado q* para o estado 3; e (8) reconfigure a ação adaptativa para um eventual consumo de um novo átomo 'a', então o autômato resultante, cuja configuração inicial é representada na figura 4, é capaz de aceitar a linguagem $a^n b^n c^n$, com $n \geq 0$, pela extensão gradativa da aceitação da cadeia 'bc' do autômato da figura 3 para 'bbcc', 'bbbccc', assim sucessivamente, até $b^n c^n$ a cada consumo adicional do átomo 'a'.

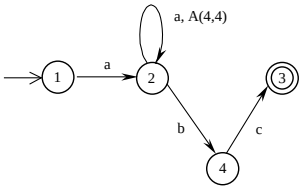

Figura 4 - Configuração inicial do autômato adaptativo que aceita a cadeia $a^n b^n c^n$. A(4,4) é a ação adaptativa com dois parâmetros de entrada que indicam, respectivamente, onde termina a transição com consumo de átomo 'b' que parte do estado 2 e de onde parte a transição com consumo de átomo 'c' que termina no estado 3.

A ação adaptativa elementar (1) gera novos estados para o autômato. As três próximas ações, (2), (3) e (4), efetuam a inclusão de um novo estado entre o estado 2 e o estado para o qual a transição com consumo de 'b' se dirigia. As próximas três ações, (5), (6) e (7), efetuam a mesma tarefa para o estado do qual a transição com consumo de 'c' saía em direção ao estado 3 e a última ação, (8), efetua a troca da ação adaptativa acoplada à transição com consumo de 'a' no estado 2 por uma nova transição adaptativa que prepara o autômato para um eventual consumo de outro átomo 'a'. Assim, dada a cadeia de entrada 'aaabbbccc' e o autômato adaptativo representado na figura 4, o consumo do primeiro átomo 'a' leva o autômato do estado 1 para o estado 2, o consumo do segundo átomo 'a' mantém o autômato no estado 2 e transforma a configuração inicial do autômato na seguinte configuração representada na figura 5a. O consumo do terceiro átomo 'a' também mantém o autômato no estado 2 e transforma a configuração da figura 5a na configuração representada na figura 5b.

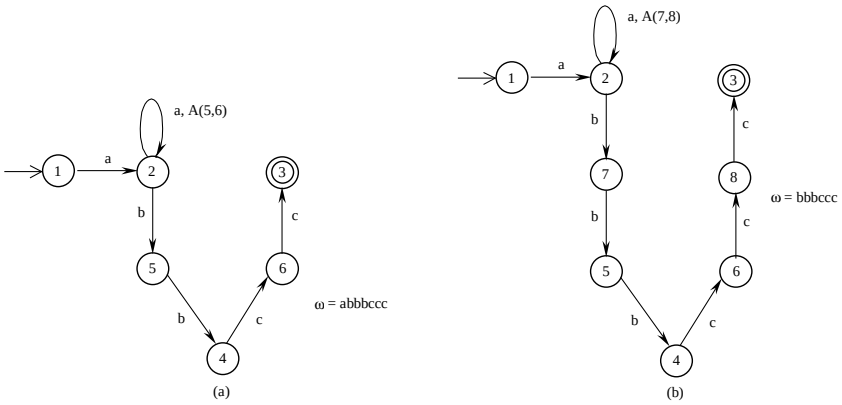

Figura 5 - (a) configuração do autômato após o consumo do segundo 'a'; (b) configuração do autômato após o consumo do terceiro 'a'. ω representa o restante da cadeia de entrada a ser consumida pelo autômato adaptativo.

Como não restam mais átomos 'a' na cadeia de entrada, a configuração do autômato da figura 5b se mantém e o autômato consome o restante da cadeia de entrada transitando do estado $2 \rightarrow 7 \rightarrow 5 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 3$, terminando por aceitar a cadeia de entrada. Este exemplo mostra como problemas difíceis podem ser resolvidos de maneira simples e eficiente com o uso de autômatos adaptativos. José Neto [17] descreve outros problemas de difícil solução resolvidos por autômatos adaptativos.

3 Algoritmo Adaptativo de Otimização *Peephole*

Escolheu-se como algoritmo de otimização *peephole* convencional, o algoritmo desenvolvido por Lamb [21]; como condição especial de entrada que condiciona a execução da ação adaptativa, a detecção da possibilidade de aplicação simultânea de duas ou mais regras de otimização, e como ação adaptativa, a aplicação concorrente de todas as possíveis regras de otimização.

