Compressão de código executável

Bruno Ribeiro*
Instituto de Computação
Universidade Estadual de Campinas
brunex.geek@gmail.com

ABSTRACT

Este artigo tem como objetivo fornecer uma revisão bibliográfica superficial de alguns trabalhos relacionados a compressão de código executável. Primeiramente é realizada uma contextualização à compressão de dados e de código executável, onde são apresentadas alguns desafios e limitações enfrentados na área. A seguir são apresentadas algumas técnicas de compressão de código, descrevendo suas características e forma de operação.

1. INTRODUÇÃO

Os sistemas computacionais embarcados tem se popularizado rapidamente nos últimos anos. O avanço da tecnologia dos microcontroladores e das ferramentas de *software* permitiu o uso de sistemas embarcados em aplicações que a computação seria inviável de outra forma. Entretanto, o custo de tais sistemas está diretamente relacionado a fatores como: o desempenho requerido, as limitações de tamanho, o consumo de energia, etc.

No sentido inverso, o *software* utilizado em tais sistemas tem se tornado cada vez maior. Cada vez mais são utilizadas linguagens de alto nível, sobretudo linguagens orientadas a objetos e as interpretadas, que apesar de fornecerem uma base mais completa de componentes reusáveis, facilitando o processo de desenvolvimento, representam um aumento significativo na quantidade de instruções que compõem um programa. Aliado a isso, os *softwares* estão se tornando cada vez mais complexos, grande parte em virtude do aumento da complexidade das soluções nas quais os sistemas embarcados estão sendo introduzidos. Esse aumento no tamanho dos programas implica, entre outros fatores, na necessidade de maior quantidade de memória disponível.

É possível identificar duas abordagens para ser obter redução na quantidade de memória necessária para manter programas. A primeira consiste em utilizar processadores que disponham de um conjunto de instruções cuja representação ocupe o mínimo possível de espaço. Esse tipo de abordagem é discutida na seção 4.1 e envolve o uso de processadores como Thumb e MIPS16.

Outra abordagem é o uso de mecanismos para compressão de código, no qual o código comprimido possa ser descomprimido em tempo de execução. Esse tipo de abordagem é discutida na seção 4.3 e pode envolver mecanismos de *software* e *hardware*.

Outra abordagem para redução do tamanho dos programas, mas que não se aplica ao problema de economia de memória principal, é o uso de técnicas voltadas para a compressão de código onde a descompressão ocorre antes do programa ser carregado em memória. A seção 4.4 cobre algumas técnicas com esse objetivo.

Este artigo está organizado em quatro seções principais. Na seção 2 é apresentada uma introdução à compressão de dados. Na seção 3 é tratada a compressão especificamente voltada para a compressão de código executável. A seguir é realizada uma revisão bibliográfica superficial sobre as técnicas de compressão de código na seção 3. Por fim, são feitas algumas considerações finais na seção 4.

2. COMPRESSÃO DE DADOS

Formalmente, compressão de dados pode ser descrita como o processo de converter um fluxo de dados de origem em um fluxo de dados de destino que apresente menor tamanho [16]. Apesar de existir uma grande variedade de métodos de compressão, todos eles partem do príncipio de remover redundância nos dados de origem. A redundância pode se apresentar de várias formas: na repetição de um símbolo ou um conjunto de símbolos, devido ao formato escolhido para a representação dos dados (como *alignment* e *padding*), etc [4].

Qualquer conjunto de dados não aleatório possui alguma estrutura lógica e tal estrutura pode ser explorada para alcançar uma representação que necessite de menos espaço. Essa nova forma de representação não é necessariamente manipulável como a estrutura dos dados originais, mas pode ser encarada como uma forma de representação intermediária que permita trafegar ou armazenar as mesmas informações de forma mais rápida e que ocupe menos espaço.

A compressão de dados, entretanto, possui algumas limitações. Primeiramente, grande parte dos arquivos não pode ser muito comprimida, independente do método de compressão adotado. Isso ocorre por que, na maioria dos casos, os arquivos não possuem muita redundância ou não são suficientemente grandes para haja um nível de redundância explorável – em alguns casos o conteúdo dos arquivos pode ainda se aproximar de dados aleatórios, não havendo redundância a ser explorada.

