DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Relatório Técnico

RT-MAC-9904

ESTUDO DE CASO: DESEMPENHO DEFICIENTE DO SISTEMA OPERACIONAL LINUX PARA CARGA MISTA DE APLICAÇÕES.

Jorge Euler Vieira, Maria do Carmo Noronha e
Dilma Menezes da Silva

Maio de 1999

Estudo de Caso: Desempenho Deficiente do Sistema Operacional Linux para Carga Mista de Aplicações

Jorge Euler Vieira, Maria do Carmo Noronha e Dilma Menezes da Silva

{euler,carmo,dilma}@ime.usp.br

Resumo

Este relatório enfoca a ineficácia dos sistemas operacionais em fornecer suporte apropriado para novas categorias de aplicações que envolvem aspectos de multimídia, computação gráfica e transmissão de vídeo. Apresentamos a utilização de CPU e o comportamento da aplicação *Controle de Aproximação de Aeronaves em Aeroportos* no ambiente Linux Debian Versão 2.0, sob as três políticas de escalonamento disponíveis neste sistema.

Abstract

This report focuses on the inefficacy of current operating systems in supporting new application categories involving aspects related to multimedia, graphics, and video transmission. We present the CPU utilization and external behavior of the application *Airport Approximation Control for Airplanes* running in the Linux Debian Version 2.0 environment, under the three scheduling policies available in this operating system.

1 Introdução

O grande avanço no desenvolvimento da tecnologia de hardware e a evolução do nível de complexidade das aplicações que esse pode suportar evidenciou a falta de estrutura dos sistemas operacionais existentes para gerenciar esta demanda. Esta inadequação pode ser observada principalmente na perspectiva da garantia de qualidade de serviço para aplicações que consomem bastante recursos da máquina, como aplicações multimídia, gráficas e de videoconferência.

A crescente onda de novas aplicações em multimídia e sua ascendente demanda evidenciou ainda mais a fragilidade dos sistemas operacionais para suportá-las. A falta de suporte dos sistemas operacionais e as características destas aplicações (como grandes consumidoras de recursos das máquinas) têm provocado a geração de aplicações de baixa qualidade. Tentativas de resolver o problema de suporte destas aplicações pelo sistema operacional, através da alteração da prioridade do processo (no *UNIX*, por exemplo, usando o comando *nice*) têm se mostrado ineficientes.

Nosso trabalho enfoca esta deficiência dos sistemas operacionais atuais nos aspectos de análise da utilização da *CPU* e do comportamento da aplicação utilizada. O *Controle de Aproximação de Aeronaves em Aeroportos* foi a aplicação escolhida. Este relatório apresenta os resultados obtidos com os testes de desempenho executados.

Os testes foram realizados em ambiente *LINUX Debian Versão 2.0*, e, para nossa avaliação, utilizamos os parâmetros tempo e iteração. No nosso caso, o termo *tempo* significa o intervalo do início ao final de uma iteração, sob o ponto de vista do usuário da aplicação, e não o tempo de uso de *CPU*.

A Sec. 2 descreve a aplicação principal utilizada nos testes. As especificações técnicas do ambiente utilizado são descritas na Sec. 3. Os testes mencionados são exibidos na Sec. 4 e as considerações finais na Sec. 5.

2 Descrição das Aplicações

A aplicação para *Controle de Aproximação de Aeronaves em Aeroportos* foi implementada em *Tcl/Tk* [5, 3]. *Tcl* é uma linguagem de *script* e *Tk* é um construtor de janelas gráficas, que trabalham em conjunto.

O objetivo da aplicação é simular, em um ambiente gráfico, a trajetória de um avião partindo de uma determinada cidade-origem até uma cidade-destino. A cada iteração, a posição da aeronave é impressa na tela juntamente com sua distância relativa a cada uma das cidades principais.

O tempo de captura do sinal pelo radar e de seu envio para o computador é de aproximadamente 500ms. A nossa implementação simula este tempo de captura de dados através de um bloqueio.

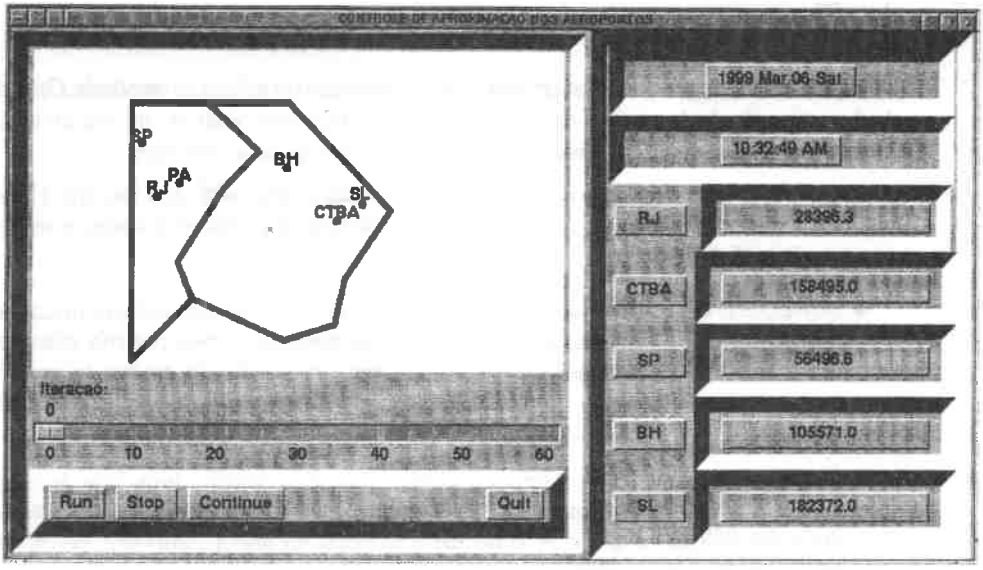

Figura 1: Tela da aplicação principal

A aplicação gera um mapa fictício com algumas cidades indicadas. A Fig. 1 retrata a tela principal da aplicação. Há um campo contendo a data e a hora atualizado em tempo-real. Uma barra com 60 iterações é mostrada para indicar o número de iterações realizadas. No rodapé da aplicação há 4 botões: *run*, *stop*, *continue* e *quit*. No momento em que o botão *run* é pressionado, inicia-se a execução. A partir de uma cidade estipulada e pré-definida, a trajetória do avião é indicada graficamente na tela. A função da trajetória da aeronave é simulada pela função $ln(x)$. A cada 4 (quatro) iterações, é calculada a distância da aeronave com as principais cidades e os campos relacionados são atualizados. Os valores da coordenada x e y, bem como o tempo gasto para computar a iteração, são armazenados em um arquivo. O botão *stop* pára a execução da aplicação e o *continue* reinicia a execução da aplicação a partir do ponto de parada. O botão *quit* aborta a aplicação em execução.

