International Journal of Computer Architecture Education (IJCAE) 2316-9915

Por um Ensino de Arquitetura de Computadores para Cursos de Sistemas de Informação

Cristian Koliver,
Cristina Meinhardt
Mateus Grellert da Silva
Universidade Federal de Santa Catarina (UFSC)
Departamento de Informática e Estatística (INE)
Florianópolis, Santa Catarina 88040-370, Telefone: (48) 3721-9498
Email: https://ine.ufsc.br/docentes/

***Abstract*—The subtle difference among the computer courses in Brazilian Universities has as common barrier the contents of computer architecture and computer organization. Engineering courses focus more on the design of integrated circuits, deepening the knowledge of circuit-level architectures. Science courses, on the other hand, seek to focus on the strategies for increasing the performance presented in the evolution of computing and the future challenges, using intermediate abstractions to circuits. However, for the information systems course, the relevant points are knowing the architecture to get the most out of it during application development. In this paper, we defend a Computer Architecture course driven to undergraduate program in Information Systems. Part of this direction is achieved through the demonstration of how studied hardware aspects can be used in the construction of more efficient and robust software. We present some examples of how this can be done.**

I. Introdução

Analisando os projetos pedagógicos dos diversos cursos de Bacharelado em Sistemas de Informação (SI) ofertados no País, dificilmente encontraremos algum no qual o estudo da Arquitetura de Computadores (AC) não esteja inserido no rol das disciplinas ofertadas. Por outro lado, tradicionalmente a unidade curricular responsável por essa área do conhecimento possui uma ementa que procura condensar, em uma carga horária menor, os mesmos tópicos de duas ou mais unidades curriculares com o mesmo fim de cursos como Ciência ou Engenharia da Computação. Assim, examina-se um lago de conhecimentos com a profundidade de uma poça d'água. Ademais, esse estudo é dissociado das demais disciplinas do curso bem como do perfil esperado do egresso, pelo uso de uma bibliografia geralmente direcionada para cursos de Engenharia Elétrica, Ciência ou Engenharia da Computação, cujas habilidades esperadas são relativamente distintas daquelas de um Bacharel em SI.

Desta forma, é nossa crença que o estudo de AC voltado para cursos de SI deve ser instrumental, no sentido de fornecer conhecimentos que auxiliem o estudante a modelar e implementar soluções de Tecnologia de Informação mais eficazes e eficientes.

Fazendo uma analogia, um motorista pode dirigir um automóvel de maneira bastante razoável desconhecendo totalmente a mecânica e o funcionamento do automóvel. Basta que ele conheça e saiba usar as "interfaces": volante, alavanca de marchas (para carros sem câmbio automático), botões e relógios do painel, pedais, freio de mão, etc. Porém, se ele também sabe como funciona a caixa de câmbio, evitará esticar as marchas em virtude do maior consumo de combustível; se ele conhece o mecanismo de embreagem, utilizará o freio de mão ou pé para segurar o carro em aclives ao invés de balancear a pressão nos pedais do acelerador e da embreagem para não reduzir a vida útil dos discos; se ele souber como funciona a partida elétrica, evitará acionar a chave com faróis ou outros dispositivos elétricos ligados evitando a redução da vida útil da bateria, entre outros. Assim, de maneira análoga, projetista e programadores que conhecem a representação interna dos tipos de dados, as instruções da arquitetura do processador, o funcionamento das memórias, poderão desenvolver aplicações ou sistemas computacionais mais eficientes e menos propensos a erros.

Além disso, considerando aplicações embarcadas, o conhecimento de aspectos do *hardware* embarcado pode ser útil ou mesmo fundamental para a consecução de aplicações mais eficientes e mesmo mais robustas. Essas aplicações são tipicamente executadas sobre um *chip* com um único processador, que varia de 8 a 64 bits, com ou sem um sistema operacional [1]. Assim, elas possuem restrições do *hardware*, tais como memória disponível, velocidade do processador ou da rede (no caso de aplicações distribuídas) e, muitas vezes, também restrições temporais. Vale notar que ainda em 2008, Ebert e Jones [2] apontavam que o volume de *software* embarcado aumentava de 10% a 20% ao ano, dependendo do domínio, à medida que mais dispositivos se tornavam automatizados e os consumidores adquiriam mais desses dispositivos. Também em 2008, o total de microprocessadores embarcados já representava mais de 98% de todos os microprocessadores produzidos. De fato, a taxa de crescimento do *software* embarcado acelerou sobremaneira nas últimas décadas. Por exemplo, os automóveis atuais têm de 20 a 70 unidades de controle eletrônico com mais de 100 milhões de instruções de código de objeto, totalizando cerca de 1 GB de *software* em um automóvel "premium". A valoração de um automóvel é determinada, principalmente, pelo *software* embarcado.

Neste artigo, descrevemos alguns tópicos de *software* que podem ser utilizados no ensino de disciplinas de organização e arquitetura de computadores em cursos de Bacharelado em SI e Tecnologia em Análise e Desenvolvimento de Sistemas. Esse tópicos estão inseridos em uma metodologia e em um livro que estão sendo desenvolvidos pelos autores com o intuito de servirem como suporte para essa disciplina nos cursos supramencionados. Esses aspectos são particularmente interessantes para os egressos que atuarão na área de desenvolvimento de aplicações embarcadas.

II. Arquitetura de Computadores para Sistemas de Informação

A preocupação com uma disciplina de AC direcionada vem desde que os primeiros curso de Bacharelado em SI surgiram, no início dos anos 2000. Em 2007, dos Santos [3] propôs uma ementa para AC levando em consideração as características do curso e, principalmente, o perfil vislumbrado para um egresso.

