4
Trabalhos Correlatos

Este capítulo apresenta trabalhos relacionados aos trabalhos descritos nos dois capítulos anteriores (LuaInterface e Lua2IL). Os trabalhos são divididos em três categorias: pontes entre o interpretador Lua e outras plataformas, pontes entre outros interpretadores e o CLR, e compiladores de outras linguagens dinâmicas para o CLR. Cada categoria é apresentada em uma seção.

As pontes são comparadas com LuaInterface, tanto em relação aos recursos oferecidos quanto à implementação do mesmos. Para as pontes com o CLR, é feita uma comparação de desempenho com LuaInterface. Os compiladores são comparados com Lua2IL, e o desempenho do código que eles geram é comparado com o desempenho do código compilado por Lua2IL.

PUC-Rio - Certificação Digital Nº 0210489/CA

4.1
Pontes entre Lua e Outras Plataformas

Nesta seção são apresentadas as pontes LuaORB e LuaJava, entre o interpretador Lua e as plataformas CORBA e Java, respectivamente. O objetivo da plataforma CORBA é a interoperabilidade entre componentes heterogêneos dentro de um ambiente distribuído, um objetivo similar ao do CLR de interoperabilidade entre linguagens de programação; LuaORB é a inspiração para o trabalho apresentado nesta dissertação. Já a plataforma Java tem recursos e implementação similares aos do CLR. Tanto LuaORB quanto LuaJava são comparadas com LuaInterface.

LuaORB

LuaORB é uma biblioteca para a linguagem Lua, implementada em C++, para controlar objetos e implementar interfaces CORBA [4, 11]. LuaORB usa os recursos reflexivos do padrão CORBA para criar proxies

para objetos CORBA em tempo de execução. Os scripts usam a sintaxe de Lua para acesso a propriedades e para chamada a métodos destes objetos. A conversão dos tipos de Lua para os tipos CORBA é automática; LuaORB também converte automaticamente tabelas Lua em estruturas CORBA. Scripts Lua também podem usar LuaORB para registrar tabelas Lua como implementações de interfaces CORBA. LuaORB registra os objetos dinamicamente, usando a CORBA Dynamic Skeleton Interface.

LuaInterface é similar a LuaORB em sua operação, oferecendo aos objetos CLR o mesmo nível de acesso que LuaORB oferece aos objetos CORBA. A exceção é a conversão automática de tabelas em estruturas, mas LuaInterface permite instanciar e usar estruturas do CLR. LuaInterface também oferece a possibilidade de usar tabelas Lua como objetos do CLR (a função `make_object`), criando dinamicamente as classes necessárias usando a API `Reflection.Emit` do CLR.

PUC-Rio - Certificação Digital Nº 0210489/CA

LuaJava

LuaJava é uma ponte entre Lua e a máquina virtual da linguagem Java, e permite que scripts Lua instanciem e usem objetos Java e que criem classes a partir de tabelas Lua [5, 6]. Para comunicação com o interpretador Lua, LuaJava usa a API de reflexão e a API de acesso a código nativo de Java. Quando um script instancia um objeto Java, LuaJava retorna um proxy para o objeto, e o script usa este proxy como um objeto Lua qualquer. LuaJava cria novas classes Java a partir de tabelas Lua por geração e carregamento dinâmico de bytecodes da máquina virtual Java.

LuaInterface é bastante similar em recursos a LuaJava, mas sua implementação é mais simples, devido às facilidades oferecidas pela CLR: as APIs PInvoke e `Reflection.Emit`. Para chamar funções da API de Lua a partir de Java, LuaJava possui *stubs* em C que fazem a conversão dos tipos Java para os tipos C. PInvoke faz as conversões automaticamente, bastando declarar as funções da API. Já `Reflection.Emit` oferece métodos que criam e carregam classes temporárias dinamicamente; Java não possui uma API similar, então LuaJava precisa criar e carregar os bytecodes que representam a classe diretamente.