3 Ambiente

Utilizamos um *PC* Pentium 266MHz, 32MB de Memória primária e *SO LINUX Debian, Versão 2.0*, para os testes. A máquina foi desconectada da rede.

4 Testes

Os experimentos foram baseados em testes de desempenho da aplicação escolhida, *Controle de Aproximação de Aeronaves em Aeroportos*, e cobriram dois cenários: um em ambiente onde a carga do sistema era baixa e, outro, com a presença de sobrecarga.

Estes testes foram realizados colocando essa aplicação em cada uma das três classes de escalonamento do sistema operacional *LINUX*, as quais são definidas abaixo e seguem o padrão POSIX 1003.4:

- SCHED_FIFO - Nesta classe o escalonador escolhe o processo de mais alta prioridade para executar. O sistema nào interrompe a sua execução exceto por três situações: um outro processo da mesma classe e de maior prioridade esta na fila de prontos, o processo ficou suspenso na espera de um evento ou o processo chama a primitiva *sched_yield* e vai para a fila de prontos [2].
- SCHED_RR - esta classe e' similar a anterior com uma exceção: uma fatia de tempo é atribuída ao processo que está executando. Quando seu tempo acaba processos dos tipos SCHED_FIFO e SCHED_RR com igual ou maior prioridade podem ser selecionados e executados.
- SCHED_OTHER - os processos nesta classe somente podem ser executados quando nenhum processo das classes de tempo real estiverem na fila de prontos. A seleção do processo que irá executar é baseada na prioridade dinâmica.

As classes de escalonamento podem ser selecionadas através de chamadas ao sistema [1].

Para realizar o experimento mencionado, foi implementado um programa em linguagem *C*, entitulado `classe.c`, que solicita os valores desejados para o *pid* do processo e sua classe de escalonamento. Através das chamadas de sistema do sistema operacional *LINUX* mudou-se a classe de escalonamento do respectivo processo para classe RT FIFO (*Real-Time FIFO*) ou RT RR (*Real-Time Round Robin*).

Posteriormente, comparamos os resultados entre os dois cenários e, também, os resultados obtidos com o mesmo teste e a mesma carga, quando a aplicação principal pertencia a classes de escalonamento diferentes.

Para que os vários testes pudessem ser reapresentados com os mesmos dados da aplicação principal, a trajetória de uma aeronave foi simulada através da mesma função e do mesmo dado inicial, ou seja, uma mesma entrada em todos experimentos. Além disto, foi aplicado o mesmo número de iterações para todos os testes.

A aplicação mencionada é simples, sua função é apenas monitorar o percurso de um avião e, regularmente, informar a distância da aeronave até os principais aeroportos. É importante salientar que a aplicação é apenas um protótipo onde esta monitoração não é real.

O tempo gasto para completar cada iteração foi armazenado em um arquivo, de maneira que, além da comparação visual de cada experimento, tivemos vários conjuntos de dados para analisar. Ressaltamos que a definição de tempo utilizada como parâmetro das medidas é aquela exibida na Sec. 1.

Além da aplicação principal, foram utilizados quatro processos secundários que disputavam os recursos do sistema: `Bomba.c` que consumia fortemente os ciclos de *CPU*, `Bomba0.c` que consumia memória, `Isin.tcl` [4] que interagia com o *Servidor de Janelas* além de possuir as funções das classes anteriores e um processo de compilação do *emacs* versão 20.2.

Primeiramente, colocou-se a aplicação principal na classe *Real-Time FIFO* e os demais processos continuaram a ser executados pelo escalonador *timesharing*. Depois, aplicou-se os mesmos testes, colocando a aplicação principal na classe *Real-Time Round Robin* e os demais processos na classe *timesharing*. Foram utilizados os mesmos parâmetros (tempo e iteração) e houve a mesma coleta de dados. Comparou-se as funções anteriores onde a aplicação principal estava na classe *timesharing* com as funções da mesma estando nas classes de escalonamento *Real-Time* sendo mantidas as mesmas cargas para cada comparação.

As seções seguintes exibem os resultados obtidos com os testes descritos aplicados para as classes *timesharing* (TS), *Real-Time FIFO* (RT-FF) e *Real-Time Round Robin* (RT-RR). Ressaltamos que nos gráficos comparativos o intervalo entre as funções representa o *delay* registrado.

4.1 Resultados na Classe *Time-Sharing* (TS)

Nesta seção, a aplicação principal e os demais processos concorrentes em cada cenário dos testes pertencem a classe TS.

O gráfico (tempo versus iteração) da aplicação principal sem nenhum outro processo disputando os recursos do sistema é exibido na Fig. 2.

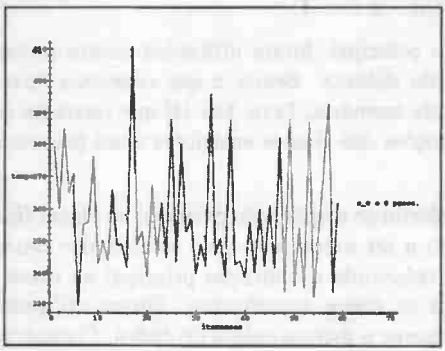

Figura 2: Gráfico (tempo versus iteração) da aplicação principal sem processos concorrentes.

