Nelson Posse Lago

Processamento distribuído de áudio em tempo real

Proposta de dissertação apresentada ao Instituto de Matemática e Estatística da Universidade de São Paulo como requisito parcial para qualificação de mestrado em ciência da computação.

Orientador: Prof. Dr. Fabio Kon

São Paulo, Agosto de 2002

Sumário

Resumo

Este projeto pretende pesquisar e implementar, em ambiente Linux, mecanismos para o processamento síncrono, distribuído e com baixas latências de áudio em redes locais, viabilizando o processamento paralelo do áudio em equipamentos de custo relativamente baixo. O objetivo primário é viabilizar o uso de sistemas computacionais distribuídos para a gravação e edição de material musical em estúdios domésticos ou de pequeno porte, mas espera-se que esses mecanismos possam ser expandidos para outros tipos de mídia (como vídeo), bem como para áreas não-relacionadas à multimídia que dependam de processamento distribuído sincronizado com baixa latência.

1 introdução

O uso do computador na produção musical já há muito tempo deixou de ser exclusividade dos centros de pesquisa na área: embora continuem existindo amplos progressos graças à pesquisa nos campos da tecnologia e da composição eletroacústica, o computador tem penetrado cada vez mais em diversas áreas da criação, produção e gravação musical.

No nível individual, o computador tem sido utilizado como ferramenta de apoio à composição por diversos músicos[1], mesmo alguns voltados à produção de música estritamente acústica, por permitir a experimentação direta e imediata de diversos elementos musicais que não poderiam ser executados pelo compositor sozinho ao piano, por exemplo. Por outro lado, um estúdio doméstico centrado em um computador e uma pequena quantidade de equipamentos de áudio permite a produção e a gravação de materiais com ótima qualidade sonora, elevando em muito o nível das gravações "demo" feitas nesse tipo de ambiente em relação ao que era possível dez anos atrás. De fato, para muitas aplicações (como trilhas sonoras para vídeo ou materiais musicais com poucos instrumentos acústicos, por exemplo), a qualidade oferecida por um estúdio doméstico desse tipo é suficiente também para a criação do produto final.

Também em estúdios de gravação profissionais, sejam de pequeno, médio ou grande porte, o computador já é uma das ferramentas de trabalho cotidianas, e muitas vezes central: operações de edição, que antigamente dependiam de manipulações diretas das fitas magnéticas de gravação, são rotineiramente realizadas através do computador; diversos tipos de correção do sinal gravado (como a eliminação de distorções pontuais ou a adequação do nível de gravação, por exemplo) podem ser executadas rapidamente e praticamente sem degradação do sinal; e a paleta de efeitos disponíveis para processamento de áudio é facilmente expansível através da instalação de novos pacotes de software, dispensando uma boa parte dos dispositivos de hardware dedicados a tarefas específicas comumente usados há poucos anos.

Essa popularização se deu, basicamente, através de aplicativos disponíveis para as plataformas MacOS e Windows (tais como Pro Tools, Logic Audio, Cakewalk, Soundforge etc.) mas, com o crescente interesse pelo sistema operacional Linux, este também começa a contar com diversos aplicativos

[1] Essa tendência é evidenciada pela quantidade de produtos no mercado voltados para esse público, que transcende os pesquisadores da área; é claramente perceptível o crescente espaço dedicado a esses produtos tanto nas lojas de instrumentos musicais como em diversas publicações voltadas para músicos em geral.

voltados à produção e pesquisa musicais[2]. Uma parte dessas ferramentas tem origem em sistemas experimentais desenvolvidos originariamente para plataformas baseadas em UNIX (como o Ceres[3], MixViews[4] e outros), mas há diversos esforços recentes voltados especificamente para o Linux (como o Ardour[5], Muse[6], Gstreamer[7] e outros).

Uma aspecto fundamental para a operação de computadores nas aplicações descritas acima (e em muitas outras na área de multimídia) é a operação em tempo real: a possibilidade de interagir diretamente com o computador durante a produção ou reprodução de um som, alterando suas características, além de simplificar o uso dos sistemas computacionais, abre as portas para novos mecanismos para a composição, execução e gravação da música.

No entanto, o processamento em tempo real demanda uma alta velocidade por parte do computador. Assim, com vistas a um aumento de desempenho em sistemas desse tipo, pretendemos fazer neste trabalho um breve levantamento sobre as tendências importantes para o desenvolvimento de softwares na área de áudio (modularização, interfaces padronizadas e processamento baseado em *callbacks*) e apontar caminhos para a implementação de mecanismos para a paralelização do processamento do áudio em uma rede local em tempo real de maneira coerente com essas tendências de desenvolvimento.

[2] PHILLIPS, D. *Linux music & sound*. São Francisco: No starch press, 2000. O autor também mantém uma lista bastante ampla de aplicações voltadas para áudio e música compatíveis com o Linux, disponível em <`http://www.sound.condorow.net`> Acesso em: 18 ago. 2002.

[3] <`http://www.notam02.no/arkiv/src/`> Acesso em: 18 ago. 2002.

[4] <`http://www.ccmrc.ucsb.edu/~doug/htmls/MiXViews.html`> Acesso em: 18 ago. 2002.

[5] <`http://ardour.sf.net/`> Acesso em: 18 ago. 2002.

[6] <`http://muse.seh.de/`> Acesso em: 18 ago. 2002.

[7] <`http://gstreamer.net/`> Acesso em: 18 ago. 2002.

2 infra-estrutura de desenvolvimento

Diversas técnicas têm se consolidado recentemente no desenvolvimento de aplicativos voltados para o processamento de áudio (em particular entre os aplicativos citados acima ou outros para as plataformas Windows e MacOS, que têm tido bastante espaço na área de multimídia): seja por tendências que têm influenciado o desenvolvimento de software de maneira geral, seja por características específicas desse processamento, ou seja por pressões do mercado, o fato é que existem hoje padrões típicos de desenvolvimento de aplicativos desse tipo. Esses padrões, por sua vez, se refletem em especificações criadas para permitir uma maior integração entre produtos de diferentes fabricantes ou para conduzir a uma maior modularidade no código.

2.1 modularização

Uma tendência que tem se consolidado recentemente nos aplicativos desse tipo é a modularização: de fato, diversas especificações padronizadas para a criação de módulos (*plugins*) para o processamento de áudio existem, e aplicações complexas que possuam suporte a essas especificações, como gravadores ou sequenciadores, podem se beneficiar dos novos módulos que têm sido criados regularmente. Essa tendência segue um movimento geral em direção ao desenvolvimento baseado em componentes, que possibilita a criação de aplicações mais flexíveis e expansíveis[8].

Nos ambientes Windows e MacOS, as especificações mais comuns para a criação de módulos são a VST e as especificações desenvolvidas para os produtos da família *Pro Tools* (AudioSuite, RTAS, TDM e HTDM)[9]:

- VST (***V****irtual* ***S****tudio* ***T****echnology*): consiste em uma especificação e um pacote de desenvolvimento (SDK - ***S****oftware* ***D****evelopment* ***K****it)* desenvolvidos pela empresa Steinberg[10] para a criação de módulos que encapsulam algoritmos de DSP para áudio (como sintetizadores, processadores de efeitos como reverberação ou distorção etc.). Módulos escritos de acordo com esta especificação são carregados como bibliotecas dinâmicas pelos aplicativos e são executados pelo

[8]Para uma visão geral sobre componentes, consulte Szyperski, C. *Component Software*: Beyond Object-Oriented Programming. Addison-Wesley, 1997.

[9]Um breve resumo sobre os padrões desenvolvidos para os sistemas *Pro Tools* (que não cita o mais novo deles, o HTDM) pode ser encontrado em Digidesign. *Digidesign Plug-Ins Guide - for Macintosh and Windows*. Disponível em: <`http://www.digidesign.com/support/docs/Digidesign_Plug-Ins_Guide.pdf`>.

[10]O sítio da empresa na Internet está disponível no endereço <`http://www.steinberg.net/`> Acesso em: 18 ago. 2002.

processador do computador no próprio contexto de execução do aplicativo que carregou o módulo; o processamento é feito em tempo real. Graças à importancia da Steinberg no mercado e ao fato de a especificação ser fornecida sem nenhum custo, a VST tem sido adotada por diversos fabricantes e está se tornando um padrão *de facto* nessas plataformas;

- AudioSuite: essa especificação, desenvolvida pela Digidesign, é semelhante à VST, mas o processamento não é executado em tempo real: o algoritmo é aplicado aos dados de um arquivo e um novo arquivo processado é criado;

- RTAS (***R****eal-****T****ime* ***A****udio****S****uite*): versão com suporte a tempo real da especificação AudioSuite;

- TDM (***T****ime* ***D****ivision* ***M****ultiplexing*): módulos escritos de acordo com esta especificação, também desenvolvida pela Digidesign, são carregados em um processador DSP dedicado localizado em uma placa de expansão do computador e realizam o processamento em tempo real; esses módulos, portanto, só funcionam em conjunto com hardware específico fabricado também pela Digidesign (fabricante dos sistemas *Pro Tools*);

- HTDM (***H****ost* ***T****ime* ***D****ivision* ***M****ultiplexing*): também desenvolvida pela Digidesign, é similar ao RTAS, mas tem uma melhor integração com o hardware proprietário da empresa: em um sistema com esse tipo de hardware, os módulos RTAS só podem ser aplicados sobre canais de áudio já armazenados em disco no computador, e módulos TDM precisam ser aplicados ao áudio "depois" dos módulos RTAS; a especificação HTDM elimina essas restrições, estabelecendo um mecanismo de comunicação direta entre o módulo (que é executado pelo processador do computador) e os processadores DSP da placa de expansão.

No ambiente Linux, existem também algumas especificações para a criação de módulos, mas duas em particular são as mais importantes:

- LADSPA (***L****inux* ***A****udio* ***D****eveloper's* ***S****imple* ***P****lugin* ***A****PI*): é uma especificação bastante semelhante à VST, e também conta com um pacote de desenvolvimento (SDK); foi criada como uma "substituta" para a VST por causa da impossibilidade presente de uso da grande maioria de módulos VST em ambiente Linux bem como da incompatibilidade entre a licença da especificação VST e a licença GPL. Diversas aplicações que rodam em Linux têm suporte a módulos

que seguem esta especificação, um grande número de módulos já existe e muitos outros estão em desenvolvimento;

- gstreamer: é um projeto em andamento para a criação de um arcabouço para o desenvolvimento de aplicações voltadas para a multimídia baseadas em componentes, bem como um conjunto de componentes de uso genérico (*codecs* para diversos formatos de áudio e vídeo, por exemplo); um dos componentes disponíveis no gstreamer é um adaptador para módulos LADSPA.

2.2 baixa latência

Um outro aspecto que tem chamado a atenção dos desenvolvedores de sistemas de tempo real para áudio é a necessidade intrínseca desse tipo de aplicação de processamento com baixa latência. De fato, em muitas aplicações o usuário interage diretamente com o computador para a produção de um resultado, e a latência do sistema nesses casos precisa ser mantida abaixo do limiar perceptível pelo usuário. Um exemplo simples dessa situação é a gravação de um instrumento acústico processado (por exemplo, uma guitarra elétrica sobre a qual é aplicado um efeito em tempo real): ao mesmo tempo em que executa seu instrumento, o músico precisa ouvir o som que está sendo produzido; se a latência do processamento é muito grande, o músico terá dificuldades para a execução. Existe alguma controvérsia sobre qual o limiar de percepção da latência, mas parece ser recomendável que ela seja menor que 5ms[11]; o *Pro Tools* 24|MIX, que até o início e 2002 era o topo de linha da família de produtos *Pro Tools* (que é líder de mercado), acrescenta 1.5ms de latência de "entrada" e mais 1.5ms de latência de "saída" ao sinal, totalizando 3ms de latência no caso do exemplo acima[12].

Para possibilitar o desenvolvimento de aplicações que satisfaçam o requisito de processamento e comunicação com o hardware de áudio com baixa latência, três especificações foram desenvolvidas:

- ASIO (***A****udio* ***S****tream* ***I****nput/* ***O****utput*): desenvolvida pela Steinberg, consiste em uma especificação e um pacote de desenvolvimento (SDK) para o desenvolvimento de aplicações que interagem com dispositivos de áudio num esquema baseado em *callbacks*, ou seja, o controlador de disposi-

[11] De fato, se o som processado for misturado de alguma maneira com o som original, distorções no sinal semelhantes às provocadas por fenômenos acústicos (tais como *comb filtering*) podem ocorrer. Sobre *comb filtering* e distorção acústica, veja D'Antonio, P. *Minimizing acoustic distortion in project studios*. Disponível em: <`http://www.rpginc.com/cgi-bin/byteserver.pl/news/library/PS_AcD.pdf`> Acesso em: 18 ago. 2002.

[12] Burger, J. Digidesign Pro Tools|24 MIX and MIXplus (Mac/Win NT). *Electronic Musician*, julho, 2000. Disponível em: <`http://emusician.com/ar/emusic_digidesign_pro_tools_2/index.htm`> Acesso em: 18 ago. 2002.

tivo força a execução de uma função específica da aplicação em intervalos de tempo regulares; a cada vez que é executada, essa função processa um fragmento de áudio correspondente ao tamanho do intervalo de tempo entre as sucessivas chamadas, garantindo que a latência do sistema seja do tamanho desse intervalo, que pode, por sua vez, ser tão pequeno quanto se queira dentro dos limites permitidos pela velocidade do hardware;

- JACK: sistema desenvolvido para Linux, consistindo em um conjunto de bibliotecas, um servidor de áudio e um pacote de desenvolvimento, que apresenta uma API bastante similar à especificação ASIO em termos de arquitetura, mas possui recursos que vão além dos objetivos da ASIO, como veremos mais abaixo;

- CoreAudio: desenvolvida pela Apple como a interface nativa entre aplicações e o hardware de áudio no MacOS X, opera de forma similar ao sistema JACK.

Além de atenderem seus respectivos objetivos, as especificações ASIO, CoreAudio e JACK indiretamente servem também como camadas de abstração de alto nível sobre o hardware, eliminando para o programador a necessidade de se preocupar com aspectos de configuração como definição de taxas de amostragem e precisão de bits, número de canais etc.

É bom lembrar que, em aplicações com necessidades de tempo real rígido, geralmente usam-se sistemas operacionais dedicados com garantias de tempo real, que costumam ser capazes de garantir tempos de resposta da ordem de poucas dezenas de microsegundos; no entanto, esses sistemas oferecem dificuldades maiores para o desenvolvimento de aplicações e não possuem controladores de dispositivos para dispositivos de hardware genéricos como placas de som de uso geral. Como as necessidades das aplicações de áudio não são tão extremas (da ordem de milisegundos e não microsegundos), as versões mais recentes dos sistemas operacionais de uso genérico (como o Windows, MacOS e Linux) têm sido adaptadas (às vezes através do uso de controladores de dispositivos especiais, como é o caso do ASIO) para oferecer garantias que sejam suficientes para esses usos[13]; assim, fica a cargo dos controladores

[13] MacMillan, K., Droettboom, M. & Fujinaga, I. (2001). "Audio latency measurements of desktop operating systems". *Proceedings of the International Computer Music Conference*, pp. 259-262. Disponível na Internet em `http://gigue.peabody.jhu.edu/~ich/research/icmc01/latency-icmc2001.pdf` (nesse texto, o sistema JACK é citado com o nome LAAGA, já que em seus estágios preliminares esse era o nome do projeto). Os autores realizam testes de latência em diversas plataformas e comprovam que, com combinações de hardware e software adequados, Linux, Windows e MacOS são capazes de bons níveis de performance em termos de latência, em torno de 3-6 milissegundos.

de dispositivos de áudio e das próprias aplicações garantir um nível de desempenho adequado para o processamento em tempo real.

2.3 comunicação inter-processos

Finalmente, da mesma maneira que um usuário de aplicações típicas de escritório (como processadores de texto e planilhas de cálculos) muitas vezes precisa trocar informações entre aplicações diferentes, o músico ou técnico de estúdio também precisa trocar informações entre sintetizadores, sequenciadores MIDI, gravadores multicanais, editores de áudio etc. Até recentemente, havia duas opções para viabilizar esse trânsito de informações: utilizar recursos sem suporte a tempo real, como "copiar e colar" ou "exportar e importar", ou transmitir sinais de áudio entre aplicações por via analógica, através da placa de som, com a correspondente perda de qualidade devido a múltiplas conversões entre os domínios analógico e digital, além das dificuldades de sincronização.