4.2 Pontes entre Interpretadores e o CLR

Esta seção apresenta pontes publicamente disponíveis entre outras linguagens interpretadas e o CLR, comparando-as com LuaInterface. São apresentadas a LuaPlus, PerlNET e Dot-Scheme, pontes respectivamente entre o CLR e os interpretadores Lua, Perl e Scheme. A seção também apresenta uma comparação de desempenho entre LuaInterface e PerlNET.

LuaPlus

A distribuição LuaPlus é uma interface de C++ para o interpretador Lua que também inclui uma interface do CLR [22]. A interface CLR possui métodos para executar código Lua, ler e atribuir valores a variáveis globais de Lua, e registrar delegates como funções Lua. Objetos CLR são passados para o interpretador Lua como userdata, mas scripts Lua não podem usá-los (chamando seus métodos, por exemplo), nem podem instanciar novos objetos. Os delegates registrados como funções também não podem possuir qualquer assinatura, mas sim uma assinatura fixa, definida pela própria interface. LuaPlus, portanto, oferece apenas uma pequena parte dos recursos presentes em LuaInterface.

PerlNET

PerlNET é uma biblioteca comercial, desenvolvida pela ActiveState, para integrar o interpretador da linguagem Perl ao CLR [16]. Ela usa PInvoke para comunicação entre o interpretador Perl 5.6 e o CLR. Scripts Perl podem, com PerlNET, instanciar e usar objetos do CLR através da mesma sintaxe usada com objetos Perl. Também podem definir novas classes do CLR, com métodos implementados em Perl. As classes são permanentes, e podem definir novos métodos, visíveis a partir de outros objetos do CLR.

PerlNET e LuaInterface são equivalentes quanto ao uso de objetos CLR; ambos são consumidores completos da Common Language Specification. PerlNET oferece mais recursos quanto à extensão do CLR: as classes criadas por LuaInterface são temporárias, e podem apenas redefinir métodos de suas classes base. PerlNET empacota o interpretador Perl, o script que define a nova classe e a classe gerada em uma única *assembly* do CLR, e as aplicações do CLR podem referenciar esta assembly para criar instâncias da nova classe e chamar os seus métodos.

PUC-Rio - Certificação Digital Nº 0210489/CA

Dot-Scheme

Dot-Scheme é uma ponte entre o interpretador PLT Scheme e o CLR. Programas Scheme podem instanciar e usar objetos do CLR, com os métodos dos objetos mapeados para funções Scheme. Dot-Scheme não é um consumidor completo do CLS: não oferece nenhuma maneira de definir delegates em código Scheme, para tratar eventos, por exemplo. Também não há nenhum recurso de criação de novas classes. Oferece, portanto, menos recursos que LuaInterface.

4.2.1
Comparação de Desempenho

Esta seção apresenta uma avaliação do desempenho de LuaInterface, comparando-o com o desempenho da biblioteca PerlNET. A biblioteca LuaPlus não oferece o recurso necessário para essa comparação (chamadas a métodos de objetos CLR), e Dot-Scheme, de acordo com o seu autor, não está otimizada para melhor desempenho, logo estas duas pontes foram deixadas de fora da comparação. A Seção 4.3.1 compara o desempenho de LuaInterface com o do código compilado por Lua2IL.

O foco dos testes de desempenho foi nas chamadas de métodos do CLR a partir de scripts, já que este é o principal recurso oferecido pelas pontes. Ao todo foram feitas chamadas a seis métodos diferentes, variando o número e tipo dos parâmetros e do valor de retorno.

Três dos métodos têm todos os parâmetros e o valor de retorno do tipo `System.Int32`, e variam no número de parâmetros (zero, um ou dois). Suas assinaturas são, respectivamente, `Int32 ()`, `Int32 (Int32)` e `Int32 (Int32, Int32)`. Os tempos para as chamadas a estes métodos estão na Figura 4.1. Todos os tempos estão em microsegundos, e foram coletados na mesma máquina, sob as mesmas condições[1]. Os tempos são uma média de dez execuções distintas de um milhão de chamadas cada.

