DEPAnalyzer: um Analisador Estático de Dependências para Programas Java

Silvana Campos de Azevedo [1], Patrícia Kayser Vargas [1,2],
Jorge Luis Victória Barbosa [1,3], Adenauer Corrêa Yamin [1,3], Cláudio Fernando Resin Geyer [1]
[1] Universidade Federal do Rio Grande do Sul, Instituto de Informática
Caixa Postal 15064 – CEP 91591-970 – Porto Alegre, RS, Brasil
{silvanac, kayser, barbosa, adenauer, geyer}@inf.ufrgs.br
[2] Centro Universitário La Salle, Departamento de Informática
Rua Victor Barreto, 2288 CEP: 92010-000– Canoas, RS, Brasil
{kayser@lasalle.tche.br}
[3] Universidade Católica de Pelotas, Escola de Informática
Rua Felix da Cunha, 412 CEP: 96010-000 Pelotas, RS, Brasil
{barbosa, adenauer}@atlas.ucpel.tche.br

Resumo—
Este trabalho apresenta um modelo de análise estática para programas Java, denominado DEPAnalyzer (*DEPendencies Analyzer*). O DEPAnalyzer tem por objetivo a geração de informações sobre dependências entre as classes de um programa. As classes são as entidades estáticas que dão origem, em execução, a grupos de objetos. A comunicação entre grupos de objetos estabelece o relacionamento de dependência entre eles. Esta informação pode ser usada na alocação destes objetos em uma arquitetura distribuída. Sabendo as dependências, pode-se manter perto as entidades que possuam um grau maior de acoplamento.
Palavras-chave— **Análise estática, Programação Distribuída e Programação Orientada a Objetos**

Abstract—
This work presents a model of static analysis of Java programs, denominated DEPAnalyzer (DEPendencies Analyzer). DEPAnalyzer has the purpose of generating information about dependencies among the classes of a program. The classes are the static entities, which give origin, in execution time, to groups of objects. The communication among the groups of objects establishes the dependence relationship among them. This information can be used in the allocation of these objects in a distributed architecture. Knowing about dependencies, it is possible to keep highly coupled entities near.
Keywords— **Static analysis, Distributed programming and Object oriented programming**

I. Introdução

O paradigma orientado a objetos é considerado como uma ferramenta adequada à estruturação de problemas do mundo real, facilitando a modelagem computacional de problemas. A união da facilidade de modelagem e de características como reusabilidade e herança, tornam a utilização deste paradigma atrativa. Há várias décadas este paradigma vem sendo estudado. Atualmente, uma das linguagens mais populares é Java.

Java é uma linguagem relativamente nova, introduzida em 1995, no entanto, os conceitos empregados por esta não o são [TYM 98]. Duas razões destacam-se na ampla disseminação de Java. A primeira é o fato de sua sintaxe ser simplificada e bastante parecida com a sintaxe da linguagem C++. A segunda, e possivelmente a principal razão, é a disponibilidade de recursos previstos para suporte à programação distribuída. Sua elevada portabilidade de código se mostra muito conveniente para aplicações que serão executadas em redes de computadores inerentemente heterogêneas. A disponibilidade cada vez maior de redes locais e o aumento da velocidade das conexões praticadas, tanto nestas como na Internet, potencializam cada vez mais esta segunda razão.

O preço pago pela portabilidade da linguagem Java, decorrente do uso da interpretação, recai no desempenho da execução dos seus programas. Diversos esforços têm sido feitos procurando aumentar o desempenho da execução de programas Java, a exemplo, a proposta de uso da compilação *Just-in-Time* [SUN 01]. Os melhoramentos feitos até agora, no entanto, não garantem que Java consiga atingir os patamares de desempenho das linguagens compiladas para código nativo do processador.

Deste modo, uma primeira análise pode levar à conclusão de ser paradoxal a escolha de Java, uma linguagem interpretada, como linguagem para uso em processamentos onde o desempenho é um aspecto importante. Outrossim, a comunidade científica vem investindo na vertente de empregar Java na área de alto desempenho e diversos grupos já atingiram resultados bastante animadores [JAV 01], [MAN 01].

Dentre as principais razões para uso de Java em processamento de alto desempenho, destacaríamos: o uso cada vez maior de redes com suporte ao processamento

paralelo, o suporte nativo a *multithreading*, os recursos de sincronização, o suporte às comunicações e a reusabilidade.