Com vistas a solucionar esse problema, surgiram em 2001 três sistemas que buscam viabilizar a comunicação entre aplicativos em tempo real e com garantias de baixa latência: nos ambientes Windows e MacOS, o ReWire (desenvolvido pela Steinberg), no MacOS X o CoreAudio e, no ambiente Linux, o JACK (esses dois últimos combinam em um único sistema capacidades para comunicação em baixa latência com o hardware de áudio e para a comunicação entre aplicações). Esses sistemas oferecem às aplicações a possibilidade de comunicação através de mecanismos semelhantes aos *pipes* do Unix, mas otimizados para o uso em tempo real. Graças a isso, aplicações inteiras podem ser tratadas como componentes que fazem parte de um conjunto mais complexo.

3 limitações dos sistemas atuais

A diversidade de recursos para o desenvolvimento de aplicativos de áudio, como vimos, é grande: especificações claras e eficientes existem tanto para a comunicação com o hardware de áudio em tempo real e com baixa latência quanto para a criação e o uso de módulos reutilizáveis que encapsulam algoritmos DSP quaisquer, além de mecanismos para comunicação inter-processos em tempo real.

No entanto, um problema aparece freqüentemente como limitação para o uso do computador na área de áudio (e multimídia em geral): boa parte das aplicações demanda uma grande capacidade de E/S ou de processamento por parte do computador; alguns algoritmos são tão complexos que mal podem ser executados em tempo real, e muitas aplicações dependem da execução de vários algoritmos simultaneamente sobre um grande número de canais de áudio, impossibilitando completamente o processamento em tempo real. A despeito da crescente velocidade dos processadores disponíveis no mercado, é de se esperar que esse problema não seja resolvido apenas pelo aumento na velocidade do hardware a médio prazo.

3.1 soluções correntemente em uso

Comumente, tem-se usado algumas técnicas para contornar as limitações do hardware nesse tipo de aplicação:

- Uso de hardware dedicado: essa é a solução tradicionalmente adotada, por exemplo, pelos produtos da família *Pro Tools* (embora a Digidesign também comercialize o *Pro Tools LE*, versão do seu sistema que funciona independentemente de hardware dedicado); quando um aumento na capacidade de processamento é necessário, novas placas processadoras podem ser acrescentadas ao sistema. Além de se tratar, quase sempre, de hardware proprietário e importado, o maior problema desta solução é o custo: cada placa processadora do sistema *Pro Tools*, por exemplo, custa U$ 4000,00 nos EUA (de acordo com o próprio sítio da Digidesign na Internet). A empresa comercializa sistemas com uma, duas ou três placas, e sugere que a compra de mais placas pode trazer benefícios;

- Aumento da capacidade de processamento: a troca periódica de equipamentos por versões mais rápidas pode aliviar, embora não resolver, o problema. Em particular, sistemas biprocessados

oferecem ganhos de desempenho sem que haja necessidade de alteração significativa do código para operação paralela[14], já que permitem distribuir entre as CPUs as diversas cargas do sistema, tanto as que não têm necessidade de processamento em tempo real (gerenciamento de E/S de disco e da interface com o usuário) quanto as que têm necessidades de processamento em tempo real (processamento do áudio e comunicação com o hardware de áudio). Isso praticamente garante que as tarefas de tempo real não serão interrompidas indevidamente pelas outras;

- Processamento em lote: ao invés de realizar o processamento necessário em tempo real, o usuário define quais passos devem ser executados para obter um resultado e solicita o processamento em lote de todos os passos; alguns editores de áudio oferecem a opção deste tipo de processamento.

3.2 ausência de mecanismos para processamento paralelo

Essas soluções diferem das soluções tradicionalmente utilizadas para a solução de problemas que envolvem grandes capacidades de processamento ou E/S; em geral, problemas dessa natureza costumam ser resolvidos através da paralelização ou da distribuição da carga. No caso do processamento de multimídia, existem diversas oportunidades para a paralelização do código:

- Novos algoritmos podem ser desenvolvidos com vistas à paralelização ou à distribuição;
- Diferentes mídias, como áudio e vídeo, podem ser processadas paralelamente: basta alocar processadores ou máquinas independentes para cada mídia;
- No caso do áudio, muitas vezes a origem do problema está no fato de diversos canais independentes precisarem ser processados simultaneamente; o processamento desses canais também pode ser feito paralelamente, da mesma forma que o processamento de diferentes mídias;
- Mesmo um conjunto de algoritmos aplicados seqüencialmente sobre um único fluxo de dados (um canal de áudio ou vídeo) pode ser distribuído em um conjunto de máquinas ou processadores e executado "paralelamente": basta "enfileirar" as máquinas ou processadores de maneira que cada

[14] Se a aplicação se utilizar de *threads* diferentes para o processamento do áudio e para a interface com o usuário, o que é típico em sistemas que procuram obter baixas latências, ela deve se beneficiar do uso de dois processadores; mas mesmo que esse não seja o caso, o próprio sistema operacional se beneficia dos dois processadores para o gerenciamento das interrupções de disco, mouse etc. e para o agendamento de tarefas.

máquina ou processador atue sobre os dados já processados pela máquina ou processador imediatamente anterior a si; o resultado é simplesmente um aumento na latência do processamento em relação ao que haveria se todo o processamento fosse realizado em uma única máquina ou processador.

A paralelização de aplicações de áudio (e de multimídia em geral) em tempo real, no entanto, não é comum; por um lado, o uso de máquinas com mais de dois processadores não é comum, já que, além do custo elevado[15], os sistemas operacionais de uso genérico com Windows e MacOS apenas recentemente incorporaram suporte para esse tipo de hardware. Por outro, a distribuição entre máquinas requer a sincronização periódica entre as máquinas envolvidas; o volume de dados e a necessidade de baixa latência intrínsecas do processamento de multimídia se apresentam como dificuldades reais para a sincronização nesse tipo de cenário.

Um dos aspectos referentes à sincronização entre máquinas pode ser resolvido com a ajuda de hardware: se todas as diversas máquinas possuírem placas de som com suporte a sincronização unificada (*world clock*), é possível garantir o processamento síncrono em todas as máquinas através das interrupções geradas pelo hardware de áudio. Essa configuração, no entanto, envolve um aumento de custo significativo em cada nó (pois as placas de som com suporte a sincronização unificada são placas de uso profissional, custando acima de US$500,00 cada nos EUA) além de não resolver completamente o problema: a menos que cada máquina processe independentemente todo o áudio que deve ser reproduzido ou gravado pela placa de som local, a necessidade de tráfego de dados sincronizado entre as máquinas continua existindo.

3.2.1 formas de distribuição paralela da carga

Embora o desenvolvimento de algoritmos distribuídos para o processamento de áudio seja interessante, essa não é a melhor forma de garantir a paralelização do processamento de áudio, pois:

- O desenvolvimento de algoritmos paralelos é mais complexo que o desenvolvimento de algoritmos "lineares";

[15] Máquinas biprocessadas, que têm custo menos elevado, constituem um caso especial pois, como vimos, apresentam benefícios sem que haja a necessidade de algoritmos paralelos nas aplicações.

- É provável que o tipo de algoritmo adequado para resolver um determinado problema paralelamente seja em geral distinto dos tipos adequados para outros problemas; isso significa que seria necessário desenvolver diretamente uma grande quantidade de algoritmos paralelos para os mais diversos fins;

- Na maioria dos casos, os limites do hardware são atingidos não pela execução de um único algoritmo extremamente complexo, mas sim pela execução de um grande número de algoritmos sobre um grande número de canais de áudio. Nesses casos, a paralelização de algoritmos não traria benefícios em relação à mera distribuição de diferentes tarefas independentes entre diferentes processadores ou máquinas, mas traria um aumento desnecessário de complexidade;

- Quaisquer algoritmos paralelos desse tipo dependeriam de algum mecanismo de comunicação e sincronização entre suas partes; no caso do processamento distribuído, seria necessário desenvolver uma infra-estrutura para permitir esse tipo de comunicação. Essa mesma infra-estrutura pode ser usada para permitir a distribuição de diferentes tarefas independentes entre máquinas de forma sincronizada.

Por essas razões, antes de investigar a paralelização e distribuição de algoritmos, deve-se procurar soluções que permitam a distribuição sincronizada de tarefas independentes referentes ao processamento do áudio em tempo real; isso deve facilitar a distribuição da carga sem a necessidade de algoritmos complexos.

4 o projeto

Tendo em vista as necessidades de processamento de aplicações de áudio e a quase total ausência de soluções no mercado que possibilitem o processamento paralelo do áudio, é objetivo deste projeto investigar técnicas de baixo custo para a distribuição de aplicações para o processamento de áudio em tempo real com garantias de baixa latência; essas técnicas basicamente consistem em especificar um modelo síncrono de processamento entre as máquinas.

Uma forma de viabilizar a distribuição sincronizada de tarefas independentes referentes ao processamento do áudio em tempo real seria através do desenvolvimento de novas aplicações ou da adaptação das aplicações já existentes para o processamento distribuído; no entanto, como vimos, a maioria das aplicações para o processamento de áudio disponíveis atualmente se utiliza de módulos para tarefas específicas, e a comunicação entre os módulos e a aplicação "principal" é feita através de interfaces padronizadas. De forma similar, algumas aplicações têm suporte para funcionar como parte de um conjunto de aplicações interligadas, como se fossem elas também componentes de um sistema maior, também através de interfaces padronizadas.

Assim, a implementação do mecanismo de distribuição das tarefas de maneira que tarefas realizadas remotamente sejam acessíveis transparentemente através de alguma dessas interfaces permite que todas as aplicações e módulos já existentes compatíveis com essas interfaces passem a se beneficiar da distribuição de carga sem a necessidade de alterações em seu código. Portanto, além da especificação de mecanismos para a comunicação, deveremos implementar soluções para a distribuição do processamento de forma acessível através de uma interface de áudio padronizada já existente.

O sistema deve poder ser adequado para diversos sistemas de comunicação entre os computadores, tanto de hardware (como redes padrão 1Gigabit ou myrinet ou portas IEEE 1394[16]) quanto de software (TCP/IP ou outros), mas a implementação inicial será baseada no protocolo UDP[17], que possui excelentes implementações em vários sistemas operacionais, e em redes *Fast Ethernet*[18] padrão 100Mbits, que têm custo bastante baixo e oferecem desempenho razoável.

Também será analisada a viabilidade e interesse da "paralelização seqüencial" (ou pipeline), ou

[16] Canosa, J. Fundamentals of FireWire. *Embedded Systems Programming*, vol. 12, nº 6, junho de 1999. Disponível em: <`http://www.embedded.com/1999/9906/9906feat2.htm`> Acesso em: 18 ago. 2002.
[17] Stevens, W. R. *TCP/IP Illustrated.* Reading, MA: 1997. v. 1: the protocols. p.143-168.
[18] Quinn, L. B.; Russell, R. G. *Fast Ethernet.* John Wiley & Sons, 1997.

seja, a paralelização através do encadeamento de máquinas onde cada máquina processa os dados já processados pela máquina anterior, com custo de aumento na latência; esse tipo de solução pode ser útil em situações onde o aumento na latência não tem conseqüências graves ou onde a latência pode ser compensada com pré-processamento. Eventualmente, técnicas que favoreçam especificamente a paralelização em uma única máquina multiprocessada poderão também vir a ser discutidas mas, devido à opção por soluções de baixo custo, o uso de máquinas multiprocessadas ou de hardware dedicado não será considerado. Pela mesma razão, o estudo deverá se centrar em computadores pessoais de uso geral (ou seja, computadores compatíveis com o IBM PC ou Apple Macintosh).

Dentre os sistemas operacionais possíveis para essas plataformas, escolhemos como infra-estrutura para a pesquisa os sistemas baseados em Linux e os padrões desenvolvidos para ele (ALSA, LADSPA, JACK etc.); há diversas razões para essa escolha:

- As implementações dos principais programas (inclusive, por exemplo, o servidor de áudio jackd) estão disponíveis sob licenças de software livre[19] , possibilitando a alteração de algum desses programas caso necessário ou interessante;

- De forma similar, todos os padrões envolvidos são abertos e desenvolvidos pela comunidade de usuários; isso pode possibilitar a incorporação e adoção imediata das técnicas desenvolvidas na pesquisa, o que dificilmente aconteceria no caso de especificações controladas por uma empresa não diretamente envolvida com o movimento de software livre;

- O próprio projeto disponibilizará seus resultados como software livre, e o incentivo ao software livre traz diversas vantagens, em particular no Brasil, já que a maioria dos softwares de áudio de uso restrito[20] são caros e importados;

- As semelhanças entre os principais padrões (ASIO/CoreAudio e JACK, VST e LADSPA) significa que não deve ser demasiadamente difícil reimplementar as técnicas levantadas neste trabalho para os padrões usados nas outras arquiteturas caso as empresas responsáveis se interessem;

- O sistema operacional em si é altamente portátil além de ser amplamente compatível com os

[19] Em geral GPL, LGPL ou BSD; para uma visão geral sobre o software livre e sobre diversas licenças que se enquadram nessa definição, veja os sítios da *Free Software Foundation* (<`http://www.fsf.org/`>) e da *Open Source Initiative* (<`http://www.opensource.org/`>), acessados em: 18 ago. 2002.

[20] Em inglês, a expressão comumente usada é "proprietary".

padrões POSIX[21], viabilizando a implementação dessas mesmas técnicas em outras plataformas de hardware, seja através do próprio Linux, seja através da adaptação para outros UNIXes;

- O sistema operacional também é altamente estável e, com um dos *patches* de baixa latência[22], oferece níveis de latência extremamente baixos[23], comparáveis ao melhor que pode ser obtido com o MacOS X, que é um sistema voltado para o mercado de multimídia;

- Finalmente, sistemas distribuídos naturalmente dependem de diversos computadores e, portanto, de diversas cópias do sistema operacional; logo, o uso do Linux reduz consideravelmente o custo de um estúdio doméstico baseado em um sistema distribuído.

Um sistema desse tipo é de interesse primariamente para estúdios pequenos ou domésticos, onde o custo de soluções dedicadas capazes de oferecer grande desempenho, como por exemplo os sistemas da família *Pro Tools*, são inviáveis. Em contrapartida, um sistema distribuído baseado em uma rede de 100Mbits e computadores usados e "obsoletos" pode ser implementado com custo muito baixo e oferecer um bom nível de desempenho, além de ser facilmente expansível através do acréscimo de novas máquinas ou da substituição das máquinas existentes por máquinas mais rápidas.

O sistema pode vir a ser expandido para abraçar outras formas de mídia além de áudio, bem como servir como um experimento inicial no desenvolvimento de aplicações distribuídas com necessidades de tempo real e baixa latência para além da multimídia. Também pode, indiretamente, apontar métodos para a escolha de ordenações entre as operações favoráveis à paralelização.

[21] POSIX é um conjunto de especificações para sistemas operacionais; seu objetivo é facilitar a transposição de aplicações entre diferentes sistemas operacionais compatíveis com essas especificações. As especificações são desenvolvidas por um grupo de trabalho do IEEE, cujo sítio é acessível em <`http://www.pasc.org/`> Acessado em: 18 ago. 2002.

[22] O *patch* de Ingo Molnar foi desenvolvido primariamente para as versões 2.2.X do núcleo do Linux: <`http://people.redhat.com/mingo/lowlatency-patches/`> Acesso em: 18 ago. 2002; o *patch* de Andrew Morton, por sua vez, foi desenvolvido para as versões 2.4.X do núcleo do Linux: <`http://www.zip.com.au/~akpm/linux/schedlat.html`> Acesso em: 18 ago. 2002; o *patch* de Robert Love (que permite a operação em baixa latência por permitir a preempção do código dentro do núcleo do Linux) é compatível com as versões 2.4.X do Linux, e foi incorporado ao Linux a partir das versões 2.5.X: <`http://kpreempt.sourceforge.net/`> Acesso em: 18 ago. 2002.

[23] MacMillan, K., Droettboom, M. & Fujinaga, I., op. cit.

5 características do processamento do áudio em tempo real

O processamento de áudio por um computador envolve uma interação complexa entre o hardware de áudio, o sistema operacional e a aplicação, e a necessidade de garantias de baixa latência aumenta consideravelmente essa complexidade. É graças a diversas características dessa interação que técnicas semelhantes têm sido usadas em plataformas tão diferentes quanto os sistemas que usam o padrão ASIO, o Linux e o MacOS X, em particular a adoção de mecanismos baseados em funções *callback*.

