5
Avaliação de Desempenho

Uma das principais preocupações no uso de linguagens dinâmicas é o seu desempenho, especialmente quando o produto sendo desenvolvido é um middleware de distribuição. Tanto a interpretação de código de uma máquina virtual, quanto a verificação de tipos durante a execução do programa e a manutenção de informações dinâmicas para reflexão computacional implicam em um custo computacional, tanto no uso de memória como em tempo de processamento. Neste capítulo, são apresentadas algumas medições do desempenho do OiL em situações específicas. O objetivo dessas medições é identificar a influência da sobrecarga computacional introduzida pelo uso de Lua em alguns aspectos de desempenho do OiL.

Em todas as medições, o OiL é utilizado exclusivamente com o suporte CORBA. O protocolo LuDO não é adequado para esse tipo de avaliação, pois apresenta um desempenho muito inferior aos protocolos de CORBA. Além disso, o uso do suporte CORBA permite a comparação com implementações de middleware de qualidade que também oferecem esse suporte. As medições são feitas através de um conjunto de *micro-benchmarks*. Os mesmos experimentos são aplicados a outras implementações de CORBA em linguagens menos dinâmicas como Java e C++. Esses experimentos funcionam como um *benchmark* para comparação do desempenho do OiL com outras implementações similares. As versões das implementações utilizadas nos experimentos são mostradas na figura 5.1.

Figura 5.1: Implementações avaliadas nos experimentos de desempenho.

Identificação	Implementação	Compilador/Interpretador
OiL	OiL 0.5 alpha	Lua 5.1.3
OiLJIT	OiL 0.5 alpha	LuaJIT 1.1.3 (Lua 5.1.2)
JacORB	JacORB 2.3	Sun JRE 1.6.0 (update 7) com JIT
SunJDK	ORB do Sun JDK 1.6	Sun JDK 1.6.0 (update 7) com JIT
Mico	Mico 2.3.12	GCC 3.4.6 (Red Hat 3.4.6-9)
Orbacus	IONA Orbacus 4.3.2	GCC 3.4.6 (Red Hat 3.4.6-9)

Todos os experimentos foram realizados em um conglomerado de 12 computadores de processador Pentium D 3.4Ghz (*dual-core*). Um dos computa-

dores possui 4 GB de memória RAM e é usado para executar o processo sendo avaliado. Todos os demais computadores possuem 2 GB de memória RAM. O sistema operacional das máquinas é o CentOS 4.4 com *kernel* 2.6.9-55.0.2.ELsmp. Todas elas estão conectadas em uma rede Gigabit Ethernet.

Os experimentos são divididos em três categorias. Na primeira são apresentadas medidas de tempo de operações com parâmetros de diferentes tipos definidos pelo padrão CORBA. O objetivo desse experimento é ilustrar o desempenho do mecanismo de codificação e decodificação de valores enviados pela rede, processo denominado *marshaling*. Na segunda categoria, os experimentos avaliam como esse tempo é influenciado pelo número de chamadas sendo realizadas concorrentemente. A última categoria avalia o uso de memória do servidor durante essas chamadas. As seções seguintes descrevem esses experimentos e apresentam os resultados obtidos.

5.1 Desempenho das Chamadas

O objetivo desse experimento é avaliar o tempo da chamada de operação de um objeto remoto, quando são utilizados diferentes tipos de parâmetros. Para tanto, é utilizado um *servant*, que oferece diferentes operações como ilustrado na IDL da figura 5.2. Cada operação recebe um parâmetro de um tipo diferente. O corpo de cada operação é vazio, ou seja, a operação não faz nada. O experimento é composto por dois processos: um processo servidor que cria o *servant* usando uma das implementações de ORB sendo avaliadas e um processo cliente, que realiza as chamadas calculando a duração delas. Ambos processos são escritos inteiramente na linguagem de programação do ORB. O apêndice A apresenta a implementação desses testes de desempenho em maior detalhe.