Para simular a concorrência, o algoritmo utilizará uma pilha para armazenar o contexto de cada execução em andamento, ou *thread*. A organização e a subseqüente execução dos contextos armazenados na pilha será análoga ao que ocorre em uma busca em profundidade [29]. A criação de tantos *threads* quantas forem as regras de otimização que se apliquem num dado instante propicia a posterior aplicação de cada uma das possíveis regras de otimização a partir da instrução corrente até o final da lista de instruções do programa.

O algoritmo processa todos os *threads* presentes na pilha de controle. Para cada *thread*, o algoritmo processa todas as instruções a partir do apontador de instruções corrente até o final da lista de instruções. No início, existe um único *thread* presente na pilha de controle, sendo o seu apontador de instruções voltado para a primeira instrução do programa.

À medida que as instruções são incluídas na lista de instruções do *thread* atual, determina-se o número de regras de otimização que se aplicam. Caso se aplique uma única regra, o algoritmo efetua a otimização e determina se existem outras possibilidades de otimização. Caso não existam, o algoritmo avança o apontador de instruções e efetua a leitura da próxima instrução. Caso existam, o algoritmo determina o número de regras que se aplicam. Caso se aplique uma única regra, o algoritmo repete o procedimento descrito anteriormente. Caso o algoritmo detecte a possibilidade de aplicação simultânea de duas ou mais regras de otimização, o algoritmo cria tantos *threads* quantas forem as regras de otimização que se apliquem, insere os novos *threads* no topo da pilha de controle e retoma o processamento do *thread* atual com a realização da otimização atribuída ao mesmo e a determinação de outras possibilidades de otimização, e assim por diante, até que a lista de instruções do programa se esgote.

Quando isso ocorre, o algoritmo imprime a lista de instruções armazenada pelo *thread* atual,

desempilha o próximo *thread* do topo da pilha de controle e repete o procedimento descrito no parágrafo anterior para todas as instruções do programa, iniciando-se pela instrução na qual o *thread* foi criado.

4 Resultados Preliminares

O algoritmo proposto na seção anterior foi implementado em uma versão do algoritmo de Lamb desenvolvida por Fraser em linguagem C [12]. Modificou-se, basicamente, o enlace principal do otimizador *peephole* original a fim de possibilitar a execução de vários *threads* pelo otimizador resultante. A figura 6 mostra as regras de otimização empregadas por este otimizador *peephole* adaptativo. As regras de otimização das figuras 6a, 6c, e 6f foram introduzidas na seção 1; a regra da figura 6b elimina uma instrução que soma zero ao operando de destino; a regra da figura 6d substitui duas instruções que efetuam armazenamento indireto por uma única instrução de armazenamento indireto, a instrução de soma de constante a registrador é preservada, pois o valor do registrador pode ser necessário no processamento subseqüente; e a regra da figura 6e elimina o processamento desnecessário de uma instrução antes da realização de um salto. O número que se segue após a linha contendo o átomo '->' indica o número de instruções eliminadas com a aplicação da regra de otimização.

mov $0,%00
%01
=
clr %00
%01
->
0
	add $0,%00
	=
	->
	1
		jeq %00
		jmp %01
		%00:
		=
		jne %01
		%00:
		->
		1
			add $%01,r%00
			mov *r%00,%02
			=
			mov %01(r%00),%02
			add $%01,r%00
			->
			0
				%01 %02,r%03
				j%04
				=
				j%04
				->
				1
					mov %00,r%01
					mov r%01,%02
					=
					mov %00,%02
					->
					1

(a)	(b)	(c)	(d)	(e)	(f)

Figura 6 - Regras de otimização empregadas pelo otimizador *peephole* adaptativo.

Antes do início do processamento do código não-otimizado, o otimizador efetua a leitura de um arquivo que armazena as regras de otimização mostradas na figura 6. A figura 7a mostra um fragmento de código não-otimizado e as figuras 7b e 7c mostram três fragmentos de código produzidos pelo otimizador *peephole* adaptativo a partir do código não-otimizado da figura 7a.

1. mov $0,r2
2. mov r2,i.(fp)
3. add $0,r2
4. add $4,r2
5. mov *r2,r2
6. jeq L1
7. jmp L2
8. L1:
	// Otimizador Peephole Adaptativo
	// resultado final do otimizador:[1]
	clr r2
	mov r2,i.(fp)
	jne L2
	L1:
		// resultado final do otimizador:[2]
		clr i.(fp)
		jne L2
		L1:
		// resultado final do otimizador:[3]
		clr i.(fp)
		jne L2
		L1:
		// Fim.

(a)	(b)	(c)

Figura 7 - (a) fragmento de código não-otimizado; (b) fragmento sub-ótimo produzido pelo otimizador *peephole* adaptativo a partir do código não-otimizado; (c) fragmentos ótimos produzidos pelo otimizador *peephole* adaptativo a partir do código não-otimizado.