A coluna *MethodBase.Invoke* mostra os menores tempos possíveis para chamadas reflexivas aos métodos, ou seja, o tempo para a chamada uma vez que já se tenha o método e os argumentos necessários. A coluna *Cache* mostra o tempo para uma chamada a partir de Lua, quando o método

[1]Athlon 1.2GHz, com 256Mb de memória, sob o sistema operacional Windows XP Professional, e a versão 1.1 do .NET Common Language Runtime. O interpretador Lua foi compilado com o compilador Microsoft Visual C++ versão 7, com todas as otimizações ativas. A biblioteca PerlNET usada é a versão que acompanha o Perl Dev Kit 5.3, da ActiveState, e já estava compilada. Nenhuma outra aplicação estava executando durante a coleta dos tempos.

PUC-Rio - Certificação Digital Nº 0210489/CA

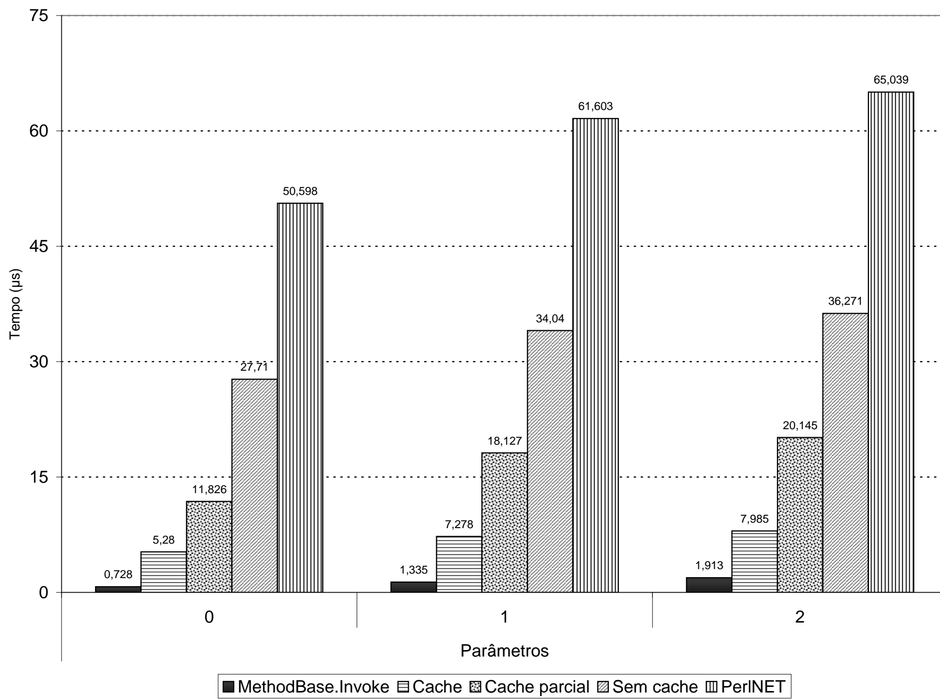

Figura 4.1: Tempos de chamada para métodos com parâmetros `System.Int32`

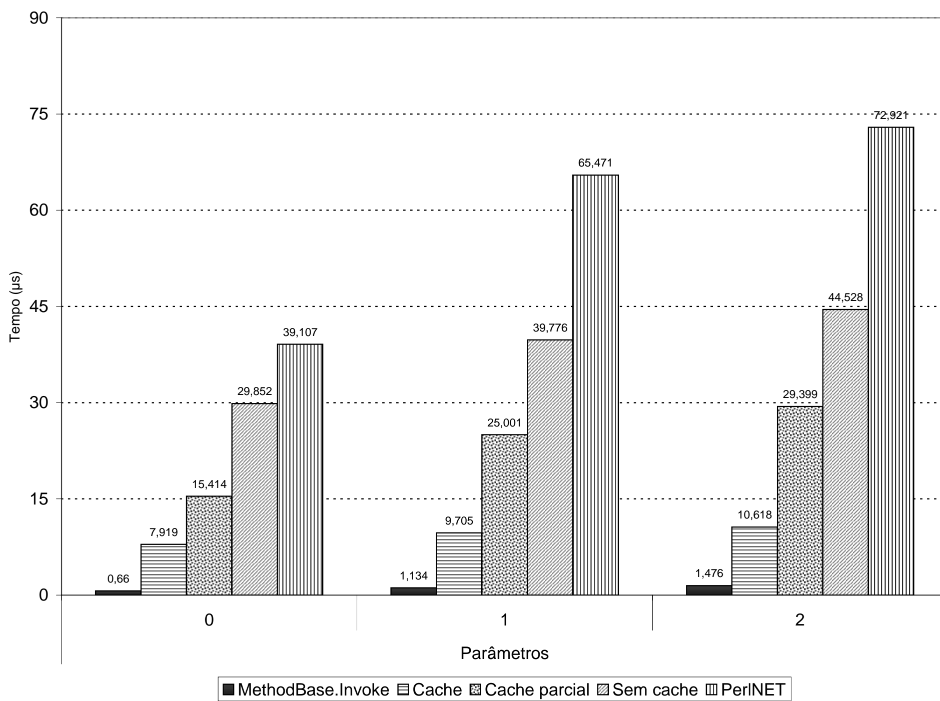

Figura 4.2: Tempos de chamada para métodos com parâmetros de tipos de objeto

PUC-Rio - Certificação Digital Nº 0210489/CA

PUC-Rio - Certificação Digital Nº 0210489/CA

já está em cache (a partir da segunda chamada). A coluna *Cache parcial* mostra o tempo com o cache interno do delegate desativado, forçando que LuaInterface, a cada chamada, descubra quais tipos o método recebe e como fazer a conversão dos valores Lua passados para esses tipos (também é o tempo para uma chamada em que ocorre incompatibilidade de tipos). Finalmente, a coluna *Sem cache* mostra o tempo com ambos os caches desativados, forçando LuaInterface a procurar pelo método a cada chamada (o tempo para a primeira chamada a um método). A coluna *PerlNET* mostra os tempos para chamadas a métodos a partir da linguagem Perl, usando a ponte PerlNET.

