X Simpósio Brasileiro de Arquitetura de Computadores

Aspectos do Comportamento Espaço-temporal de Programas Funcionais em Uni e Multiprocessadores

Francisco Vieira de Souza e *Rafael Dueire Lins*
Universidade Federal de Pernambuco - CCEN - DI
Av. Prof. Luis Freire, s/n - Caixa Postal 7851 - CEP 50.732-970
Cidade Universitária - Recife - Pe
e-mails: {*fvs,rdl*}*@di.ufpe.br*

Abstract

This paper analyses the performance of two of the most important techniques for automatic dynamic memory management in lazy functional languages. Our results show that, as garbage collection enforces data locality, it is cheaper to garbage collect than to avoid it by increasing heap size. We observed this behaviour both in uni and multiprocessors running on operating systems with virtual memory.

1 Introdução

O estudo empreendido neste artigo visa conhecer o comportamento dinâmico espaço-temporal de programas funcionais lazy representando um passo inicial necessário para a descoberta de novas técnicas de gerenciamento de memória [3] em compiladores funcionais. Além disso, os resultados obtidos desmistificam a crença, ainda existente, de que o gerenciamento dinâmico automático de memória representa um custo adicional e, por consequência, perda em desempenho.

A *nofib suite* [4] desenvolvida para analisar o desempenho do compilador Haskell [2] tem se mostrado adequada para análises de programas funcionais pela diversidade de operações envolvidas. Neste artigo, foram utilizados o programas de teste Hanói para 15 peças e sete outros programas da *nofib suite* (*pseudoknot, parafins, primes, queens, rfib, boyer, fish* e *mult*). O tamanho do código fonte variou de 8 a 3312 linhas, o código executável de 1MB a 9MB e o número de células alocadas na execução de 25 a 186 mega.

Os programas de teste foram executados variando-se os tamanhos da *heap* desde a *heap* mínima de execução (*heap* inicial) até a *heap* onde não se faz chamadas ao *Garbage Collector* (*heap* final). O coletores geracional e de cópia foram utilizados neste estudo. Os tempo de usuário e de sistema foram obtidos pelo comando **time** do sistema operacional Unix. Cada execução foi repetida, no mínimo 3 vezes, ou até que se obtivesse uma produtividade (tempo CPU/tempo decorrido) acima de 90%, eliminando a interferência de outros processos nos tempos obtidos. As execuções foram realizadas em três máquinas de arquitetura SPARC, sob o sistema operacional SunOS 5.5.1, conforme a tabela 1.

2 Chamadas aos coletores

Verifica-se que o gráfico de "*quantidade_de_coletas* × *tamanho_de_heap*" obedece a um padrão bem definido, tanto para as coletas geracionais quanto para as coletas por cópia.

Tabela 1: As máquinas empregadas nas execuções.

Nome	Arq.	modelo	RAM(MB)	VM(MB)	clock(MHz)	No. Proc.
zumbi	Sparc	Axil 320	256	285	90	2
ingazeira	Sparc	Axil 320	64	270	90	1
paulista	Sparc	Ultra 1	320	139	167	1

Para todos os programas, a quantidade de chamadas aos coletores é muito alta para *heap* muito pequenas, havendo um decréscimo muito acentuado a medida que o tamanho da *heap* é incrementado. Este comportamento está evidenciado na figura 1, onde a grande concentração das linhas no gráfico formam uma linha bastante densa evidenciando que todas elas têm o mesmo comportamento.

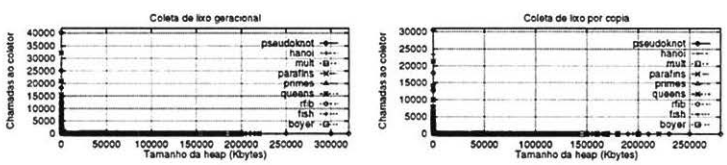

Figura 1: Resumo das coletas geracionais e de cópia.

Esse rápido decréscimo atinge um ponto, a partir do qual, o número de chamadas ao coletor diminui muito lentamente. Este ponto de inflexão se situa bem próximo à *heap* inicial.

3 Tempos de usuário, de sistema e total

Em relação ao tempo de CPU, para todos os programas, verificou-se a existência de um ponto de tempo mínimo em torno de um determinado tamanho de *heap* bem próximo, quando não coincidente, com o ponto de inflexão do gráfico de coletas de lixo.

Os tempos de sistema e total são bastante altos próximos à *heap* inicial havendo uma queda acentuada com pequenos incrementos no tamanho da *heap*. Ao atingir o ponto de mínimo, o tempo de sistema começa a crescer e, por consequência, o tempo de CPU. O crescimento a partir desse ponto se apresenta muito próximo do linear e permanece assim até atingir a *heap* máxima.

Esse comportamento se verificou tanto nas execuções com coletas geracionais quanto nas execuções com coletas por cópia. O gráfico das figuras 2 e 3 apresenta *pseudoknot* como representante do comportamento comum a todos os programas de teste.