A primeira versão de código otimizado da figura 7b foi obtida pela aplicação sucessiva das seguintes regras: (1) regra 6a às instruções das linhas 1 e 2; (2) regra 6b à instrução da linha 3; (3) regra 6d às instruções das linhas 4 e 5; (4) regra 6e ao resultado de (3) e à instrução da linha 6; (5) regra 6e ao resultado de (3) e de (4); (6) regra 6c ao resultado de (5) e às instruções das linhas 7 e 8.

A segunda versão de código otimizado da figura 7c foi obtida pela aplicação sucessiva das seguintes regras: (1) regra 6f às instruções das linhas 1 e 2; (2) regra 6a ao resultado de (1) e à instrução da linha 3; (3) regra 6b ao resultado de (2); (4) regra 6d às instruções das linhas 4 e 5; (5)

regra 6e ao resultado de (4) e à instrução da linha 6; (6) regra 6e ao resultado de (4) e de (5); (7) regra 6c ao resultado de (6) e às instruções das linhas 7 e 8.

A terceira versão de código otimizado da figura 7c foi obtida pela aplicação sucessiva das seguintes regras: (1) regra 6f às instruções das linhas 1 e 2; (2) regra 6b à instrução da linha 3; (3) regra 6a ao resultado de (1) e à instrução da linha 4; (4) regra 6d ao resultado de 3 e à instrução da linha 5; (5) regra 6e ao resultado de (4) e à instrução da linha 6; (6) regra 6e ao resultado de (4) e de (5); (7) regra 6c ao resultado de (6) e às instruções das linhas 7 e 8.

A superposição de regras de otimização igualmente aplicáveis às linhas 1 e 2 do código não-otimizado, regras 6a e 6f, proporcionou a derivação de uma primeira e uma segunda versão de código otimizado e uma inesperada superposição de regras de otimização igualmente aplicáveis, regras 6a e 6b, proporcionou a derivação de uma terceira versão de código otimizado a partir da segunda versão. Caso as regras da figura 6 fossem empregadas por um otimizador *peephole* convencional, não se poderia garantir a obtenção de uma versão ótima de código otimizado, uma vez que a ordem de análise das regras de otimização influencia decisivamente o resultado final conforme descrito em 1.3.

5 Conclusão

Neste trabalho se apresentou um otimizador *peephole* adaptativo, utilizando mais memória e tempo de execução, capaz de procurar uma versão de código otimizado com a melhor taxa de compressão entre várias versões de código otimizado que resultam da superposição de regras de otimização igualmente aplicáveis.

Referências

[1] AHO, A.V.; SETHI, R.; ULLMAN, J.D. *Compilers: principles, techniques and tools*. Reading, MA: Addison-Wesley, 1986.

[2] COOPER, K. D.; MCINTOSH, N. Enhanced code compression for embedded RISC processors. *ACM SIGPLAN Notices*, v.34, n.5, p.139-149, May 1999.

[3] COOPER, K. D.; SCHIELKE, P. J.; SUBRAMANIAN D. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices*, New York, v.34, n.7, p.1-9, July 1999.

[4] DAVIDSON, J.W.; FRASER, C.W. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, New York, v.2, n.2, p.191-202, Apr.1980.

[5] DAVIDSON, J.W.; FRASER, C.W. Automatic generation of peephole optimizations. *ACM SIGPLAN Notices*, v.19, n.6, p.111-116, June 1984.

[6] DAVIDSON, J.W.; FRASER, C.W. Automatic inference and fast interpretation of peephole optimization rules. *Software-Practice and Experience*, New York, v.17, n.11, p.801-812, Nov.1987.

[7] DAVIDSON, J.W.; WHALLEY, D.B. Quick compilers using peephole optimization. *Software-Practice and Experience*, New York, v.19, n.1, p.79-97, Jan.1989.

[8] DEBRAY, S.K.; EVANS, W.; MUTH, R.; DE SUTTER, B. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, New York, v.22, n.2, p.378-415, Mar. 2000.

[9] DRINIC, M.; KIROVSKI, D.; VO, H. Code optimization for code compression. In: International Symposium on Code Generation and Optimization, 1st, San Franscisco, CA, USA, Mar. 2003. *Proceedings*. Los Alamitos, CA: IEEE Computer Society Press, 2003, p.315-324.

[10] ERNST, J.; EVANS, W.; FRASER, C.W.; LUCCO, S.; PROEBSTING, T.A. Code compression. *ACM SIGPLAN Notices*, v.32, n.5, p.358-365, May 1997.

[11] FRANZ, M.; KISTLER, T. Slim Binaries. *Communications of the ACM*, New York, v.40, n.12, p.87-94, Dec. 1997.