5.1 interação entre o computador e o hardware de áudio

As placas de som normalmente possuem *buffers* de dados, usados tanto para o armazenamento dos dados referentes ao áudio capturado quanto para o armazenamento dos dados "em espera" para serem reproduzidos. Esses *buffers* são periodicamente esvaziados ou preenchidos, conforme o caso, através da comunicação, via barramento de E/S, com o computador; essa comunicação é regida pela geração de requisições de interrupção por parte da placa de som quando um de seus *buffers* está "cheio" ou "vazio". A placa normalmente precisa alocar ao menos dois *buffers* para cada sentido da comunicação: no caso da saída de dados, por exemplo, enquanto os dados contidos em um *buffer* são convertidos para o domínio analógico e reproduzidos, o outro *buffer* é preenchido com dados pelo computador; um mecanismo semelhante ocorre na comunicação no sentido inverso.

Assim, supondo que todos os *buffers* de entrada e saída têm tamanho igual e que, no caso de operação "full-duplex", os *buffers* de entrada estão alinhados no tempo com os de saída (e supondo, portanto, que uma única interrupção corresponde a uma solicitação para que a CPU preencha um *buffer* de saída e esvazie um *buffer* de entrada), a menor latência total possível de uma aplicação que simplesmente copia os dados da entrada para a saída (ou seja, o menor tempo possível para que um sinal injetado na entrada seja reproduzido na saída) corresponde exatamente à soma das durações correspondentes ao tamanho de um *buffer* de entrada e um de saída. Como o tamanho dos *buffers* usados pela placa de som normalmente é configurável via software (dentro de alguns limites), é possível definir com clareza a latência de um sistema de áudio computadorizado. Para que isso seja verdade, no entanto, a aplicação não pode introduzir nenhum outro tipo de atraso ao sinal: ela necessariamente deve processar tanto os dados fornecidos pela placa de som quanto os dados que devem ser fornecidos para a placa de som no momento em que eles são fornecidos ou solicitados, ou seja, no momento da

geração da requisição de interrupção por parte da placa.

Em um sistema operacional de uso geral, no entanto, essa não é uma tarefa simples: o tratamento das interrupções geradas pelo hardware é papel do núcleo do sistema operacional, e é ele quem realiza a transferência de dados entre a placa de som e a aplicação. Como o agendamento de tarefas num sistema operacional de uso geral não é determinístico, o programa que se comunica com o hardware de áudio pode eventualmente ser agendado para execução apenas após um grande número de interrupções.

Como solução para esse problema, a comunicação entre o hardware de áudio e as aplicações de áudio é tradicionalmente intermediada pelo núcleo, que possui *buffers* adicionais acessíveis pela aplicação através de uma API padronizada, geralmente centrada em torno da abstração de leitura e escrita com bloqueio: a aplicação continuamente processa uma certa quantidade de dados que devem ser reproduzidos e executa uma chamada de função para a escrita desses dados em um descritor de arquivos (ou alguma abstração semelhante) correspondente ao controlador de dispositivo; essa chamada de função bloqueia sempre que o *buffer* interno do sistema operacional está cheio, e desbloqueia quando há espaço no buffer. De forma semelhante, a passagem de dados do núcleo para a aplicação é baseada em uma chamada de função de leitura que bloqueia quando o *buffer* interno do sistema operacional está vazio e desbloqueia quando há dados disponíveis para a aplicação. Esse modelo é bastante adequado para aplicações de áudio simples, que podem ser executadas como quaisquer outros processos do sistema, mas apresenta o inconveniente de acrescentar uma grande latência ao processamento do sinal, o que não é aceitável em aplicações interativas.

Com uma abordagem diferente, os sistemas ASIO, CoreAudio e JACK procuram justamente possibilitar que as aplicações possam processar os dados referentes à comunicação com o hardware de áudio no momento em que o hardware solicita uma interrupção, ou seja, procuram dar garantias de tempo real rígido a sistemas operacionais de uso geral. Esses sistemas são baseados na idéia de *callbacks*: quando uma interrupção é gerada pelo hardware, esses sistemas realizam uma ou mais chamadas de função pré-definidas, em aplicações que são executadas como processos de usuário, que deverão processar imediatamente os dados de áudio referentes ao último *buffer* lido e ao próximo *buffer* a ser reproduzido da placa de som. Esse mecanismo é, basicamente, uma transposição do mecanismo do tratamento de interrupções interno do núcleo do sistema operacional para o nível das aplicações de usuário, permitindo que o tratamento das interrupções seja feito por aplicativos complexos quaisquer,

independentemente de alteração no código do núcleo.

Mas se, por um lado, esses sistemas são baseados em chamadas *callback*, por outro, aplicações para o processamento de áudio desse tipo se beneficiam do desenvolvimento modular; isso significa que o funcionamento dos módulos deve ser compatível com um sistema baseado em *callbacks*. As especificações de desenvolvimento LADSPA e VST solucionam esse problema implementando o mecanismo de comunicação entre as aplicações e os módulos também sob a forma de *callbacks*: após carregar a biblioteca que implementa um determinado módulo e inicializá-lo, a aplicação cria e registra junto ao módulo *buffers* de entrada e saída de dados; a partir desse momento, executa chamadas periódicas a uma função desse módulo; a cada chamada, o módulo processa os dados dos *buffers* de entrada e coloca os resultados do processamento nos *buffers* de saída.

Para que esse expediente baseado em *callbacks* funcione adequadamente para *buffers* de tamanho pequeno, é necessário que, após a detecção de uma interrupção, o núcleo do sistema suspenda sem praticamente nenhum atraso a execução de quaisquer outros processos da máquina e agende o programa de nível de usuário responsável pelo tratamento da interrupção para execução. Embora o núcleo "padrão" do Linux não satisfaça esse requerimento (ele pode demorar algum tempo antes de suspender outros processos), diversos *patches* de baixa latência para o núcleo do linux, desenvolvidos por Ingo Molnar, Andrew Morton e Robert Love, são capazes de trazer o tempo de resposta do sistema para valores entre um e dois milisegundos[24].

5.2 um exemplo: o sistema JACK

O JACK, que além de permitir a comunicação com a placa de som também estabelece um mecanismo para a comunicação entre as aplicações de áudio, é baseado em um servidor, jackd, que é executado com prioridade de "tempo real" no Linux (SCHED_FIFO). Seguindo a API padronizada pelo sistema operacional para comunicação com o hardware de áudio (no Linux, essa API é a ALSA - ***A****dvanced* ***L****inux* ***S****ound* ***A****rchitecture*), esse servidor interage com o controlador da placa de áudio através das abstrações de leitura e escrita, mas solicita ao controlador da placa que o sistema operacional utilize apenas um *buffer* intermediário; assim, cada vez que a placa de áudio preenche o *buffer* de captura de

[24] Graças ao programa latencytest (<`http://www.gardena.net/benno/linux/audio/`>), desenvolvido por Benno Senoner, é bastante simples realizar testes de desempenho para verificar o tempo de resposta do Linux. Por exemplo, os resultados do *patch* para preempção do núcleo aplicado ao Linux versão 2.4.6 estão disponíveis em <`http://kpreempt.sourceforge.net/benno/linux+kp-2.4.6/3x256.html`>.

áudio e gera uma interrupção, o servidor jackd é agendado para execução imediata, já que é executado com prioridade SCHED_FIFO. Ele então passa a agir como um agendador no nível das aplicações de usuário, gerenciando a execução de outros aplicativos SCHED_FIFO (os "clientes" do servidor jackd) que, assim como ele, permanecem bloqueados entre uma interrupção e outra do hardware de áudio. Os clientes então realizam o processamento necessário e retornam os resultados para o servidor jackd, que preenche o *buffer* de reprodução de áudio do sistema operacional que, finalmente, redireciona esses dados para o *buffer* da placa de som.

Os clientes do servidor jackd são, na verdade, *threads* de programas de áudio quaisquer que têm suporte ao sistema JACK por terem sido ligados dinamicamente à biblioteca libjack. Essas *threads* são criadas pela libjack e executam o processamento referente a cada aplicativo conectado ao servidor através da leitura e escrita de dados em segmentos de memória compartilhada criados pelo servidor jackd, o que dá ao servidor jackd a possibilidade de realizar a "conexão" desses aplicativos entre si além de com a placa de som: basta fazer dois aplicativos compartilharem um mesmo segmento de memória para que um escreva os dados que serão lidos pelo outro. Para que isso funcione adequadamente, os programas clientes precisam definir exatamente qual código deve ser executado pela *thread* criada pela libjack; para isso, o programa define uma função (que não pode realizar chamadas que bloqueiem, já que será executada com necessidades estritas de tempo) e a registra junto à libjack como uma função *callback*, ou seja, uma função que deverá ser executada quando do agendamento para execução pelo servidor jackd. Muito provavelmente, essa função precisa ter acesso a dados compartilhados entre as *threads*; é muito importante que essa comunicação seja feita de forma independente de sistemas de bloqueios (*locks*) ou semáforos para que não haja atrasos na execução das chamadas de *callback*.

O jackd, então, funciona como um agendador de tarefas que faz com que as sucessivas chamadas de *callback* definidas pelos aplicativos sejam executadas em ordem. Quando os clientes se registram junto ao jackd, o servidor estabelece uma ordenação topológica entre as chamadas de *callback* de forma que qualquer aplicativo cuja saída seja processada por um outro seja executado antes deste. Para realizar o agendamento, o jackd simplesmente força o agendador do sistema operacional a agendar o processo adequado para execução: todas as *threads* criadas pela libjack nos clientes estão bloqueadas aguardando a leitura de um byte a partir de um *named pipe* criado pelo servidor jackd; depois que o jackd é ativado para execução pelo núcleo, ele coloca os dados lidos da placa de som no segmento

de memória de "entrada" do primeiro cliente a ser agendado, escreve um byte no *named pipe* sobre o qual essa aplicação está bloqueada em espera e bloqueia na leitura de um outro *named pipe*. Como a aplicação tem prioridade SCHED_FIFO e o servidor jackd bloqueia nesse momento, a aplicação é agendada para execução, processa os seus dados de "entrada" e os escreve na "saída"; a seguir, a aplicação escreve um byte em um outro *named pipe* e bloqueia novamente na leitura do *named pipe* anterior. É sobre esse segundo named pipe que a segunda aplicação a ser executada está bloqueada, o que faz que essa aplicação desbloqueie e inicie sua execução. Esse processamento continua até que a última aplicação escreva um byte no *named pipe* sobre o qual o servidor jackd está bloqueado. Nesse momento, o servidor é desbloqueado, envia os dados para a placa de som e bloqueia novamente, agora sobre o descritor de arquivo do hardware de áudio, aguardando pela próxima interrupção.

Vale observar que, se o JACK foi descrito aqui como um sistema totalmente controlado pelo hardware de áudio (e, de fato, a implementação atual tem essa característica), esse sistema de "controle" por parte do hardware de áudio é implementado por um módulo dentro do servidor jackd, que pode eventualmente ser substituído por algum outro tipo de controlador, já que o autor procurou manter o código referente ao controlador de áudio claramente separado do resto do servidor justamente para possibilitar outras aplicações para o sistema.

6 distribuição do processamento do áudio em tempo real

Se, por um lado, nosso objetivo é distribuir o processamento do áudio em tempo real através de uma rede local, por outro, os sistemas de rede comumente usados têm algumas limitações que dificultam seu uso para essas aplicações: as taxas de transferência de dados das redes, embora crescentes, ainda são relativamente pequenas para o tráfego de vários canais de áudio, e essas velocidades relativamente baixas podem introduzir latências na comunicação que precisam ser eliminadas ou, pelo menos, levadas em conta; além disso, a maioria dos sistemas de rede de baixo custo não oferece mecanismos para a transmissão de dados com garantias de tempo real.

6.1 redes e tempo real

Para que seja possível o acesso transparente a aplicações distribuídas em tempo real através de uma interface semelhante às descritas acima, o sistema de comunicação entre as máquinas deve ser adequado ao mecanismo de funções *callback*, ou seja, os dados devem poder ser enviados e recebidos sincronamente, em blocos de tamanho pré-determinado e a intervalos de tempo fixos e relativamente curtos, e o processamento remoto deve ser o mais rápido possível para que o dado processado retorne à máquina "de origem" antes da próxima chamada de *callback*, que normalmente deve ocorrer quando o hardware de áudio provocar uma nova interrupção. Essas características pressupõem que a utilização da rede siga um padrão "half-duplex" e podem oferecer dificuldades de desempenho no nível do hardware de rede; de fato, dependendo da velocidade nominal da rede, da taxa de amostragem e do número de bits de cada amostra de áudio, podemos calcular o número máximo de canais que pode ser enviado e recebido pela rede em tempo real em modo "half-duplex":

$$numero\,de\,canais = \frac{velocidade\,da\,rede\,(em\,bits/s)}{2 * taxa\,de\,amostragem(em\,Hertz) * bits\,por\,amostra}$$

Se tomarmos como exemplo uma rede de 100Mbits de velocidade nominal e uma aplicação que processe o áudio como amostras de 32 bits com freqüência de 96KHz[25], veremos que o número máximo de canais que podem ser enviados e recebidos pela rede em tempo real é $\frac{100000000}{2*96000*32} \cong 16$. Esse número,

[25] Os sistemas de gravação de áudio modernos têm trabalhado com resolução de 24 bits, e as amostras são processadas como números de ponto flutuante de 32 bits, o que praticamente elimina problemas de distorções causados por ajustes de ganho muito grandes ou pequenos que, no caso de inteiros, resultariam em perda de definição no sinal. Por outro lado, taxas de amostragem acima de 48KHz ainda não são tão comumente usadas, mas também oferecem ganhos de qualidade para o sinal, especialmente em relação à percepção espacial. O sistema *Pro Tools HD*, atualmente topo-de-linha da família *Pro Tools*, opera com taxas de amostragem de até 192KHz.

no entanto, é um limite superior que não leva em conta diversos fatores:

- a comunicação entre o computador e a placa de rede consome algum tempo, tanto para a transmissão quanto para a recepção, e os dados precisam ser transmitidos e recebidos tanto pela máquina de origem quanto pela máquina que deve realizar o processamento remoto;
- o sistema operacional gasta algum tempo para processar as interrupções provocadas pelo hardware de rede;
- o sistema operacional também gasta algum tempo para processar as interrupções provocadas pelo hardware de áudio e para se comunicar com o hardware de áudio;
- o tempo de resposta do sistema operacional às interrupções provocadas pelo hardware de rede não é constante: dependendo do contexto de execução e de outras interrupções de hardware (por exemplo, por parte do disco), esse tempo pode variar;
- de forma semelhante, no caso de redes como a Ethernet, que não oferece garantias de tempo real (diferentemente do padrão IEEE 1394, por exemplo), o trânsito de dados pela rede não gasta sempre exatamente o mesmo tempo: mesmo em uma rede dedicada para esse fim, pode haver colisões capazes de causar atrasos na transmissão de pacotes;
- todos os fatores acima são agravados pelo fato de ser desejável manter um nível baixo de latência, o que significa que centenas de mensagens precisam ser enviadas e recebidas pela rede a cada segundo, todas elas responsáveis por um pequeno consumo de tempo para o gerenciamento da comunicação e suscetíveis a sofrer atrasos por conta das variações nos diversos tempos de resposta;
- o envio de dados para processamento remoto pressupõe que esse processamento deve tomar um tempo não-trivial; caso contrário, ele poderia ser realizado localmente.

Ainda no caso de uma rede de 100Mbits de velocidade nominal, se considerarmos, por exemplo, que para o tráfego de dados entre as máquinas podemos dispor, com segurança, de 20% do tempo total disponível para o processamento (o que significa que temos necessariamente bem menos de 80% do tempo disponível para o processamento remoto em si), teremos velocidade na rede para o envio e recebimento de apenas $\frac{100000000*0.2}{2*96000*32} \cong 3$ canais de áudio em tempo real (no caso da taxa de amostragem

de 44100Hz, esse número sobe para cerca de 7). Qualquer tentativa de aumentar esse número de canais se reverte numa redução no tempo disponível para o processamento remoto. Se, por um lado, esse valor pode ser adequado para algumas aplicações, não é difícil imaginar situações onde o processamento distribuído se beneficiaria de um número maior de canais. O uso com outras mídias, como vídeo por exemplo, pode demandar velocidades ainda maiores para o tráfego de dados.

