Ciências Exatas e Tecnológicas

Curso de Graduação em Engenharia da Computação

Estudo de performance de requisições por meio de ponteiros remotos em protocolo de camada de aplicação

Autor: João Pedro Bezerra Socas

Orientador: Prof. Me. Gabriel Baião

Americana – SP

2023

Resumo

Pretende-se avaliar o ganho ou a perda de desempenho através do uso de uma técnica em que ponteiros para funções são previamente permutados entre processos remotos e posteriormente utilizados em cabeçalhos de requisições, evitando assim uma rotina condicional para identificar qual procedimento responde ao o que foi requerido. Por se tratar de uma otimização pouco impactante, será apenas de valor em aplicações com alto consumo de recursos que necessitam de baixa ou baixíssima latência.

Expressões chave: comunicação entre processos, chamada de procedimento remoto, otimização.

Abstract

The intent is to evaluate the gain or loss of performance through the use of a technique that function pointers are permutated in advance between remote process and then used on request headers, avoiding this way, a conditional routine to identify which procedure is the response for what was requested. This being a low impact optimization, it will be most valuable in high resource consumption applications that need low or very low latency.

Key phrases: inter-process communication, remote procedure call, optimization.

Sumário

Lista de Símbolos do Artigo Científico

Grandeza	**Nome**	**Símbolo**
tempo	segundo	s
tempo	milissegundo	ms
tempo	microssegundo	µs
resolução	quatro K	4k
cadencia	quadros por segundo	fps

(Tabela 1) – lista de símbolos do artigo cientifico.

Fonte: (o autor)

Capítulo 1 – Introdução

Justificativa

Tomando como exemplo um conjunto de aplicações que permeiam o funcionamento de um óculos de realidade virtual conectado por meio sem fio a um computador pessoal, onde se pode observar a transmissão de até 120 (cento e vinte) imagens 4K para cada olho por segundo, todas renderizadas em tempo real com base nos dados de sensores do óculos que também trafegaram por até dois trechos sem fio (óculos a roteador e roteador a computador); podemos perceber que se trata de uma tecnologia que depende de capacidades computacionais que estão no limiar do alcance de empresas e usuários comuns. Sendo assim, até uma pequena economia de recursos significará a possibilidade de uma experiência mais responsiva ou de uma qualidade melhor de imagem fornecida pela aplicação final.

Objetivo

Averiguar se o emprego dessa técnica é de fato uma otimização, ou uma "*pessimização*", e mensurar o impacto positivo ou negativo.

Alcances e Limitações

Apenas serão avaliadas questões de desempenho e em uma única arquitetura computacional com um mesmo sistema operacional. Outras arquiteturas e diferentes sistemas operacionais poderão apresentar resultados diferentes. Não serão levadas em consideração, questões de segurança da informação.

Metodologia

Para o presente trabalho foram realizadas as seguintes etapas:

- Pesquisa bibliográfica a respeito de otimizações parecidas em sites, artigos e fóruns online.
- Criação de uma biblioteca para aferir o tempo de execução de cada iteração
- Criação de um projeto de exemplo com conceito simplificado para análise inicial com rotinas de controle e rotinas modificadas.
- Criação de um projeto de exemplo com simulação do uso real da técnica com rotinas de controle e rotinas modificadas.

- Acumulação de resultados de medições de tempo das rotinas de requisições.
- Cálculo da diferença de eficiência entre o controle e as modificações.

Capítulo 2 – Revisão da Literatura

2.1 – Fundamentação Teórica

A cláusula "*switch*"

A cláusula "*switch*" "*transfere o controle para uma de várias rotinas, dependendo do valor da condição." (*cppreference.com, tradução nossa), sendo assim, quando um sistema recebe uma requisição, e precisa transferir o controle para uma determinada rotina de resposta de acordo com o que foi requisitado, esta clausula se encaixa como uma resolução adequada.

Entretanto existe um movimento na comunidade de programadores que defende a refatoração de algoritmos que utilizam "*switch"* para soluções baseadas em polimorfismo.

"Switch" contra polimorfismo

O polimorfismo se caracteriza por "*duas ou mais classes distintas têm métodos de mesmo nome, de forma que uma função possa utilizar um objeto de qualquer uma das classes polimórficas, sem necessidade de tratar de forma diferenciada conforme a classe do objeto*" (Wikipedia), ou seja, ao invés de um parâmetro utilizado para a decisão, ter-se-ia um objeto que se comporta diferentemente conforme sua classe.