Também em 2007, Sobreira et al. [4] descreveram os resultados do uso de Linguagens de Descrição de *Hardware* (VHDL) e de ferramentas de automação de projetos na disciplina de AC para Bacharelado em SI com o intuito de aumentar a motivação dos estudantes.

Em Brito [5], é descrito um simulador de sistemas computacionais também usado em sala de aula com o intuito de engajar mais os estudantes no aprendizado. O simulador foi usado no contexto da disciplina Organização de Computadores para ilustrar conceitos relacionados ao funcionamento da Unidade Central de Processamento (UCP).

A principal diferença do presente trabalho em relação relação aos acima mencionados é que enquanto nestes o foco é em relação aos conteúdos ou recursos para o ensino de AC para estudantes de SI, no nosso trabalho procuramos complementar o ensino de hardware mostrando como os conhecimentos relacionados podem ser aplicados no desenvolvimento de *softwares*.

III. AC na Programação

Nesta seção, apresentamos um conjunto de exemplos nos quais conceitos de AC são utilizados em programas para aumentar a eficiência ou diminuir a possibilidade de erros de execução. Em todos nossos exemplos, assumimos que as palavras são de 4 bytes e os tipos de dados seguem o número de bytes identificados na Tabela I.

TABLE I
Número de bytes para os tipos de dados

Tipo de Dado	Número de Bytes
char	1
short int	2
signed	4
int	4
unsigned int	4
float	4
long int	8
double	8
long double	16

A. Representação Interna de Números

O estudo das representações de números em ponto fixo e ponto flutuante pode ser usado para enfatizar a importância da escolha correta dos tipos dos dados da aplicação. Essa escolha influencia na eficiência da aplicação e no corretismo dos resultados. Vejamos alguns exemplos que podem ser usados para ilustrar isso.

Quando do estudo de aritmética nessas duas notações, podemos demonstrar como os tempos de execução de operações sobre dados em ponto fixo, ou seja, tipos **int** (e suas variações), e **char** da linguagem C podem ser muito menores do que sobre dados em ponto flutuante (**float**, **double**, **long double**) pelo fato dos circuitos eletrônicos que implementam as operações aritméticas sobre esses últimos serem mais complexos do que sobre os primeiros. O trecho de código em C do Exemplo 1 teve, em um certo computador, um tempo de execução de um pouco mais de 16 segundos. Trocando apenas o tipo da variável para **unsigned int**, e executando no mesmo computador, o tempo caiu para menos da metade, executando em cerca de 7 segundos apenas, mantendo a mesma configuração do computador utilizado para obter o tempo de execução do Exemplo 1.

Exemplo 1. Tipo de dado *versus* tempo de txecução

```
for(long double i = 0; i < 4000000000; i++);
```

Os tipos de dados escolhidos, por terem diferentes tamanhos, podem influenciar sobremaneira a memória consumida por um programa. Supondo que precisamos armazenar milhões de dados que estão na faixa de valores menores que 255, por exemplo, para representar idades de pessoas. A escolha do tipo correto fará grande diferença. Por exemplo, a declaração da linha 1 do Exemplo 2 consumirá 3 MB mais de memória em relação à declaração da linha 2. Este exemplo ilustra para os estudantes o impacto em memória da escolha correta do tipo de dados para cada variável do sistema a ser desenvolvido.

Exemplo 2. Boa utilização de memória

```
unsigned int idade[1000000]; //ocupa 4 MB
unsigned char idade[1000000];//ocupa 1 MB
```

A influência na precisão, ou seja, do número de algarismos significativos da aproximação, e acurácia (quão próximo do valor verdadeiro está uma aproximação) de dados representados em ponto flutuante pode ser demonstrada por meio de exemplos em programas. Por exemplo, seja a seguinte soma:

$$\frac{1}{3}+\frac{1}{3}+\frac{1}{3}+\frac{1}{3}+\cdots+\frac{1}{3}+\frac{1}{3}.$$

Matematicamente, ela equivale à expressão aritmética

$$\frac{1+1+1+1+\cdots+1+1}{3}.$$

Portanto, se o número de termos n é igual a 90.000.000, temos

$$\frac{90.000.000}{3}=30.000.000.$$

No Exemplo 3, usamos um laço para executar o cálculo. Em ambos os casos, ocorreu uma perda de precisão a cada iteração que vai sendo acumulada, mas no caso do tipo **float**, essa perda é muito maior.

Algo que causa surpresa à maioria dos programadores iniciantes é saber que $\frac{1}{10}$ (assim como $\frac{1}{100}$, $\frac{1}{1.000}$, etc.) não é representável de forma exata no formato ponto flutuante binário padrão. Isso pode ser demonstrado por meio de outro contraexemplo em C (Exemplo 4). O primeiro **printf** escreve 1 (**verdadeiro**) para o resultado da comparação $v1 == 0.3$ ($v1$ igual a $0,3$). Porém, na execução do segundo **printf** é escrito 0 (**falso**) para o resultado da comparação $v2 == 0.5$ em virtude do arredondamento das frações decimais.

