Sobre o Emprego de Tabelas Hash em Sistemas Operacionais de Tempo Real

Rômulo Silva de Oliveira

Departamento de Automação e Sistemas – Universidade Federal de Santa Catarina (DAS-UFSC)
Caixa Postal 476 – 88040-900 – Florianópolis-SC – Brasil
romulo@das.ufsc.br

Abstract. *Operating systems use hash tables for many different purposes. Hash tables have an excellent average-case behavior but, in the worst-case, it degrades to something like a chaining list. Because of that, the use of hash tables in real-time operating systems is not usual, since those systems may be required to guarantee deadlines. This paper discusses the use of hash table in real-time systems, considering that when the probability of a undesirable behavior is low enough, it can be ignored. It also compares approaches simple and 2-choice for the table design.*

Resumo. *Sistemas operacionais empregam tabelas hash para diversas finalidades. Tabelas hash apresentam excelente comportamento no caso médio mas, no pior caso, degradam para algo semelhante a uma lista encadeada. Em função disto, tabelas hash não são usuais em sistemas operacionais de tempo real onde existe a necessidade de garantir deadlines. Este artigo discute o emprego de tabelas hash em sistemas de tempo real, considerando que, quando a probabilidade de um comportamento indesejado for suficientemente baixa, ele pode ser ignorado. Também são comparadas as abordagens simples e 2-choice para a construção da tabela.*

1. Introdução

Sistemas computacionais de tempo real são definidos na literatura como aqueles submetidos a requisitos de natureza temporal. Nestes sistemas, os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto. Na maioria dos sistemas os requisitos temporais aparecem na forma de deadlines (prazos máximos) para a execução de determinadas tarefas [Liu 2000].

Na literatura os sistemas de tempo real são, em geral, classificados conforme a criticalidade dos seus deadlines. Nos sistemas tempo real críticos (hard real-time) o não cumprimento de um deadline pode resultar em conseqüências catastróficas no sentido econômico, ambiental ou mesmo em risco para seres humanos. Para sistemas deste tipo é necessária uma análise de escalonabilidade ainda em tempo de projeto (off-line). Esta análise procura determinar se o sistema vai ou não atender os requisitos temporais mesmo em um cenário de pior caso, quando as demandas por recursos computacionais são maiores. Quando os deadlines não são críticos (soft real-time) eles apenas descrevem o comportamento desejado.

Em geral, qualquer técnica de programação convencional pode ser empregada no desenvolvimento de sistemas de tempo real não críticos. Entretanto, em sistemas de tempo real críticos, em função da criticalidade dos seus deadlines, apenas devem ser empregadas técnicas de programação que apresentem um comportamento temporal de pior caso que não comprometa o atendimento dos deadlines.

Tabelas estão entre as estruturas de dados mais utilizadas em qualquer sistema operacional, e uma das estruturas de dados mais utilizadas é a tabela por difusão ou tabela hash [Cormen et al 1990]. Tabelas hash apresentam excelente comportamento no caso médio. Entretanto, no pior caso, tabelas hash apresentam um comportamento que degrada para algo semelhante a uma lista encadeada não ordenada. Em função disto, o uso de tabelas hash não é comum em sistemas operacionais de tempo real onde existe a necessidade de garantir deadlines mesmo em cenários de pior caso. Em geral, para uma tabela hash, o comportamento de pior caso é muito pior que o comportamento de caso médio, e garantir o pior caso representa uma enorme sub-utilização de recursos. Esta sub-utilização pode ser compensada em parte pela utilização de servidores que coletam o tempo reservado porém não utilizado por uma tarefa (*gain time*) e o utilizam para executar tarefas com deadlines brandos (soft). Porém, a necessidade de dimensionar o hardware para o pior caso inviabiliza economicamente muitos sistemas.

Na literatura de tempo real é geralmente recomendado o uso de algum tipo de árvore ordenada no lugar de tabela hash, pois os vários tipos de árvores descritas na literatura apresentam um comportamento de pior caso mais próximo do comportamento médio. Entretanto, o caso médio da tabela hash é melhor que o caso médio das árvores balanceadas. Embora o gasto de memória seja em geral semelhante, é mais fácil implementar a tabela hash, além de não ser necessário tempo de processador para manter a estrutura de dados balanceada, como no caso das árvores.

Considerando-se uma tabela hash com N elementos inseridos, o pior caso acontecerá quando a função hash retornar o mesmo valor para todas as N chaves associadas com estes N elementos. Na maioria das implementações isto degradar a operação de consulta para uma pesquisa seqüencial sobre todos os N elementos. Por outro lado, a probabilidade de ocorrer tal colisão total é insignificante na maioria das aplicações. A probabilidade de uma falta no hardware que comprometa todo o sistema será provavelmente muito maior do que uma falta temporal gerada pela colisão de uma grande quantidade de elementos da tabela hash.