Os três métodos restantes para os quais foram feitos testes de desempenho têm parâmetros e valor de retorno de um tipo de objeto; o seu tipo é a própria classe na qual os métodos estão definidos. Os três métodos também variam na quantidade de parâmetros, de zero a dois, e suas assinaturas são `PerfTest ()`, `PerfTest (PerfTest)` e `PerfTest (PerfTest, Perftest)`, respectivamente. A Figura 4.2 mostra os tempos para estes três testes. Fica evidente, na diferença entre os tempos das chamadas a partir de Lua, o alto custo de se procurar reflexivamente o método a ser chamado, e o custo de determinar como cada argumento deve ser convertido. Os tempos com os caches ativos são de cerca de um quinto do tempo sem os caches.

O pequeno aumento nos tempos das chamadas a partir de Lua, da Figura 4.1 para a Figura 4.2, se deve aos testes que LuaInterface precisa fazer nos valores de tipos de objeto, antes de trazê-los para o CLR. Valores destes tipos são representados em Lua por userdata, e quando LuaInterface encontra um userdata ela checa primeiro se o userdata realmente corresponde a um objeto CLR.

Finalmente, a própria API PInvoke representa um custo. Cada chamada PInvoke gera de dez a trinta instruções, possivelmente mais, a depender do tipo dos argumentos envolvidos [23]. Individualmente as chamadas contam pouco, mas, como a chamada de um método envolve várias chamadas à API de Lua, os tempos individuais se somam até chegar a cerca de um quinto do tempo total da chamada ao método.

4.3 Compiladores de Linguagens Dinâmicas para o CLR

Esta seção compara alguns compiladores de outras linguagens dinâmicas para o CLR com o compilador Lua2IL. Como o único compilador totalmente implementado é o da linguagem JScript, mesmo compiladores

em desenvolvimento (ou abandonados) são apresentados. A seção também apresenta uma comparação de desempenho entre o código gerado pelos compiladores, e uma comparação do desempenho das chamadas a métodos CLR entre os compiladores e LuaInterface.