É visível em [TOP 01] o crescimento da adoção de arquiteturas de memória distribuída como plataforma para processamento de alto desempenho. Uma evidência neste sentido é o equipamento ASCI White. Este sistema é o que atualmente registra maior desempenho na lista dos 500 mais poderosos ([TOP 01] - atualização de novembro de 2000), e apresenta uma arquitetura de memória distribuída organizada na forma de clusters hierarquicamente interligados.

Tendo em vista este cenário de arquiteturas de memória distribuída, onde os custos de comunicação são bastante heterogêneos dependendo dos nodos envolvidos, considerar aspectos de localidade é crucial para o desempenho geral de uma aplicação distribuída. Particularmente para um ambiente de programação distribuída, baseado em uma linguagem orientada a objetos como Java, objetos que compartilham dados, ou que apresentam um perfil de intensa troca de mensagens precisam ficar o mais "próximo possível" (ou no mesmo nodo processador, ou em nodos conectados através de um canal de comunicação rápido).

As políticas de balanceamento dinâmico de carga têm na sua atuação custos inevitáveis, sejam para transferir contextos, como para realizar a realocação dos objetos propriamente ditos. Deste modo, um estado inicial de execução que encaminhe para um menor número de migrações, tanto para os agentes de execução, como para os seus dados, aponta para um melhor desempenho global da execução.

Por outro lado, a redistribuição de carga de trabalho baseada somente em informações dinamicamente coletadas durante o processamento, isto é, escalonamento dinâmico sem conhecer estados futuros, também apresenta problemas. Neste tipo de escalonamento pode acontecer de, tão logo uma migração de objetos/dados seja feita, o perfil de execução (comunicações e/ou acesso a dados) se alterar, o que implicaria em um custo adicional inútil.

Uma análise estática do relacionamento das estruturas de um programa orientado a objetos, como feita em [ARO 01], [DEW 01], [AGR 99], gera informações que podem contribuir fortemente para um melhor desempenho das ações dinâmicas de balanceamento de carga.

Tendo como meta o auxílio ao escalonamento, este trabalho apresenta um modelo de análise estática das dependências (comunicações) entre as entidades de um programa, escrito em um subconjunto da linguagem Java, o qual denomina-se DEPAnalyzer (DEPendencies Analyzer). A análise estática tem o objetivo de determinar, estaticamente, informações sobre o comportamento que o programa terá em tempo de execução [AZE 99], [COR 97], [DAM 97].

O artigo possui a seguinte organização. A seção 2 apresenta o modelo, primeiramente dando uma visão geral e depois descrevendo cada módulo. A seção 3 mostra os aspectos da implementação atual. A seção 4 analisa alguns trabalhos relacionados. Finalmente a seção 5 mostra as conclusões do trabalho.

II. DEPANALYZER

O DEPAnalyzer é um analisador estático de dependências. As dependências são os relacionamentos entre suas entidades, ou seja, os relacionamentos entre as classes de um programa. As classes de um programa orientado a objetos representam as entidades estáticas, as quais em tempo de execução darão origem a conjuntos de objetos. O relacionamento de dependência é estabelecido através das invocações de variáveis e de métodos entre classes.

As dependências entre conjuntos de objetos podem ser abstraídas como dependências entre classes. Estabelecer as dependências entre as classes, de um programa seqüencial Java, pode auxiliar no escalonamento de objetos em uma arquitetura distribuída. Considerando que as dependências revelam comunicações entre classes, é possível separar estes conjuntos de objetos em grupos comunicantes, isto é, que possuem dependências. Deste modo, o enfoque central deste trabalho é a obtenção de informações estáticas sobre as classes de um programa que sejam relevantes para sistemas de escalonamento.

Tendo em vista este objetivo, o DEPAnalyzer baseia-se na abordagem do pior caso. Nesta, se uma dependência pode ocorrer em tempo de execução, mas sua ocorrência não é confirmada em tempo de compilação, considera-se que esta irá ocorrer. Esta abordagem evita que ocorra em tempo de execução alguma comunicação não considerada pelo escalonamento, o que pode levar a comunicações entre os nodos da arquitetura, isto é, a um custo adicional de processamento. No entanto, as informações geradas por esta abordagem podem levar a uma limitação na exploração do paralelismo da aplicação.