Seja P(e) a probabilidade do evento “e” ocorrer durante a vida útil de um dado sistema. É possível definir uma probabilidade minimamente relevante Pr tal que, para quaisquer fins práticos, um evento “e” com $P(e) < Pr$ pode ser considerado como um evento impossível. Desta forma, se for possível mostrar que a probabilidade de ocorrer durante a vida útil do sistema um dado número de colisões na tabela hash é menor que Pr, será possível então considerar este nível de colisão como impossível e utilizar, para fins de análise de pior caso, um número menor de colisões.

Este artigo discute o emprego de tabelas hash em sistemas operacionais de tempo real, a partir da idéia de que, quando a probabilidade de um comportamento indesejado for suficientemente baixa, ele pode ser ignorado. Desta forma, é proposta a utilização do “pior caso relevante” no lugar do “pior caso possível” nos testes de escalonabilidade. Neste artigo será suposto que o número de elementos presentes na

tabela hash a cada momento será N no pior caso, um valor conhecido. Também será suposto que o tratamento de colisões é feito através de encadeamento.

2. Tabelas Hash

Uma tabela hash ou tabela por difusão é uma estrutura de dados que associa chaves com valores. A chave é transformada através de uma função hash (função de dispersão) em um número que é usado como índice dentro da tabela para localizar a posição na qual se encontra os dados procurados [Cormen et al 1990]. Tabelas hash suportam a inserção eficiente de novas entradas, com tempo esperado O(1). O tempo gasto em uma busca depende da função hash e da carga na tabela hash (número de registros). Tanto a inserção como a busca terão tempo esperado O(1) com uma implementação cuidadosa.

Na maioria das análises de complexidade é assumida a premissa simples de que a função hash apresenta uma distribuição uniforme, ou seja, cada chave tem a mesma probabilidade de gerar cada um dos valores possíveis da função hash, ou seja:

- Para quaisquer duas chaves k1 e k2, em duas entradas quaisquer da tabela y1 e y2, a chance de h(k1)=h(k2) é exatamente $1/m^2$, onde m é o número de entradas da tabela.

- Para duas chaves k1 e k2, a probabilidade de h(k1)=h(k2) é exatamente 1/m.

Para o caso geral não é possível garantir que uma dada função hash exibirá tal comportamento. Uma maneira de atacar este problema é usando hash universal. Neste caso, a função hash é escolhida aleatoriamente de maneira independente das chaves, dentro de um conjunto finito H de funções hash. O conjunto H é dito universal se para cada par de chaves distintas k1 e k2, o número de funções hash h em H para as quais h(k1)=h(k2) é no máximo |H|/m, onde |H| é a cardinalidade do conjunto H. Hash universal resulta em um bom desempenho para o caso médio que pode ser demonstrado. Entretanto, como o conjunto de chaves é dinâmico e desconhecido, existe uma probabilidade não nula da tabela apresentar um péssimo comportamento.

No caso de um hash perfeito, a função hash é construida de tal forma que jamais duas chaves resultam no mesmo valor de hash. Embora possível e mesmo eficiente para conjuntos estáticos e conhecidos de chaves, não existem métodos eficientes para gerar funções hash perfeitas para conjuntos dinâmicos de chaves.

2.1. Tratamento de Colisão

O domínio das chaves de uma tabela hash é tipicamente muito maior do que o número de entradas da tabela. É inevitável que duas chaves diferentes acabem sendo mapeadas para a mesma entrada da tabela pela função de dispersão. Neste caso é dito que as duas chaves colidiram.

Uma forma de lidar com o problema das colisões é usar endereçamento aberto. Nesta abordagem, todos os registros são armazenados na própria tabela. Colisões são resolvidas através da localização do próximo espaço livre na tabela após o endereço fornecido pela função de dispersão. Esta procura é feita de forma circular na tabela.

Outra possibilidade de lidar com colisões é encadear as entradas da tabela cujas chaves resultaram no mesmo endereço cálculado pela função de dispersão. Desta forma, cada endereço da tabela corresponde ao início de uma lista encadeada de registros.

2.2. Abordagem 2-Choice

Considere uma tabela hash normal que trata colisões com encadeamento e suponha a premissa de um hash uniforme. Neste caso, após n chaves serem inseridas seqüencialmente, o tamanho do encadeamento mais longo será O(log n / log log n) com alta probabilidade. Alta probabilidade significa com probabilidade pelo menos 1– $O(1/n^{alfa})$ para alguma constante alfa [Gonnet 1981].