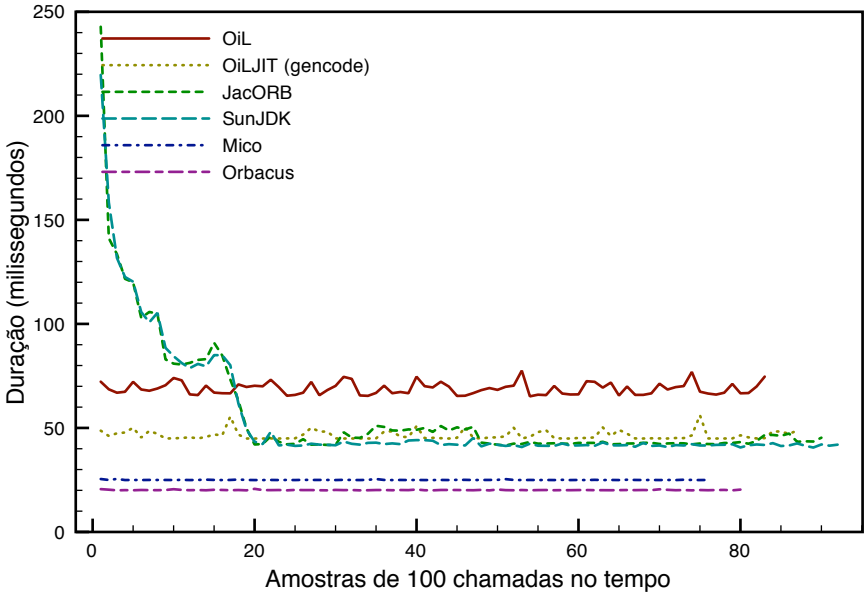
As medidas apresentadas são a duração de 100 chamadas consecutivas da operação sendo avaliada. Todos os valores apresentados nesta seção são valores médios de 30 medições consecutivas, ou seja, totalizando três mil chamadas. O desvio padrão dos valores medidos são apresentados como barras de erro. Essa medição é feita após um período de iniciação e estabilização do seu comportamento. Esse período de estabilização é necessário para evitar que otimizações como uso de *caches* para manter valores recalculados com freqüência contribuam negativamente nos experimentos. A figura 5.3 mostra o padrão típico das medidas obtidas durante o período de estabilização. Nesse gráfico, cada ponto da curva representa a duração de uma amostra de 100 chamadas consecutivas no tempo. As chamadas do OiL apresentam uma variação significativa, porém limitada a intervalo constante, desde as

Figura 5.2: IDL da interface utilizada nos experimentos de desempenho.

```
1  module ORBTests {
2    exception EmptyExcept {};
3    enum TypeEnum { /* ... */ };
4    struct FullStruct {
5      unsigned short ushort_value;
6      string string_value;
7      octet octet_value;
8      short short_value;
9      long long_value;
10     double double_value;
11     char char_value;
12     float float_value;
13     TypeEnum enum_value;
14     boolean boolean_value;
15     unsigned long ulong_value;
16   };
17   typedef sequence<FullStruct> structSeq;
18   typedef sequence<Object> objrefSeq;
19   interface Sender {
20     void send_void();
21     void send_objref(in Object value);
22     void send_objrefs(in objrefSeq value);
23     void send_structs(in structSeq value);
24     void raise_empty() raises (EmptyExcept);
25   };
26 };
```