*Registro Acadêmico: 096149

Suponha que, dado um arquivo, um método de compressão que gere um novo arquivo com tamanho igual ou inferior à metade do original poderia ser considerado um bom método. Existem 2^n arquivos de *n-bits* que podem ser comprimidos em outros 2^n diferentes arquivos com tamanho menor ou igual $\frac{n}{2}$. Porém o número total desses arquivos é

$$N = \sum_{i=0}^{\frac{n}{2}} 2^i = 2^{1+\frac{n}{2}} - 1 \approx 2^{1+\frac{n}{2}}$$

logo apenas N dos 2^n arquivos podem ser comprimidos de forma eficiente. Se o arquivo possuir, por exemplo, 1000 *bits*, dos 2^{1000} arquivos apenas 2^{501} poderão ser comprimidos eficientemente. Torna-se evidente que não se deve esperar que um método de compressão possa comprimir eficientemente todos os arquivos ou mesmo uma parcela muito grande deles.

Em segundo lugar, a forma na qual surge a redundância e sua quantidade depende diretamente do tipo de dado que é manipulado. Por exemplo, o tipo e a quantidade de redundância existente em um arquivo de texto puro – basicamente um conjunto de *bytes* descrevendo caracteres – é muito diferente da encontrada em um arquivo de imagem bitmap – onde pode-se ter *bytes* representado triplas RGB. Assim, é preciso elaborar métodos de compressão adequados para comprimir a representação de um determinado tipo de dados, considerando suas características próprias.

Os métodos de compressão, de uma forma geral, podem classificados em quatro categorias [16]:

Run Length Encoding representam um conjunto de símbolos equivalentes e contíguos por um par *ns*, onde n é o número de ocorrências e s é o símbolo em questão.

Baseados em dicionário utilizam um dicionário para armazenar símbolos ou conjuntos de símbolos comuns.

Estatísticos utilizam análises estatísticas sobre os dados para identificar a melhor forma de representação.

Transformadas utilizam transformadas para descorrelacionar dados e permitir que sejam codificados independentemente.

Os métodos comumente utilizados na compressão de texto são os baseados em dicionário e os estatísticos. Os métodos baseados em dicionário exploram a repetição de padrões, substituindo-os por *codewords* de tamanhos fixos, que são símbolos especiais comumente mais curtos que o padrão original. Tais *codewords* são utilizados como índices de um dicionário que contém, efetivamente, o conjunto de símbolos originais do padrão. Assim, o algoritmo de descompressão apenas deve fazer a substituição dos *codewords* pelo conteúdo, no dicionário, que eles representam.

Os métodos estatísticos, por sua vez, utilizam modelos estatísticos dos dados originais para obter os dados comprimidos, onde a qualidade da compressão depende de quão bom é o modelo adotado. Os métodos estatísticos usam *codewords* de tamanho variável, com os mais curtos associados aos símbolos ou grupos de símbolos mais freqüentes nos dados originais [16]. Note que o projeto de um método estatístico envolve dois problemas fundamentais: a escolha de *codewords* que podem ser descomprimidos de forma unívoca e a atribuição de *codewords* com o tamanho médio mínimo.

Em geral, os métodos baseados em dicionário possuem descompressores mais rápidos e simples, enquanto os métodos estatísticos apresentam as melhores taxas de compressão. Além disso, para cada método baseado em dicionário existe um método estatístico equivalente, em termos de taxa de compressão, e que pode ser otimizado para alcançar taxas ainda melhores [4]. Entretanto, os métodos estatísticos apresentam maior custo computacional na descompressão.

Os métodos de compressão podem, ainda, serem distinguidos quanto a dois aspectos: se o método é adaptativo e se o possui ou não alguma tolerância a perda. A adaptatividade refere-se ao comportamento do método em relação aos dados de entrada. Um método não-adaptativo é rígido e não modifica suas operações ou parâmetros durante todo o processo de compressão. Em contrapartida, existem os métodos adaptativos, que modificam suas operações e/ou parâmetros segundo análises sobre os dados de entrada, explorando assim as suas características específicas. Existem ainda os métodos denominados semi-adaptativos que efetuam a análise e a compressão em dois passos separados: primeiro realizam uma análise sobre os dados e depois utilizam tais informações no processo de compressão.

A perda refere-se ao fato de o método permitir ou não alguma perda de dados em benefício da compressão. Nos casos de métodos com perda, também conhecidos como métodos irreversíveis, os dados descomprimidos não são iguais aos dados originais, mas sim aproximações. Em tipos de dados como imagens e amostras de áudio, pequenas perdas - muitas vezes imperceptíveis - são aceitáveis e propiciam grande melhora na eficiência do método. Já os métodos sem perda, ou métodos reversíveis, garantem que os dados descomprimidos serão iguais aos dados originais. Esse tipo de compressão deve ser usado sobre dados que, comumente, não toleram qualquer aproximação, como arquivos de texto ou binários executáveis.