Na execução de quase todos os programas na máquina *ingazeira* (Axil 320, Sun4m), verifica-se o surgimento de uma forma parecida com um "joelho". Nas execuções com coleta de lixo geracional esse "joelho" se encontrava em *heaps* com tamanhos em torno de

100MBytes e nas execuções com coleta de lixo por cópia ele foi verificado nas *heaps* com tamanhos em torno de 50MBytes. Essa forma significa um aumento rápido nos tempos de sistema para em seguida acompanhar a forma anterior, motivado pela utilização de toda a memória principal e passando a usar a memória virtual com acesso mais lento. Esse comportamento não foi verificado nos programas *parafins*, *primes* e *fish*, uma vez que suas *heaps* finais são muito menores que as outras e não chegam a 100MBytes. A RAM da máquina *ingazeira* é de 64MBytes, no entanto, uma parcela dessa memória é dedicada ao sistema operacional, ficando em torno de 50MBytes disponíveis para a execução de programas. Esse valor coincide com o tamanho da *heap* que apresenta o joelho nas execuções com coletas por cópia. O motivo dele aparecer em torno de 100MBytes nas execuções com coletas geracionais credita-se ao projeto do coletor geracional de Glasgow onde a memória inicial em uso é a metade da *heap* e, nesse caso, ainda está na memória principal, ficando os outros 50MBytes alocados na memória virtual.

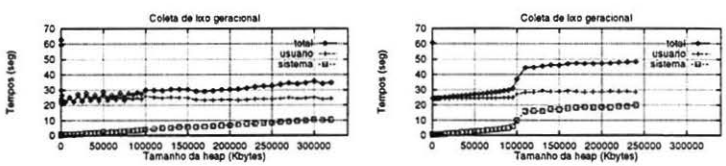

Figura 2: pseudoknot executado em zumbi e em ingazeira com coleta geracional.

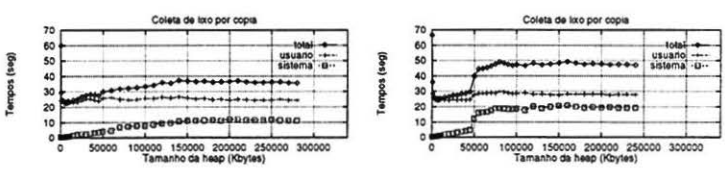

Figura 3: pseudoknot executado em zumbi e em ingazeira com coleta por cópia.

A figura 4 mostra todos os tempos de sistema plotados em um mesmo gráfico, onde pode-se verificar a concentração de linhas próximo à *heap* inicial.

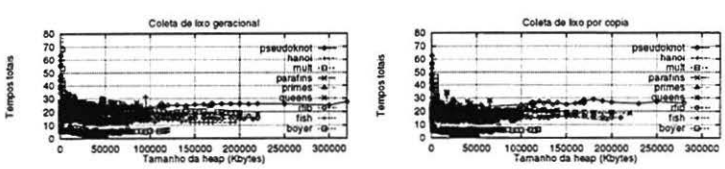

Figura 4: Resumo dos tempos de CPU em coletas geracionais e por cópia.

4 Monoprocessadores *vs* multiprocessadores

Verifica-se que a máquina *zumbi* (multiprocessada) proporcionou um aumento considerável de desempenho em relação à máquina *ingazeira* (monoprocessada). Nas execuções com coletas de lixo geracionais, o aumento de desempenho chegou a ser de 466% para o programa *boyer* (maior) e de 41% para o programa *parafins*. Em média, esse aumento foi de 156%.

Nas execuções com coletas de lixo por cópia, o aumento de desempenho verificado foi bem menor. O maior valor de melhoria verificado foi na execução do programa *mult* (82%) e o menor também foi na execução do programa *parafins* (36%). Em média, o aumento verificado foi de 60%, em contraposição ao aumento de 156% verificado nas coletas geracionais. Essas melhorias verificadas se devem ao fato de uma máquina ter um processador e a outra ter dois, uma vez que ambas têm a mesma arquitetura, modelo e *clock*.

Comparando os desempenhos das máquinas *zumbi* e *paulista*, verifica-se uma inversão nos valores dessas melhorias. Apesar das arquiteturas serem as mesmas, elas apresentam modelos diferentes e um *clock* bem maior em *paulista*. Essa diferença de *clock* proporcionou uma melhoria de desempenho em *paulista*, onde a média nas coletas geracionais foi de 378% e nas coletas por cópia foi de 564%.

5 Conclusões

Os resultados verificados neste trabalho mostram que, além de portabilidade, maior grau de abstração, modularidade e baixo custo de manutenção, o gerenciamento dinâmico de memória de forma automática representa um ganho em termos de tempo de sistema. O aumento da *heap* traz, como consequência, uma perda de localidade dos dados, ocasionando um aumento de *cache misses* e *page faults*, que acarreta degradação do tempo de execução do programa usuário. Maiores evidências experimentais poderão ser encontradas em [5].

Referências

[1] P. H. Hartel et al; *Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark.* J. Functional Programming 1 (10:1-000, Cambridge University Press. Jan. 1993.

[2] P. Hudak, S. L. Peyton Jones, and P. L. Wadler. *Report on the Programming Language Haskell: a Non-Strict, Purely Functional Language*, SIGPLAN Notices, 16(5), May 1992.

[3] R. E. Jones & R. D. Lins; *Garbage Collection : Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons; 1996.

[4] W. Partain; *The nofib Benchmark Suite of Haskell Programs.* University of Glasgow.

[5] F.V.Souza & R.D.Lins; *Analysing the Space Behaviour of Functional Programs.* in preparation.