Suponha agora que sejam usadas duas funções hash uniformes. No momento de inserir um elemento na tabela, ambas as funções hash são calculadas e portanto duas entradas possíveis da tabela são identificadas (2-choice). O novo registro será inserido na entrada com a lista encadeada menor. Se n chaves forem inseridas seqüencialmente na tabela, o tamanho do maior encadeamento é O(log log n) com alta probabilidade. No caso de uma consulta, novamente será necessário calcular o valor das duas funções hash e pesquisar as duas listas encadeadas indicadas, uma vez que a chave procurada pode ter sido inserida em qualquer uma das duas. Nesta abordagem, o tempo de pesquisa está relacionado com a varredura das duas listas encadeadas associadas com uma dada chave. De qualquer modo, a consulta também apresentará uma complexidade O(log log n) com alta probabilidade. [Mitzenmacher et al 2000]

A abordagem 2-choice tem a vantagem de usar apenas duas funções hash, ser fácil de paralelizar e não necessitar re-hashing de entradas inseridas antes. Ela também oferece robustez no caso das funções hash não serem perfeitamente uniformes [Karp et al 1996]. O uso de alocações balanceadas (visão mais abrangente desta mesma abordagem) é discutido em [Azar et al 1994].

2.3. Uso de Tabelas Hash em Sistemas de Tempo Real

Em [Friedman et al 2003] é considerado o emprego de tabelas hash em sistemas de tempo real embutidos. É proposta uma abordagem incremental para a reorganização de tabelas hash que é similar à coleta de lixo incremental. A abordagem proposta aplica-se a tabelas hash com encadeamento. O artigo é motivado pelos problemas de desempenho que surgem quando uma tabela hash com encadeamento está muito carregada, resultando em muitas colisões e pesquisa linear através de listas longas. A solução convencional é construir uma nova tabela, maior que a original, de uma única vez, removendo os registros da tabela original e fazendo sua inserção na nova tabela. Porém isto deixa a tabela indisponível enquanto ela é completamente reconstruída. A proposta em [Friedman et al 2003] faz as inserções de maneira incremental, minimizando o tempo de tabela indisponível.

Em [Parson 2004] é discutido um algoritmo para reorganizar tabelas hash cujo tratamento de colisão utiliza a técnica de "tabela aberta" (*open hash table*). O desempenho de tabelas hash abertas degrada após muitas inserções e remoções sendo necessário reorganizar toda a tabela. Tipicamente isto é feito de forma monolítica, isto é, toda a tabela permanece indisponível enquanto toda a tabela é reorganizada de uma só vez. O algoritmo proposto em [Parson 2004] alterna entre a construção incremental de uma nova tabela através da cópia seletiva das entradas, e a limpeza incremental da tabela original através do esvaziamento das entradas. Isto limita o tempo de resposta no pior caso a um evento que requer uma busca na tabela.

Poucos trabalhos podem ser encontrados sobre tabelas hash na literatura de tempo real. No presente artigo é suposto que o número máximo de registros na tabela hash, a qualquer tempo, é conhecido. Desta forma a tabela já é criada com um tamanho apropriado, o qual nunca muda. Ampliação da tabela em tempo de execução não é necessário. A questão abordada neste artigo é a identificação de um comportamento que, embora melhor que o pior caso (onde todas as chaves acabam na mesma entrada da tabela), seja probabilisticamente tão improvável que possa ser considerado, para fins práticos, como o pior comportamento possível do sistema. Trata-se portanto de uma questão diferente daquelas abordadas na literatura citada.

3. Critérios de Projeto

Como proposto neste artigo, o emprego de tabelas hash em sistemas operacionais de tempo real requer os seguintes passos:

- Definição de uma probabilidade a partir da qual os eventos são considerados irrelevantes, por serem muito raros;

- Cálculo da probabilidade de uma lista de tamanho k ocorrer durante a vida do sistema;

- Determinação do tamanho t da maior lista a qual ainda possui probabilidade relevante, este será o tamanho da fila no pior caso para fins de análise de pior caso do sistema.