Exemplo 3. Precisão de cálculos

```
#include<stdio.h>
#define N 90000000
 void main() {
   long double soma_d = 0.0;
   float soma_f = 0;
   unsigned long int i;
   for(i = 0; i <= N; i++) {
     soma_d = soma_d + 1/3.0;
     soma_f = soma_f + 1/3.0;
   }
   printf("%.30Lf\n", soma_d);
   printf("%.30f\n", soma_f);
}
```

```
Soma com long double:
30000000.333351281920840847305953502655
Soma com float:
8388608.000000000000000000000000000000
```

Exemplo 4. Imprecisão em frações decimais

```
#include <stdio.h>
  void main() {
    double v1 = 0.1 + 0.1 + 0.1;
    double v2 = 0.1 + 0.1 + 0.1 + 0.1 + 0.1;
    printf("%i\n", v1 == 0.3);
    printf("%i\n", v2 == 0.5);
 }
```

Outro aspecto importante de demonstrar, é que a ocorrência de *overflow* nem sempre faz com que a execução de um programa seja abortada, principalmente em linguagens fracamente tipadas. Em vez disso, ele continua a execução mas produzindo dados incorretos. No Exemplo 5, analisando superficialmente o código em C, pode-se pensar que o programa escreve os inteiros de 0 a 300 e encerra sua execução. Porém, o que de fato ocorre é que o programa entra em um laço infinito: ele escreve os inteiros de 0 a 255, depois i volta a ser 0 e ele volta a escrever os inteiros de 0 a 255, repetindo esse processo infinitamente.

Exemplo 5. Loop devido a *overflow* de variável

```
for (unsigned char i = 0; i <= 300; i++)
  { printf("%i\n", i); }
```

O *underflow* pode ocorrer durante a execução de um programa com dados representados em ponto flutuante quando o resultado real de uma operação em magnitude é menor (ou seja, mais próximo de zero) do que o menor valor representável como um número de ponto flutuante normal no tipo de dados de destino. Por exemplo, se a parte do expoente puder representar valores de -128 a 127, um resultado com um valor menor que -128 poderá causar *underflow*. Seja o cálculo da raiz quadrada de x usando o Método de Newton realizado pela Equação 1:

$$t_{n+1} = \frac{1}{2} \times \left(t_n + \frac{x}{t_n} \right). \tag{1}$$

Em C, esta equação pode ser descrita pelo código do Exemplo 6. Matematicamente, a sequência de iterações converge para x. Para $x = 2$, os valores de t são 2, $\frac{3}{2}$, $\frac{17}{12}$, $\frac{577}{408}$ e $\frac{665.857}{470.832}$. O último destes valores é aproximadamente $1,414213562374$ na forma decimal e coincide com $\sqrt{2}$, numa convergência rápida com onze casas decimais de precisão.

Exemplo 6. Cálculo da raiz quadrada

```
long double x, t;
scanf("%Lf", &x);
for (t = x; t*t >= x; ) {
  t = (x/t + t)/2.0;}
printf("%Lf\n", t);
```

O programa, no entanto, entra em um laço infinito porque a condição de saída do laço nunca é satisfeita. Isso porque em determinado momento os termos são arredondados para 0 porque eles são menores do que a *precisão da máquina*, o menor número ϵ tal que 1,0 + $\epsilon \neq$ 1,0. Uma maneira confiável para fazer o cálculo é escolher alguma tolerância de erro ϵ, por exemplo 1^{-15}, e tentar encontrar um valor t tal que

$$\left| t - \frac{x}{t} \right| < \epsilon \times t.$$

Assim, o código implementando essa abordagem é apresentado no Exemplo 7.

Exemplo 7. Cálculo da raiz quadrada com tolerância a erro

```
long double x, t;
long double epsilon = 0.000000000000001;
scanf("%Lf", &x);
for (t=x; t*t - x >= epsilon; ) {
   t = (x/t + t)/2.0;}
printf("%Lf\n", t);
```

Assim como a raiz quadrada, muitos outros cálculos numéricos envolvem a soma de milhares de termos pequenos. Toda vez que um programa executa uma operação aritmética envolvendo números representados como numerais em ponto flutuante, um erro adicional de pelo menos ϵ é introduzido. Brian Kernighan[1] e Phillip James Plauger[2] disseram [6]:

[1]Um dos criadores da linguagem C.

[2]Pioneiro no uso e na padronização da linguagem C.

> "*Números de ponto flutuante são como pilhas de areia: cada vez que você move uma pilha, você perde um pouco de areia e pega um pouco de sujeira*."

Em [7] são enumeradas algumas catástrofes ocorridas no mundo real devido a problemas de precisão e arredondamento que podem ser usadas pelo docente para reforçar a importância do conhecimento da representação interna dos números:

- explosão do foguete Ariane 5 em 1996;
- acidente com o míssil estadunidense *Patriot* durante a Guerra no Golfo em 1991, que atingiu um quartel do exército estadunidense, matando 26 pessoas;
- *Pentium FDIV bug*, um defeito ocorrido nas unidades de processamento em ponto flutuante da Intel em 1994;
- naufrágio da plataforma de petróleo *Sleipner* em 1991 devido à imprecisão no cálculo que subestimou em 47% a tensão de cisalhamento do valor de referência;
- acúmulo de erros de arredondamento durante 22 meses no índice da bolsa de valores de Vancouver que causou sua subestimação em mais de 50%.

A adoção de exemplos de fatos motiva os estudantes quanto a realidade do impacto das decisões tomadas no nível de programação durante o desenvolvimento de aplicações, fazendo um reforço do conhecimento.

B. Memória Principal

Quando do estudo da memória principal, pode ser interessante explorar a questão de como os dados são alocados e endereçados para tentar diminuir o consumo desse recurso. Aqui também podemos novamente explorar o conhecimento da representação interna dos dados.

Algumas linguagens de programação permitem que os dados que compõem uma estrutura ocupem valores menores do que 1 byte. Em C, podemos especificar o tamanho (em bits) dos campos de uma estrutura. A ideia é usar a memória com eficiência quando sabemos que o valor de um campo ou grupo de campos nunca excederá um determinado limite ou está dentro de um pequeno intervalo. Por exemplo, consideremos o Exemplo 8. A saída será *Tamanho da data: 12 bytes* e *Data: 31/12/2014*.