Essa abordagem tem os seguintes benefícios:

> - Essa técnica adere ao princípio de "Diga-Não-Pergunte": ao invés de perguntar a um objeto seu estado e tomar ações sobre a resposta, é muito mais fácil dizer ao objeto o que precisa feito e deixa-lo decidir como fazê-lo.
> - Remoção de código duplicado. Você se livra de várias condicionais quase que idênticas.
> - Se você precisar de uma nova variação de execução, tudo que você precisa fazer é adicionar uma nova subclasse sem tocar o código existente (Princípio do Aberto/Fechado).
>
> REFACTORING GURU (tradução nossa)

No entanto, quando se recebe dados de uma requisição remota, para que o polimorfismo funcione nestes casos, seria necessário que no cabeçalho viessem todas as informações para a

recriação do objeto, assim como toda a rotina de construí-lo, tornando essa solução totalmente inviável.

Ponteiros para funções.

Tendo em vista os benefícios perdidos pela impossibilidade de uso de polimorfismo, uma outra possibilidade de substituir a clausula switch seriam os ponteiros para funções:

> "[...] também chamados de ponteiros de sub-rotina, ou ponteiro para procedimentos, é um ponteiro referenciando um código executável, e não dados. A desreferenciação do ponteiro produz a função referenciada, a qual pode ser invocada e receber argumentos assim como uma chamada de função normal." WIKIPEDIA (tradução nossa).

Para substituir a clausula "*switch"* por ponteiros para funções, ao invés de receber um valor que definirá o resultado da condição, o que receber-se-á será um ponteiro para o procedimento que está sendo requisitado. Dessa forma todos os benefícios da refatoração com o polimorfismo também são resgatados.

Requisições remotas por ponteiros para funções

Não foi encontrado no momento da confecção deste artigo, algum exemplo de uso ou outro documento que descreva um algoritmo que se assemelha à técnica descrita neste artigo.

Tabela de saltos

O principal mecanismo de otimização que os compiladores usam para clausulas "*switch*" é a tabela de saltos. Um dos modos de implementação mais eficientes de uma tabela de saltos é "[...] usar uma lista de ponteiros para obter o endereço da função desejada" (ACADEMIC ACCELERATOR). Traz-se a tona então, que usar diretamente o ponteiro provido pelo cabeçalho da requisição, já é o resultado que seria obtido pelo código otimizado, dando indícios de que além da possibilidade de manter os benefícios do algoritmo polimórfico, também se trata de uma otimização válida.

Capítulo 3 – Experimento

3.1 – O Projeto

Será criada uma aplicação cliente-servidor para simular requisições sendo feitas tanto pelo método tradicional quanto pelo método a ser testado. Essa aplicação consistirá em dois executáveis, um simulando um cliente e o outro um servidor. Serão desenvolvidas em C++ e compilados com MSVC. Além disso será utilizada a versão do Boost da biblioteca ASIO para a criação de sockets, conexão e broadcast.

A parte que representará o servidor, justamente antes e depois das seções que irão diferir entre os modos de resposta de uma requisição, será gravado os horários dos dois momentos, permitindo assim aferir quanto tempo foi gasto em cada sessão.

Por fim, o servidor exibirá o cálculo da média do tempo gasto por inúmeras requisições feitas pelo cliente em cada modo, mostrando assim o percentual de diferença de tempo.

3.2 – Detalhamento

3.2.1 – Servidor

O servidor inicialmente fará o **"***broadcast***"** do próprio IP permitindo que um cliente que escute o broadcast peça uma conexão. A conexão terá **três** "*sockets*" para simplificar o filtro dos dados: um para averiguar os status da conexão, outro para as requisições que serão respondidas por um "*switch*" e outro para requisições que serão respondidas pela execução do ponteiro.

Ao ser estabelecida uma conexão com um cliente, o servidor enviará pelo socket de requisições por ponteiro, uma estrutura com todos os ponteiros referentes às funções que simularão as diferentes requisições, o quais serão usados pelo cliente como cabeçalho para requirir.

Por fim, quando receber uma requisição, a depender de qual "*socket*" for usado, o servidor pelo meio da implementação relacionada iniciará o procedimento requerido. Será nesse estágio mensurado os períodos de tempo gasto entre a finalização da aquisição do buffer recebido pelo servidor e o início da chamada da função requisitada, que é exatamente a parte que difere entre os dois métodos sendo comparados.

3.2.2 – Cliente

O cliente iniciará escutando o canal de "*broadcast*" da rede, e exibirá como opções de conexão os servidores compatíveis que se apresentarem. Com o input do usuário, será

requisitada uma conexão com o endereço de IP referente a escolha, e caso aceita pelo servidor, consumada.

Como descrito no detalhamento do servidor, haverá o "*socket"* de status que por meio de trocas de mensagens temporizadas, manterá atualizada a condição da conexão; também haverão mais dois soquetes, um para requisições que serão respondidas pela execução do ponteiro e outro para as requisições que serão respondidas por um "*switch*". No primeiro momento depois de conectado ao servidor, o cliente aguardará no "*socket"* de requisição por ponteiro, os ponteiros referentes aos endereços dos procedimentos no servidor, que serão usados para compor o cabeçalho das requisições.