Observou-se que há pequenos picos periódicos que é proveniente de um maior processamento da aplicação. Nestes momentos, além da aplicação atualizar, graficamente, a posição do avião que está sendo rastreado, ainda efetua atualizações de distâncias (em milhas) da posição atual do avião até os principais aeroportos. Detectou-se, claramente, que o sistema não ficou sobrecarregado e, repetindo este teste, foi possível notar que ao acionar o *mouse*, abrir novos *shell* ou entrar com novos comandos, o sistema teve um tempo de resposta muito bom e a aplicação continuava a executar normalmente. Não foram coletados os dados destes testes repetidos, sendo estas conclusões baseadas totalmente na impressão de tempo de resposta percebida pelo usuário.

A próxima figura apresenta um gráfico da função de comportamento da aplicação principal disputando os ciclos de *CPU* com um processo de compilação do *emacs*. Observou-se, visualmente, que o *mouse* e os processos iterativos tinham um bom tempo de resposta. Além disto, aparentemente, a aplicação não degradou.

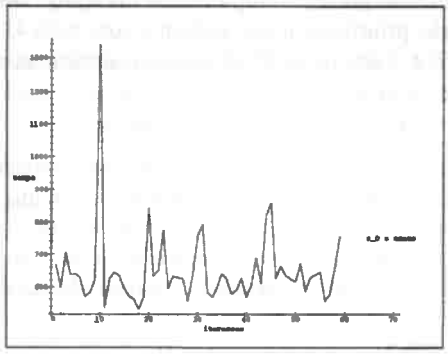

Figura 3: Gráfico (tempo vesus iteração) da aplicação principal com a compilação do emacs.

Adicionalmente, também foram coletados os dados comparativos do comportamento da aplicação nos dois cenários anteriores, os quais são apresentados na Fig. 4.

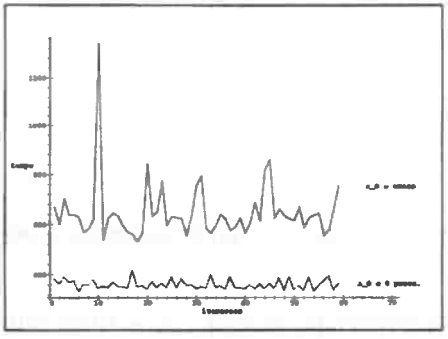

Figura 4: Gráfico comparativo da aplicação principal

A Fig. 5 apresenta outro gráfico (tempo versus iteração). Desta vez, exibindo o comportamento da aplicação principal em um ambiente com mais 41 processos. Todos disputando os ciclos de *CPU* e, deste total, 20 processos convencionais acessando endereços de memória, 20 processos executando atribuições simples e de acesso rápido e um processo gráfico com intensa iteração com o *Servidor de Janelas*.

Registrou-se uma demora maior na execução de cada iteração do processo principal. Visualmente foi verificado que os processos iterativos tinham um tempo de resposta ruim. O *mouse* levava um longo tempo para responder ao evento, ou seja, o sistema estava sobrecarregado. Nas iterações onde havia o processo de atualização das distâncias entre aeroportos, a aplicação principal praticamente parava, demonstrando uma situação de sobrecarga.

Não foi possível uma interrupção imediata dos processos concorrentes, quando o teste foi encerrado, devido a dificuldade do sistema em responder a mais outros processos.

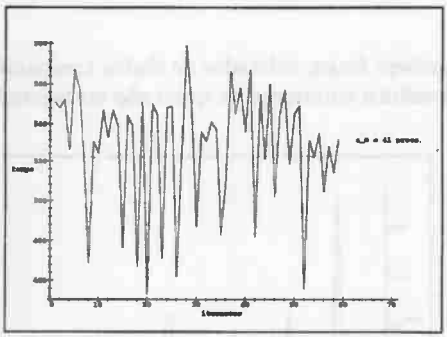

Figura 5: Gráfico (tempo versus iteração) da aplicação principal com 41 processos.

As Fig. 6 e 7 ainda referenciam análises comparativas entre cenários. Exibe-se na Fig. 6 o gráfico (tempo versus iteração) da aplicação principal sem nenhum processo concorrente e da mesma aplicação executando concorrentemente com 41 processos. Na Fig. 7 retrata-se comportamento da aplicação principal com 41 processos, bem como, desta com a compilação do *emacs*.

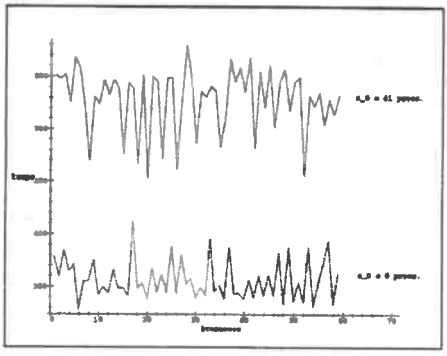

Figura 6: Gráfico (tempo versus iteração) da aplicação com 41 e 0 processos concorrentes.

Observa-se, a partir do gráfico da Fig. 7 que a função da aplicação principal levou mais tempo para executar cada iteração. Contudo, na prática, conseguimos neste cenário obter um melhor tempo de resposta para as aplicações iterativas.

Como o processo de compilação do *emacs* foi feito através do utilitário *make*, então computá-lo significou executar uma seqüência de comandos. Esta situação implicou em grande consumo de CPU e conseqüentemente o sistema operacional aumentou o valor da sua prioridade. De acordo com a política adotada pelo *LINUX*, onde: a prioridade de todos os processos que consomem bastante ciclos de CPU para executar diminui durante a computação [2].

Adicionalmente, os processos iterativos foram executados quando iniciados, o que justifica o melhor tempo de resposta para processos iterativos neste cenário.

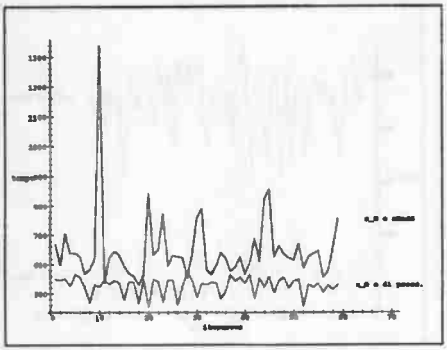

Figura 7: Gráfico (tempo versus iteração) da aplicação principal com 41 processos e a compilação do *emacs*.