Por exemplo, suponha que a tabela hash é alterada a cada segundo. Vamos supor que a qualquer momento a tabela hash possui no máximo N elementos, sendo N uma característica da aplicação. Ao longo do tempo ocorrem inserções e remoções, mas o número de elementos na tabela a qualquer momento nunca é maior do que N. Desta forma, podemos ter a cada 2s uma nova configuração para a tabela com N elementos. É possível calcular o número de diferentes configurações que são geradas ao longo de 100 milhões de anos: 10^8 ano × 365,25 dia/ano ×24 hora/dia × 3600 s/hora × 0,5 config/s
$\leq 0{,}16 \times 10^{16}$ config

Um critério de projeto possível é considerar que qualquer evento que ocorra em média uma vez a cada 100 milhões de anos é irrelevante para o projeto do sistema computacional em questão. Por exemplo, a colisão de um meteoro gigante com o planeta Terra que destrua a maior parte da vida no planeta. Desta forma, uma probabilidade de 10^{-16} seria considerada o ponto de corte e eventos com probabilidades menores que esta são possíveis, porém irrelevantes.

4. Solução Numérica

Com o propósito de mostrar o conceito, foi implementada em Java uma ferramenta que calcula a PMF (*Probability Mass Function*) da variável aleatória $L_h(N)$, isto é, o tamanho da lista encadeada associada com a entrada h da tabela hash, quando existem N elementos inseridos na tabela. Para o exemplo numérico será adotada como probabilidade de corte o valor 10^{-16}, a partir da discussão apresentada na seção anterior. Serão supostos para a tabela hash os valores M=1000 para o número de entradas da tabela e N=750 para o número máximo de registros na tabela em qualquer momento da vida útil do sistema. Colisões são resolvidas por encadeamento.

Para calcular a função de massa de probabilidade do tamanho das listas encadeadas será feita a operação de adição sobre variáveis aleatórias, isto é, será feita a

convolução das respectivas funções de massa de probabilidade. Seja X a variável aleatória tamanho da lista encadeada de uma entrada qualquer da tabela. Partindo de N=0, temos que a função massa de probabilidade da variável X é definida por P(X=0)=1.

Por exemplo, supondo que a função hash é perfeitamente uniforme (esta restrição será discutida na seção 5 e removida nas experiências), podemos definir como Ph a probabilidade de um novo registro ser inserido em uma dada entrada h da tabela. Este valor pode ser calculado por Ph=1/M, ele independe do número de elementos já inseridos na tabela. Sempre que for feita uma nova inserção, a probabilidade da lista associada com a entrada h aumentar de tamanho é 1/M, e a probabilidade da lista associada com a entrada h ficar com o mesmo tamanho é 1-1/M. A variável aleatória Ih define o aumento de tamanho da lista associada com a entrada h da tabela, e sua PMF é: $P(Ih=0) = 1-1/M$; $P(Ih=1) = 1/M$; $P(Ih>1) = 0$.

Para determinar as probabilidades de um sistema com N=1, basta somar a variável aleatória Lh(0) do tamanho da lista quando N=0, com a variável Ih, a qual determina a alteração que a lista associada com a entrada h da tabela sofrerá após uma nova inserção. Como tanto Lh(0) como Ih são variáveis aleatórias, o resultado da soma Lh(1) também será uma variável aleatória. A PMF de Lh(1) será dada pela convolução das PMFs de Lh(0) e Ih. Inicialmente temos a seguinte PMF de Lh(0):

$P(Lh(0) = 0) = 1$ $P(Lh(0) > 0) = 0$

Após a convolução de Lh(0) com Ih temos a PMF para Lh(1):

$P(Lh(1) = 0) = P(Lh(0) = 0) \times P(Ih = 0) = 1 \times (1-1/M) = (1-1/M)$
$P(Lh(1) = 1) = P(Lh(0) = 0) \times P(Ih = 1) = 1 \times (1/M) = (1/M)$
$P(Lh(1) > 1) = 0$

Após a convolução de Lh(1) com Ih temos a PMF para Lh(2):

$P(Lh(2) = 0) = P(Lh(1) = 0) \times P(Ih = 0) = (1-1/M) \times (1-1/M)$
$P(Lh(2) = 1) = P(Lh(1) = 1) \times P(Ih = 0) + P(Lh(1) = 0) \times P(Ih=1) =$
$= (1/M) \times (1-1/M) + (1-1/M) \times (1/M)$
$P(Lh(2) = 2) = P(Lh(1) = 1) \times P(Ih = 1) = (1/M) \times (1/M)$
$P(Lh(2) > 2) = 0$

Percebe-se que a medida que N cresce, as probabilidades individuais para os possíveis valores de N diminuem, a medida que as PMFs tornam-se mais alongadas. É exatamente para valores grandes de N que a probabilidade de ocorrência torna-se menor do que a probabilidade relevante.