[12] FRASER, C.W.; HANSON, D.R. *A retargetable C compiler: design and implementation*. Menlo Park, CA: Addison-Wesley Publ. Co., 1995.

[13] FRASER, C.W.; MYERS, E.W.; WENDT, A.L. Analyzing and compressing assembly code. *ACM SIGPLAN Notices*, v.19, n.6, p.117-121, June 1984.

[14] GILL, A. A novel approach towards peephole optimisations. In: Annual Glasgow Workshop on Functional Programming, 4th., Portree, Isle of Skye, Scotland, Aug. 1991. *Functional Programming, Glasgow 1991*. Berlin: Springer Verlag, 1992, p.100-111.

[15] JOSÉ NETO, J. *Contribuições à metodologia de construção de compiladores*. São Paulo, 1993, 272p. Tese (Livre-Docência) Escola Politécnica, Universidade de São Paulo. (Em Português)

[16] JOSÉ NETO, J. Adaptive automata for context-dependent languages. *ACM SIGPLAN Notices*, New York, v.29, n.9, p.115-124, Sept. 1994.

[17] JOSÉ NETO, J. Solving complex problems efficiently with adaptive automata. In: International Conference on Implementation and Application of Automata - CIAA 2000, 5th., London, Ontario, Canada, July 2000. *Lectures Notes on Computer Science 2088*. Berlin: Springer-Verlag, 2001, p.340-342.

[18] JOSÉ NETO, J. Adaptive rule-driven devices - general formulation and case study. In: International Conference on Implementation and Application of Automata - CIAA 2001, 6th., Pretoria, South Africa, July 2001. *Lectures Notes on Computer Science 2494*. Berlin: Springer-Verlag, 2002, p.234-250.

[19] KIM, D.H.; LEE, H.J. Iterative procedural abstraction for code size reduction. In: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems - CASES'02, 3rd., Greenoble, France, Oct. 2002. *Proceedings*. New York, NY: ACM Press, 2002, p.277-279.

[20] KIM, J.; OH, S. EM-code optimization algorithm using tree pattern matching. In: 1997 International Conference on Information, Communications and Signal Processing - ICICS'97, Singapore, Sept. 1997. *Proceedings*. Singapore: IEEE, 1997, v.2, p.917-923.

[21] LAMB, D.A. Construction of a peephole optimizer. *Software-Practice and Experience*, New York, v.11, n.6, p.639-647, June 1981.

[22] LEWIS, H.R.; PAPADIMITRIOU, C.H. *Elements of the Theory of Computation*. Englewood Cliffs, NJ, Prentice-Hall, 1981.

[23] McKEEMAN, W.M. Peephole optimization. *Communications of the ACM*, New York, v.8, n.7, p.443-444, July 1965.

[24] McKENZIE, B.J. Fast peephole optimization techniques. *Software-Practice and Experience*, New York, v.19, n.12, p.1151-1162, Dec.1989.

[25] MORGAN, R. *Building an optimizing compiler*. Woburn, MA: Butterworth-Heinemann, 1998.

[26] NYSTRÖM, S.O.; RUNESON, J.; SJÖDIN, J. Optimizing code size through procedural abstration. In: ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems - LCTES'2000, Vancouver, BC, Canada, June 2000. *Lectures Notes on Computer Science 1985*. Berlin: Springer-Verlag, 2001, p.204-205.

[27] PROEBSTING, T.A. Optimizing an ANSI C iterpreter with superoperators. In: ACM SIGPLAN-SIGACT Symposium of Programming Languages, 22nd., San Franscisco, CA, USA, Jan. 1995. *Proceedings*. New York, NY: ACM Press, 1995, p.322-332.

[28] RUNESON, J. *Code compression through procedural abstraction before register allocation*. Uppsala, 2000, 21p. Thesis (Master's) Computer Science Dept., Uppsala University.

[29] RUSSELL, S.; NORVIG, P. *Artificial Intelligence: a modern approach*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1995.

[30] SPINELLIS, D. Declarative peephole optimization using string pattern matching. *ACM SIGPLAN Notices*, New York, v.34, n.2, p.47-51, Feb.1999.

[31] TANENBAUM, A.S.; VAN STAVEREN, H.; STEVENSON, J.W. Using peephole optimization on intermediate code. *ACM Transactions on Programming Languages and Systems*, New York, v.4, n.1, p.21-36, Jan.1982.

[32] VAN DE WIEL, R. *The code compaction bibliography*. http://www.extra.research.philips.com/ccb.

[33] WULF, W.; JOHNSSON, R.K.; WEINSTOCK, C.B.; HOBBS, S.O.; GESCHKE, C.M. *The design of an optimizing compiler*. New York, NY: American Elsevier Publ. Co., Inc., 1975.