O gráfico (tempo versus iteração) da aplicação principal executando concorrentemente com 10 processos gráficos é apresentado na Fig. 8. Verificou-se uma degradação na *performance* da aplicação principal. Contudo, as ações com o *mouse* e os processos iterativos tinham tempo de resposta muito bons.

Igualmente ao ocorrido com a compilação do *emacs* e segundo a política adotada pelo *LINUX*, como o *Servidor de Janelas* estava altamente requisitado, o sistema operacional aumentou o valor da sua prioridade.

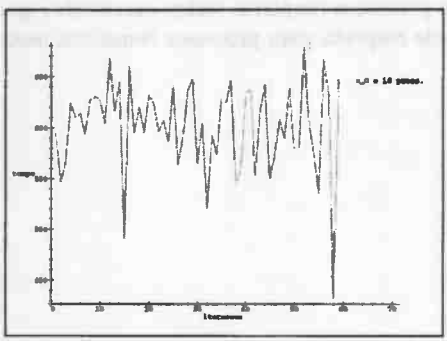

Figura 8: Gráfico (tempo versus iteração) da aplicação na classe com 10 processos gráficos.

Novamente, dados comparativos entre cenários foram coletados. Desta vez, as funções

de comportamento são da aplicação principal sem nenhum processo concorrente e da mesma aplicação executando concorrentemente com 10 processos gráficos. A Fig. 9 retrata esta situação e a Fig. 10 demonstra o comportamento da aplicação principal quando a mesma concorria com um processo de compilação do *emacs* e quando concorria com 10 processos gráficos.

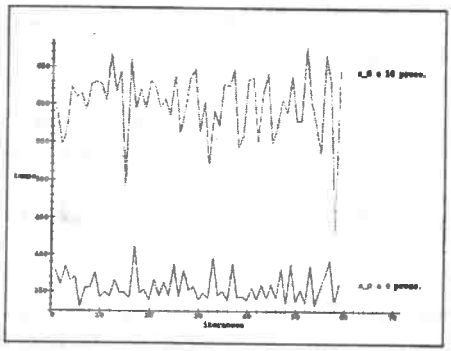

Figura 9: Gráfico (tempo versus iteração) da aplicação com 10 e zero processos.

Observou-se na Fig. 10 que, na média, consumiam tempos semelhantes em cada iteração mas na função onde a aplicação principal disputava os ciclos de *CPU* com o processo de compilação verificou-se um maior desvio padrão. O maior desvio padrão se justifica devido a própria característica do processo de compilação do *emacs*. Analisando, visualmente, verificamos um mesmo comportamento com relação aos processos iterativos, porém quando ao desempenho da aplicação principal, esta quando disputava os ciclos de *CPU* com o processo de compilação do *emacs* teve um desempenho melhor. Novamente, pela política adotada pelo sistema com relação a processos que consomem bastante CPU, o valor da prioridade do processo de compilação do *emacs* foi reduzido.

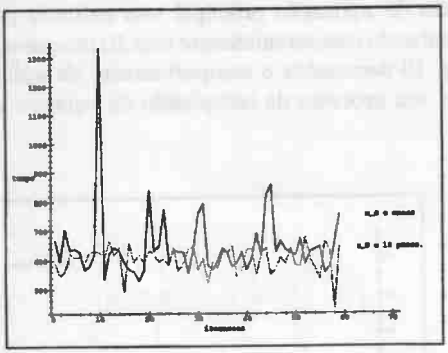

Figura 10: Função comparativa (tempo versus iteração) da aplicação principal

Na Fig. 11 a comparação é efetuada entre a aplicação principal computando com 41 e 10 processos, respectivamente. Verificou-se que, na computação com 10 processos, as iterações demoraram mais tempo para finalizar, provavelmente, devido a uma maior utilização da *CPU*. Apesar de observado um *delay* menor entre as duas funções de comportamento, o que pressupõe um tempo de resposta similar na iteração com o *mouse*, na prática, este tempo foi bem melhor com 10 processos.

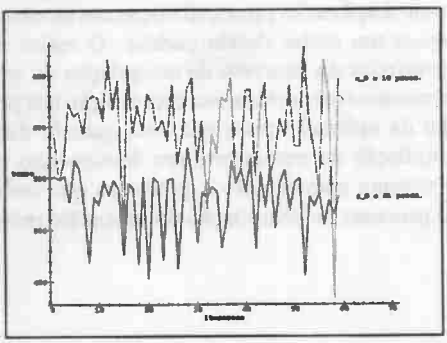

Figura 11: Gráfico (tempo versus iteração) da aplicação principal com 10 e 41 processos.

Os mesmos testes foram realizados colocando-se a aplicação principal na classe de escalonamento *Real-Time FIFO* (RT-FF) e os resultados são apresentados na próxima seção.

4.2 Resultados na Classe *Real-Time FIFO* (RT-FF)

Esta seção exibe os mesmos testes anteriores apenas alterando a classe da aplicação principal para RT-FF.

A Fig. 12 exibe o gráfico (tempo versus iteração) da aplicação principal sem outros processo disputando os recursos do sistema. Embora, graficamente, a função apresentada na Fig. 2 da Sec. 4.1 seja diferente da Fig. 12, visualmente, as funções tiveram comportamento semelhantes. Provavelmente, devido a baixa carga do sistema.

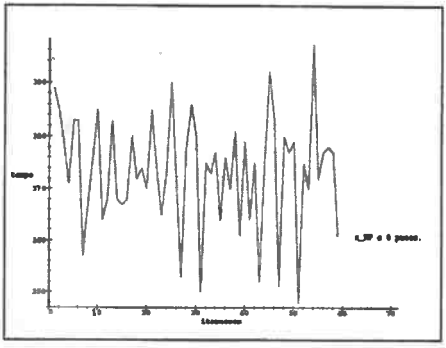

Figura 12: Gráfico (tempo versus iteração) da aplicação principal.