O desempenho de um método de compressão pode ser medido através da razão de compressão ou da taxa de compressão. A razão e a taxa de compressão são dadas pelas equações 1 e 2, respectivamente. Note que na razão de compressão, quanto menor o valor melhor é a compressão, enquanto que na taxa de compressão, quanto maior o valor melhor é a compressão.

$$razao\ de\ compressao = \frac{(tamanho\ comprimido)}{(tamanho\ original)} \quad (1)$$

$$taxa\ de\ compressao = 1 - (razao\ de\ compressao) \quad (2)$$

3. COMPRESSÃO DE CÓDIGO

A escolha de um método de compressão de código deve considerar o objetivo da redução. Basicamente, um programa pode ser comprimido com o intuito de reduzir o espaço de armazenamento necessário ou melhorar os tempos de transferências quando transmitindo programas por um meio de

comunicação. Tais abordagens tendem a efetuar a descompressão do código na medida em que o carregam na memória principal, de forma que o processador possa então executar as instruções como faria com qualquer outro programa não comprimido. Note, entretanto, que o benefício nessas abordagens concentra-se em reduzir o consumo de memórias secundárias ou de tráfego de rede, não afetando o consumo da memória principal, que tende a ser um recurso mais caro e escasso.

O problema causado pelo consumo de memória principal é ainda mais agravado devido às ferramentas adotadas e ao aumento da complexidade. O uso de linguagens de alto nível, principalmente as que necessitam de interpretadores, representam um aumento significativo na quantidade de instruções que compõem um programa. Aliado a isso, há um crescimento gradativo da complexidade dos *softwares*.

Sendo assim, existe um conjunto de técnicas que objetivam permitir que um programa possa ser carregado em memória ainda comprimido e que o processo de descompressão possa ser realizado na medida em que é executado. Nessas técnicas os métodos tradicionais de compressão reversível, em sua maioria, não podem ser diretamente utilizados na compressão de código devido a dois aspectos inerentes do problema: a necessidade de descompressão aleatória e a restrição de recursos no processo de descompressão [4, 12].

O primeiro aspecto refere-se ao fato de que o código de um programa é composto por, além das instruções que são executadas seqüencialmente, instruções de desvio condicional ou incondicional. Estudos estatísticos, baseados na análise das instruções de programas variados para uma dada arquitetura de conjunto de instruções, indicam que cerca de 30% das instruções de um programa são desvios. Dessa forma, o método de compressão deveria permitir que o processo de descompressão pudesse ser iniciado a partir do início de qualquer bloco básico do programa.

O segundo aspecto refere-se à quantidade de tempo e recursos exigidos pelo algoritmo de descompressão, sobretudo de memória. Esse é um aspecto crítico quando a descompressão é realizada em tempo de execução, pois tal processo deve impactar o mínimo possível no tempo de execução do programa em si. A fim de solucionar esse problema, muitas propostas de mecanismos para execução de código comprimido prevêem a descompressão por um *hardware* dedicado e também o uso de algoritmos mais simples, como os baseados em dicionário.

As subseções a seguir apresentam uma visão geral de alguns métodos comumente utilizados pelas técnicas de compressão de código.

3.1 Compressão por dicionário

A compressão por dicionário permite reduzir o tamanho de um programa criando um dicionário de símbolos comuns. Um símbolo pode ser uma instrução inteira ou parte dela, ou ainda um conjunto de instruções. O dicionário contém todos os símbolos únicos do dicionário e cada símbolo do programa é substituído por um índice do dicionário [12].

Se um símbolo composto por várias instruções for muito recorrente, o algoritmo de compressão irá criar uma única entrada de dicionário para esse símbolo e dessa forma substituirá um conjunto de instruções do programa original por apenas um índice. Assim, se em média o índice é menor que o símbolo e o *overhead* do dicionário não é muito grande, é possível redução em relação ao programa original.

$$nw > n\lceil log_2(d) \rceil + dw \tag{3}$$

A equação 3 apresenta a relação que deve ser satisfeita para que se obtenha alguma redução quando usando compressão por dicionário, onde n é o número de instruções estáticas do programa, w é o número de *bits* de cada instrução e d o número de símbolos do dicionário. Note que a equação desconsidera a área física ocupada pelo descompressor.

Uma técnica de compressão baseada em dicionário pode, propositalmente, manter alguns símbolos menos recorrentes fora do dicionário, ou seja, o código comprimido passa a ter não apenas índices, mas também símbolos do programa original. Essa abordagem permite reduzir o tamanho final do dicionário evitando a indexação de símbolos pouco recorrentes e que, potencialmente, não se beneficiariam em termos de compressão. Dessa forma, tais técnicas precisam adotar algum código ou construção especial que permita diferenciar um símbolo de um índice de dicionário, quando descomprimindo um programa.