Exemplo 8. Adoção de variáveis para Data

```
#include <stdio.h>
 typedef struct {
    unsigned int d;
    unsigned int m;
    unsigned int a;
 } data_t;

void main() {
  data_t dt= {31, 12, 2014};
  printf("Tamanho␣da␣data:␣%lu␣bytes\n",
  sizeof(data_t));
  printf("Data:␣%d/%d/%d", dt.d, dt.m, dt.a);
}
```

Podemos otimizar o espaço usando *campos de bits* (*bit fields*) como mostra o Exemplo 9. Como sabemos que o valor do campo *d* vai de 1 a 31 (no máximo), na estrutura acima definimos que ele ocupará apenas 5 bits (e não 4 bytes), já que $2^5 = 32$; como *m* vai de 1 a 12, 4 bits ($2^4 = 16$) são suficientes. A saída desse programa será será *Tamanho da data: 8 bytes* e *Data: 31/12/2014*.

Exemplo 9. Campos de bits para Data

```
#include <stdio.h>
 typedef struct {
    unsigned int d: 5;
    unsigned int m: 4;
    unsigned int a;
} data_t;

void main() {
  data_t dt= {31, 12, 2014};
  printf("Tamanho␣da␣data:␣%lu␣bytes\n",
  sizeof(data_t));
  printf("Data:␣%d/%d/%d", dt.d, dt.m, dt.a);
}
```

No entanto, se o mesmo código for escrito usando **signed int** (ou, simplesmente, **int**) e o valor dos campos for além dos bits alocados à variável, algo interessante poderá acontecer como demonstra o Exemplo 10. A saída será *Tamanho da data: 8 bytes* e *Data: -1/-4/2014*. O problema é que o programador se "esqueceu" que mesmo reduzindo o número de bits de **int**, o bit mais significativo continua sendo o do sinal. Os valores correspondem aos números negativos em complemento de 2.

Exemplo 10. Inteiros para representação de Data

```
#include <stdio.h>
 typedef struct {
   int d: 5;
   int m: 4;
   int a;
 } data_t;

 void main() {
   data_t dt = {31, 12, 2014};
   printf("Tamanho␣da␣data:␣%lu␣bytes\n",
   sizeof(data_t));
   printf("Data:␣%d/%d/%d", dt.d, dt.m, dt.a);
 return 0;
 }
```

Uma mensagem de erro de execução bastante desagradável é *Segmentation fault*, que ocorre quando o processo tenta acessar um endereço de memória que ele não tem permissão para acessar. Quando utilizamos aritmética de apontadores em um programa, esse erro não é raro. Entretanto, um problema ocorre quando o processo faz acesso não intencional a uma área que *pertence* a ele. Nesse caso, o processo não é abortado mas produz resultados inesperados. Isso pode ocorrer quando do processamento de vetores e tem relação íntima de como os dados são alocados na memória. Seja o trecho de código em C do Exemplo 11. O acesso à quinta posição do vetor (que não existe) faz com que a variável *a* seja modificada, recebendo 'F'. Isso ocorre porque, em C, as variáveis locais são colocadas na pilha na ordem inversa na qual elas foram declaradas no programa.

Exemplo 11. Acesso a posições de vetores

```
char a = 'A';
char vet[4] ={'B', 'C', 'D', 'E'};
int i;
for (i = 0; i <= 4; i++) {
     vet[i] = 'F';
}
```

No Exemplo 12, o programa entra em um laço infinito porque cada vez que *i* chega a 4, é atribuído '\0' (código 0 em ASCII) à posição 4 de *vet*, que é, na verdade, onde está a variável *i*. Assim, *i* volta para 0 e a contagem recomeça.

Exemplo 12. Laço infinito

```
int i;
char vet[4] ={'B', 'C', 'D', 'E'};
for (i = 0; i <= 4; i++) {
     vet[i] = '\0';
}
```

A aritmética de apontadores potencialmente também causa problemas de extrapolamento de memória e estouro de *buffers* se não for bem pensada. Vejamos essa situação por meio de exemplo bem simples no qual um vetor é percorrido (Exemplo 13). Sua execução produziu a seguinte saída:

```
40 41 42 43 44 32764 0 0 4195712 0 655526807
32681 1 0 1771285128 32764 32768 1 4195568 0
```

Exemplo 13. Aritmética de apontadores

```
#include <stdio.h>
 void main() {
 int buf[5] = {40, 41, 42, 43, 44};
 int *b = buf;
 while (b < buf + sizeof(buf)) {
   printf("%i", *b);
   b++;
 }
}
```

Supondo que *buf* é o nome do endereço 2000, então, os elementos de valor *buf*[0] = 40, *buf*[1] =41, *buf*[2] =42, *buf*[3] =43 e *buf*[4] = 44 estão nos endereços 2000, 2004, 2008, 2012 e 2016, respectivamente. Quando o compilador encontra uma operação aritmética envolvendo um apontador, ele gera um código que faz a operação relativa ao tamanho do alvo (tipo apontado) do apontador. Portanto, quando somamos uma constante a um apontador para um objeto, o resultado é um apontador para o próximo objeto do mesmo tamanho na memória. Então, a instrução *b*++ incrementa o conteúdo de *b* de 4 em 4, equivalente a *b* += 4.