O comportamento da aplicação principal disputando os ciclos de *CPU* com um processo de compilação do *emacs* é apresentado na Fig. 13. Comparando com a Fig. 3 da Sec. 4.1, observamos que, graficamente, são distintas, mas, ambas retratam um desempenho aceitável, visto que o tempo de resposta foi bom.

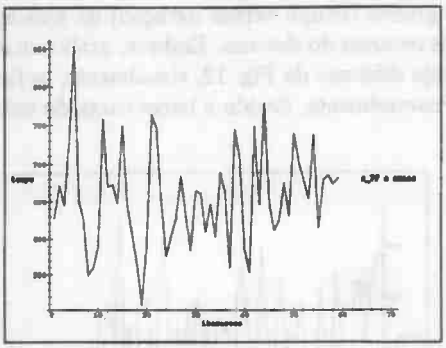

Figura 13: Gráfico (tempo versus iteração) da nossa aplicação e a compilação do *emacs*.

Como na seção anterior, registramos na Fig. 14 o comportamento da aplicação principal nos dois cenários anteriores.

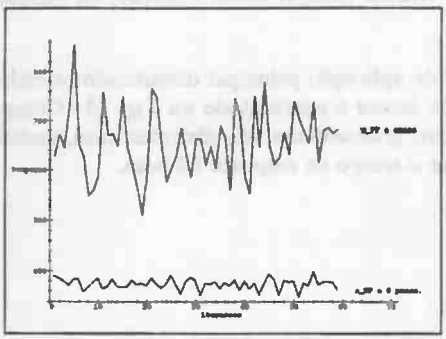

Figura 14: Gráfico (tempo versus iteração) da aplicação principal com 0 processos e compilando o *emacs*.

A Fig. 15 apresenta o gráfico (tempo versus iteração) do comportamento da aplicação principal com 41 processos concorrentes. Visualmente, não se observou nenhuma melhora significativa em relação ao mesmo teste efetuado com todas as aplicações na classe de escalonamento TS.

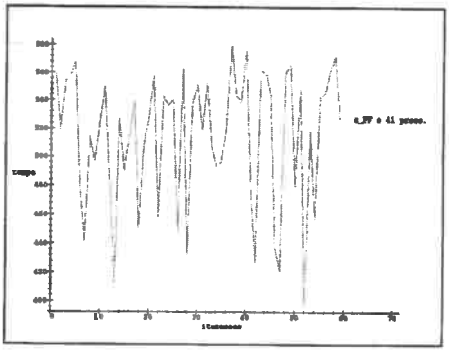

Figura 15: Gráfico (tempo versus iteração) da aplicação principal com 41 processos.

Assim como na classe TS, discutida na Sec. 4.1, exibimos, a seguir, uma análise comparativa do comportamento da aplicação principal executando sem processos concorrentes e da mesma com 41 processos concorrentes.

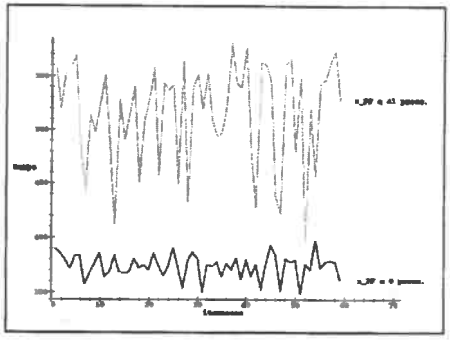

Figura 16: Gráfico (tempo versus iteração) da aplicação principal com 0 e 41 processos.

A Fig. 17 apresenta o gráfico (tempo versus iteração) do comportamento da aplicação principal pertencendo a classe RT-FF e executando concorrentemente com 10 processos gráficos. Observou-se que as ações com o *mouse* e os processos iterativos tinham um tempo de resposta equivalente àquele do mesmo teste com a aplicação principal na classe TS.

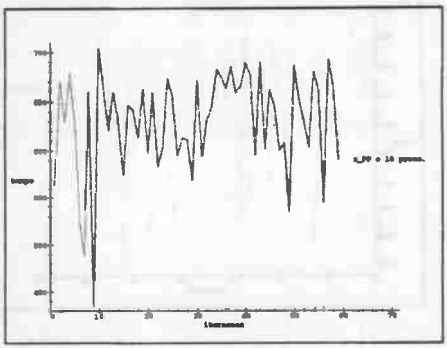

Figura 17: Gráfico (tempo versus iteração) da aplicação principal com 10 processos.

Uma comparação entre a aplicação principal sem nenhum processo concorrente e da mesma aplicação executando concorrentemente com 10 processos gráficos é exibida na Fig. 18.

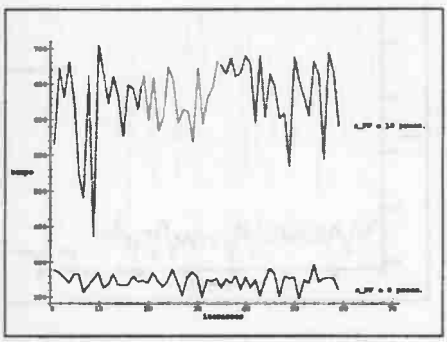

Figura 18: Gráfico (tempo versus iteração) da aplicação principal com 0 e 10 procs.

Ainda sob a perspectiva de uma análise comparativa entre cenários, apresenta-sena Fig. 19 o comportamento da aplicação principal executando com 41 e 10 processos concorrentes.

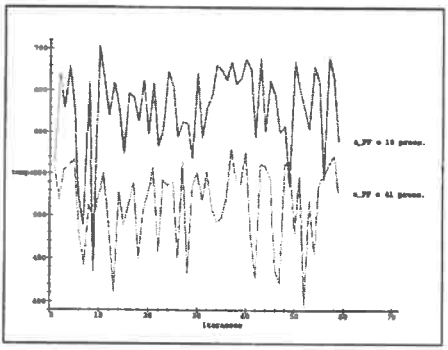

Figura 19: Gráfico (tempo versus iteração) da aplicação principal com 10 e 41 processos.

Os experimentos finais são apresentados na próxima seção.