6.2 melhorias para a distribuição

Vale a pena encarar o processamento do áudio dessa maneira como sendo dividido em três etapas (vamos pensar simplificadamente em apenas duas máquinas, mas um número maior de máquinas não mudaria o cenário):

1. a transferência dos dados da máquina de origem para a máquina remota;
2. o processamento remoto dos dados;
3. a transferência dos dados da máquina remota para a máquina de origem.

Como vimos, essas três etapas precisam ser completadas em tempo inferior ao intervalo entre chamadas sucessivas da função *callback*. Observemos, no entanto, que esse modo de operação é bastante pouco eficiente, tanto em termos de utilização da CPU remota, que permanece ociosa durante todo o tempo em que há transferência de dados em algum sentido, quanto em termos de utilização da rede, que permanece ociosa enquanto há processamento em execução por parte da CPU remota. Esse modo de operação pode ser melhorado aplicando um mecanismo semelhante ao sistema de "janelas deslizantes" utilizado em diversos protocolos de rede, como o PPP e o TCP/IP[26].

Num sistema desse tipo, a cada chamada *callback* (geralmente, a cada interrupção gerada pelo hardware de áudio), a máquina onde a chamada é executada envia dados para o processamento remoto e recebe dados processados pela máquina remota de volta; no entanto, os dados recebidos da máquina remota correspondem ao resultado do processamento dos dados enviados na chamada *callback* anterior. Esse esquema acrescenta à latência total dos canais processados remotamente o tempo correspondente ao tamanho de um *buffer* do hardware de áudio mas, por outro lado, aumenta o número de canais

[26] O sistema de janelas deslizantes do protocolo TCP/IP é descrito em Stevens, W. R. *TCP/IP Illustrated*. Reading, MA: 1997. v. 1: the protocols. p.280-284.

que podem ser processado remotamente: num sistema desse tipo, precisamos garantir que o tempo total para o envio, recebimento e processamento dos dados seja menor que o tempo correspondente ao tamanho de dois *buffers* do hardware de áudio ao invés de apenas um, desde que cada uma das três fases consuma menos que o tempo correspondente ao tamanho de um *buffer* para se completar[27]. Se considerarmos, por exemplo, que o processamento remoto de um bloco de dados toma 80% do tempo correspondente a um *buffer* do hardware de áudio e que as variações listadas acima podem tomar mais 20% desse tempo, dispomos de 100% do tempo correspondente a um *buffer* para o tráfego dos dados, ou seja, 50% em cada sentido. Isso corresponde a dizer que o limite teórico de canais de dados que podemos processar remotamente é o limite calculado acima, $\frac{100000000}{2*96000*32} \cong 16$.

No entanto, esse sistema melhora, mas não maximiza, o uso da rede e da CPU remota: se a transmissão de dados em cada um dos dois sentidos consumir tempo próximo ao correspondente ao tamanho do *buffer*, não haverá quase nenhum tempo disponível para o processamento remoto; por outro lado, se o processamento remoto consumir tempo próximo ao correspondente ao tamanho do *buffer*, a rede será utilizada como se em modo "half-duplex". Assim, seguindo o mesmo modelo, podemos imaginar um sistema onde a máquina de origem recebe, a cada chamada *callback*, os dados processados remotamente que foram enviados dois ciclos atrás. Dessa forma, precisamos garantir que o tempo total para o envio, recebimento e processamento dos dados seja menor que o tempo correspondente ao tamanho de três *buffers* do hardware de áudio ao invés de apenas um, desde que cada uma das três fases consuma menos que o tempo correspondente ao tamanho de um *buffer* para se completar[28]. Com isso, podemos operar com o máximo de efeciência da rede e da CPU remota, mas por outro lado acrescentamos o tempo correspondente a dois *buffers* do hardware de áudio à latência total do processamento. Nesse cenário, se considerarmos uma perda de 20% do tempo no gerenciamento da rede, o limite teórico para o número de canais que podem ser enviados para processamento remoto é $\frac{100000000}{96000*32} * 0.8 \cong 25$, já que a rede pode operar em modo "full-duplex".

[27] A transmissão dos dados da máquina de origem para a máquina remota precisa demorar menos tempo que o correspondente ao tamanho de um *buffer* porque, caso contrário, a máquina de origem tentaria iniciar a transmissão de um novo conjunto de dados antes de a transmissão anterior ter sido terminada; o mesmo ocorre na transmissão de dados da máquina remota para a máquina de origem. Por outro lado, o processamento na CPU remota precisa ser terminado em tempo inferior ao tempo correspondente ao tamanho de um *buffer* porque, caso contrário, a máquina receberia um novo conjunto de dados para processamento antes do término do processamento anterior.

[28] Da mesma forma que no caso anterior.

Não podemos deixar de observar que essa exposição não leva em conta fatores como o tempo de CPU gasto no gerenciamento da comunicação via rede, a variação nos tempos de resposta do sistema operacional às interrupções dos hardwares de áudio e de rede e a variação no tempo de transmissão dos dados via rede; portanto, é claro que cada uma das três fases não pode demorar um tempo muito próximo ao tempo máximo "teórico" calculado. Por outro lado, vale lembrar que, em algumas aplicações, o aumento da latência pode ser compensado através de pré-processamento, desde que a latência total seja conhecida.

7 o mecanismo de comunicação

A comunicação de dados entre aplicações via rede pode envolver muitos tipos de dados; especificações como a CORBA[29] procuram garantir um alto nível de abstração nesse tipo de comunicação, permitindo que uma ampla gama de tipos de dados sejam enviados e recebidos transparentemente via rede. Os custos de sistemas desse tipo, no entanto, são a complexidade e a perda de desempenho. O presente projeto não pretende competir com sistemas como CORBA em termos de recursos, nível de abstração ou flexibilidade; pelo contrário, a intenção do projeto é apenas estabelecer um mecanismo simples e eficiente para a transmissão de dados sincronizadamente em tempo real entre máquinas diferentes. Por esta razão, ele lida apenas com *buffers* de dados, ou seja, não trata os dados de maneira orientada a objetos e, pela mesma razão, deve ser limitado a tipos de dados nativos da linguagem C: inteiros, números de ponto flutuante ou caracteres. Por outro lado, aliado a sistemas como CORBA, o sistema pode permitir a combinação de um bom nível de flexibilidade com um bom desempenho para aplicações distribuídas com necessidades de processamento sincronizado em tempo real.

7.1 implementação

Para viabilizar a comunicação de dados segundo o modelo síncrono descrito acima, pretendemos definir e desenvolver um pequeno conjunto de classes:

- uma classe abstrata que deve corresponder à representação do endereço de uma "conexão" numa máquina em uma rede;
- uma classe concreta, derivada da classe anterior, capaz de representar um par endereço IP e porta, para redes TCP/IP;
- um par de classes concretas para a comunicação de dados através do protocolo UDP (uma para o envio e outra para o recebimento de dados); essas classes não devem ser efetivamente derivadas de nenhuma outra por razões de desempenho, mas suas interfaces devem ser bem definidas para que classes equivalentes possam ser desenvolvidas para dar suporte a outros tipos de rede além do TCP/IP. Deve ser possível utilizar a classe de recebimento de dados em modo com ou

[29] Henning, M.; Vinoski, S. *Advanced CORBA programming with C++*. Addison-Wesley, 2001, p. 9-35.

sem bloqueios (ou seja, em modos *blocking* e *non-blocking*), bem como ter acesso ao descritor de arquivos sobre o qual a classe opera caso seja interessante o uso das funções `poll()` ou `select()`;

- uma classe mestre (*master*) que deve ser usada como gerenciadora na comunicação com uma ou mais máquinas remotas; essa classe se utiliza das classes de comunicação de dados através do protocolo UDP ou outras, e deve permitir que o usuário defina *buffers* de entrada e saída de dados e chame periodicamente uma função *callback* para o envio e recebimento dos dados que devem ser processados remotamente. Ela também deve numerar os blocos de dados enviados para as máquinas remotas e verificar se a numeração de blocos recebidos é coerente com os blocos enviados. Finalmente, o usuário deve poder escolher se a comunicação com as máquinas remotas vai ou não se utilizar do mecanismo de "processamento desalinhado" descrito anteriormente;

- uma classe escrava (*slave*) que deve ser usada na comunicação com uma máquina onde roda uma classe mestre; essa classe também se utiliza das classes de comunicação de dados através do protocolo UDP ou outras e deve permitir ao usuário tanto registrar uma função *callback* quanto definir *buffers* de entrada e saída de dados; essa classe deve então colocar os dados recebidos via rede nos *buffers* de entrada e chamar a função *callback* quando houver novos dados a processar recebidos da rede; após o processamento da função *callback*, os dados dos *buffers* de saída devem ser enviados de volta à máquina mestre. Caso haja mais de uma instância dessa classe em execução por um único processo, este pode se beneficiar do uso de funções como `poll()` ou `select()` para gerenciar a recepção de dados de múltiplas máquinas mestres e, portanto, a classe escrava deve prover acesso ao descritor de arquivo sobre o qual opera a classe de comunicação de dados que está usando;

- possivelmente, uma classe auxiliar para encapsular o gerenciamento do mecanismo de espera baseado em funções POSIX como `poll()` ou `select()`.

7.2 distribuindo o processamento de módulos LADSPA

Como as classes descritas anteriormente implementarão um mecanismo genérico de comunicação, para que alguma aplicação específica possa fazer uso delas é preciso que sejam desenvolvidas classes para adaptar a interface dessas classes à interface esperada pela aplicação; ou seja, é preciso desenvolver

classes que desempenhem o papel descrito no padrão de projeto "adaptador"[30]. Como o objetivo primário deste projeto é viabilizar o processamento distribuído de áudio por aplicações com suporte a APIs padronizadas, vamos desenvolver um sistema baseado nessas classes para o processamento distribuído de módulos LADSPA de maneira transparente para aplicações capazes de interagir com módulos LADSPA. Esse sistema deverá consistir de:

- uma biblioteca carregável dinamicamente[31] com "pseudo-módulos" LADSPA. Quando uma aplicação carrega essa biblioteca e solicita a instanciação de um desses módulos, o módulo cria um novo processo (através do par de instruções `fork()` e `exec()`) que deve inicializar os mecanismos necessários para a comunicação entre o módulo e o processo remoto que implementa de fato o processamento dos dados. Esse passo deve necessariamente ser executado por um processo diferente para que o laço principal da sua interface com o usuário não interfira com o laço principal da interface com o usuário da aplicação que solicitou a instanciação do módulo, que pode ser baseada em um pacote para construção de interfaces diferente. Após a configuração ser preparada, a biblioteca instancia a classe mestre para a comunicação com a máquina remota e registra sua função *callback* junto à aplicação;

- uma classe que funcione como um adaptador entre a classe escrava e módulos LADSPA, de maneira que dados recebidos via rede possam sejam processados pela função *callback* de um módulo LADSPA;

- um cliente CORBA responsável por solicitar a um servidor, possivelmente através do serviço de negociações CORBA[32], a instanciação de um adaptador para um conjunto de módulos LADSPA. Esse cliente é executado quando da instanciação de um "pesudo-módulo" LADSPA na máquina que vai atuar como mestre da comunicação;

- um servidor CORBA capaz de criar instâncias da classe adaptadora quando solicidado de acordo com diversas especificações recebidas: quais módulos LADSPA deve ser manipulados pela classe adaptadora, endereço de rede da máquina mestre etc. Para evitar interferências entre o laço

[30] Gamma, E. et al. *Design Patterns*: elements of reusable object-oriented software. Reading: Addison-Wesley, 1995, p. 139-150.

[31] na especificação LADSPA, módulos são carregados em tempo de execução não através do mecanismo normal do sistema operacional, mas diretamente pelo programa através da função `dlopen()`.

[32] Henning, M.; Vinoski, S. *Advanced CORBA programming with C++*. Addison-Wesley, 2001, p. 827-834.

principal do servidor CORBA e a classe adaptadora, esta deve ser inicializada em uma *thread* independente da *thread* que gerencia a comunicação CORBA. Essa nova *thread* deve ser executada com prioridade de tempo real (SCHED_FIFO) e rodar as diversas instâncias de classes adaptadoras em um laço infinito (mas que pode ser interrompido ou alterado pela outra *thread*) gerenciado por uma função como `poll()` ou `select()`, permitindo que esse servidor possa processar dados solicitados por mais de um mestre.

Diversos detalhes sobre a implementação desse sistema deverão ser definidas ao longo do desenvolvimento do projeto; em particular, a especificação LADSPA pressupõe que a aplicação tem acesso a informações como número, nome e tipo das portas de um módulo LADSPA antes de sua instanciação, o que vai tornar problemática a criação de "pseudo-módulos" LADSPA capazes de encapsular quaisquer conjuntos de módulos LADSPA remotos.

8 estado atual do projeto

Alguns resultados de interesse para o projeto já foram obtidos:

- um par de clientes para o servidor JACK capazes, respectivamente, de enviar e receber dados de áudio para processamento remoto, foram escritos e testados. Embora eles não tivessem nenhum mecanismo para a numeração dos blocos de dados, foi possível verificar auditivamente que a transmissão de dados via rede em modo síncrono funcionou a contento;
- para o desenvolvimento desses clientes, foram implementadas classes para a comunicação via protocolo UDP e para a representação de pares de endereço IP/porta; sua adaptação para este projeto deverá ser trivial;
- alguns testes de velocidade de transmissão de dados em uma rede local foram realizados, oferecendo uma visão do impacto de diversos fatores sobre a eficiência da comunicação.

testes de velocidade da rede

Os testes foram realizados usando três máquinas: duas baseadas em processadores AMD Athlon, de 1.1GHz e 1.4GHz com 256Mb de memória e a outra baseada em um processador AMD K6-2 de 450MHz com 192Mb de memória. As máquinas de 1.1GHz e de 450MHz estavam equipadas com placas de rede padrão 100Mbits baseadas no chip Realtek RTL-8139, que são de muito baixo custo (em torno de R$40,00); a máquina de 1.4GHz possui um controlador de rede embutido na própria placa-mãe, baseado no chip SiS 900; o concentrador de rede usado foi um *switching hub* de baixo custo da marca Encore. É de se esperar que placas de rede mais sofisticadas reduzam o custo em termos de processamento da CPU para a comunicação de dados.

Os testes consistiram na execução de dois programas em máquinas distintas com prioridade de tempo real no Linux (SCHED_FIFO). O primeiro deles periodicamente enviava um bloco de dados via protocolo UDP pela rede para a máquina remota e passava a aguardar a resposta dessa máquina, além de registrar o tempo necessário para as diversas operações de comunicação. O segundo, ao receber um bloco de dados, simplesmente o reenviava de volta à máquina de origem. Para eliminar o efeito de mudanças de contexto ou interrupções de hardware inesperadas na máquina remota, o programa "eco" funcionava em modo de espera em ciclo (*busy-wait*); de forma similar, o programa de origem, após

enviar um bloco de dados, passava a esperar pela resposta também em modo de espera em ciclo. Na máquina de origem, no entanto, outros processos eram responsáveis por gerar uma grande atividade de disco, de forma a introduzir aleatoriamente a necessidade de tratamento de interrupções durante os ciclos de espera, a transmissão de dados etc. Foram testados blocos de tamanho entre 128 e 55296 bytes; tamanhos maiores não podem ser transmitidos em um único bloco UDP. Para esses diversos tamanhos de blocos de dados, foram medidos:

- O tempo para a execução da função `write()`;
- O tempo para a execução da função `read()`;
- O tempo total para o envio e recebimento de um bloco de dados;
- A partir desses dados, o tempo dos dados "no fio" foi estimado: do tempo total entre o envio e o recebimento de um bloco foram subtraídos os tempos usados na execução das funções `read()` e `write()` nas duas máquinas envolvidas em cada teste. Não foi possível excluir desse resultado o tempo necessário para o tratamento da interrupção gerada pela placa de rede quando do recebimento dos dados via rede e, portanto, o tempo dos dados "no fio" erroneamente inclui esse tempo.