Importante ressalvar que a função **sizeof** devolve o tamanho da estrutura *em bytes*. Destarte, **sizeof**(*buf*) devolve $4 \times 5 = 20$. Portanto, na linha 5, o compilador gera um código equivalente a **sizeof**(*buf*)$\times$ 4. Em tempo de execução, o valor gerado para toda expressão *buf* + **sizeof**(*buf*) é 2.000 + $(4 \times 20) \times 4$ = 2.080. No final, após a impressão do 5º elemento de *buf*, foram impressos 15 inteiros contendo sujeira de memória.

Poderíamos resolver esse problema da seguinte forma, apresentada no Exemplo 14, onde definimos como condição do laço de repetição (*buf*) mais o tamanho definido.

Exemplo 14. Laço com limite dinâmico

```
#include <stdio.h>
#define TAM 5
void main(void) {
  int buf[TAM] = {40, 41, 42, 43, 44};
  int *b = buf;
  while (b < buf + TAM) {
    printf("%i", *b);
    b++;
  }
}
```

C. A. R. Hoare [3] [8] lamentou que as linguagens posteriores ao Algol60 sacrificaram a verificação de limites de vetores em tempo de execução para melhorar a eficiência dos programas.

C. A Palavra do Computador

A palavra (*word*) da ISA (*Instruction Set Architecture*) se reflete em muitos aspectos da organização e da operação de um computador: a maioria dos registradores do processador geralmente possui o tamanho da palavra; o maior dado que pode ser transferido entre a memória e a UCP em uma única operação é uma palavra em muitas, mas não todas, arquiteturas; o tamanho dos endereços de memória geralmente é o da palavra; em algumas arquiteturas de processador, as instruções de máquina têm sempre o tamanho da palavra. Além disso, ela também tem influência na execução dos programas. Vejamos alguns exemplos a seguir.

O tamanho da palavra determina os endereços de memória acessíveis, que devem ser múltiplos ou *alinhados* à palavra. Tentar o acesso a um endereço de memória desalinhado pode causar um erro de barramento. Esse erro faz com que o sistema operacional aborte a execução do programa que o causou. Na maioria dos casos, o compilador simplesmente não consegue identificar se os dados do programa estão ou não alinhados.

Instâncias da estrutura podem cair em locais não alinhados na memória com mais facilidade, sem gerar nenhum aviso de compilação. Uma regra básica é que nenhum membro da estrutura deve começar em endereços ímpares. Por exemplo, seja uma arquitetura com palavra de 32 bits e o código do Exemplo 15, onde são definidas quatro estruturas.

Na estrutura *A*, o primeiro elemento é **char**, que é alinhado a um byte, seguido de um **short int**, alinhado por 2 bytes. Se o elemento **short int** for alocado imediatamente após o elemento **char**, ele começará em um limite de endereço ímpar. O compilador inserirá, então, um bloco (*pad*) de um byte de preenchimento após o caractere para garantir que o **short int** tenha um endereço múltiplo de 2 (ou seja, 2 bytes alinhados). O tamanho total de *A* será *sizeof*(**char**) + *padding* + 2 *sizeof*(**short int**) = 1 + 1 + 2 = 4 bytes.

[3] Criador do algoritmo de ordenação *quicksort* e um dos membros da equipe no projeto da linguagem Algol60.

O primeiro membro da estrutura *B* é **short int** seguido por **char**. Como **char** pode estar em qualquer limite de bytes, não é necessário preenchimento entre **short int** e **char** que, juntos, ocupam 3 bytes. O próximo membro é **int**. Se o **int** fosse alocado imediatamente após os outros dois membros da estrutura, ele começaria em um limite de bytes ímpares. Precisamos do preenchimento de 1 byte após o membro **char** para tornar o endereço do próximo membro **int** alinhado com 4 bytes. No total, *B* requer 2 + 1 + 1 (*padding*) + 4 = 8 bytes.

Na estrutura *C*, o preenchimento para garantir o alinhamento dos membros será maior. Como o tamanho do membro **double** é 8 bytes, teríamos que colocar o bloco de 7 bytes após o membro **char**. Assim, o tamanho de *C* seria *sizeof*(**char**) + *padding* + *sizeof*(**double**) + *sizeof*(**int**) = 1 + 7 + 8 + 4 = 20 bytes. No entanto, *B* ocupará, de fato, 24 bytes. Isso ocorre porque, assim como os membros da estrutura, as variáveis de tipo *structc_t* também terão alinhamento natural.

Exemplo 15. Estruturas de dados

```
typedef {
  char c;
  short int s;
} structa_t; // Estrutura A


typedef {
  short int s;
  char c;     // 1 byte
  int i;      // 4 bytes
} structb_t; // Estrutura B


typedef {
  char c;
  double d;  // 8 bytes
  int s;
} structc_t; // Estrutura C


typedef {
  double d;
  int s;
  char c;
} structd_t; // Estrutura D
```

Vamos entender isso por meio do Exemplo 16. Suponha que o endereço base de *vetc* seja 0x0000 para facilitar os cálculos. Se *vetc* ocupasse 20 (0x14) bytes como calculamos, o segundo elemento de *vetc* (indexado em 1) estará em 0x0000 + 0x0014 = 0x0014. É o endereço inicial do elemento do índice 1 do vetor. O membro *d* (**double**) dessa estrutura será alocado em 0x0014 + 0x1 + 0x7 = 0x001C (28 decimal) que não é múltiplo de 8 e está em conflito com os requisitos de alinhamento de **double** que é de 8 bytes. Assim, na estrutura *C* haverá o preenchimento de 4 bytes após o membro **int** para tornar o tamanho da estrutura múltiplo de seu alinhamento. Logo, **sizeof**(*structc_t*) = 24 bytes.