4.3 Resultados na Classe *Real-Time Round Robin* (RT-RR)

Esta seção exibe os mesmos testes anteriores, apenas alterando a classe da aplicação principal para *Real-Time Round Robin* (RT-RR).

Abreviando o relato dos experimentos realizados, visto que as seções anteriores já retrataram detalhadamente os cenários usados, e os resultados obtidos não apresentaram diferenças significativas, apenas enumeraremos as figuras e suas respectivas etapas de execução.

Execução apenas da aplicação principal

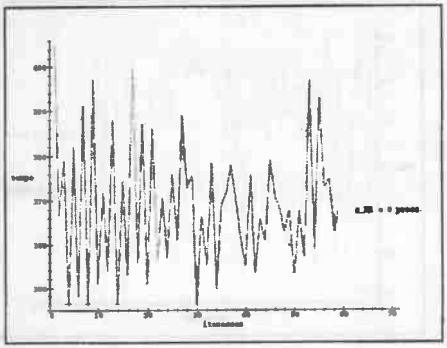

Figura 20: Gráfico (tempo versus iteração).

Execução da aplicação principal com a compilação do *emacs*

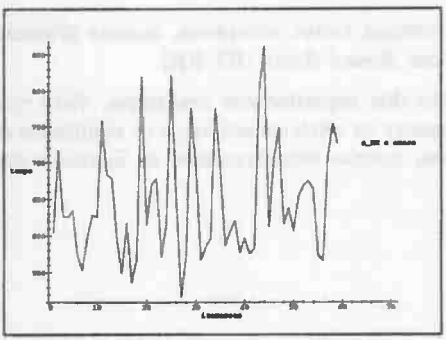

Figura 21: Gráfico (tempo versus iteração).

Comparação das duas etapas anteriores

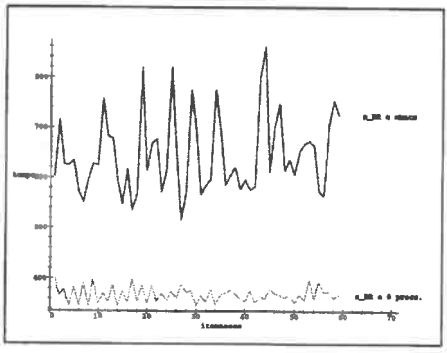

Figura 22: Gráfico (tempo versus iteração).

Execução da aplicação principal com 41 processos concorrentes

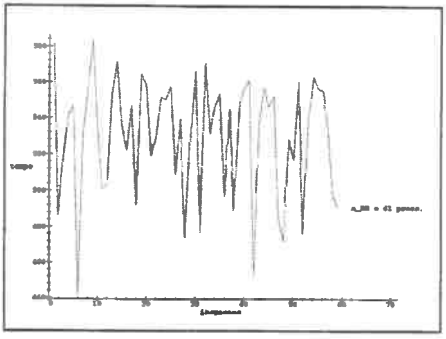

Figura 23: Gráfico (tempo versus iteração).

Comparação da etapa anterior com a execução da aplicação principal

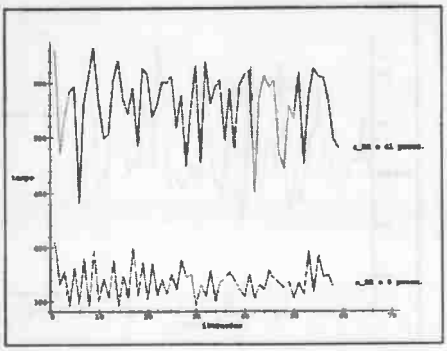

Figura 24: Gráfico (tempo versus iteração).

Execução da aplicação principal com 10 processos concorrentes

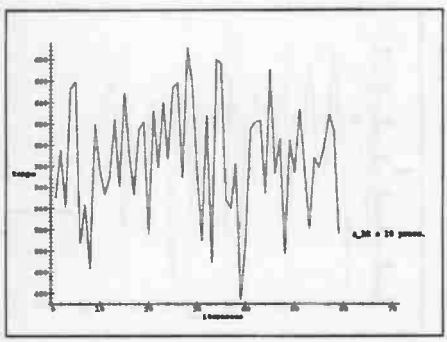

Figura 25: Gráfico (tempo versus iteração).

Comparação da etapa anterior com a execução da aplicação principal

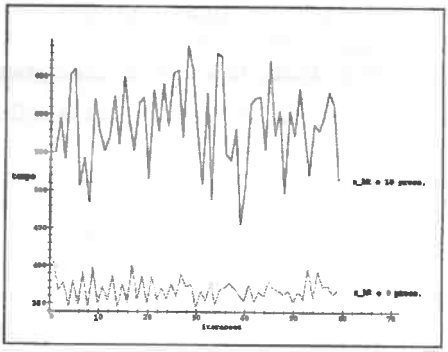

Figura 26: Gráfico (tempo versus iteração).

Comparação entre a aplicação com os 10 processos e a compilação do *emacs*

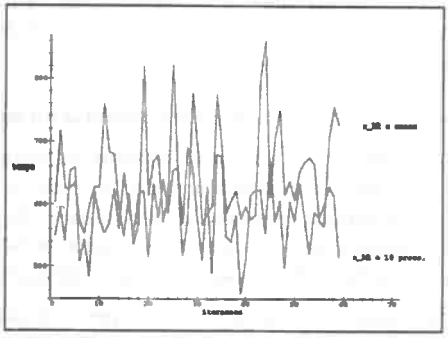

Figura 27: Gráfico (tempo versus iteração).

A seguir, apresentamos nossas conclusões sobre os experimentos relatados.

5 Considerações Finais

Baseados nos testes efetuados, observamos que o sistema operacional *Linux* não oferece suporte apropriado para as aplicações estudadas. Não havia como informar o sistema sobre a importância da nossa aplicação *Controle de Aproximação de Aeronaves nos Aeroportos*, de maneira que na disputa fosse dada a mesma uma maior prioridade.