3.2 Codificação diferencial

A codificação diferencial, também conhecida como *codificação delta*[1], consiste em comprimir um conjunto de valores em termos de suas diferenças. O primeiro valor é codificado explicitamente, fornecendo assim um ponto de partida. Os demais valores, por sua vez, são codificados como sendo a diferença entre si e o valor anterior. Quando a diferença entre os valores é pequena, pode-se representá-la com um número menor de *bits* do que o valor original [12].

Esta técnica não tem muito uso prático na compressão de código, uma vez que a maioria das instruções de uma máquina arbitrária é composta por um identificador de instrução e um ou mais campos. Mesmo que esses dados fossem considerados como valores inteiros, a diferença obtida não seria pequena o bastante para justificar o uso da técnica. Comumente, essa técnica é utilizada para reduzir o tamanho ocupado por tabelas de decodificação de programas comprimidos.

3.3 Empacotamento de dados

Os *codewords* gerados por uma técnica de compressão de código podem ter tamanhos que não se alinham ao tamanho da palavra de uma máquina. Para solucionar esse problema pode-se efetuar um *empacotamento*[2] dos *codewords* de forma que dois ou mais *codewords* juntos estejam alinhados á palavra da máquina.

Suponha, por exemplo, que um *codeword* de uma técnica baseada em dicionário possui 12 *bits*, sendo 4 *bits* de controle

[1] Do inglês *delta encoding*.
[2] Do inglês *data packing*

e 8 *bits* para índice, poder-se-ia empacotar dois *codewords* de forma que nos primeiros 8 *bits* ficariam o par de *bits* de controle e nos próximos 16 *bits* ficariam os dois índices. Se a máquina para qual o código foi gerado possui uma palavra de 8 *bits*, o acesso aos *codewords* poderia ser feito naturalmente pelo processador e de uma forma eficiente [12].

4. TÉCNICAS DE COMPRESSÃO

Nas subseções a seguir, serão apresentadas algumas técnicas utilizadas a fim de reduzir o tamanho do código de um programa, a fim de obter uma menor representação em memória principal ou secundária. Muitas das técnicas apresentadas são baseadas em vários conceitos ou métodos em comum, o que dificulta uma classificação considerando tais características. Além disso, a maioria dos métodos realiza a compressão uma única vez quando o binário executável é gerado, mas o momento de descompressão pode variar bastante: imediatamente antes de executar uma instrução, quando um procedimento é invocado, quando o programa é carregado na memória, etc. Sendo assim, optou-se por classificar as técnicas de acordo com o momento em que ocorre o processo de descompressão.

4.1 Descompressão de instruções individuais

As técnicas de compressão que atuam sobre o conjunto de instruções da arquitetura podem ser classificadas em duas categorias.

A primeira compreende as técnicas relativas ao projeto do conjunto de instruções de uma arquitetura, onde é possível especificar uma forma de codificação que permita representar, em uma quantidade menor de *bits*, todas as informações necessárias para a execução de uma instrução pelo processador. Essas instruções são decodificadas normalmente pelo estágio de decodificação do processador, comumente não exigindo o uso de algoritmos de compressão ou *hardwares* adicionais.

Outra categoria refere-se aos métodos por *software* que fazem uso de interpretadores. Tais métodos adotam uma codificação muito compacta para as instruções, mas que diferentemente das técnicas da categoria anterior, não podem ser diretamente decodificadas pelo processador. Para a decodificação e execução de um programa usa-se um interpretador, que é escrito em linguagem nativa e fornece um ambiente de emulação.

Thumb e MIPS16

Os processadores Thumb e MIPS16 foram definidos como um subconjunto das arquiteturas ARM e MIPS-III, respectivamente. A escolha das instruções que fariam parte do subconjunto, em ambos os casos, foi realizada através de uma análise das instruções mais freqüentes, considerando as que não necessitam de 32 *bits* em sua codificação e que sejam relevantes para que os compiladores pudessem gerar um código objeto menor.

As instruções originais, anteriormente codificadas em 32 *bits*, passaram a ser codificadas em 16 *bits*, reduzindo-se o número de *bits* que codificam os registradores – resultando em menos registradores disponíveis – e os imediatos – resultando em um menor intervalo de valores possíveis [4].

Durante a execução do programa, instruções de 16 *bits* são buscadas na memória, decodificadas para suas correspondentes em 32 *bits* e então executadas pelo processador. Assim, esses processadores não possuem efetivamente um conjunto de instruções específico em 16 *bits*, mas ao invés possuem um mecanismo em *hardware* que permite mapear instruções de 16 *bits* nas instruções originais em 32 *bits* [13, 15]. Note, entretanto, que apesar das instruções de 16 bis serem decodificadas para 32 *bits*, a limitação no número de registradores e valores de imediatos possíveis ainda permanece.