Exemplo 16. Vetor

```
structc_t vetc[3];
```

Apesar do preenchimento ser inevitável, existe uma maneira de minimizar seu custo em bytes. Para isso, o programador deve declarar os membros da estrutura em sua ordem crescente ou decrescente de tamanho. Um exemplo é a estrutura *D* acima. Apesar de ter os mesmos membros de *C*, seu tamanho é 16 bytes em vez de 24.

D. Memória Cache

A programação ciente da cache, explorando as localidades espacial e temporal, permite alguma melhora na eficiência dos programas. As estratégias são válidas também para a memória principal, que é, de fato, uma cache do disco. Contudo, elas são ainda mais promissoras para a memória cache dada sua capacidade de armazenamento limitada.

Um caso "clássico" utilizado para demonstrar a exploração da localidade espacial é no processamento de matrizes. Como a memória do computador é inerentemente linear, o mapeamento de uma estrutura bidimensional para unidimensional pode ser feito de várias maneiras ou *layouts* de memória. Os dois mais comuns são por linha e por coluna. No *layout* de linha (C, C++, Python, Pascal, Mathematica, dentre outras linguagens), a primeira linha é colocada na memória contiguamente, depois a segunda linha logo após ela, depois a terceira e assim por diante. O *layout* de coluna (Fortran, Matlab, R, Julia...) coloca a primeira coluna na memória contígua, depois a segunda, etc.

Vamos supor o *layout* de linha, as linhas de cache com duas palavras de 4 bytes e que a cache realiza a busca antecipada do bloco seguinte ao requisitado. Vamos também supor que o computador tem uma cache para dados e outra cache para instruções e que ambas têm apenas 4 linhas. Seja o seguinte trecho do Exemplo 17.

Quando inicia o processamento da matriz, as variáveis *i* e *j* são lidas na cache, ocupando uma linha, assim como a variável *soma*, que ocupará sozinha uma linha inteira da cache. Assim, sobrarão duas linhas de cache para a armazenar os elementos da matriz. No processamento, ocorrerão 9 ausências de cache. Agora vamos supor que invertamos os laços conforme o Exemplo 18. Essa modelagem, do ponto de vista do algoritmo, é equivalente à anterior, mas ocorrerão 18 ausências compulsórias de cache, contra apenas 6 ausências compulsórias quando o processamento é por linha.

Exemplo 17. *Layout* de linha

```
int mat[4][4] = {{0x00, 0x01, 0x02, 0x03},
                 {0x04, 0x05, 0x06, 0x07},
                 {0x08, 0x09, 0x0A, 0x0B},
                 {0x0C, 0x0D, 0x0E, 0x0F}};
long int soma = 0;
for (int i = 0; i < 4; i++) {
  for (int j = 0; j < 4; j++ ) {
    soma = soma + mat[i][j];
  }
}
```

Exemplo 18. *Layout* de coluna

```
for (int j = 0; j < 4; j++) {
  for (int i = 0; i < 4; i++ ) {
    soma = soma + mat[i][j];
  }
}
```

Geralmente, várias operações podem ser executadas sobre elementos de um vetor. Se elas são operações aritmética envolvendo valores escalares (variáveis simples ou constantes), a ordem em que essa operações são executadas pode ser significante. No Exemplo 19, o código fará com que sejam feitos dois acessos a cada posição *i* do vetor para a divisão e, depois, para a multiplicação se não houver cache.

Exemplo 19. Operações com vetores 1

```
for (int i = 0; i < N; i++) {
  soma = vet[i]/n + vet[i] * m + soma;
}
```

Ora, a associatividade da multiplicação permite que codifiquemos nosso trecho como no Exemplo 20. Essa alteração fará com que seja feito um único acesso a cada posição *i*do vetor. Mesmo se o segundo acesso gerasse um acerto na cache L1, o tempo desta operação ainda é superior ao de operações com valores imediatos.

Exemplo 20. Operações com vetores 2

```
for (i = 0; i < 1000; i++) {
  soma = vet[i]*(1/n + m) + soma;
}
```

A fusão de vetores é um estratégia que também pode ser usada para melhorar o desempenho da cache. No Exemplo 21, a cada iteração do laço serão feitos acessos a dois blocos de dados que não são contíguos: um contendo *cpf*[*i*] e outro contendo *nome*[*i*].

Porém, usando uma estrutura ou registro (ou um objeto, em uma linguagem orientada a objetos) para fundir os dois vetores em um único vetor, podemos fazer apenas um acesso. Isso é demonstrado no Examplo 22. Mesmo que a estrutura seja grande demais para caber em um único bloco, ainda será vantajoso se a controladora de cache faz busca antecipada.

Exemplo 21. Fusão de vetores

```
...
int cpf[N];
char nome[N][50];
...
int i, cpf_lido= 56;
for (i = 0; i < N; i++) {
  if (cpf_lido == cpf[i]) {
    printf("nome:␣%s\n", nome[i]);
  }
}
```

Exemplo 22. Fusão utilizando estruturas

```
typedef struct Pessoa {
  char nome[50];
  int cpf;
} pessoa_t;
...
pessoa_t pessoa[N];
...
void main() {
  int cpf_lido;
  for (int i = 0; i < N; i++) {
    if (cpf_lido == pessoa[i].cpf) {
      printf("nome:␣%s\n", pessoa[i].nome);
    }
  }
}
```