Python for .NET

Em 1999 e 2000 a Microsoft financiou uma pesquisa para a criação de um compilador Python para o CLR, o Python for .NET [8]. Python for .NET percorre a árvore sintática gerada pelo interpretador CPython, emitindo código da Common Intermediate Language do CLR através da API `Reflection.Emit`. A implementação tem similaridades com a de Lua2IL: Python for .NET define uma estrutura `PyObject` para seus valores, e uma interface IPyType que define as operações que podem ser feitas sobre os valores (os equivalentes de Lua2IL são a estrutura LuaValue e a classe LuaReference, respectivamente).

Cerca de 95% do núcleo da linguagem Python está disponível para os scripts, de acordo com o autor. Ficaram de fora do compilador os tipos primitivos sem um correspondente direto no CLR (números de tamanho arbitrário, números complexos e elipses), além de métodos embutidos das classes Python, usados para extensão dinâmica de classes e objetos. A sintaxe da linguagem não foi modificada. O desenvolvimento do compilador Python for .NET foi interrompido há cerca de dois anos. O último protótipo disponível tem data de abril de 2002, com partes datando de abril de 2000.

Perl for .NET

Perl for .NET é um compilador da linguagem Perl para o CLR, desenvolvido pela ActiveState entre 1999 e 2000 [7]. O compilador funciona como um *back-end* para o interpretador Perl 5.6, gerando código C# que chama um runtime Perl para as operações. Não há informações sobre o quanto da linguagem Perl é coberto pelo compilador, e o código-fonte para o mesmo não está disponível. A última versão data de junho de 2000, e funciona apenas com uma versão beta do CLR, não mais disponível.

JScript .NET

JScript .NET é uma extensão da linguagem JScript (ou EcmaScript) com um compilador para o CLR, parte do .NET Software Development

PUC-Rio - Certificação Digital Nº 0210489/CA

Script	Parâmetro
ack	Cálculo da função de Ackermann, parâmetros 3 e 8
fibo	Cálculo dos números de Fibonacci, o 30º número
random	Gerador de números aleatórios, gerar 1.000.000 de números entre 0 e 100
sieve	Crivo de Eratóstenes, de 2 até 8.192, 10 execuções
matrix	Multiplicação de matrizes $30x30$, 100 execuções
heapsort	Heapsort em um vetor de 10.000 números gerados aleatoriamente

Tabela 4.1: Scripts para o teste de desempenho dos compiladores

Kit da Microsoft [25]. O compilador estende a linguagem com classes e declarações opcionais de tipos, mas mantém os recursos dinâmicos de JScript. Para uso integral da interface do código JScript com o CLR, entretanto, são necessárias declarações de tipos; os scripts podem instanciar e usar classes do CLR sem declarações, mas delegates devem ser declarados com a assinatura correta, e dentro de uma classe. O código gerado pelo compilador usa os tipos do CLR nativamente, não os encapsulando dentro de outros valores. O resultado é que todas as operações envolvem checagem de tipos e *casts*.

PUC-Rio - Certificação Digital Nº 0210489/CA

S#.NET

S# é uma versão da linguagem Smalltalk desenvolvida pela SmallScript Corporation, e S#.NET é um compilador de S# para o CLR. Segundo o autor, o compilador e o runtime da linguagem estão prontos, mas ainda será feita a integração com o ambiente Visual Studio.NET. Não há uma versão pública para avaliação. O compilador vem sendo desenvolvido desde 1999.

4.3.1 Comparação de Desempenho

O primeiro teste de desempenho foi a execução de seis scripts do *The Great Win32 Computer Language Shootout* [26], envolvendo principalmente operações aritméticas, recursão e operações com vetores. O objetivo foi avaliar o desempenho do código gerado pelos compiladores para as operações simples das linguagens. A descrição de cada script de teste e os parâmetros de sua execução estão na Tabela 4.1.