Os resultados mostraram que, num cenário onde a latência do hardware de áudio é da ordem de poucos milisegundos, o trânsito de dados correspondente a um número razoável de canais gera uma carga adicional para o gerenciamento da comunicação via rede não-desprezível; no entanto, essa carga adicional é aceitável (gráficos dos dados obtidos estão disponíveis no apêndice A). Por exemplo, se a latência correspondente a um *buffer* da placa de som é de 1.5ms, para o "processamento desalinhado" com os dados defasados em dois ciclos, sabemos que a transmissão e a recepção dos dados devem tomar, no máximo, 1.5ms cada. Isso corresponde, nos testes, a um tempo de ida-e-volta próximo de 3ms. Tomando as medidas feitas com a máquina principal com processador de 1.1GHz e a máquina "remota" com processador de 1.4GHz, vemos que foi possível transmitir 14336 bytes em menos de 2852μsecs, o que corresponde, assumindo uma taxa de transferência de 96KHz e amostras de 32bits, a cerca de 26 canais de áudio($\frac{\frac{14336*1000000}{2852}}{96000*4}$).

Nesse contexto, a máquina mestre gastou 187μsecs para enviar os dados para a máquina remota (função `write()`) e 127μsecs para processar os dados recebidos da máquina remota (função `read()`),

sem contar o tempo necessário para processar a interrupção gerada pelo hardware de rede quando do recebimento dos dados remotos; ou seja, a cada 1.5ms a máquina gastou pelo menos 314μsecs para gerenciar a comunicação com a rede, ou cerca de 20% de sua capacidade de processamento. Por outro lado, a máquina remota gastou 115μsecs para o envio dos dados via rede (função `write()`) e 98μsecs para processar os dados recebidos (função `read()`), ou seja, gastou pelo menos 213μsecs no gerenciamento da comunicação via rede, equivalente a cerca de 15% de sua capacidade de processamento.

No caso de latências maiores, por exemplo 4.5ms, a utilização da rede é mais eficiente, mas o custo do processamento é ligeiramente maior: esse caso corresponde a um tempo de ida-e-volta nos testes de 9ms; tomando como exemplo as mesmas máquinas, vemos que foi possível transmitir 49152 bytes em menos de 8806μsecs, o que corresponde, também assumindo uma taxa de transferência de 96KHz e amostras de 32bits, a cerca de 29 canais de áudio ($\frac{\frac{49152*1000000}{8806}}{\frac{2}{96000*4}}$).

Nesse contexto, a máquina mestre gastou 764μsecs para enviar os dados para a máquina remota (função `write()`) e 395μsecs para processar os dados recebidos da máquina remota (função `read()`), sem contar o tempo necessário para processar a interrupção gerada pelo hardware de rede quando do recebimento dos dados remotos; ou seja, a cada 4.5ms a máquina gastou pelo menos 1159μsecs para gerenciar a comunicação com a rede, ou cerca de 25% de sua capacidade de processamento. Por outro lado, a máquina remota gastou 463μsecs para o envio dos dados via rede (função `write()`) e 277μsecs para processar os dados recebidos (função `read()`), ou seja, gastou pelo menos 740μsecs no gerenciamento da comunicação via rede, equivalente a cerca de 17% de sua capacidade de processamento. Vale lembrar que outros hardwares de rede podem propiciar resultados melhores.

9 tarefas a serem cumpridas

Diversas tarefas ainda devem ser desenvolvidas para o término adequado do projeto:

- É necessário realizar um levantamento bibliográfico adequado, de forma a tornar possível a discussão sobre outros trabalhos relacionados ao tema deste projeto e a redação dos capítulos introdutórios da dissertação, que dependem de informações de caráter mais genérico. Em particular, alguns temas devem ser alvo de mais pesquisas:
  - sistemas distribuídos, sistemas de tempo real e, principalmente, sistemas distribuídos de tempo real;
  - outros projetos comparáveis, principalmente o ORB CORBA TAO e o VST System Link, mas também outros sistemas voltados para multimídia em rede;
  - possivelmente, princípios de paralelização de aplicações a algoritmos;
  - algoritmos geralmente usados e nível de carga que eles produzem, de forma a mostrar que a paralelização pode trazer benefícios mesmo a médio prazo;
  - relações entre latência e processamento de áudio em tempo real.
- Outros produtos disponíveis no mercado voltados para o processamento do áudio além do *Pro Tools* devem ser investigados e apresentados. Em particular, é importante mostrar que a maioria ou a totalidade dos sistemas baseados em hardware dedicado são de implementação restrita (*proprietary*) e de alto custo. Da mesma forma, é preciso realizar um levantamento dos equipamentos com suporte a sincronização unificada e seu custo;
- Os cálculos realizados aqui para redes de 100Mbits devem ser repetidos para redes padrão 1Gigabit ou IEEE 1394, e deve-se mostrar como o modo isócrono do padrão IEEE 1394 pode ser usado pelo sistema;
- É preciso verificar se existem placas de som que não seguem o esquema de comunicação com o computador aqui descrito;
- Também é preciso descrever com mais detalhes as características de sistemas de tempo real e como essas características são utilizadas nos sistemas compatíveis com o padrão POSIX, como o

Linux; em particular, o funcionamento do escalonamento com prioridade SCHED_FIFO deve ser descrito;

- O sistema como um todo deve ser implementado e testado, e as dificuldades de implementação devem ser documentadas e solucionadas. Em particular, o problema da ordenação de bytes (*endianess*) entre máquinas de arquiteturas diferentes precisa ser resolvido, bem como o mecanismo para adaptação entre o sistema de comunicação e o padrão LADSPA;

- Será útil levantar as possibilidades da distribuição de carga através da "paralelização seqüencial" e as conseqüências do aumento na latência inerente a esse sistema. Também será útil discutir aplicações em sistemas de áudio que utilizem hardware com suporte a sincronização unificada, bem como observar que outras aplicações para o sistema são possíveis, por exemplo baseadas na API do sistema JACK, e discutir possíveis aplicações fora do campo da multimídia que também possam se beneficiar de um sistema de comunicação semelhante ao que será usado neste projeto.

10 trabalhos relacionados

Até o momento foram encontrados cinco trabalhos com alguma relação ao tema proposto:

- VST System Link: é um produto, desenvolvido pela Steinberg, com objetivos semelhantes ao deste projeto, mas através de mecanismos diferentes: a transmissão de dados é feita através de interfaces de áudio com conexões digitais (AES/EBU, S/PDIF, ADAT) e um ou mais bits do sinal de áudio são utilizados para a codificação de informações auxiliares, como MIDI, contagem de tempo etc. O produto ainda não foi de fato lançado, e não há muitas informações disponíveis no sítio do fabricante na Internet;

- TAO (***T****he* ***A****CE* ***O****RB*): é um ORB CORBA com extensões para suporte a tempo real, cujas características ainda precisam ser comparadas com os objetivos deste projeto;

- aRts (***A*** ***R****eal* ***T****ime* ***S****ynthesizer*): é um servidor de áudio com diversas características interessantes e com suporte à execução de fluxos de áudio recebidos via rede; no entanto, ele não provê recursos para a execução sincronizada ou em baixa latência de fluxos de áudio independentes;

- MAS (***M****edia* ***A****plication* ***S****erver*): é um projeto para o desenvolvimento de um servidor de mídia (basicamente áudio e vídeo) capaz de operar transparentemente via rede, complementando as capacidades do sistema de janelas X. Um dos objetivos do projeto é garantir que haja mecanismos para impedir que fluxos de áudio e vídeo sejam executados "fora de sincronia"; no entanto, isso deverá ser implementado através da correção de erros cumulativos, e não através de um modelo de execução verdadeiramente síncrono. Além disso, o sistema não se preocupa diretamente com a latência da comunicação, já que a aplicação primária do sistema é permitir a visualização de conteúdo multimídia via rede de forma transparente;

- RTP (***R****eal* ***T****ime* ***P****rotocol*): é uma especificação para a transmissão e o controle de multimídia via rede; a principal aplicação almejada pela especificação é a criação de sistemas para vídeo-conferência, mas ela também serve para a transmissão de conteúdo multimídia; por outro lado, não oferece suporte estrito para processamento sincronizado nem para latências abaixo do máximo adequado para sistemas de vídeo-conferência.

11 cronograma esperado de trabalho

atividade/período	set	out	nov	dez	jan	fev	mar
levantamento bibliográfico, leitura e redação dos capítulos introdutórios	X	X	X				
desenvolvimento do sistema			X	X	X		
testes, experimentos e redação final					X	X	
entrega e defesa							X

12 bibliografia

12.1 livros e artigos

rede e multimídia

- Quinn, L. B.; Russell, R. G. *Fast Ethernet*. John Wiley & Sons, 1997.
- Stevens, W. R. *TCP/IP Illustrated*. Reading, MA: 1997. v. 1: the protocols.
- Canosa, J. Fundamentals of FireWire. *Embedded Systems Programming*, vol. 12, nº 6, junho de 1999. Disponível em: <`http://www.embedded.com/1999/9906/9906feat2.htm`> Acesso em: 18 ago. 2002.
- Wiffen, P. An introduction to mLAN, part 1. *Sound on Sound*, agosto, 2000. Disponível em: <`http://www.sospubs.co.uk/sos/aug00/articles/mlan.htm`> Acesso em: 18 ago. 2002.
- Wiffen, P. An introduction to mLAN, part 2. *Sound on Sound*, setembro, 2000. Disponível em: <`http://www.sospubs.co.uk/sos/sep00/articles/mlan.htm`> Acesso em: 18 ago. 2002.
- Schulzrinne, H.; Casner, S.; Frederick, R.; Jacobson, V. *RTP: A Transport Protocol for Real-Time Applications (RFC1889)*. Disponível em: <`http://www.ietf.org/rfc/rfc1889.txt`> Acesso em: 18 ago. 2002.
- Schulzrinne, H. *RTP Profile for Audio and Video Conferences with Minimal Control (RFC1890)*. Disponível em: <`http://www.ietf.org/rfc/rfc1890.txt`> Acesso em: 18 ago. 2002.
- Andrews, M. *The X.org Audio Project*: Feature Specification. Disponível em: <`http://mediaapplicationserver.net/mas/documents/files/x-audio-feature-specification_xorg-audio_1_3_std.pdf`> Acesso em: 18 ago. 2002.

programação

- Gamma, E. et al. *Design Patterns*: elements of reusable object-oriented software. Reading: Addison-Wesley, 1995.
- Henning, M.; Vinoski, S. *Advanced CORBA programming with C++*. Addison-Wesley, 2001.

- Silberschatz, A.; Galvin, P. B. *Sistemas operacionais: conceitos*. São Paulo: Prentice Hall, 2000.

- Szyperski, C. *Component Software*: Beyond Object-Oriented Programming. Addison-Wesley, 1997.

áudio e áudio digital

- D'Antonio, P. *Minimizing acoustic distortion in project studios*. Disponível em: <`http://www.rpginc.com/cgi-bin/byteserver.pl/news/library/PS_AcD.pdf`> Acesso em: 18 ago. 2002.

- Mac, J.; Gallagher, M. High-definition audio. *Keyboard*, junho 1998, p. 34-42.

- Burger, J. Digidesign Pro Tools|24 MIX and MIXplus (Mac/Win NT). *Electronic Musician*, julho, 2000. Disponível em: <`http://emusician.com/ar/emusic_digidesign_pro_tools_2/index.htm`> Acesso em: 18 ago. 2002.

- Digidesign. *Digidesign Plug-Ins Guide:* for Macintosh and Windows. Disponível em: <`http://www.digidesign.com/support/docs/Digidesign_Plug-Ins_Guide.pdf`> Acesso em: 18 ago. 2002.

- Apple Computer. *Audio and MIDI on Mac OS X*. Disponível em: <`http://developer.apple.com/audio/pdf/coreaudio.pdf`> Acesso em: 18 ago. 2002.

- Phillips, D. *Linux music & sound*. São Francisco: No starch press, 2000.

- Roads, C. *The computer music tutorial*. Cambridge: MIT press, 1999.

tempo real

- Gallmeister, B. O. *POSIX. 4: Programming for the real world*. O'Reilly & Associates, 1995.

- Brandt, E.; Dannenberg, R. B. *Low-latency music software using off-the-shelf operating systems*. Disponível em: <`http://www-2.cs.cmu.edu/~rbd/papers/latency98/latency98.htm`> Acesso em: 18 ago. 2002.

- MacMillan, K.; Droettboom, M.; Fujinaga, I. "Audio latency measurements of desktop operating systems". *Proceedings of the International Computer Music Conference*, 2001, p. 259-262. Disponível em: <http://gigue.peabody.jhu.edu/~ich/research/icmc01/latency-icmc2001.pdf> Acesso em: 18 ago. 2002.

- Phillips, D. *Low latency in the linux kernel*. Disponível em: <http://linux.oreillynet.com/pub/a/linux/2000/11/17/low_latency.html?page=1> Acesso em: 18 ago. 2002.

- Williams, C. *Linux scheduler latency*. Disponível em: <http://www.linuxdevices.com/articles/AT8906594941.html> Acesso em: 18 ago. 2002.

- Dankwardt, K. *Comparing real-time Linux alternatives*. Disponível em: <http://www.linuxdevices.com/articles/AT4503827066.html> Acesso em: 18 ago. 2002.

12.2 sítios de interesse na Internet

- O sítio da *Free Software Foundation*: <http://www.fsf.org/> Acessado em: 18 ago. 2002.

- O sítio da *Open Source Initiative*: <http://www.opensource.org/> Acessado em: 18 ago. 2002.

- O sítio do grupo de trabalho responsável pelo desenvolvimento dos padrões POSIX, *The Portable Application Standards Committee*: <http://www.pasc.org/> Acessado em: 18 ago. 2002.

- Uma ampla listagem de aplicativos para áudio e música, mantida por Dave Phillips, compatíveis com Linux: <http://www.sound.condorow.net> Acesso em: 18 ago. 2002.

- O *patch* de baixa latência de Ingo Molnar, desenvolvido primariamente para as versões 2.2.X do núcleo do Linux: <http://people.redhat.com/mingo/lowlatency-patches/> Acesso em: 18 ago. 2002.

- O *patch* de baixa latência de Andrew Morton, desenvolvido para as versões 2.4.X do núcleo do Linux: <http://www.zip.com.au/~akpm/linux/schedlat.html> Acesso em: 18 ago. 2002.

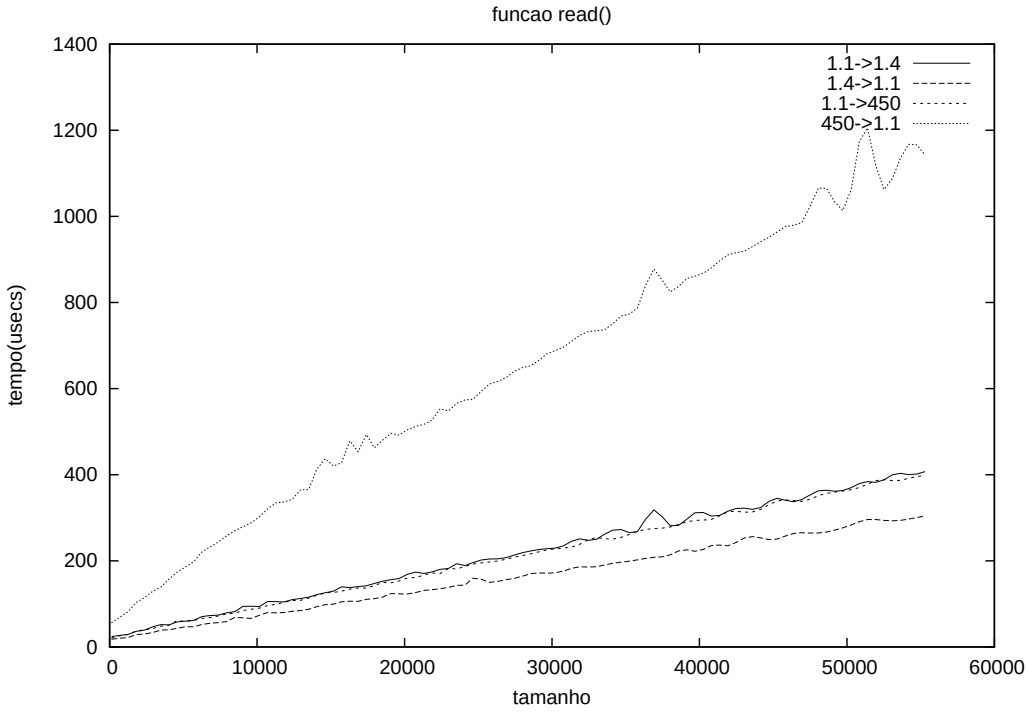
- O *patch* de Robert Love para permitir a preempção do código dentro do núcleo do Linux: <http://kpreempt.sourceforge.net/> Acesso em: 18 ago. 2002.