Larin e Conte

Larin e Conte [11] introduzem uma abordagem diferenciada de codificação do conjunto de instruções. Os autores assumem que a codificação do conjunto de instruções de uma arquitetura pode ser otimizada para um programa específico. Através de um compilador específico, os autores conseguem gerar o código comprimido do programa e também uma lógica programável capaz de fazer a busca e decodificação das instruções. Essa abordagem permite que se possa, por exemplo, reduzir o tamanho de campos de uma instrução. Assim, supondo que uma aplicação utilize no máximo 8 registradores, o campo que define o registrador que uma instrução deve manipular poderia ser codificado em apenas 3 *bits*.

Directly Executed Languages

Flynn [6] introduziu o conceito de linguagens diretamente executáveis (DEL[3]), onde a representação do código pode ser personalizada para cada programa e linguagem. Uma linguagem diretamente executável é uma representação intermediária entre a linguagem do código fonte (de alto nível) e a linguagem de máquina, e sua execução é realizada através de um interpretador. Essa representação tem a vantagem de prover um modo eficiente de representar programas e sua representação é menor devido a três razões. Primeiramente, assumindo que a linguagem de código fonte é a representação ideal do programa, seus operadores são os mesmos encontrados na linguagem de alto nível.

Em segundo lugar, uma linguagem diretamente executável não faz uso operadores de *load* e *store* como linguagens de máquina. Ao invés disso, são utilizadas referências diretas aos elementos do código fonte. Assim, se um programa especifica uma variável, por exemplo, o interpretador é responsável por encontrar o local de armazenamento e manipulá-la (lendo de, ou gravando em um registrador).

Por fim, todos os operadores e operandos das instruções são alinhados a 1 *bit*. O tamanho dos operandos é definido de acordo com o número de elementos que se pode referenciar no escopo corrente. Assim, se um procedimento referenciar 8 variáveis, essas variáveis podem ser codificadas como operandos de 3 *bits*.

O autor afirma que, comparando o tamanho de representações em linguagem de máquina e representações DEL, foi possível identificar que programas representados em DEL são cerca de 2,6 à 5,5 vezes menores.

[3]Do inglês *Directly Executed Languages*.

Byte-coded RISC

Ernst *et al* [5] desenvolveram também uma abordagem baseada em interpretadores, denominada BRISC, que é um formato de programa comprimido que pode ser executado através da Omniware Virtual Machine (OmniVM). O BRISC adiciona macro-instruções ao conjunto de instruções da OmniVM e obtém redução no tamanho do programa substituindo seqüências repetidas de instruções no programa por *codewords* que se referem as macro-instruções. Tais macro-instruções atuam como modelos e possuem campos que podem ser utilizados para passagem de argumentos.

Entretanto, pelo fato do BRISC ser interpretado, é preciso considerar o custo de tamanho do interpretador. Assim, o uso do BRISC torna-se viável se o tamanho do interpretador somado ao tamanho do programa comprimido for inferior ao tamanho do programa original compilado diretamente em linguagem de máquina.

4.2 Descompressão na chamada de procedimento

Esta seção apresenta uma técnica de compressão que trata procedimentos como unidade básica de compressão. Sendo assim, o processo de descompressão é realizado somente quando é preciso chamar um procedimento, e quando ocorre, todas as instruções do procedimento em questão são descomprimidas.

Kirovski et al

Kirovski *et al* [10] apresentam uma técnica de compressão onde cada procedimento é comprimido utilizando o algoritmo Ziv-Lempel. A técnica faz uso de dois elementos: um serviço de diretório, que mapeia procedimentos entre o espaço de endereçamento comprimido e descomprmido; e um *cache* de procedimento, que é um local reservado onde os procedimentos descomprimidos são colocados.

Quando um procedimento é invocado, o serviço de diretório localiza o procedimento no espaço de endereçamento comprimido e o descomprime no *cache* de procedimento. Os procedimentos são colocados em endereços arbitrários do *cache* de procedimentos. Caso haja falta de espaço disponível no *cache* de procedimentos, uma rotina de gerenciamento de memória desapropria um ou mais procedimentos para liberar recursos.

Os desvios cujo alvo esteja dentro do escopo do procedimento, que são os tipos mais freqüentes, são automaticamente encontrados na forma usual uma vez que são relativos. Entretanto, chamadas de procedimento devem sempre fazer uso do serviço de diretório, pois o procedimento pode estar localizado em qualquer endereço do *cache* de procedimentos.

Uma das vantagens desse método é que ele pode ser implementado com um suporte de *hardware* mínimo e não requer qualquer alteração no conjunto de instruções da arquitetura. Como desvantagem, as chamadas de procedimentos podem se tornar muito custosas, já que podem envolver uma descompressão a cada chamada.