Quando uma PMF for uniforme ou mesmo parcialmente uniforme, é possível acelerar o cálculo da convolução, aproveitando-se as simetrias existentes. Por exemplo, no caso de uma PMF perfeitamente uniforme, todas as entradas da tabela hash apresentam o mesmo comportamento. Logo, é possível calcular para apenas uma entrada a PMF do seu tamanho de lista encadeada, e considerar este resultado válido para todas as entradas. Da mesma forma, uma PMF onde toda uma seção de valores apresentam a mesma probabilidade, é possível realizar os cálculos para apenas um representante desta seção e usar os resultados para todas as entradas desta seção. Nas experiências numéricas deste artigo foram utilizadas funções hash com grandes seções uniformes, o que resultou em redução significativa do tempo de execução.

4.1. Cálculo da PMF

Uma dificuldade encontrada na implementação da ferramenta está na resolução do tipo de dado *double*. Por exemplo, em Java o tipo de dado *double* corresponde ao formato de ponto flutuante com precisão dupla (64 bits) definido pelo padrão IEEE 754. Nesta representação, a resolução fica em torno de 2×10^{-16}, ou seja, não é capaz de representar com precisão números pequenos, mas essenciais para o cálculo correto da PMF.

Para superar esta dificuldade foi utilizada a classe java.math.BigDecimal, a qual implementa números decimais com precisão arbitrária. Um objeto BigDecimal consiste de um número inteiro com precisão arbitrária (valor base) e um inteiro de 32 bits que fornece sua escala. Se a escala for zero ou positiva, ela representa o número de dígitos existentes do lado direito do ponto decimal. Se a escala for negativa, significa que o valor base é multiplicado por 10 elevado a potência do valor da escala. O número representado por um objeto BigDecimal é dado por: valorBase $\times$ 10^{escala}.

A classe BigDecimal prove operações aritméticas, manipulação de escalas, arredondamento, comparações, etc. A classe permite o controle completo sobre arredondamento. Se nenhum modo de arredondamento for especificado e o resultado exato não pode ser representado, uma exceção é lançada.

No caso do cálculo de uma PMF, algum modo de arredondamento é necessário por duas razões. Primeiramente, valores com um número infinido de dígitos são possíveis uma fez que a própria probabilidade de P(Ih = 1) = 1/M pode gerar isto, especialmente por que usualmente M é escolhido entre os números primos. Em segundo lugar, mesmo que seja possível representar todos os valores de interesse com um número finito de dígitos, o número de dígitos cresce de tal forma que o tempo de computação torna-se proibitivo.

Arredondamentos em PMFs são possíveis, mas exigem cuidados especiais. Toda PMF apresenta como propriedade fundamental o fato de que o somatório de todas as probabilidades dos valores individuais soma 1. Por exemplo, com M=3 e função hash com distribuição perfeitamente uniforme temos:

P(Ih = 0) = 1-1/3 = 2/3 P(Ih = 1) = 1/3 = 1/3 P(Ih > 1) = 0

Claramente o somatório das probabilidades é 1. Entretanto, caso o arredondamento seja feito sempre para cima, com 10 casas, a propriedade é perdida:

P(Ih = 0) = 0.6666666667 P(Ih = 1) = 0.3333333334 P(Ih > 1) = 0

A propriedade de somatório igual a 1 poderia ser mantida através de compensações nos arredondamentos, tal como:

P(Ih = 0) = 0.6666666667 P(Ih = 1) = 0.3333333333 P(Ih > 1) = 0

Mas, neste caso, um grave erro é incluído no modelo. A probabilidade da lista associada com a entrada h ter 1 elemento foi artificialmente reduzida. Ou seja, os resultados que seriam obtidos a partir desta análise seriam artificialmente otimistas, o que não é aceitável em um sistema de tempo real que requer garantias para o pior caso.

No sentido de preservar a análise de pior caso, ao mesmo tempo que arredondamentos são viabilizados, optou-se neste estudo por fazer arredondamentos seguidos de compensações que mantém a propriedade de somatório igual a 1, sempre

aumentando a probabilidade do pior comportamento possível. Embora esta solução introduza um pequeno pessimismo na análise, ela preserva as garantias de comportamento no pior caso oriundas desta análise.