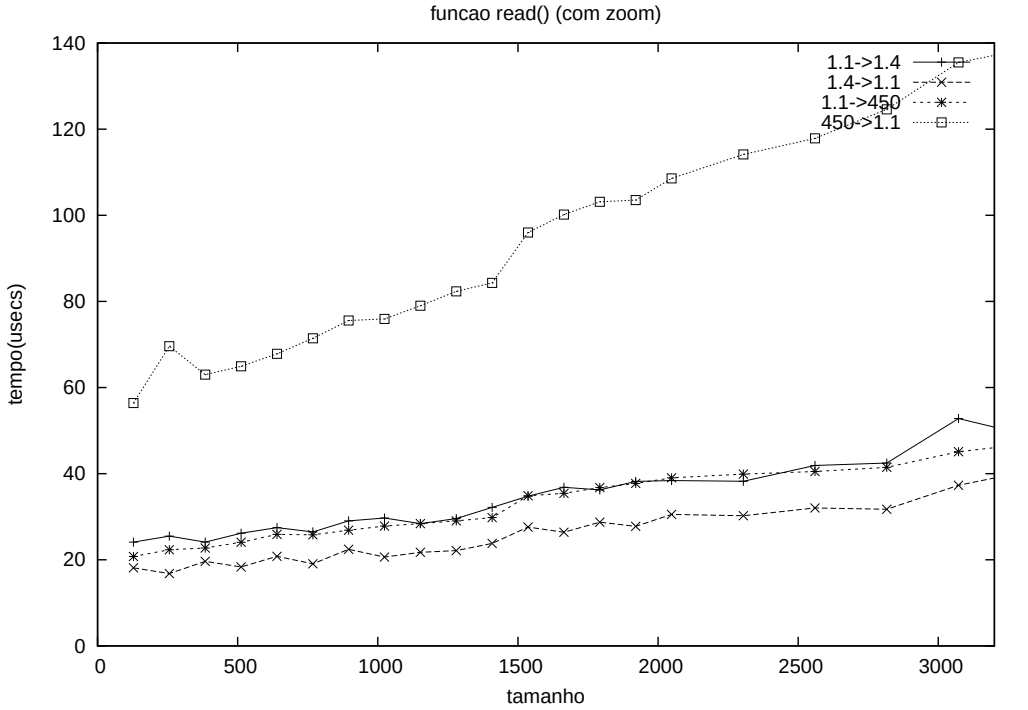
- O programa latencytest: <http://www.gardena.net/benno/linux/audio/> Acesso em: 18 ago. 2002.
- O sistema de controladores de áudio ALSA: <http://www.alsa-project.org/> Acesso em: 18 ago. 2002.
- A especificação VST: <http://www.steinberg.net/developers/VST2SDKAbout.phtml> Acesso em: 18 ago. 2002.
- A especificação ASIO: <http://www.steinberg.net/developers/ASIO2SDKAbout.phtml> Acesso em: 18 ago. 2002.
- A especificação LADSPA: <http://www.ladspa.org/> Acesso em: 18 ago. 2002.
- O sistema JACK: <http://jackit.sf.net/> Acesso em: 18 ago. 2002.
- O projeto MAS (Media Application Server): <http://mediaapplicationserver.net/> Acesso em: 18 ago. 2002.
- O projeto aRts: <http://www.arts-project.org/> Acesso em: 18 ago. 2002.
- O projeto gstreamer: <http://gstreamer.net/> Acesso em 18 ago. 2002.
- O ORB TAO: <http://www.cs.wustl.edu/~schmidt/TAO.html> Acesso em: 18 ago. 2002.
- Informações preliminares sobre o sistema VST System Link: <http://www.steinberg.net/products/up/vstsl/index.phtml?sid=06487939&id=1003>, <http://www.steinberg.net/products/up/vstsl/applications.phtml?sid=06487939&id=100301> Acessados em: 18 ago. 2002.
- Um levantamento dos problemas e objetivos que motivaram o desenvolvimento do sistema JACK (sob o nome LAAGA): <http://www.eca.cx/laaga> Acesso em: 18 ago. 2002.
- O programa ardour, um dos mais ambiciosos editores de áudio para Linux, compatível com os sistemas JACK e LADSPA: <http://ardour.sf.net/> Acesso em: 18 ago. 2002.

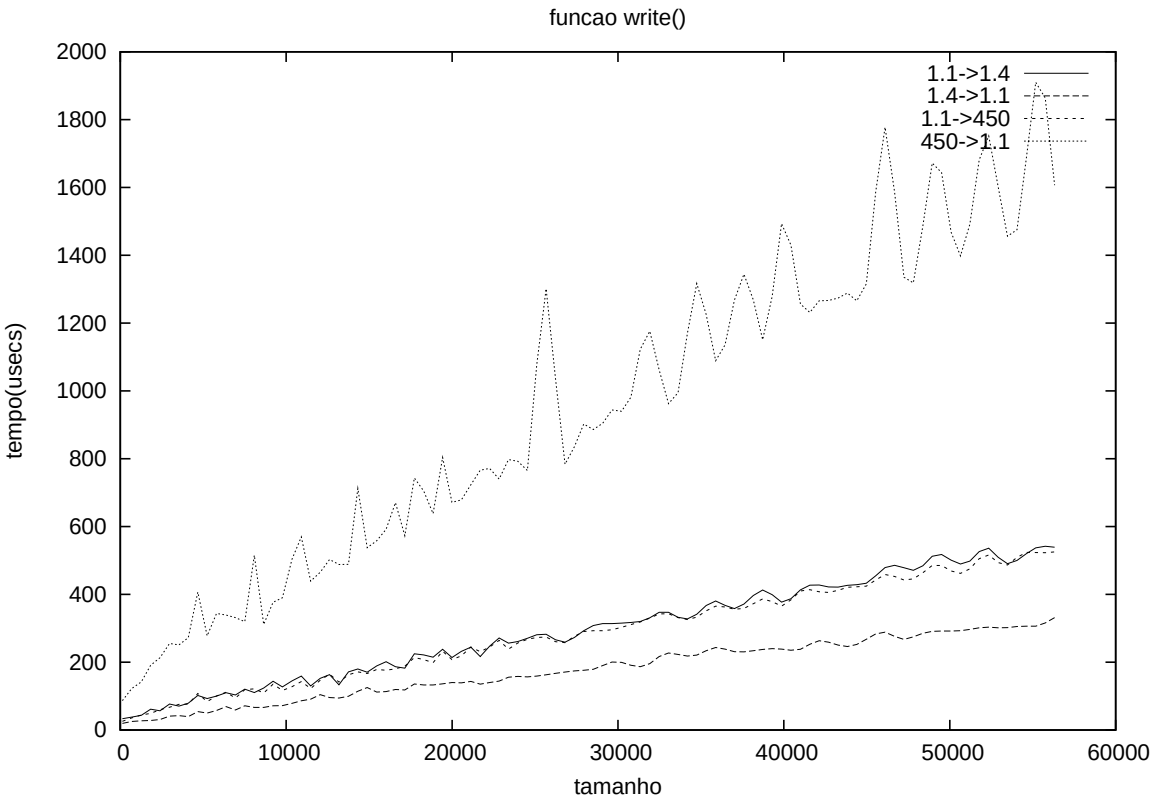
13 apêndice A

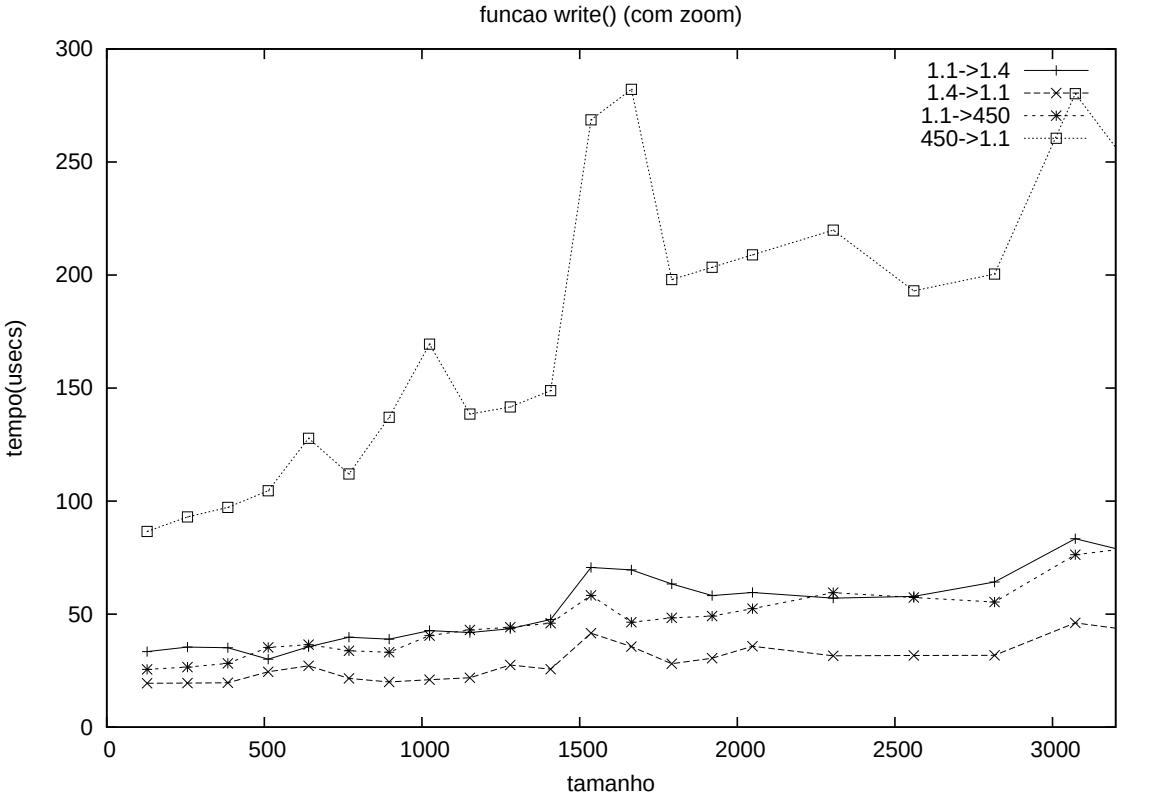
Aqui estão reproduzidos gráficos relativos ao desempenho da rede de acordo com os testes realizados. Vários deles representam diversas medições, em máquinas diferentes ou em testes diferentes, de acordo com a legenda. Cada teste envolve a comunicação entre duas máquinas, uma operando como mestre e outra como escrava; isso foi indicado nas legendas com números que representam as velocidades das CPUs das máquinas e setas indicando a relação mestre/escravo (por exemplo, 1.1->1.4 indica que o teste foi realizado com as máquinas com processadores de 1.1GHz e 1.4GHz, e que a máquina de 1.1GHz operou como mestre). Como os tempos de resposta e a comunicação com o hardware tem efeito maior sobre os resultados com blocos de dados menores, os gráficos foram feitos em duas versões, uma completa (onde podemos verificar que a custo da transmissão e recepção de dados cresce de forma aproximadamente linear em todos os testes) e outra mostrando apenas os resultados para blocos de até 3200 bytes (como um "zoom" sobre esses dados, que compreendem os casos onde um bloco de dados é menor que uma, duas ou três vezes 1500 bytes, que é tamanho máximo do pacote permitido em uma rede Ethernet). Os gráficos foram criados com o programa gnuplot, alguns usando o recurso de curvas suavizadas com o algoritmo cspline.

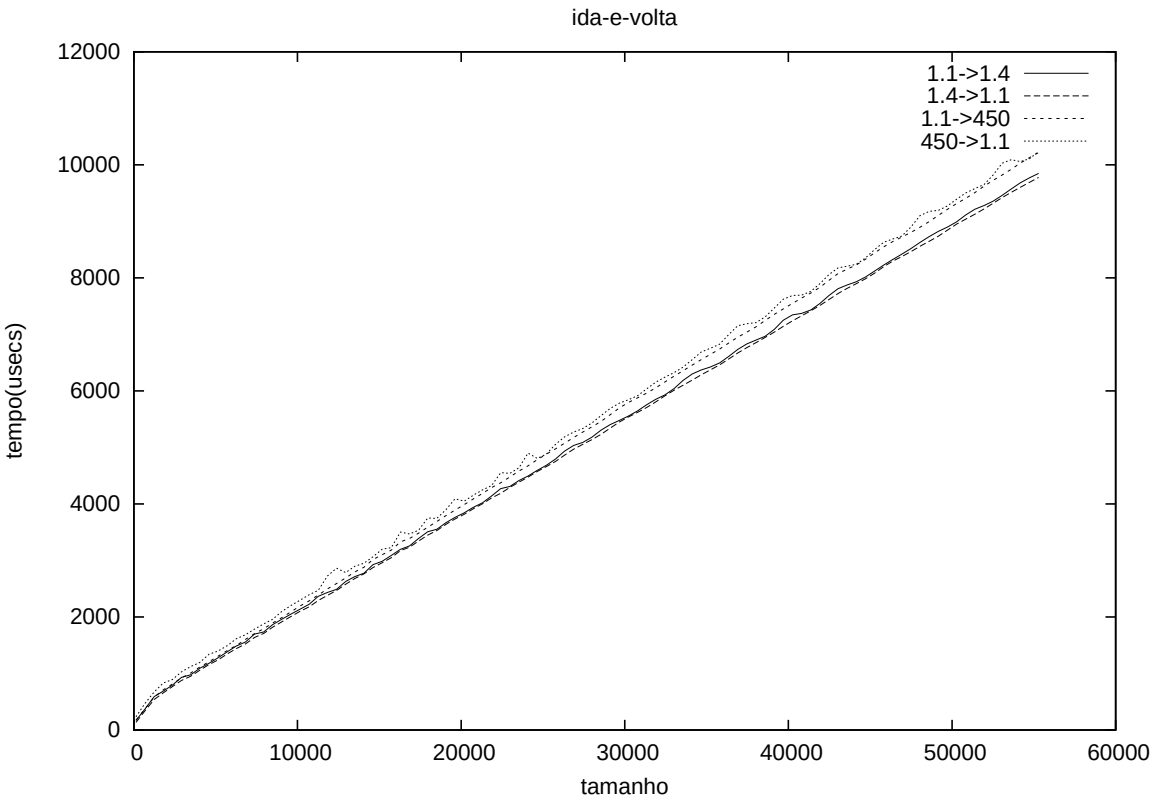
- Tempos para execução das funções `read()` e `write()`, tempo de ida-e-volta dos dados e tempo estimado dos dados "no fio":