Atualmente, o DEPAnalyzer analisa programas *CPU bound*, ou seja, sem interação com o usuário. Além disso, os programas analisados possuem somente polimorfismo do tipo sobrecarga e o uso de interfaces ainda não é tratado. O polimorfismo tipo sobrecarga permite que existam dois métodos, da mesma hierarquia, com o mesmo nome mas com assinaturas diferentes. As interfaces são construções abstratas, cujos métodos possuem somente a definição da assinatura. A implementação dos métodos das interfaces são definidas em classes que implementam estas interfaces.

A. A. Visão Geral do Modelo

O DEPAnalyzer analisa o código fonte de um programa Java basicamente através de duas etapas: a coleta das informações e a análise de dependências, como mostra a figura 1. É importante lembrar que um programa Java é formado por várias classes, cada uma das quais é

normalmente armazenada em um arquivo separado. Portanto, para realizar a análise, todas as classes envolvidas no programa devem ser submetidas ao DEPAnalyzer.

Durante a etapa de coleta das informações é realizada uma análise das classes declaradas no programa, gerando uma estrutura de dados, a qual retrata esta hierarquia juntamente com os elementos (atributos e métodos) declarados em cada classe.

Por sua vez, a análise de dependências realiza a análise do fluxo das invocações com o intuito de detectar as comunicações entre as classes do programa. Como resultado desta análise, tem-se o grafo das dependências entre as classes declaradas pelo programador. Além disso, como decorrência da análise de dependências, pode-se obter o grafo das invocações do programa.

A geração do grafo das invocações do programa é possível devido ao fato da análise de dependências percorrer o fluxo das invocações dos métodos do programa para detectar as comunicações entre as classes. Isto leva a uma detecção do fluxo das invocações geradas entre as classes declaradas no programa.

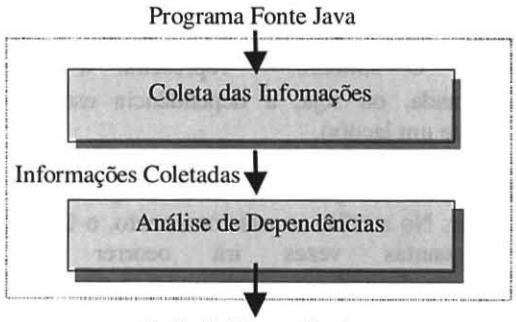

Fig. 1 Estrutura do DEPAnalyzer

B. Coleta das Informações

As classes de um programa orientado a objetos especificam os novos tipos de dados definidos pelo programador. Estes tipos modelam as entidades que compõem o problema a ser resolvido. Ou seja, em uma linguagem tipada como Java, existem os tipos de dados primitivos da linguagem como: int, float, String. No entanto, os tipos primitivos não são suficientes para a implementação de um sistema, havendo a necessidade de definir e utilizar estruturas (classes) mais complexas. Estas estruturas representam as partes do sistema que está sendo modelado, das quais serão instanciados os objetos.

A análise das classes de um programa proporciona uma visão das classes envolvidas na computação, tornando possível o conhecimento prévio dos tipos definidos pelo programador. Esta visão é sugerida por Agesen, pois facilita a análise de tipos em programas orientados a objetos [AGE 95].

O ambiente Java oferece um conjunto de classes, o que faz com que as classes de um programa Java possam ser divididas em duas categorias: as classes/tipos primitivos (classes da biblioteca Java) e as classes/tipos definidos pelo programador.

Para ocorrer a distribuição de um programa Java em uma arquitetura multiprocessada, deve-se ter o ambiente Java (juntamente com sua biblioteca de classes) em cada máquina da arquitetura. Isto faz com que cada máquina tenha o conjunto de classes primitivas, tornando locais as comunicações entre classes primitivas e classes definidas pelo programador. Em decorrência deste fato, os objetos de estudo do DEPAnalyzer são as dependências entre as classes definidas pelo programador.

Em um programa orientado a objetos existem três formas de se estabelecer uma dependência entre os conjuntos de objetos. Estas formas são apresentadas na tabela 1.

TABELA I

FORMAS DE ESTABELECER DEPENDÊNCIAS

1	quando um atributo ou método de classe é invocado por um método de outra classe;
2	quando um método de instância é invocado por um método de uma instância de outra classe;
3	quando uma instância de classe é passada como argumento em uma invocação de método.