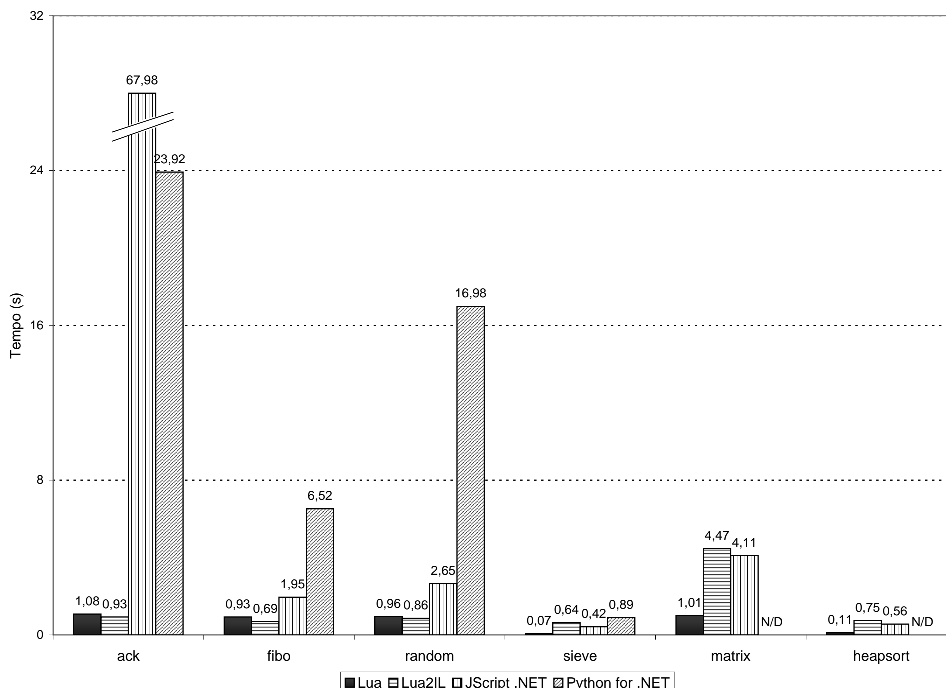

Figura 4.3: Comparação entre compiladores para o CLR

Foram testados os compiladores Lua2IL, JScript .NET e Python for .NET. Os mesmos scripts compilados pelo Lua2IL também foram executados pelo interpretador Lua 5.0. Os resultados são mostrados na Figura 4.3. Os tempos são em segundos, e todos os scripts foram executados na mesma máquina, sob as mesmas condições[2].

Os scripts *matrix* e *heapsort* não foram compilados pelo Python for .NET, mas não apresentavam erros de sintaxe (foram executados normalmente pelo interpretador Python).

Não é surpresa que o compilador Python for .NET tenha apresentado os piores tempos, já que seu desenvolvimento foi abandonado antes mesmo dele implementar toda a linguagem, e antes de se otimizar o código gerado por ele. O desempenho de JScript .NET nos três primeiros scripts também foi o esperado, por todas as operações envolverem casts e checagens de tipos. Já o pior desempenho de Lua2IL nos três últimos scripts é devido ao fato dos vetores serem implementados por tabelas hash, no protótipo atual de Lua2IL. As tabelas do interpretador Lua têm tanto uma parte vetor quanto uma parte hash, e índices numéricos usam a parte vetor. Uma otimização

[2]Athlon 1.2GHz, com 256Mb de memória, sob o sistema operacional Windows XP Professional. O código gerado pelos compiladores foi executado pela versão 1.1 do .NET Common Language Runtime. O interpretador Lua foi compilado com o compilador Microsoft Visual C++ versão 7, com todas as otimizações ativas. Nenhuma outra aplicação estava executando durante a coleta dos tempos.