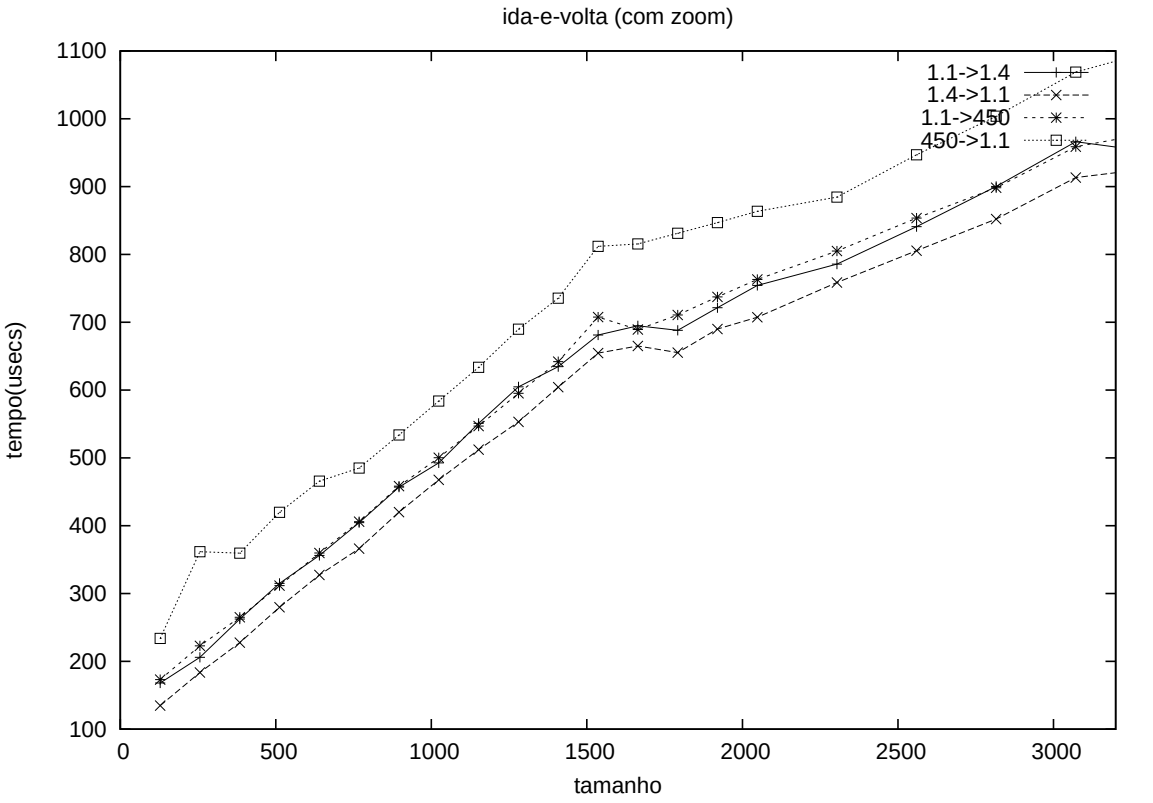


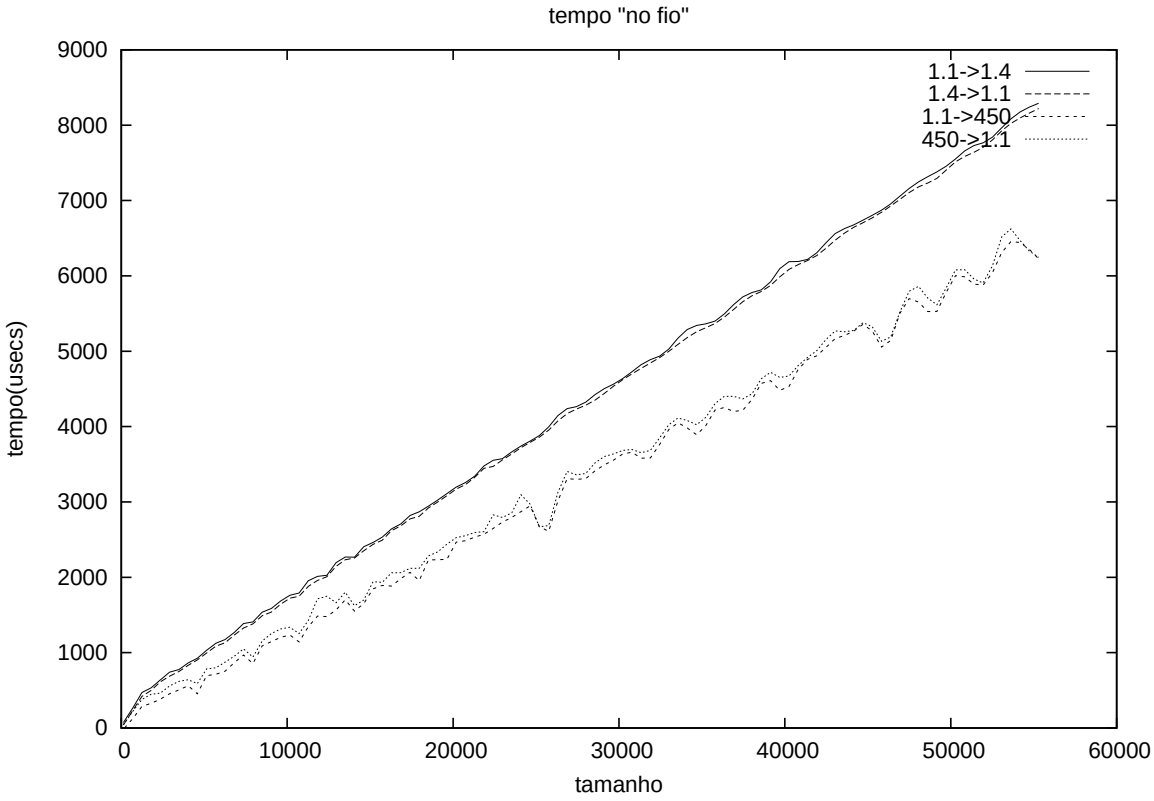


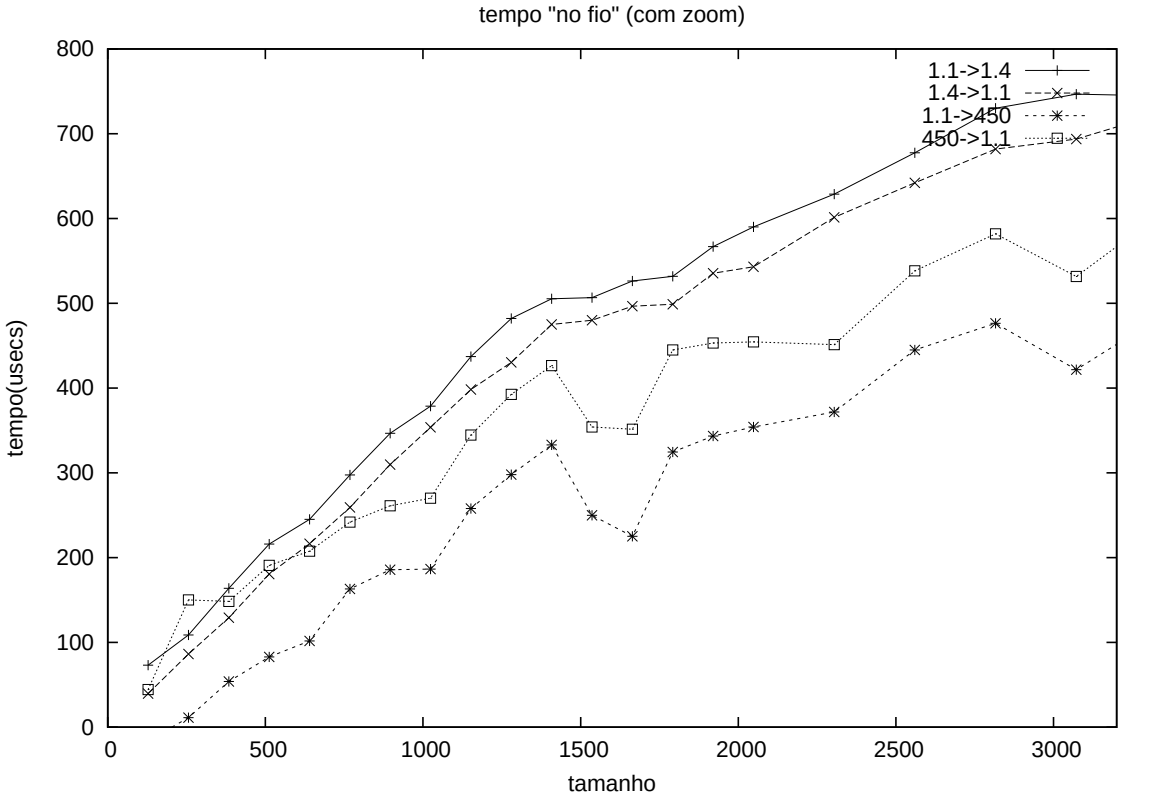

funcao write()
tempo(usecs)
tamanho
1.1->1.4
1.4->1.1
1.1->450
450->1.1
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
10000
20000
30000
40000
50000
60000


funcao write() (com zoom)
tempo(usecs)
tamanho
1.1->1.4
1.4->1.1
1.1->450
450->1.1
0
50
100
150
200
250
300
0
500
1000
1500
2000
2500
3000

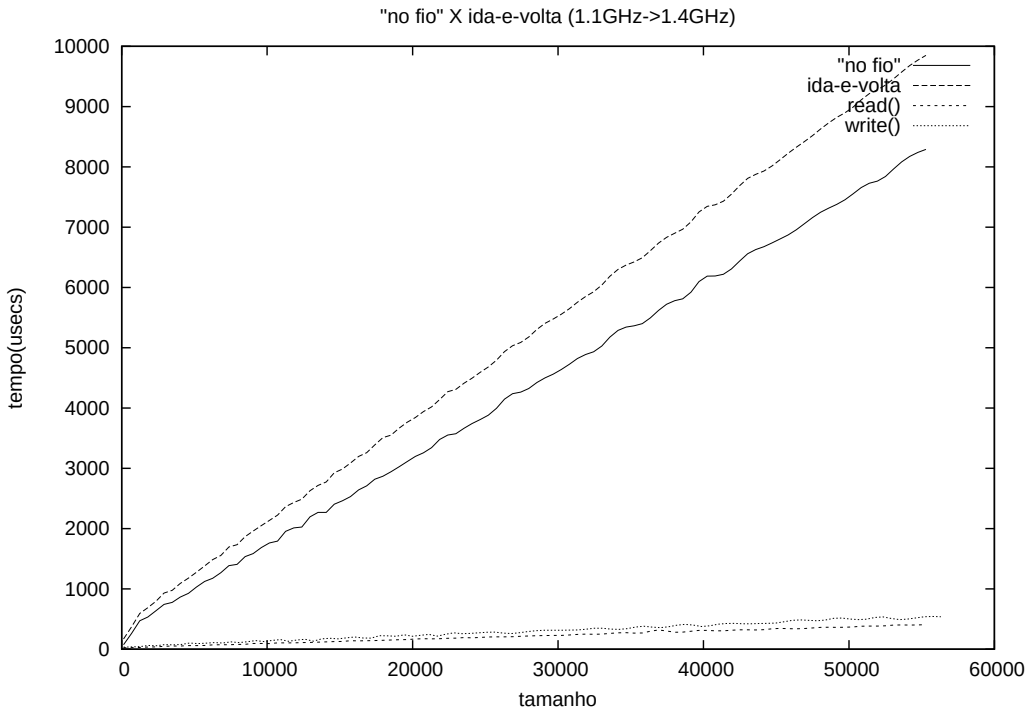

ida-e-volta
1.1->1.4
1.4->1.1
1.1->450
450->1.1
tempo(usecs)
tamanho
0
2000
4000
6000
8000
10000
12000
0
10000
20000
30000
40000
50000
60000

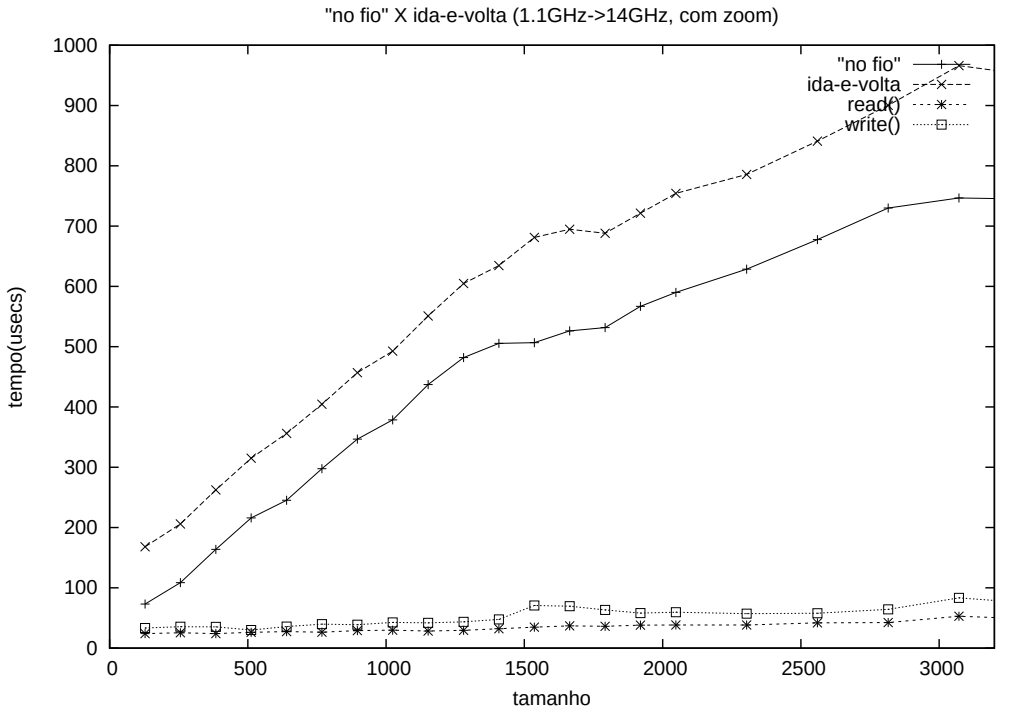

ida-e-volta (com zoom)
1.1->1.4
1.4->1.1
1.1->450
450->1.1
tempo(usecs)
tamanho
100
200
300
400
500
600
700
800
900
1000
1100
0
500
1000
1500
2000
2500
3000

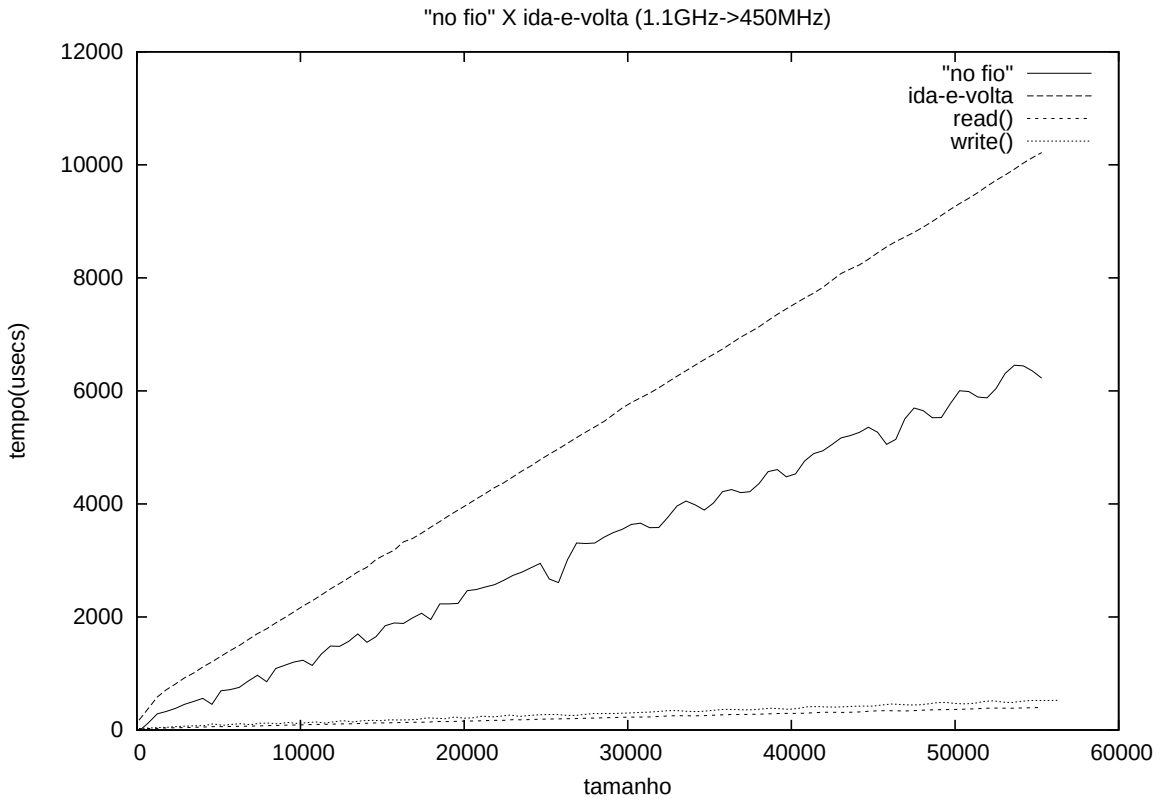

tempo "no fio"
tempo(usecs)
tamanho
1.1->1.4
1.4->1.1
1.1->450
450->1.1
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
0
10000
20000
30000
40000
50000
60000