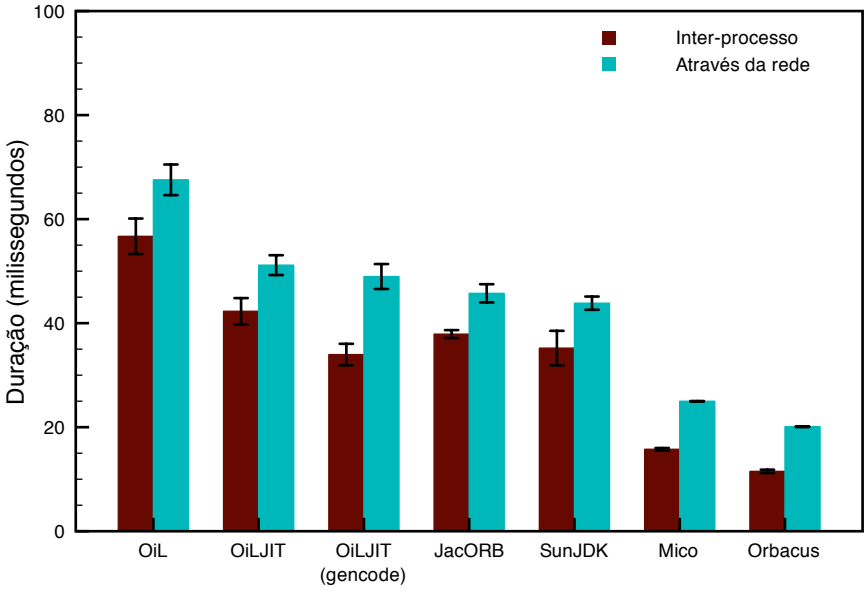

primeiras chamadas. Uma possível razão para essa variação é o coletor de lixo incremental de Lua, que executa automaticamente durante a execução da aplicação para liberar a memória que não está mais sendo usada. Ambos os ORBs em Java apresentam uma duração muito elevada das chamadas iniciais, que vão progressivamente sendo realizadas de forma mais rápida, atingindo uma duração constante apenas depois de duas mil chamadas.

Figura 5.3: Duração de amostras de 100 chamadas durante a estabilização do ORB.



A figura 5.4 mostra a duração de 100 chamadas sem parâmetro (`void`), tanto quando o processo servidor e o processo cliente residem na mesma máquina utilizando comunicação inter-processo, quanto quando eles residem em máquinas diferentes conectadas através de uma rede. As implementações em C++ são as mais rápidas, pois não mantém informações dinâmicas e executam instruções de máquina, sem interpretação. O OiL com interpretador padrão de Lua, apresenta o pior tempo, chegando a ser três vezes mais lento que as implementações em C++. Parte dessa diferença pode ser justificada pela interpretação de instruções (*bytecodes*) da máquina virtual de Lua. Isso pode ser amenizado através da utilização de um interpretador de Lua com compilador JIT (*Just In Time*) como o LuaJIT, ou seja, que realiza a compilação dos *bytecodes* em instruções nativas da máquina antes de serem executados. Com o LuaJIT, a duração das chamadas do OiL apresenta uma redução significativa.

Figura 5.4: Duração de 100 chamadas sem parâmetro.



Outra parte da sobrecarga de processamento nas chamadas feitas pelo OiL é devido ao uso da reflexão computacional para consultar informações de tipo necessárias para codificação das mensagens GIOP, enquanto as demais implementações utilizam geração de código. Para amenizar esse problema é utilizada uma otimização, que consiste em gerar dinamicamente um código especializado de codificação para cada novo tipo de valor a ser enviado pela rede (`gencode`). Dessa forma, a primeira vez que um valor de um dado tipo é codificado resulta na criação de uma função de codificação especializada com base nas informações de tipo obtidas através de reflexão computacional. Contudo,

quando um valor de mesmo tipo é codificado novamente, essa informação não é consultada novamente. Esse recurso permite que a duração das chamadas do OiL sejam similares à duração da chamadas de implementações em Java. Contudo, como a geração de código feita pelo OiL é dinâmica, através do suporte a interpretação de código, não é necessário conhecer na etapa de desenvolvimento todos os tipos tratados pela aplicação, como é necessário nas demais implementações em Java e C++.

É importante notar que a redução da duração das chamadas devido à geração de código de *marshaling* especializado é mais significativa em chamadas inter-processo do que através da rede. Isso é devido ao atraso do envio de dados pela rede, que implica em uma sobrecarga fixa entre as chamadas que oculta parte da redução do tempo das chamadas.