Além disto, não há como o sistema dar *feedback* para as aplicações e informá-las sobre a carga do sistema. Em conseqüência, a aplicação não podia se adaptar, tanto atualizando as distâncias em 8 iterações ao invés de 4, como diminuindo a qualidade de imagem, o que degradou a sua qualidade de serviço. Neste contexto, quando o sistema entrava em sobrecarga, todas as aplicações degradavam, o sistema demorava em responder às ações do *mouse* e havia grande dificuldade de executar comandos iterativos. A aplicação principal, principalmente nos períodos de cálculo das distâncias entre os outros aeroportos, chegava a " congelar " por alguns segundos.

Outro aspecto observado foi o limitado gerenciamento, por parte do usuário, sobre o controle de recursos do sistema. Não havia muito que fazer para evitar que a aplicação principal diminuísse o seu desempenho, além de interromper alguns processos, ou bloquear alguns outros, quando o sistema estava em degradação.

Na classe de escalonamento *timesharing* detectou-se apenas os problemas mencionados acima, nas demais classes, outras situações merecem destaque.

Esperava-se que o desempenho da aplicação, quando colocada na classe de tempo real sozinha fosse muito melhor do que quando disputando na mesma classe *timesharing*. Entretanto, isto não aconteceu. Provavelmente porque a aplicação ficava bloqueada por um período para simular a chegada de dados do radar. Em conseqüência, outros processos ocupavam a CPU e quando este processo estava pronto para rodar tinha que esperar o processador ser liberado. Naturalmente, a aplicação sofria atrasos.

O outro motivo e' que mesmo a aplicação estando em uma classe de tempo real, o servidor de janelas estava na classe *timesharing* e no caso de uma sobrecarga estaria disputando a CPU com os demais processos de sua classe. Se houvesse mais processos gráficos, a situação ficaria ainda pior porque o escalonador veria que o servidor de janelas estava gastando todo seu *quantum* e diminuiria a sua prioridade.

Ao colocarmos a aplicação na classe de escalonamento *Real-Time FIFO*, observamos uma pequena melhora na aquisição de recursos pela aplicação. Contudo, se, em paralelo com a aplicação principal, fosse executada uma aplicação consumidora de grande parte dos recursos da máquina com mesma prioridade, initerruptamente e por muito tempo, quando nossa aplicação entrasse no estado bloqueado, devido a espera das informações provenientes do radar, perderia o processador para a outra aplicação até que esta finalizasse sua execução. Isto porque ambas tinham a mesma prioridade, e estavam na classe de escalonamento *Real-Time FIFO*. Este cenário mostra uma das possíveis situações que retrata inadequação desta solução para esta classe de aplicações.

É possível construir um cenário similar para a classe *Real-Time Round Robin*, basta, na situação descrita acima, introduzir o novo processo com prioridade maior do que aquela da aplicação principal.

Também observamos que o tempo no cenário com 41 processos foi menor que aquele com 10, em todas as classes. Este resultado não era esperado e para efetuar uma análise melhor deve-se verificar os serviços que estes requisitam. Nesta perspectiva, o que diferencia os dois e' que no cenário com 10 processos solicita-se muito mais trabalho para o servidor de janelas e em função disto a prioridade deste e' cada vez mais baixa. Enquanto no ambiente com 41 processos o servidor de janelas não fica sobrecarregado. Acreditamos que esta seja uma razão plausível, embora uma melhor ferramenta de análise seja necessária. É importante observar, porém, que os processos iterativos tinham muito melhor tempo de resposta no ambiente com 10 processos, um comportamento esperado.

Referências

[1] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *"Linux Kernel Internals"*. Addison-Wesley, second edition, 1998.

[2] R. Card, É. Dumas, and F. Mével. *"The Linux Kernel Book"*. John Wiley, 1997.

[3] Eric Foster-Johnson. *"Graphical Applications with Tcl and Tk"*. Division of Henry Holt and Company, Inc, second edition, 1997.

[4] Myunggyu Kim. Ising model simulator. http://hana.etri.re.kr/ mgkim/tcltk.html.

[5] John K. Ousterhout. *"Tcl and the Tk Toolkit"*. Addilson-Wesley Publishing Company, first edition, 1994.

RELATÓRIOS TÉCNICOS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Instituto de Matemática e Estatística da USP

A listagem contendo os relatórios técnicos anteriores a 1996 poderá ser consultada ou solicitada à Secretaria do Departamento, pessoalmente, por carta ou e-mail(mac@ime.usp.br).

Daniela V. Carbogim and Flávio S. Corrêa da Silva
FACTS, ANNOTATIONS, ARGUMENTS AND REASONING
RT-MAC-9601, janeiro de 1996, 22 pp.

Kunio Okuda
REDUÇÃO DE DEPENDÊNCIA PARCIAL E REDUÇÃO DE DEPENDÊNCIA GENERALIZADA
RT-MAC-9602, fevereiro de 1996, 20 pp.

Junior Barrera, Edward R. Dougherty and Nina Sumiko Tomita
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES BY DESIGN OF STATISTICALLY OPTIMAL OPERATORS IN THE CONTEXT OF COMPUTATIONAL LEARNING THEORY.
RT-MAC-9603, abril de 1996, 48 pp.

Junior Barrera e Guillermo Pablo Salas
SET OPERATIONS ON CLOSED INTERVALS AND THEIR APPLICATIONS TO THE AUTOMATIC PROGRAMMINIG OF MMACH'S
RT-MAC-9604, abril de 1995, 66 pp.

Kunio Okuda
CYCLE SHRINKING BY DEPENDENCE REDUCTION
RT-MAC-9605, maio de 1996, 25 pp.

Julio Stern, Fabio Nakano e Marcelo Lauretto
REAL: REAL ATTRIBUTE LEARNING FOR STRATEGIC MARKET OPERATION
RT-MAC-9606, agosto de 1996, 16 pp.

Markus Endler
SISTEMAS OPERACIONAIS DISTRIBUÍDOS: CONCEITOS, EXEMPLOS E TENDÊNCIAS
RT-MAC-9607, agosto de 1996, 120 pp.