Após as prévias etapas realizadas, o cliente apresentará para o usuário opções para realizar as simulações de requisição ao servidor.

3.3 – Código

O código pode ser obtido no seguinte endereço: <github.com/Joao-Socas/PointerBasedRequests>

Capítulo 4 – Resultados obtidos

4.1 – Dados iniciais

Primeiro teste com 100000 (cem mil) interações de 3 distintas requisições.

Teste Nº	Clausula Switch (segundos)	Ponteiro para função (segundos)
1	0,0006963	0,0006709
2	0,0005575	0,0007238
3	0,0005619	0,0006664
4	0,0005373	0,0007113
5	0,0006671	0,0007325
6	0,0005592	0,000666
7	0,0005564	0,000714
8	0,0005875	0,0006811
9	0,0005522	0,0006825
10	0,0005744	0,0006851
Média	0,00058498	0,00069336

Segundo teste com 100000 (cem mil) interações de 26 distintas requisições.

Teste Nº	Clausula Switch (segundos)	Ponteiro para função (segundos)
1	0,0005836	0,0010076
2	0,000549	0,0010396
3	0,0005487	0,0009761
4	0,0005766	0,0009611
5	0,0005661	0,0009612
6	0,0005627	0,0009655
7	0,0006237	0,00096
8	0,0005525	0,0011046
9	0,0005634	0,0009615
10	0,0005637	0,0009604
Média	0,000569	0,00098976

4.2 – Diferença percentual

Primeiro teste com 100000 (cem mil) interações de 3 distintas requisições: as requisições por ponteiro de funções foram 18,53% mais demoradas do que as pela cláusula "*switch"*.

Segundo teste com 100000 (cem mil) interações de 26 distintas requisições: as requisições por ponteiro de funções foram 73,95% mais demoradas do que as pela cláusula "*switch"*.

Capítulo 5 – Conclusão e propostas de trabalhos futuros

5.1 – Considerações iniciais

Independentemente dos resultados obtidos, tomando-se apenas a perspectiva da organização de código e do cumprimento de princípios reconhecidos pela comunidade, a requisição por meio de ponteiros de função se mante válida. Porém, com o objetivo principal de otimização, com vista nos resultados, se tem certeza agora que não é uma técnica viável.

5.2 – Conclusão

Infere-se que o principal fator que explica o menor desempenho das requisições feitas com ponteiro para função, são as operações de “deferenciamento” do ponteiro.

Mesmo mantendo a mesma ordem de tempo (décimos de milésimos a cada cem mil interações), podemos perceber que existe uma queda significativa de desempenho quando se aumentam a quantidade de opções de requisição. Isso paravelmente ocorre devido a impossibilidade de o processador prever que métodos ele usará para manter no cache. O contrário é dito sobre a clausula “*switch*” que manteve seu desempenho em ambos os cenários.

5.3 – Propostas de trabalhos futuros

A principal incógnita que deveria ser avaliada por trabalhos sobre o tema se dá a respeito de questões de segurança que podem vir a surgir provenientes de trafegar um endereço de memória pela rede, mesmo que este esteja dentro do escopo da tabela “*hash*” de um único processo. Pode-se citar por exemplo, utilizando-se de matemática de ponteiros, prever endereço de requisições que não deveriam ser utilizadas. Sem tal estudo, deve-se limitar o uso da técnica apenas para aplicações de redes locais.

Referências bibliográficas

Página dedicada à clausula "*switch*" – cppreference: Disponível em: <https://en.cppreference.com/w/cpp/language/switch>. Acesso em: 20/11/2023.

Artigo sobre "branch *table*" (tabela de desvio) – Academic Accelerator: Disponível em: <academic-accelerator.com/encyclopedia/branch-table>. Acesso em: 20/11/2023.

Exemplo de "*jump table*" (tabela de saltos) – Wikipedia: Disponível em: <en.wikipedia.org/wiki/Branch_table#Jump_table_example_in_C>. Acesso em: 20/11/2023.

Página sobre polimorfismo – Wikipedia: Disponível em: <pt.wikipedia.org/wiki/Polimorfismo_(ciência_da_computação)>. Acesso em: 20/11/2023.

"*Replace Conditional with Polymorphism*" (Substituindo condicionais por polimorfismo) – Refactoring Guru: Disponível em: <refactoring.guru/replace-conditional-with-polymorphism>. Acesso em: 20/11/2023.

Página sobre ponteiro para funções – Wikipedia: Disponível em: <en.wikipedia.org/wiki/Function_pointer>. Acesso em: 20/11/2023.