Esse mesmo padrão de duração das chamadas se repete quando o parâmetro da chamada é um valor simples como um número ou uma *string*. Mesmo para tipos de dados estruturados, como uma estrutura contendo valores simples ou uma referência de objeto, o desempenho do OiL é comparável ao de implementações em Java. Porém, tanto o OiL como implementações em Java apresentam uma demora maior nas chamadas para tipos estruturados do que implementações em C++. É importante notar, que a resolução automática de referências locais não implica em uma sobrecarga significativa no *marshaling* de referências, pois o mesmo padrão de duração é visto nas chamadas com outros tipos estruturados.

No caso de chamadas que recebem como parâmetro uma seqüência de valores, o desempenho do OiL fica pior que o de implementações em Java, especialmente com tipos estruturados, como é ilustrado na figura 5.5. Isso é porque o OiL utiliza a tabela para representar os dados estruturadas. Tabelas são estruturas dinâmicas que são inspecionadas para obter os campos da estrutura a ser enviada pela rede. Em linguagens como Java e C++, os dados estruturados são representados por estruturas estáticas de acesso mais eficiente.

Apesar da maior duração relativa das chamadas com seqüências de valores do OiL, seqüências de referências de objeto, que são codificadas como informações estruturadas, apresentam uma duração igual ou menor do que em implementações Java, como é ilustrado na figura 5.6. Contudo, a razão para esse comportamento não é clara.

5.2 Utilização de Memória

Outro aspecto importante da avaliação de desempenho do OiL é a utilização de memória. Linguagens dinâmicas geralmente impõem um maior

Figura 5.5: Duração de 100 chamadas cujo parâmetro é uma seqüência de 10 estruturas com valores simples.

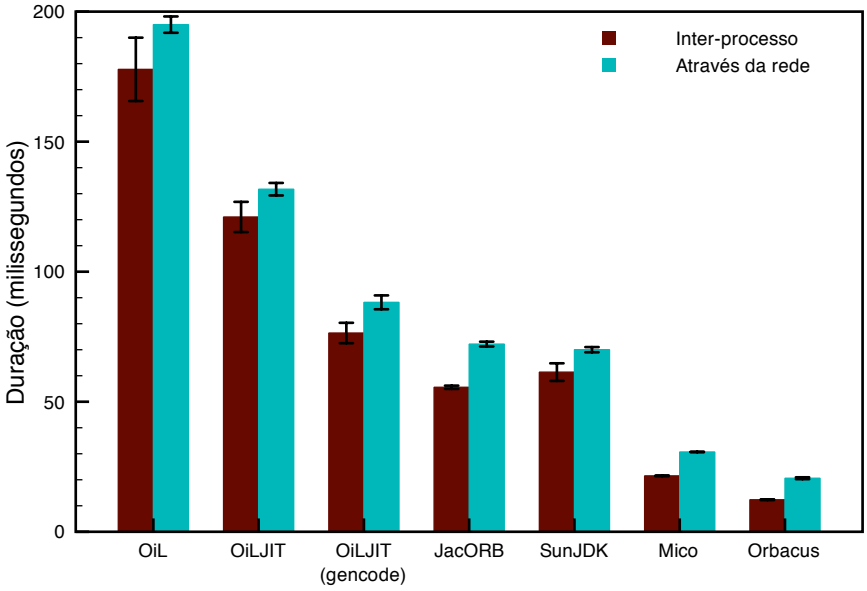
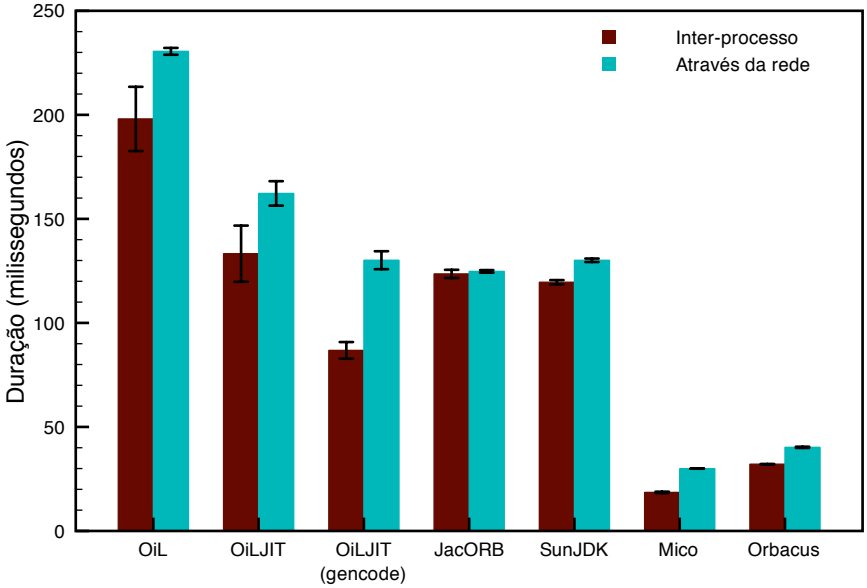