Os autores relatam uma razão de compressão de 60% em instruções SPARC, entretanto não esclarecem se essa razão considera o overhead do serviço de diretório, o código do algoritmo de descompressão, o código de gerenciamento *cache* de procedimentos e a quantidade de espaço disponível nesse *cache*.

4.3 Descompressão em memória

As técnicas de descompressão em memória atribuem a tarefa de descompressão ao sistema de memórias, ocultando tal processo do processador. Nesse modelo, a descompressão das instruções é realizada quando o processador solicita uma instrução à memória. Dessa forma, o processo de descompressão fica totalmente transparente ao processador e o tempo necessário pela descompressão passa a ser encarado como uma latência de acesso à memória [12].

Uma das vantagens dessas técnicas é que não é necessária nenhuma modificação no processador, permitindo inclusive que a compressão de código seja independente do conjunto de instruções da arquitetura.

Compressed Code RISC Processor

O CCRP é uma técnica que emprega um *cache* de instruções modificado para executar instruções comprimidas. Quando um programa é compilado, as linhas de *cache* são comprimidas utilizando a codificação Huffman. Em tempo de execução as linhas de *cache* são obtidas da memória principal, descomprimidas e colocadas no *cache* de instruções. As instruções obtidas desse *cache* têm os mesmos endereços que a as instruções no programa original, assim não é requerida nenhuma modificação no núcleo do processador para suportar o mecanismo.

Quando ocorre um *cache* miss no *cache* de instruções, o CCRP utiliza uma tabela denominada *Line Accress Table* (LAT) para mapear endereços do *cache* para endereços da memória principal, onde o código comprimido reside. Isso é necessário por que as instruções faltantes no *cache* não estão nos mesmos endereços na memória principal. Considerando que acessos freqüentes a LAT podem reduzir o desempenho do sistema, os autores adicionaram um buffer, denominado *Cache Line Address Lookaside Buffer* (CLB) que armazena os mapeamentos mais recentemente utilizados.

Os autores relatam uma razão de compressão de 73% para MIPS, entretanto não consideram o espaço físico necessário para a máquina de descompressão. Um trabalho completo de Bennes *et al* [2, 1] relata uma implementação em CMOS, com tecnologia 0,8 μm, que ocupa 0,75mm^2 e é capaz de descomprimir 560 Mbits/s.

4.4 Descompressão em tempo de carregamento

As técnicas de compressão apresentadas nessa seção são caracterizadas por efetuar o processo de descompressão durante o carregamento do programa, seja através de rede ou de uma memória secundária. Note, entretanto, que essas técnicas não podem ser utilizadas na solução de problemas relacionados a limitação de memória principal, como as técnicas apresentadas nas seções anteriores, pois o programa, quando em memória principal, já estará totalmente descomprimido para seu tamanho original.

Apesar dessa limitação, as técnicas que efetuam descompressão em tempo de carregamento possuem algumas aplicações muito relevantes. Primeiramente, elas podem ser adotadas quando o problema reside na economia de memória secundária ou de banda da rede. No caso de carregamento através da rede, o menor tamanho do programa proporcionado por tais técnicas permite proporcional redução no tempo de transmissão.

Outra aplicação desse tipo de técnica é a possibilidade de introduzir o conceito de conjunto de instruções independente de máquina. Nesses moldes, torna-se possível que, durante o carregamento, um programa em uma linguagem de intermediária possa ter suas instruções traduzidas para o conjunto de instruções da arquitetura na qual está sendo executado.

Slim binaries

Franz and Kistler [7, 8] desenvolveram um formato de distribuição de programas independente de máquina denominado *slim binaries*. Um *slim binary* é uma representação comprimida da árvore sintática de um programa, gerada pelo compilador. Assim, quando um módulo de carregamento carrega o programa, a árvore sintática é descomprimida e é gerado o código de máquina do programa.

A técnica é muito vantajosa do ponto de vista de redução do programa, já que a árvore sintática pode ser muito comprimida. Apesar de o código de máquina ser gerado durante o carregamento, esse tempo pode ser parcialmente oculto se a geração de código for realizada simultaneamente com a execução do programa. Os autores, inclusive, afirmar que o tempo de carregamento de um *slim binary* e quase tão rápido quando de um programa em código de máquina.

Os autores relatam que a representações em *slim binary* obtiveram um tamanho médio abaixo de um terço do tamanho original dos programas em código de máquina para PowerPC. Essas medições, entretanto, não consideram o tamanho do programa gerador de código que, como alternativa, poderia ser um componente do próprio sistema operacional. Cabe ressaltar que, apesar da grande redução no tamanho dos programas, essa abordagem não se aplica em casos onde a quantidade de memória principal é limitada, já que a descompressão imediatamente no carregamento.