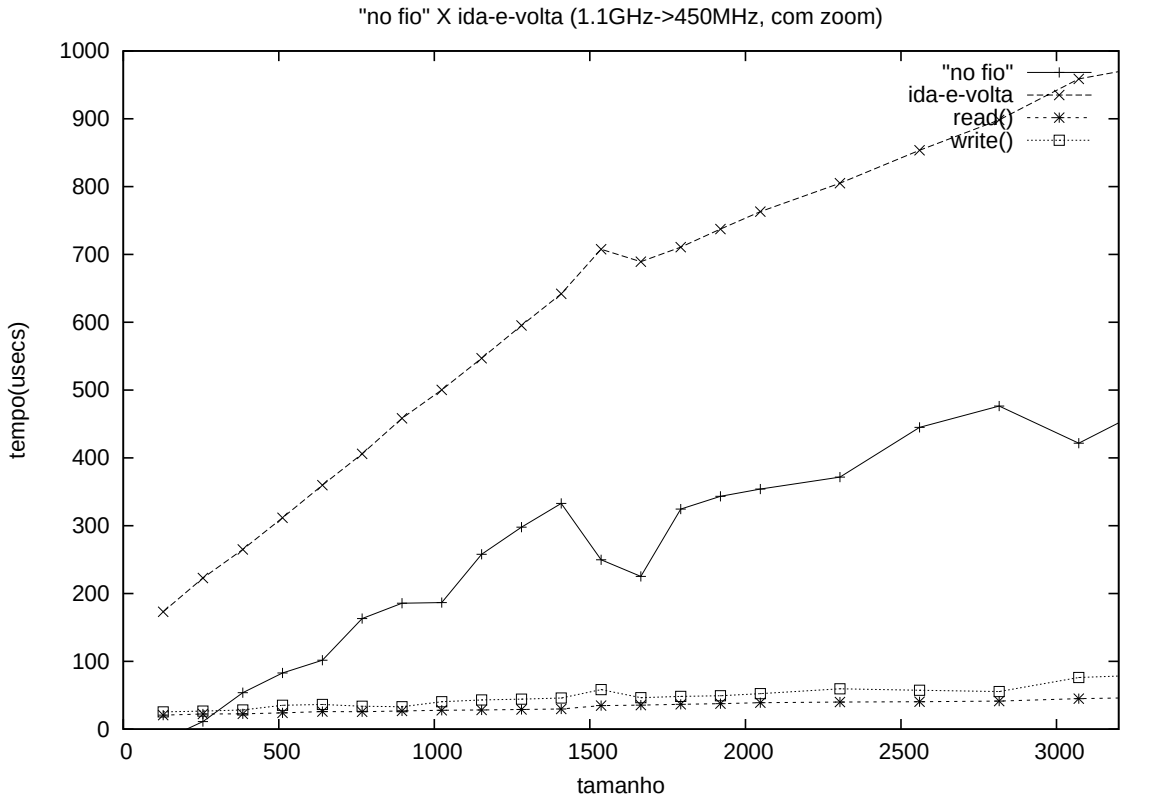

tempo "no fio" (com zoom)
tempo(usecs)
tamanho
1.1->1.4
1.4->1.1
1.1->450
450->1.1
0
100
200
300
400
500
600
700
800
0
500
1000
1500
2000
2500
3000

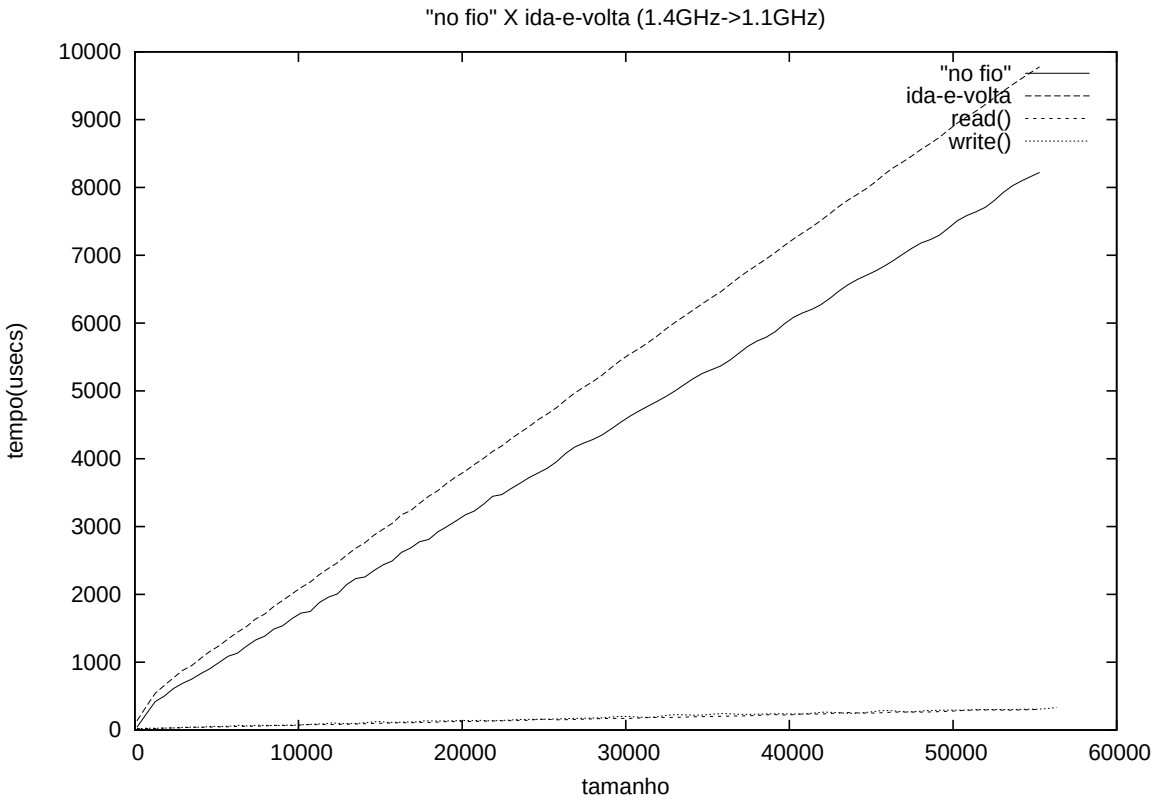
- Comparação entre as diversas medições:

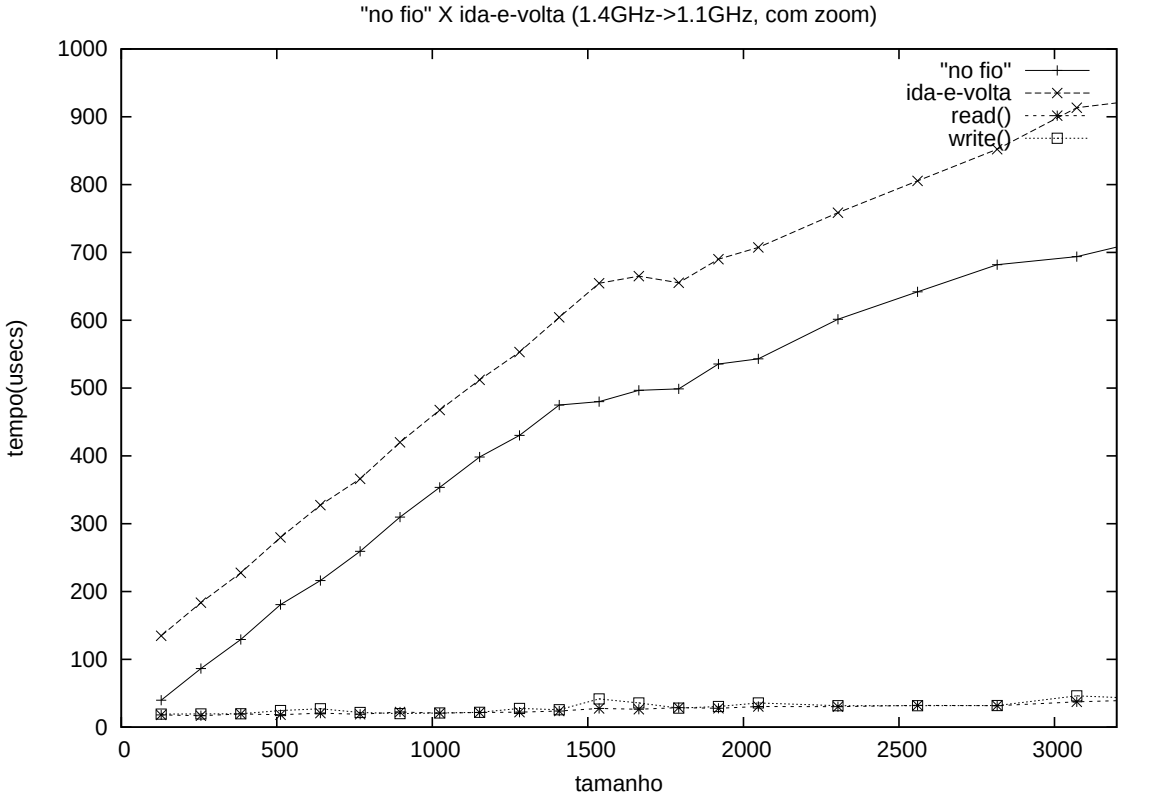


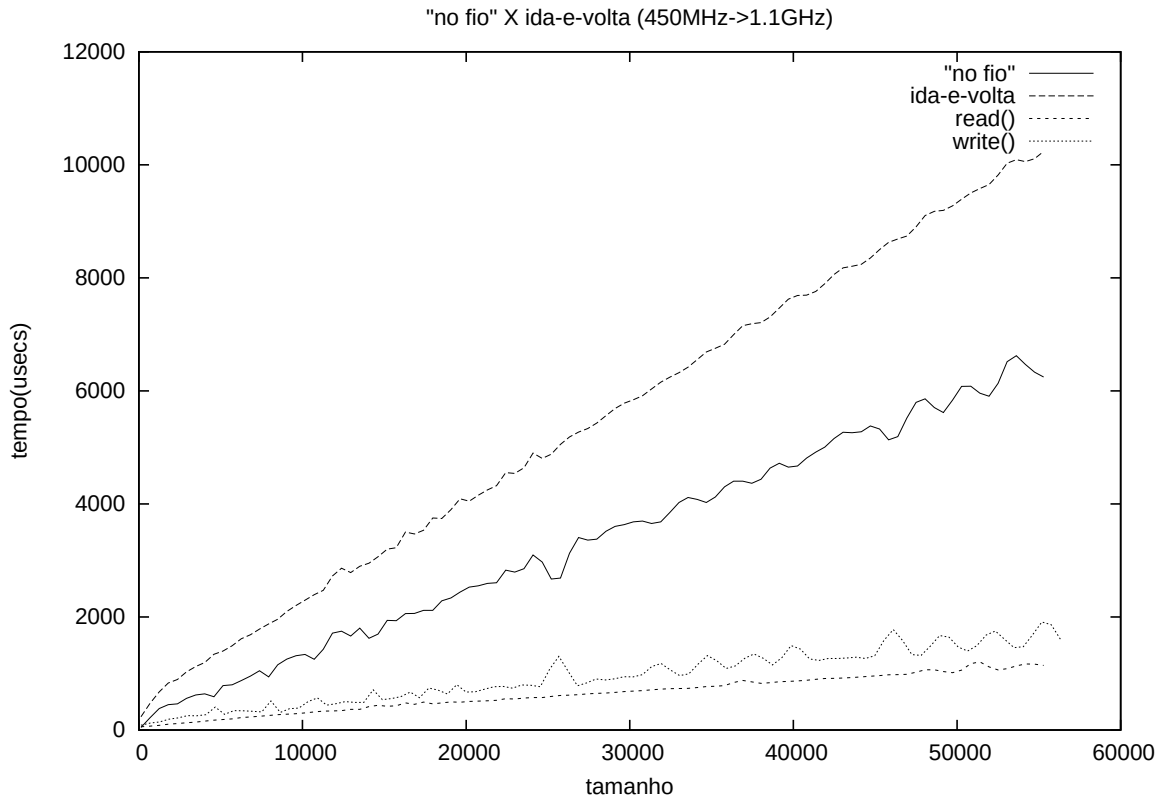


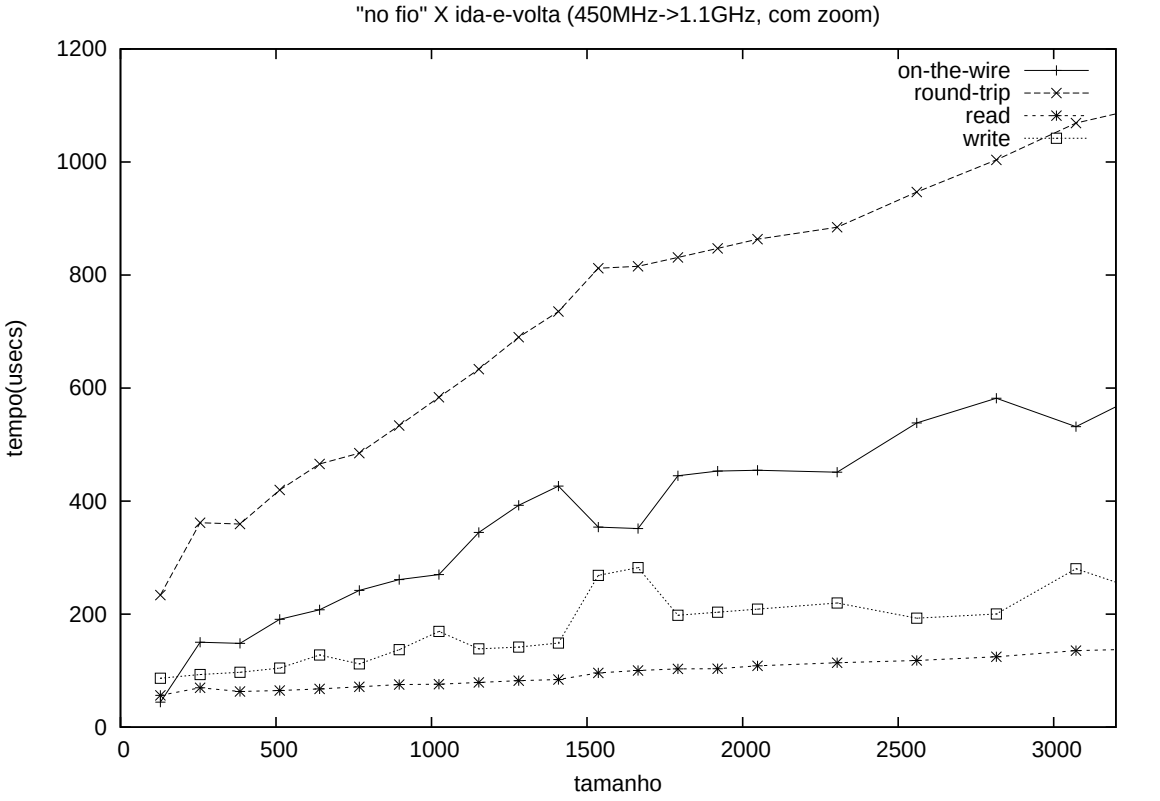

"no fio" X ida-e-volta (1.1GHz->450MHz)
"no fio"
ida-e-volta
read()
write()
tempo(usecs)
tamanho
0
2000
4000
6000
8000
10000
12000
0
10000
20000
30000
40000
50000
60000


"no fio" X ida-e-volta (1.1GHz->450MHz, com zoom)
"no fio"
ida-e-volta
read()
write()
tempo(usecs)
tamanho
0
100
200
300
400
500
600
700
800
900
1000
0
500
1000
1500
2000
2500
3000


"no fio" X ida-e-volta (1.4GHz->1.1GHz)
"no fio"
ida-e-volta
read()
write()
tempo(usecs)
tamanho
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
10000
20000
30000
40000
50000
60000


"no fio" X ida-e-volta (1.4GHz->1.1GHz, com zoom)
"no fio"
ida-e-volta
read()
write()
tempo(usecs)
tamanho
0
100
200
300
400
500
600
700
800
900
1000
0
500
1000
1500
2000
2500
3000


"no fio" X ida-e-volta (450MHz->1.1GHz)
"no fio"
ida-e-volta
read()
write()
tempo(usecs)
tamanho
0
2000
4000
6000
8000
10000
12000
0
10000
20000
30000
40000
50000
60000


"no fio" X ida-e-volta (450MHz->1.1GHz, com zoom)
on-the-wire
round-trip
read
write
tempo(usecs)
tamanho
0
200
400
600
800
1000
1200
0
500
1000
1500
2000
2500
3000