Figura 5.6: Duração de 100 chamadas cujo parâmetro é uma seqüência de 10 referências de objeto.



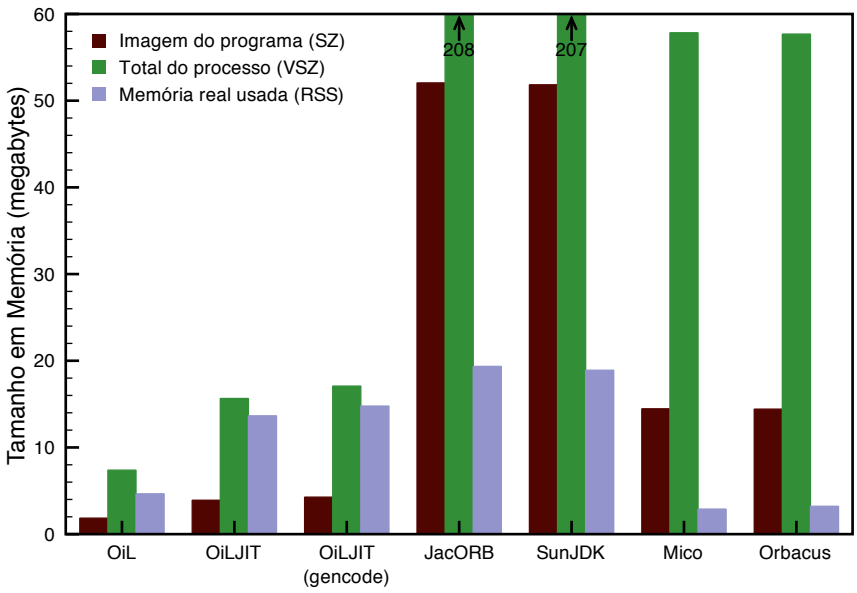
consumo de memória, pois é necessário manter informações adicionais sobre o programa, seja sobre os tipos dos valores utilizado nas verificações de tipo ou sobre seu código e estrutura interna para o suporte a reflexão computacional. Além disso, o suporte a interpretação também implica que o interpretador ou compilador da linguagem fazem parte da infra-estrutura necessária para a execução do programa.

Para avaliar o uso de memória, foram coletadas informações de uso

de memória dos servidores utilizados nos experimentos apresentados nas seções anteriores. Essas informações foram obtidas através do comando `ps` oferecido pelo sistema. Através desse comando é possível obter o tamanho em memória da imagem do programa executado pelo servidor (SZ), assim como a quantidade total de memória virtual alocada pelo processo (VSZ) e a quantidade real de memória física utilizada pelo processo (RSS). Essas medidas de memória incluem toda memória utilizada pelo processo, inclusive a memória que pode ser compartilhada com outros processos.