Coffing and Brown

Coffing and Brown [12] propõem um mecanismo de compressão automática para sistemas de arquivos utilizando o gzip. A idéia geral baseia-se em manter comprimidos os arquivos menos recentemente utilizados e descomprimi-los quando houver algum acesso. Os arquivos menos recentemente utilizados são identificados e 75% deles (iniciando dos menos utilizados) são comprimidos.

A fim de melhorar o acesso aleatório e reduzir o custo de descompressão em arquivos muito grandes, os autores adotaram uma abordagem na qual dividem os arquivos em partes que podem ser comprimidas independentemente. Assim, quando uma parte do arquivo é acessada, não é preciso descomprimir todo o arquivo, o que poderia resultar em perda de desempenho para arquivos muito grandes.

O autores realizaram, também, alguns refinamentos na técnica. Os arquivos pequenos, sobre os quais a aplicação da compressão potencialmente não traria ganho significativo, foram mantidos descomprimidos. Além disso, é adotada uma descompressão especulativa que permite ir descomprimindo outras partes de um arquivo que esteja sendo acessado, assumindo que poderão ser lidas em acessos subseqüentes. Os autores, porém, ressaltam que um problema apresentado pelo mecanismo proposto é um aumento significativo da fragmentação do disco na medida em que os arquivos são comprimidos e descomprimidos.

4.5 Compactação em tempo de compilação

Diferentemente dos métodos apresentados nas seções anteriores, que fazem uso de compressão de código, existem um conjunto de técnicas que visam a redução do código de um programa através de *compactação de código*[4]. Johnson [9] apresenta a compactação de código, em termos formais, como sendo um processo que transforma um programa P em outro programa equivalente P', sendo que $|P'| < |P|$ e ambos podem ser diretamente executados por um processador. Note que pelo fato de P' também ser executável, significa que nenhum pré-processamento é requerido, como a aplicação de um algoritmo de decodificação.

A compactação de código é obtida durante o processo de compilação através dos métodos de otimização do código, dessa forma atuando sobre estruturas intermediárias (árvores sintáticas, código intermediário, etc.) ao invés do programa objeto (binário executável). Nas seções a seguir serão apresentadas algumas das técnicas de otimização de código que permitem reduzir o tamanho do código de um programa.

Abstração de procedimentos

O método de otimização por abstração de procedimentos consiste em reduzir o tamanho do código de um programa através da transformação padrões de códigos comuns em funções geradas pelo compilador e, posteriormente, substituindo todas as ocorrências dos padrões em chamadas à essas funções [9]. Dessa forma, quanto mais ocorrências desses padrões existirem no código do programa, maior será a redução obtida.

Apesar do ganho em redução do tamanho do executável, existem custos associado às chamadas de função. Primeiramente, existem custos de desempenho, como os relativos aos desvios necessários para se alcançar o início do procedimento e retornar, que podem resultar em *cache misses* e exigir mais acessos à memória principal.

Além disso, os procedimentos podem efetuar modificações em registradores e os valores originais de tais registradores precisam ser preservados para que o código imediatamente após a chamada de procedimento possa continuar executando com dados válidos. Assim, normalmente é necessária a inclusão de instruções para o salvamento dos registradores antes da chamada de procedimento e também para a recuperação dos mesmos após o retorno.

Cooper e McIntosh [3] apresentam uma técnica nos moldes

[4] Alguns autores consideram o termo como equivalente à compressão sem perda.

de abstração de procedimentos, onde acrescentam ainda a aplicação de renomeação de registrados (seção 4.5) na tentativa tornar idênticos padrões de código que eventualmente possam se apresentar ligeiramente diferentes. Os autores afirmam terem obtido uma razão de compressão média de 95% sobre um conjunto de programas para processamento de sinais e para codificação e compressão de áudio, vídeo e dados.

Liao *et al* [14] propõem uma técnica similar à abstração de procedimentos denominada de mini sub-rotinas. Nessa técnica, são utilizadas sub-rotinas no lugar dos procedimentos. Tais sub-rotinas não necessitam de passagem de parâmetros ou salvamento dos registradores, uma vez que o código da sub-rotina é executado como se ele estivesse no lugar da instrução que efetuou a chamada. O autor relata uma razão de compressão média de 88% sobre um conjunto de programas (`compress`, `gzip`, `gnucrypt`, entre outros) em um processador TMS320C25. A técnica foi aplicada sobre uma representação já otimizada pelo compilador.