Hae Yong Kim
CONSTRUÇÃO RÁPIDA E AUTOMÁTICA DE OPERADORES MORFOLÓGICOS E EFICIENTES PELA APRENDIZAGEM COMPUTACIONAL
RT-MAC-9608, outubro de 1996, 19 pp.

Marcelo Finger
NOTES ON COMPLEX COMBINATORS AND STRUCTURALLY-FREE THEOREM PROVING
RT-MAC-9609, dezembro 1996, 28 pp.

Carlos Eduardo Ferreira, Flávio Keidi Miyazawa e Yoshiko Wakabayashi (eds)
ANAIS DA I OFICINA NACIONAL EM PROBLEMAS DE CORTE E EMPACOTAMENTO
RT-MAC-9610, dezembro de 1996, 65 pp.

Carlos Eduardo Ferreira, C. C. de Souza e Yoshiko Wakabayashi
REARRANGEMENT OF DNA FRAGMENTS: A BRANCH-AND-CUT ALGORITHM
RT-MAC-9701, janeiro de 1997, 24 pp.

Marcelo Finger
NOTES ON THE LOGICAL RECONSTRUCTION OF TEMPORAL DATABASES
RT-MAC-9702, março de 1997, 36 pp.

Flávio S. Corrêa da Silva, Wamberto W. Vasconcelos e David Robertson
COOPERATION BETWEEN KNOWLEDGE BASED SYSTEMS
RT-MAC-9703, abril de 1997, 18 pp.

Junior Barrera, Gerald Jean Francis Banon, Roberto de Alencar Lotufo, Roberto Hirata Junior
MMACH: A MATHEMATICAL MORPHOLOGY TOOLBOX FOR THE KHOROS SYSTEM
RT-MAC-9704, maio de 1997, 67 pp.

Julio Michael Stern e Cibele Dunder
PORTFÓLIOS EFICIENTES INCLUINDO OPÇÕES
RT-MAC-9705, maio de 1997, 29 pp.

Junior Barrera e Ronaldo Fumio Hashimoto
COMPACT REPRESENTATION OF W-OPERATORS
RT-MAC-9706, julho de 1997, 13 pp.

Dilma M. Silva e Markus Endler
CONFIGURAÇÃO DINÂMICA DE SISTEMAS
RT-MAC-9707, agosto de 1997, 35 pp

Kenji Koyama e Routo Terada
AN AUGMENTED FAMILY OF CRYPTOGRAPHIC PARITY CIRCUITS
RT-MAC-9708, setembro de 1997, 15 pp

Routo Terada e Jorge Nakahara Jr.
LINEAR AND DIFFERENTIAL CRYPTANALYSIS OF FEAL-N WITH SWAPPING
RT-MAC-9709, setembro de 1997, 16 pp

Flávio S. Corrêa da Silva e Yara M. Michelacci
MAKING OF AN INTELLIGENT TUTORING SYSTEM (OR METHODOLOGICAL ISSUES OF ARTIFICIAL INTELLIGENCE RESEARCH BY EXAMPLE)
RT-MAC-9710, outubro de 1997, 16 pp.

Marcelo Finger
COMPUTING LIST COMBINATOR SOLUTIONS FOR STRUCTURAL EQUATIONS
RT-MAC-9711, outubro de 1997, 22 pp.

Maria Angela Gurgel and E.M.Rodrigues
THE F-FACTOR PROBLEM
RT-MAC-9712, dezembro de 1997, 22 pp.

Perry R. James, Markus Endler, Marie-Claude Gaudel
DEVELOPMENT OF AN ATOMIC-BROADCAST PROTOCOL USING LOTOS
RT-MAC-9713, dezembro de 1997, 27 pp.

Carlos Eduardo Ferreira and Marko Loparic
A BRANCH-AND-CUT ALGORITHM FOR A VEHICLE ROUTING PROBLEM WITH CAPACITY AND TIME CONSTRAINTS
RT-MAC-9714, dezembro de 1997, 20 pp.

Nami Kobayashi
A HIERARCHY FOR THE RECOGNIZABLE M-SUBSETS
RT-MAC-9715, dezembro de 1997, 47 pp.

Flávio Soares Corrêa da Silva e Daniela Vasconcelos Carbogim
A TWO-SORTED INTERPRETATION FOR ANNOTATED LOGIC
RT-MAC-9801, fevereiro de 1998, 17 pp.

Flávio Soares Corrêa da Silva, Wamberto Weber Vasconcelos, Jaume Agustí, David Robertson e Ana Cristina V. de Melo.
WHY ONTOLOGIES ARE NOT ENOUGH FOR KNOWLEDGE SHARING
RT-MAC-9802, outubro de 1998, 15 pp.

J. C.de Pina e J. Soares
ON THE INTEGER CONE OF THE BASES OF A MATROID
RT-MAC-9803, novembro de 1998, 16 pp.

K. Okuda and S.W.Song
REVISITING HAMILTONIAM DECOMPOSITION OF THE HYPERCUBE
RT-MAC-9804, dezembro 1998, 17pp.

Markus Endler
AGENTES MÓVEIS: UM TUTORIAL
RT-MAC-9805, dezembro 1998, 19pp.

Carlos Alberto de Bragança Pereira, Fabio Nakano e Julio Michael Stern
A DYNAMIC SOFTWARE CERTIFICATION AN VERIFICATION PROCEDURE
RT-MAC-9901, março 1999, 21pp.

Carlos E. Ferreira e Dilma M. Silva
BCC DA USP: UM NOVO CURSO PARA OS DESAFIOS DO NOVO MILÊNIO
RT-MAC-9902, abril 1999, 12pp.

Ronaldo Fumio Hashimoto and Junior Barrera
A SIMPLE ALGORITHM FOR DECOMPOSING CONVEX STRUCTURING ELEMENTS
RT-MAC-9903, abril 1999, 24 pp.

Jorge Euler, Maria do Carmo Noronha e Dilma Menezes da Silva
ESTUDO DE CASO: DESEMPENHO DEFICIENTE DO SISTEMA OPERACIONAL LINUX PARA CARGA MISTA DE APLICAÇÕES.
RT-MAC-9904, maio 1999, 27 pp.