Outra questão ligada ao arredondamento é o número de casas decimais a serem preservadas nos valores arredondados. Uma vez que os valores arredondados são probabilidades e trabalha-se aqui com o conceito de probabilidade relevante, optou-se por trabalhar com um número de casas decimais que permita a representação exata de probabilidades 10000 vezes menores do que a menor probabilidade relevante. Por exemplo, caso a menor probabilidade relevante seja 10^{-16}, os arredondamentos preservam 20 casas decimais. A título de ilustração, considere a PMF abaixo:

P(X = 0) = 0.367 P(X = 1) = 0.322 P(X = 2) = 0.211
P(X = 3) = 0.100 P(X > 3) = 0

Um arredondamento para 2 casas decimais geraria a PMF mostrada abaixo. Observa-se que em ambos os casos o somatório das probabilidades é 1, e que a PMF arredondada inclui um pequeno pessimismo ao aumentar a probabilidade de x assumir o valor 3. No entanto, a probabilidade de X assumir valores maiores que 3 continua zero. No caso de uma tabela hash com N registros, teremos sempre que a probabilidade de uma entrada da tabela ter associada a ela mais do que N registros é zero.

P(X = 0) = 0.36 P(X = 1) = 0.32 P(X = 2) = 0.21
P(X = 3) = 0.11 P(X > 3) = 0

É preciso notar que o arredondamento tem como efeito um deslocamento das probabilidades para a direita, na medida que colunas da esquerda perdem probabilidade para colunas da direita. Desta forma, é inserido um pequeno pessimismo no comportamento do sistema, o que invalida o resultado para comportamentos otimistas, mas mantém a validade do resultado para o comportamento de pior caso, aquele no qual estamos interessados neste trabalho.

5. Exemplo Numérico

Para o exemplo numérico será adotada como probabilidade de corte o valor 10^{-16}, a partir da discussão apresentada na seção anterior. Serão supostos para a tabela hash os valores M=1000 para o número de entradas da tabela e N=750 para o número máximo de registros na tabela em qualquer momento da vida útil do sistema. Colisões são resolvidas por encadeamento.

A qualidade de uma função de dispersão (função hash) está associada com a sua capacidade de fazer um espalhamento perfeito das chaves que ela recebe. Desta forma, uma função hash ideal distribui as chaves uniformemente entre as entradas da tabela. Na prática, funções hash apresentam um comportamento aquém do ideal.

A princípio, qualquer função hash é vulnerável a uma situação de grande número de colisões. Em [Carter e Wegman 1979] é proposto o Hash universal (*universal hashing*), uma forma para melhorar o desempenho no caso médio da função hash. A função hash a ser usada é escolhida de maneira aleatória no início da execução, independentemente das chaves, a partir de um conjunto de funções hash projetadas.

Mesmo o hash universal não garante um comportamento ideal durante a execução. O método numérico descrito neste artigo permite que o projetista inclua na

análise as imperfeições da função hash empregada. Desta maneira, é possível antever o comportamento das listas da tabela hash mesmo quando a função hash não é perfeita, o que vem a ser a maioria dos casos. Serão considerados três tipos de funções:

- Função hash perfeitamente uniforme, todas as entradas possuem a mesma probabilidade de receber uma dada chave: $P(h = x) = 1/M$, $1 \leq x \leq M$.

- Função hash quase uniforme, metade das entradas recebem 2/3 das chaves, enquanto a outra metade das entradas recebe 1/3 das chaves, dentro de cada metade a distribuição é uniforme: $P(h=x) = 4/(3 \times M)$ caso $1 \leq x \leq M/2$; $P(h=x) = 2/(3 \times M)$ caso $M/2 < x \leq M$.

- Função não uniforme, 3 entradas recebem 30% das chaves, ao passo que as demais M-3 entradas da tabela recebem 70% das chaves, é suposto $M > 3$:
$P(h = 1) = 0.1$ $P(h = 2) = 0.1$ $P(h = 3) = 0.1$ $P(h = x) = (0.7/M)$, $4 \leq x \leq M$.

No caso da abordagem 2-choice, duas funções hash H1 e H2 são necessárias. Se as funções hash forem uniformes, então H1 é igual a H2. No caso de funções hash quase uniformes ou não uniformes, é preciso decidir se as entradas com maior probabilidade são as mesmas ou são diferentes em H1 e H2. Foram feitos os cálculos supondo as duas situações. As funções H1 e H2 são ditas não correlacionadas quando as entradas de maior probabilidade de cada uma forem diferentes. Elas são ditas correlacionadas quando as entradas de maior probabilidade foram as mesmas nas duas funções.