PUC-Rio - Certificação Digital Nº 0210489/CA

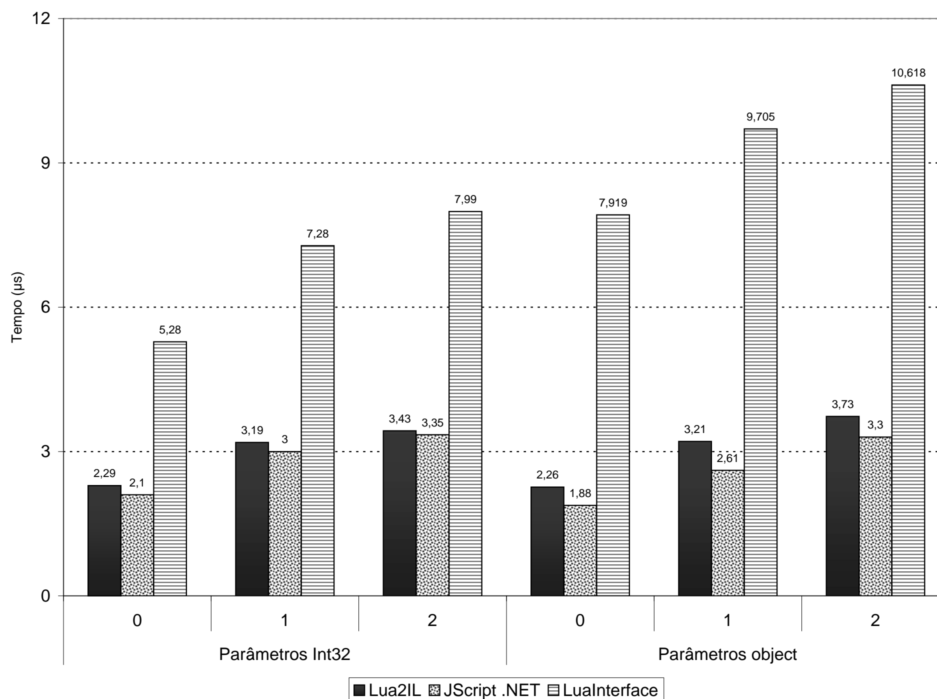

Figura 4.4: Tempos de chamada para métodos

PUC-Rio - Certificação Digital Nº 0210489/CA

similar, no código gerado por Lua2IL, deve reduzir bastante os tempos dos últimos três scripts.

O segundo teste de desempenho é uma extensão da comparação de desempenho da Seção 4.2.1, e avalia os tempos para chamadas a métodos de um objeto CLR, a partir do código gerado pelo compilador Lua2IL e pelo compilador JScript .NET (usando o recurso de *late binding* do compilador, ou seja, sem declarar tipos). O compilador Python for .NET ficou fora deste teste, pois os scripts compilados por ele só podem instanciar tipos da biblioteca padrão do CLR, e não puderam instanciar os tipos da assembly usada nos testes.

Os tempos são mostrados na Figura 4.4, e estão em microsegundos. Foram coletados na mesma máquina e sob as mesmas condições (as mesmas do teste anterior). As colunas *LuaInterface* são os tempos das chamadas a partir do interpretador Lua, usando a biblioteca LuaInterface, com o cache ativo (coluna *Cache* das Figura 4.1 e 4.2). As colunas *Lua2IL* são os tempos para chamar um método a partir do código gerado pelo compilador Lua2IL, também com o cache ativo. As colunas *JScript .NET* são os tempos para chamar um método a partir do código gerado pelo compilador JScript .NET.

Para este teste, o código gerado pelo compilador JScript .NET tem uma ligeira vantagem, por usar os valores do CLR diretamente, enquanto o código gerado por Lua2IL mapeia todos os valores seguindo as regras da

Seção 3.2; todos os valores passados para um método do CLR têm que ser mapeados de `LuaValue` para o valor do CLR correspondente, e o valor de retorno do método tem que ser mapeado para um valor `LuaValue`.

O teste também indica que a maior parte do tempo das chamadas a partir do interpretador Lua se deve aos custos de passar valores entre o interpretador Lua e o CLR, além dos custos das operações sobre valores userdata, envolvendo metatables; o código gerado por Lua2IL não precisa pagar nenhum destes custos.

PUC-Rio - Certificação Digital Nº 0210489/CA