Renomeação de registradores

Em geral, uma pequena variedade de *opcodes* são utilizados pelos compiladores na geração de código, sendo que as instruções tendem a diferir entre si em relação aos registradores que utilizam. A renomeação de registradores permite em aumentar a repetição de instruções similares através da renomeação dos registradores utilizados por elas. Dessa forma, a renomeação de registradores permite aumentar a redundância no código do programa de forma que outras técnicas possam alcançar melhores resultados.

A técnica de Cooper e McIntosh [3], apresentada na seção anterior, utiliza renomeação de registradores com essa finalidade. Os autores utilizam uma variação do mecanismo de coloração de grafos na renomeação dos registradores e indicam que a adoção desse mecanismo permitiu evoluir de uma razão de compressão de aproximadamente 99,3% para 95%.

Multiple Memory Access Optimization

Vários processadores possuem instruções que permitem carregar e armazenar dois ou mais valores em memória. Tais instruções são denominadas de instruções de *múltiplo acesso à memória*[5] e, dependendo da arquitetura, variam na forma com a qual especificam os registradores envolvidos no carregamento e armazenamento. Algumas arquiteturas utilizam um mapa de *bits* para definir os registradores a serem utilizados, enquanto outras permitem indicar um intervalo de registradores [9].

Um compilador para um processador que suporte MMA poderia otimizar o código de forma que várias leituras e gravações individuais pudessem ser realizadas numa única instrução. Isso é especialmente útil quando trabalhando em vetores, estruturas ou com qualquer conjunto de dados contíguos.

Johnson [9] introduz um algoritmo denominado *SolveMMA* que efetua otimizações no código de forma a explorar o uso do MMA. O autor apresenta os resultados em termos do número de instruções em funções individuais e indica que enquanto o compilador GCC para um processador Thumb de 32 *bits* gerou, em média, 243 instruções por função, com a otimização proposta obteve-se uma média de 230 instruções por função.

[5]Do inglês *Multiple Memory Access (MMA).*

5. CONSIDERAÇÕES FINAIS

Este artigo permitiu vislumbrar uma parcela das diversas propostas de solução para o problema de redução do tamanho de programas. A adoção de uma ou mais técnicas em um projeto requer uma profunda análise e a escolha correta deve considerar as vantagens e desvantagens de cada técnica, bem como as suas limitações e implicações em termos de custo do projeto.

É importante destacar, também, que a aplicação das diversas técnicas existentes não é mutuamente exclusiva. Na realidade, muitos dos trabalhos e artigos tem se concentrado na união de diversas técnicas a fim de explorar as suas vantagens e/ou compensar suas limitações.

6. REFERENCES

[1] M. Benes, S. M. Nowick, and A. Wolfe. A fast asynchronous huffman decoder for compressed-code embedded processors. In *Proceedings of the 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 43–56, Washington, DC, USA, 1998. IEEE Computer Society.

[2] M. Benes, A. Wolfe, and S. M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI '97)*, ARVLSI '97, pages 219–, Washington, DC, USA, 1997. IEEE Computer Society.

[3] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded risc processors. 1999.

[4] P. C. Ducatte. *Compressão de Programas Usando Árvores de Expressão*. PhD thesis, Universidade Estadual de Campinas, Campinas, 1999.

[5] J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, and S. Lucco. Code compression. *SIGPLAN Not.*, 32:358–365, May 1997.

[6] M. Flynn and L. Hoevel. Execution architecture: The deltran experiment. *IEEE Transactions on Computers*, 32:156–175, 1983.

[7] M. Franz. *Code-Generation On-the-Fly: A Key for Portable Software*. PhD thesis, Institute for Computer Systems, ETH Zurich, 1994.

[8] M. Franz and T. Kistler. Slim binaries. *Commun. ACM*, 40:87–94, December 1997.

[9] N. E. Johnson. Code size optimization for embedded processors. Technical Report 607, University of Cambridge, November 2004.

[10] D. Kirovski, J. Kin, and W. H. Mangione-smith. Procedure based program compression. *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 204–211, Dezembro 1997.

[11] S. Y. Larin and T. M. Conte. Compiler-driven cached code compression schemes for embedded ilp processors. *Microarchitecture, IEEE/ACM International Symposium on*, page 82, 1999.

[12] C. R. Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, The University of Michigan, 2000.

[13] J. Lemieux. Introduction to arm thumb, 2003. Acessado em 10 de Junho de 2011.

[14] S. Y.-H. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, Massachusetts Institute of Technology, 1996.

[15] MIPS Techonologies. *MIPS32 Architecture for Programmmers Volume IV-a: The MIPS16e Application-Specific Extension to the MIPS32 Architecture*, 2003.

[16] D. Salomon. *Data Compression: The Complete Reference*. Springer, 2007.