5.1. Resultados

Para uma tabela simples com função hash uniforme, o tamanho máximo relevante da lista encadeada chegou a 16 entradas, embora 750 inserções sejam feitas na tabela hash. As probabilidades calculadas para este caso foram:

[0 registros]0.47676059996147070471
[1 registros]0.35333048416116927176
[2 registros]0.13075322758731027045
[3 registros]0.03221456331861269999
[4 registros]0.00594473290489221967
[5 registros]0.00087643690653154137
[6 registros]0.00010753384311034233
[7 registros]0.00001129378589414100
[8 registros]0.00000103647269260709
[9 registros]8.443815743448E-8
[10 registros]6.18267536145E-9
[11 registros]4.1099351096E-10
[12 registros]2.501022747E-11
[13 registros]1.40297537E-12
[14 registros] 7.298059E-14
[15 registros] 3.53786E-15
[16 registros] 1.6000E-16
[17 registros]6.12E-18
[18 registros] 1E-20
[19 a 749 registros]1E-20
[750 registros] 1E-20
[751 registros] 0

Os resultados fundamentais dos cálculos realizados são apresentados na tabela 1. Quando a função hash utilizada não é perfeitamente uniforme, mas quase uniforme, nos termos definidos neste artigo, o tamanho máximo relevante para as listas encadeadas aumenta para apenas 17, ou seja, um registro a mais. Entretanto, ao ser utilizada uma função hash não uniforme, com uma elevada concentração de probabilidade em poucas entradas, o tamanho máximo relevante da lista encadeada aumenta para 151, quase dez vezes maior que o caso uniforme, usualmente considerado na literatura. Neste caso, a previsibilidade temporal do sistema ficaria seriamente comprometida, caso o sistema fosse dimensionado para uma função uniforme.

A qualidade de uma função hash está associada com a sua capacidade de fazer um espalhamento perfeito das chaves que ela recebe. Desta forma, uma função hash ideal distribui as chaves uniformemente entre as entradas da tabela. Na prática, funções hash apresentam um comportamento aquém do ideal. É necessário conviver com o fato da função utilizada não espalhar perfeitamente as chaves nas entradas da tabela.

A abordagem 2-Choice trás robustez ao sistema. Primeiramente, o tamanho máximo relevante das listas encadeadas cai de 16 para 3 se a função hash for perfeitamente uniforme. No caso de funções hash quase uniformes, sejam elas correlacionas ou não, o tamanho máximo relevante para uma lista encadeada permanece em 3. No caso da abordagem 2-choice, no pior caso é necessário percorrer duas listas encadeadas. Logo, um tamanho máximo relevante de 3 para as listas indica que será necessário percorrer 3+3=6 registros a cada operação na tabela. Ainda assim, este número é bem menor que os 16 ou 17 da tabela simples.

Tabela 1. Tamanho máximo relevante das listas encadeadas

Tabela	Tipo de Função		Tamanho máx. relevante	Registros a pesquisar
Simples	Uniforme		16	16
Simples	Quase Uniforme		17	17
Simples	**Não Uniforme**		**151**	**151**
2-choice	Uniforme		3	6
2-choice	Quase Uniforme	Não Correlacionada	3	6
2-choice	Quase Uniforme	Correlacionada	3	6
2-choice	Não Uniforme	Não Correlacionada	8	16
2-choice	**Não Uniforme**	**Correlacionada**	**10**	**20**

No caso de funções hash não uniformes, assim como no caso da tabela hash simples, o tamanho máximo relevante das listas encadeadas aumenta. Entretanto, mesmo no caso de funções hash correlacionadas, ele fica em 10, o que resulta em pesquisar 10+10=20 registros da tabela 2-choice, no lugar dos 151 registros da tabela hash simples. As probabilidades calculadas para o caso de tabela 2-choice com funções hash não uniformes e correlacionadas, considerando-se uma entrada de alta probabilidade, foram:

[0 registros] 0.00900853291867478015
[1 registros] 0.02367672240017487672
[2 registros] 0.04528701275583918460
[3 registros] 0.06499467606307381824
[4 registros] 0.08976225758159696449
[5 registros] 0.12795405064570978937
[6 registros] 0.19983082784151369890
[7 registros] 0.30209411092767075707
[8 registros]0.13556813347534509603
[9 registros]0.00182367409035382306
[10 registros]1.30004720397E-9
[11 registros]1E-20
[12 a 749 registros]1E-20
[750 registros]1E-20
[751]0

As experiências realizadas indicam claramente que é possível adotar uma abordagem probabilista para o comportamento das tabelas hash no pior caso. Embora

exista uma probabilidade não nula de uma mesma entrada da tabela hash receber todos os 750 registros, o que tornaria o processo de pesquisa muito lento, esta probabilidade não nula é tão pequena (menor que 10^{-20} no nosso caso) que pode ser considerada irrelevante. Tamanhos máximos de lista encadeada mais realistas podem ser usados, ainda com boa margem de segurança.