Outra estratégia utilizada para otimizar o uso da cache é o *azulejamento de laço* (*loop tiling*) ou *blocagem de laço* (*loop blocking*). Nessa técnica, o espaço de iteração de um laço é particionado em blocos ou azulejos (*tiles*) para ajudar a garantir que os dados usados no laço permaneçam na cache até que sejam reutilizados. O particionamento do espaço de iteração do laço leva ao particionamento de uma matriz grande em blocos menores, assim ajustando os elementos da matriz acessada ao tamanho da cache, aprimorando a reutilização do cache e diminuindo a taxa de ausência de cache. Seja o processo de cálculo da multiplicação de uma matriz por um vetor. Para exemplificar, seja $A_{3\times3}$ a matriz multiplicando e $B_{3\times1}$ o vetor multiplicador. O vetor produto $C_{3\times1}$ é obtido da seguinte forma:

$$\begin{bmatrix} A_{0,0}\times B_{0,0} + A_{0,1}\times B_{1,0} + A_{0,2}\times B_{2,0} \\ A_{1,0}\times B_{0,0} + A_{1,1}\times B_{1,0} + A_{1,2}\times B_{2,0} \\ A_{2,0}\times B_{0,0} + A_{2,1}\times B_{1,0} + A_{2,2}\times B_{2,0} \end{bmatrix}$$

Por exemplo,

$$\begin{bmatrix} 3 & 1 & 2 \\ 4 & 2 & 1 \\ 6 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} 4 \\ 0 \\ 5 \end{bmatrix} = \begin{bmatrix} 22 \\ 21 \\ 69 \end{bmatrix}$$

O código do Exemplo 23 calcula o produto matricial de qualquer matriz *a* com N linhas e N colunas por um vetor de *b* com N posições. O espaço de iteração do laço é $N \times N$. O pedaço acessado da matriz *A*[*i*, *j*] também é $N \times N$. Quando *N* é muito grande e a cache do processador é muito pequena, os elementos da matriz e dos vetores referenciados na *i*-ésima iteração (por exemplo, quando *i* = 1, *j* = 0 a *N*-1) podem exceder ao tamanho da cache, causando ausências por capacidade.

Exemplo 23. Produto matricial de matriz

```
int i, j, A[N][N], B[N], C[N];
for (i = 0; i < N; i++) {
  C[i] = 0;
  for (j = 0; j < N; j++) {
    C[i] = C[i] + A[i][j] * B[j];
  }
}
```

Porém, podemos alterar nosso código para que os acessos sejam feitos em blocos de tamanho 2×2, como mostra o Exemplo 24. Estruturas maiores podem exigir mais do que uma linha de cache para serem carregadas. Para reduzir o tamanho de uma estrutura, temos que dedicar atenção ao problema do alinhamento de dados, que já comentamos anteriormente. Outra aspecto que devemos cuidar é a escolha adequada dos tipos para os membros da estrutura. Estruturas cujos campos possuem tipos que exigem mais bytes podem implicar em menos campos se encaixando em uma linha de cache, causando mais buscas na linha de cache e uma taxa de perda de cache mais alta.

Exemplo 24. Produto matricial em blocos

```
#define N 100
 int i, j, x, y, A[N][N], B[N], C[N];
 for (i = 0; i < N; i += 2) {
   C[i] = 0;
   C[i + 1] = 0;
   for (j = 0; j < N; j += 2) {
     for (x = i; x < min(i + 2, N); x++) {
       for (y = j; y < min(j + 2, N); y++) {
         C[x] = C[x] + A[x][y] * B[y];
       }
     }
   }
}
```

Problema similar ocorre quando temos campos pouco utilizados numa estrutura. Considere a estrutura do Exemplo 25. Apesar de todos os campos poderem ser usados pela aplicação, há um laço mais crítico de desempenho no qual apenas *P* e *Q* são usados. No entanto, os campos não utilizados também são trazidos para o cache do processador a cada iteração.

Exemplo 25. Exemplo de estrutura 1

```
struct pqrs {
  int P;
  int Q;
  int R;
  int S;
};
...
struct pqrs vet[N];
for (int i = 0; i < N; i++) {
  vet[i].P = vet[i].Q;
}
```

Supondo que possamos reescrever o código para que os campos raramente usados sejam movidos para uma estrutura separada conforme Exemplo 26. Agora o laço crítico de desempenho carregará apenas para a cache os campos realmente usados. Essa otimização deve ser usada apenas quando for realmente necessária. A maioria dos programadores achará o código modificado feio e difícil de ler, e a mudança vai contra tudo o que é ensinado sobre programação. No entanto, quando usada com cuidado, ela pode resultar em grandes melhorias de desempenho.

Exemplo 26. Exemplo de estrutura 2

```
struct pq {
  int P;
  int Q;
};
struct rs {
  int R;
  int S;
};
...
struct pq vet_PQ[SIZE];
struct rs vet_RS[SIZE];

for (int i = 0; i < SIZE; i++) {
  vet_PQ[i].P = vet_PQ[i].Q;
}
```

E. Pipelining

Existem várias estratégia de modelagem de um programa que podem ser usadas para melhorar o desempenho do *pipeline* do processador, ao diminuir a ocorrência de *hazards* estruturais, de dados ou de controle. Vamos mostrar apenas um exemplo de mudança que beneficia quando da predição de desvios.

Vamos supor que queremos saber quais servidores públicos federais têm vencimentos maiores do que R$ 20.000,00 por mês. Como são centenas de milhares de servidores, vamos carregar seus dados em um vetor com centenas de milhares de posições. O algoritmo é trivial e podemos modelá-lo como no Exemplo 27.