Como se pode perceber, na tabela 1, as formas de relacionamento entre as entidades de um programa estão expressas nas estruturas de comportamento (métodos) declaradas em cada classe. A identificação das classes de um programa é feita através da leitura do código fonte em busca das estruturas sintáticas de declarações destas.

O DEPAnalyzer ao detectar as estruturas sintáticas de declaração das classes, recolhe em uma estrutura de dados as informações apresentadas na figura 2.

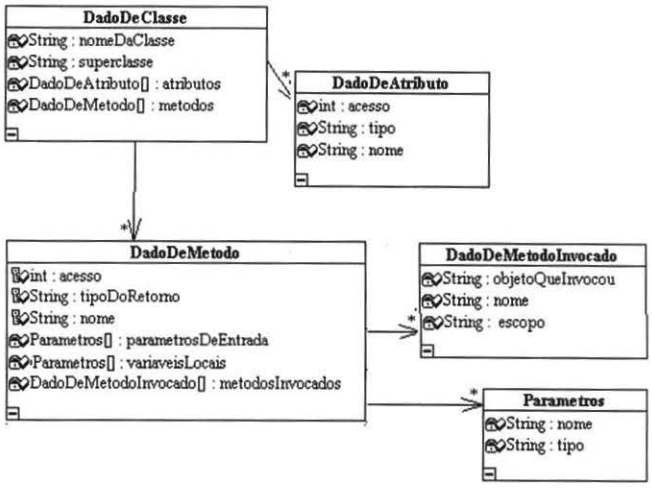

Fig 2 Estrutura de dados contendo as informações coletadas

Cada classe definida pelo programador é um elemento da estrutura de dados *DadoDeClasse*, que contém os seguintes elementos:

- *nomeDeClasse*: representa o identificador da classe declarada.
- *superclasse*: informação que contém o identificador da superclasse. Em Java, todas as classes descendem implicitamente da classe *Object*. Então, diante da ausência da descendência explícita de uma classe, é atribuída a este campo a descendência implícita *Object*.
- *atributos*: estrutura de dados onde cada elemento é a descrição de uma variável declarada na classe. Estes elementos possuem as seguintes informações: *acesso*, que representa o tipo de visão que as outras entidades (classes) do programa terão deste atributo, o *tipo*, que representa o tipo da variável declarada e o *nome*, que representa o identificador da variável declarada.
- *metodos*, estrutura de dados onde cada elemento é uma descrição de um método da classe declarada. Estes elementos possuem as seguintes informações: *acesso, nome, tipoDoRetorno, parametrosDeEntrada, variáveisLocais* e *metodosInvocados*. O *acesso* representa o tipo de visão que as outras entidades (classes) do programa analisado terão deste método. O *nome* representa o nome do método declarado. O *tipoDoRetorno* representa o tipo/classe da variável retornada pelo método. Cada elemento da estrutura de dados *parametrosDeEntrada* é um dos parâmetros recebidos pelo método. Estes elementos possuem as seguintes informações: Nome e Tipo. Cada elemento da estrutura de dados *variaveisLocais* representa uma variável declarada no método. Estes elementos possuem as seguintes informações: *nome* e *tipo*. Cada elemento da estrutura de dados *metodosInvocados* é um método invocado no método declarado. Estes elementos possuem as seguintes informações: *objetoQueInvocou*, *nome* do método invocado, o qual representa a assinatura do mesmo e *escopo*, o qual representa o contexto de invocação (se ocorreu em um contexto iterativo ou não).

C. Análise de Dependências

A análise de dependências visa gerar um grafo representando as dependências entre os conjuntos de objetos de um programa.

O processo de análise tem início com a criação do nodo da classe principal, a qual contém o método principal, que desencadeia o fluxo de execução do programa. Para isto, o usuário deve indicar qual é a classe principal.

O funcionamento da análise baseia-se no fato de que, em uma invocação de método, existe um método invocador (ou que causa a invocação) e um método invocado. O controle da análise utiliza estas informações para simular o fluxo das invocações do programa e assim, juntamente com a estrutura *DadoDeClasse*, detectar as dependências estabelecidas entre as classes.

Como resultado da análise, será obtido o grafo de dependências. Os nodos do grafo de dependência são as classes declaradas no programa e as arestas representam as dependências entre estas. As dependências podem ser classificadas conforme mostra a tabela 2.