A figura 5.7 mostra a memória utilizada pelos processos servidores desenvolvidos utilizando as diferentes implementações de ORB avaliadas nesse trabalho. Os valores apresentados são os valores máximos de memória registrados durante toda execução do processo. As medidas são obtidas através de um processo executando na mesma máquina do processo servidor, que executa o comando `ps` a cada 0,01 segundos.

Figura 5.7: Utilização de memória dos diferentes servidores durante os experimentos.



A barra vertical indicando o tamanho total dos processos Java não é mostrada integralmente na figura para permitir uma escala adequada a uma melhor visualização dos dados, visto que a diferença do uso total de memória de um processo Java e os demais é muito grande. Os processos servidores em Java usando JacORB e SunJDK alocam um total de mais de 208 MB e 207 MB de memória virtual, respectivamente. Contudo, menos de 20 MB dessa memória reside efetivamente na memória física. Grande parte dessa memória utilizada pelas implementações Java pertence ao JRE (*Java Runtime Environment*) que é compartilhado com outros processos Java.

O montante total de memória virtual utilizado pelas implementações em C++ é próximo a 60 MB, porém menos de 4 MB é efetivamente usado da memória física. A imagem do programa, que armazena código do programa e bibliotecas e espaço para pilha de execução, representa pouco mais de 14 MB. Os dados efetivamente manipulados pela implementação em C++ é muito pequeno e a gerência de memória é feita de forma extremamente eficiente.

O total de memória utilizada por todas as variações do OiL é menor que as demais implementações. Em particular, o tamanho da imagem do programa é pequeno, ficando em torno de 4 MB. Isso é devido à máquina virtual (VM) de Lua ser muito reduzida e incluir poucas bibliotecas C. Por outro lado, a maior parte da memória utilizado pelo OiL é mantida na memória real pois se refere a instruções Lua (*opcodes*), que são interpretadas pela VM, ou por dados acessados freqüentemente pelo coletor automático de lixo de Lua. Em especial, o uso do interpretador com JIT aumenta o consumo de memória em até três vezes o utilizado pelo interpretador padrão de Lua. Isso representa uma decisão importante na escolha do interpretador de Lua utilizado com OiL, pois essa escolha deve priorizar uma menor utilização memória ou menor velocidade de execução.

Sem JIT, a memória física realmente utilizada pela implementação com o OiL é próxima à implementação em C++ nesse experimento particular. Entretanto, isso não é sempre verdade, pois estruturas de dados podem ser implementadas mais eficientemente em C++ do que em Lua. Portanto, se o tamanho dos dados manipulados pela aplicação com o OiL aumentar, a utilização de memória da implementação em Lua pode variar consideravelmente. Em alguns casos, a memória utilizada pelo OiL pode crescer muito além da memória necessária pela implementação C++, de acordo com o tamanho dos dados manipulados.

O OiL apresenta um desempenho bastante razoável quando comparado a implementações em Java, tanto em relação ao tempo das chamadas como no uso de memória. Quando comparado a implementações em C++, o OiL apresenta menor uso total de memória em aplicações que manipulam poucos dados, o que é particularmente interessante em sistemas embarcados que não utilizam grandes porções de memória compartilhada entre processos ou não oferecem suporte de memória virtual.

Por outro lado, é importante ressaltar que essa avaliação de desempenho leva em consideração apenas o desempenho de implementações do protocolo GIOP de CORBA, que apresenta inúmeras facilidades para implementação em C e C++. Em particular, muitos dos tipos de dados do GIOP são codificados de forma idêntica ao formato dos dados em memória dos tipos de C++. Isso

permite que a codificação de valores em C++ possa ser implementada mais eficientemente do que em linguagens como Java e Lua, que definem outros tipos de dados. Portanto, é possível obter resultados diferentes utilizando outros protocolos mais ajustados para cada linguagem.