Exemplo 27. Uso do *pipeline*

```
typedef struct ficha_pessoal {
  double salario;
  char nome[50];
} t_ficha;
...
for (int i = 0; i < N; i++ ) {
  if (ficha[i].salario > 20000) {
    printf("Nome:␣%s\n",ficha[i].nome);
  }
}
```

Supondo que os dados dos servidores estejam ordenados pelo nome, o vetor *salario* estará completamente desordenado do ponto de vista do salário. No entanto, se ele for ordenado pelo salário antes da verificação, então haverá uma sequência de casos, provavelmente majoritários, nos quais a execução do programa não entrará no corpo do **if** seguida por uma sequência de casos em que será executado o corpo do **if**. Esse tipo de regularidade no resultado de instruções de desvio beneficia mecanismos de predição do *pipeline* como preditores de 1 ou 2 bits, pois eles comumente tendem a prever que as decisões mais recentes serão tomadas novamente.

F. Programação Assembly

Se, por um lado, as otimizações realizadas por compiladores consolidados como o GCC geram códigos às vezes até melhores do que aqueles escritos diretamente em *assembly* por

programadores, por outro, é praticamente um consenso de que a programação *assembly* ajuda sobremaneira os estudantes a entenderem como as instruções da máquina e, de maneira mais geral, como o computador funciona.

Tópicos fundamentais como aritmética binária, alocação de memória, processamento de pilha, codificação de conjunto de caracteres, processamento de interrupção e projeto de compiladores seriam difíceis de estudar em detalhes sem uma compreensão de como um computador opera no nível do *hardware*. Como o comportamento de um computador é fundamentalmente definido por sua ISA, a maneira lógica de aprender esses tópicos é estudar uma linguagem *assembly*. Mas a programação *assembly* também pode ser usada para aprimorar as habilidades dos estudantes em programação em linguagem de alto nível (LAN).

Um exemplo são as operações *bit-a-bit* (*bitwise*). Enquanto na programação em *assembly* seu uso é corriqueiro, causa espanto em muitos docentes o desconhecimento dos estudantes e mesmos egressos de cursos de Sistemas de Informação em relação ao seu uso em LANs. Assim, por exemplo, a operação para determinar se um inteiro é par ou ímpar, necessária em uma grande gama de algoritmos, é geralmente implementada usando o operador de resto da divisão em vez da verificação do bit menos significativo por meio de mascaramento. Da mesma forma, dados passíveis de serem modelados como *bitmaps* são geralmente estruturados como *arrays*, usando mais memória e demandando mais tempo de processamento.

Exemplos de programas e exercícios em *assembly* usando *bitmaps* e mascaramento e mapeamento do código *assembly* para LAN podem ser usados para que o estudante se familiarize e passe a usar, de fato, essas estratégias para o desenvolvimento de aplicações nos quais o tempo de resposta e os requisitos de memória são limitados.

IV. Conclusões

Neste artigo descrevemos brevemente como contextualizar disciplinas de AC em cursos de Sistemas de Informações e Análise e Desenvolvimento de Sistemas. Essa contextualização é feita mostrando como os conceitos de AC podem ser aplicados no desenvolvimento de *softwares* mais eficientes e robustos. Entretanto, neste trabalho são ilustradas apenas algumas possibilidades para motivar a interligação da programação com os conhecimentos de arquitetura e organização de computadores.

Muitos outros conceitos de Arquitetura de Computadores podem ser mapeados para a programação. A demonstração do funcionamento de apontadores, raramente estudados nos cursos de Tecnologia da Informação nos quais as linguagens de programação utilizadas como meio são orientadas a objetos, pode subsidiar o estudo de endereçamento da memória. Estruturas representadas por meio de mapas de bits podem ser usadas para reforçar o entendimento da representação interna de números bem como de operações *bit a bit*. Na programação *assembly*, a tradução manual de trechos em linguagem de alto nível pode ser usada para demonstrar o custo das instruções da linguagem.

A viabilização dessa proposta exige o concentração de determinados conteúdos que não são estritamente necessários para que os egressos tenham as habilidades e competências descritas nas Diretrizes Curriculares Nacionais para os cursos de Sistemas de Informação[4].

Referências

[1] N. B. Dale and J. Lewis, *Computer science illuminated*. Jones & Bartlett Learning, 2016.

[2] C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," *Computer*, vol. 42, no. 4, pp. 42–52, 2009.

[3] R. R. dos Santos, "Ensino de arquitetura de computadores em cursos de sistemas de informação," in *Workshop sobre Educação em Arquitetura de Computadores (WEAC 2007)*. SBC, out 2007, pp. 1–6.

[4] P. de L. Sobreira, M. A. M. Dórea, M. E. de Lima, M. Torres, T. O. Motta, V. da C. Wanderley, A. da C. Sena, and C. S. de Alencar, "Competição como uma técnica motivacional no ensino de arquitetura de computadores," in *Workshop sobre Educação em Arquitetura de Computadores (WEAC 2007)*. SBC, out 2007, pp. 39–42.

[5] A. V. Brito, "Simulação baseada em atores para no ensino de arquitetura de computadores," in *Workshop sobre Educação em Arquitetura de Computadores (WEAC 2009)*. SBC, out 2009, pp. 60–63.

[6] B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style*, 2nd ed. USA: McGraw-Hill, Inc., 1982.

[7] R. Sedgewick and K. Wayne, *Introduction to Programming in Java: An Interdisciplinary Approach*, 1st ed. USA: Addison-Wesley Publishing Company, 2007.

[8] C. A. R. Hoare, "The emperor's old clothes," in *ACM Turing award lectures*, 2007, p. 1980.

[4] http://portal.mec.gov.br/escola-de-gestores-da-educacao-basica/323-secretarias-112877938/orgaos-vinculados-82187207/12991-diretrizes-curriculares-cursos-de-graduacao