O maior risco desta abordagem são funções hash que apresentem comportamento patológico para o conjunto de registros observados na prática. Por exemplo, a concentração de 30% dos registros em apenas 3 entradas da tabela hash fez o tamanho máximo relevante de uma lista da tabela saltar de 16 ou 17 para 151 no caso da tabela hash simples. Neste caso, a tabela 2-choice representa uma segurança quanto a falhas de projeto da função hash. Na mesma situação, e considerando as suas duas funções hash correlacionadas, o tamanho máximo relevante ficou em 10, o que significaria uma pesquisa sobre 20 entradas (duas listas).

Foram necessárias cerca de 3,5 horas para computar todos os casos apresentados neste artigo, em um Pentium de 2GHz e 500Mbytes de memória principal, sendo a ferramenta programada em Java. O tempo de cálculo depende dos formatos da funções hash e do tipo de tabela. Por exemplo, o caso da tabela 2-choice com funções hash não uniformes correlacionadas demorou cerca de 40 minutos neste computador.

6. Considerações Finais

Este artigo discutiu o uso de tabelas hash em sistemas operacionais de tempo real, onde o tempo de resposta das tarefas é uma restrição importante e, por vezes, precisa ser garantido para aplicações críticas. Mostra-se que é possível definir uma probabilidade minimamente relevante tal que, para quaisquer fins práticos, um evento com probabilidade menor do que a probabilidade minimamente relevante pode ser considerado como um evento impossível.

O artigo descreve uma ferramenta de software construída para calcular numericamente a função massa de probabilidade do tamanho das listas encadeadas de uma tabela hash que resolve colisão através de encadeamento. Esta ferramenta emprega diversas técnicas para controlar a precisão na representação dos dados e para reduzir o tempo de cálculo. Foram consideradas tabelas hash simples e baseadas na abordagem 2-choice.

As experiências mostram que faz sentido adotar uma abordagem probabilista para o comportamento das tabelas hash no pior caso. Embora exista uma probabilidade não nula de uma mesma entrada da tabela hash receber todos os registros inseridos, esta probabilidade não nula é tão pequena que pode ser considerada irrelevante. Desta forma, tamanhos máximos de lista encadeada mais realistas podem ser usados, ainda com boa margem de segurança. Cabe ao projetista do software definir a probabilidade minimamente relevante do seu sistema.

Uma questão não abordada neste trabalho é o tempo para calcular uma segunda função hash, a cada operação sobre a tabela. Este é o preço pago pela abordagem 2-choice. Se o custo computacional da segunda função hash for elevado, ele poderá compensar o ganho da abordagem 2-choice no tempo de pesquisa. Entretanto, mesmo neste caso, pode ser vantajoso usar uma tabela 2-choice, em função da robustez que ela confere à tabela na ocorrência de funções hash com comportamento patológico.

Outra questão em aberto é a caracterização precisa da PMF das funções hash utilizadas, levando em consideração os dados reais da aplicação. Qualquer informação disponível sobre os valores de chaves a serem utilizados na prática pode permitir uma melhor caraterização das PMFs, levando a resultados mais precisos na análise numérica.

References

Azar, Y., Broder, A. Z., Karlin, A. R., Upfal, E. (1994) "Balanced Allocations", Proceedings of the 26th Annual ACM Symposium on the Theory of Computing (STOC 94), pages 593-602.

Carter, J. L. and Wegman, M. N. (1979) "Universal classes of hash functions", Journal of Computer and System Sciences, 18(2), pages 143–154.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. (1990) "Introduction to Algorithms", The MIT Press, Cambridge, United States.

Friedman, S., Krishnan, A., Leidenfrost, N., Brodie, B. C., Cytron, R. K., and Niehaus, D. (2003) "Hash tables for embedded and real-time systems", Technical Report 2003-15, Department of Computer Science & Engineering, Washington University in Saint Louis.

Gonnet, G. H. (1981) "Expected Length of the Longest Probe Sequence in Hash Code Searching", Journal of the ACM, 28(2), pages 289-304.

Liu, J. (2000) "Real-Time Systems", Prentice-Hall, United States.

Karp, R. M., Luby, M., Heide, F. M. (1996) "Efficient PRAM Simulation on a distributed memory Machine", Algorithmica, 16, pages 245-281.

Mitzenmacher, M., Richa, A., and Sitaraman, R. (2000) "The power of two random choices: A survey of the techniques and results", In Handbook of Randomized Computing, P. Pardalos, S. Rajasekaran, and J. Rolim, Eds. Kluwer.

Parson, D. (2004) "Incremental Reorganization of Open Hash Tables", Work-in-progress Section of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), May 25 - May 28, 2004.