TABELA II
CLASSIFICAÇÃO DAS DEPENDÊNCIAS

Sem dependência	$\perp$
Dependência	*
Dependência indeterminada	$\top$

O símbolo $\perp$ representa a ausência de dependência. O símbolo * representa a dependência entre duas classes, sendo esta livre de qualquer escopo, ou seja, independente de laços. O símbolo $\top$ representa a dependência indeterminada, ou seja, a dependência estabelecida no contexto de um laço(s).

As dependências indeterminadas evidenciam o fato da indefinição sobre a quantidade de vezes que a comunicação irá ocorrer. No auxílio ao escalonamento, o fato de não se saber quantas vezes irá ocorrer determinada comunicação/dependência é um impasse para a tomada de decisões. No entanto, o DEPAnalyzer sugere que quando isto ocorrer o escalonador assuma a abordagem do pior caso, na qual o modelo é baseado. Esta sugestão, parte do pressuposto que estas comunicações/dependências irão acontecer um número máximo de vezes, ou seja, um valor *upper bound*. Uma condição assim causaria inúmeras comunicações remotas, o que tende a ser muito custoso e conseqüentemente degradaria o desempenho do sistema como um todo. Isto representaria para o escalonamento que estes conjuntos de objetos não devem se encontrar em nodos distintos de processamento.

Na presença de comandos condicionais o DEPAnalyzer assume um caráter conservativo, ou seja, os dois ramos são analisados.

C.1 Detalhamento do algoritmo de análise de dependências

A análise tem início no método main. Neste método são verificadas as ocorrências das invocações. Na invocação de um método a análise de dependências detecta o escopo desta, ou seja, se esta ocorreu dentro de um comando iterativo (laço) ou não.

Após isto, é verificado o tipo/classe do elemento que está realizando a invocação. Este elemento pode ser uma

classe, o que representa uma invocação de classe ou pode ser uma instância (variável/objeto).

Em ambos os casos, o tipo do elemento que realiza a invocação é comparado com o tipo do método invocador.

Se os tipos forem iguais não se estabelece nenhuma dependência, ou seja, não é criado nenhum nodo e nenhuma aresta no grafo de dependências.

Se os tipos forem diferentes, o tipo do elemento que invoca o método é comparado com os tipos definidos pelo programador. Este processo é realizado para se verificar se esta dependência é entre a classe do método invocador e alguma classe primitiva do pacote Java, ou se esta dependência é entre duas classes declaradas no programa.

Quando o tipo da variável invocadora é definido pelo programador, se estabelece uma dependência entre a classe do elemento que realiza a invocação e a classe do método invocador. Neste caso, é criado o nodo correspondente à classe do elemento que invoca o método, se este ainda não existir, e a aresta que representa a dependência entre as classes.

Depois de verificar o tipo do elemento que realiza a invocação, é necessário analisar os tipos dos argumentos recebidos pelo método invocado e a ordem destes. Isto deve-se à presença do polimorfismo tipo sobrecarga. Uma vez identificado o método, procura-se na hierarquia da classe, do elemento que realiza a invocação, a declaração deste método.

O escopo de invocação do método é propagado juntamente com o fluxo de execução para dentro do corpo do método invocado. É importante salientar que inicialmente, antes da primeira invocação do método principal, não existem dependências. Após a primeira invocação, mesmo que esta não cause dependência, se o escopo da mesma for iterativo, esta informação é propagada para as invocações decorrentes da primeira invocação. Isto quer dizer que, sendo $\Delta = \{\bot, *, \top\}$, qualquer dependência $K \in \Delta$, $K \wedge \top = \top$, $K \wedge \bot = K$.

Se algum dos tipos dos argumentos do método invocado for diferente do tipo do elemento que o invoca, compara-se novamente este com os tipos definidos pelo programador.

Quando a igualdade entre o tipo do argumento do método invocado e algum dos tipos definidos pelo programador é detectada, uma dependência é estabelecida. Este relacionamento refere-se ao tipo do argumento e o tipo do elemento que invoca o método. Então é criado um nodo representando o tipo do argumento, se este não existir, e uma aresta entre o nodo correspondente ao tipo do elemento invocador e o tipo do argumento.

Outro aspecto relevante da análise é a detecção de invocações recursivas, as quais produzem dependências indeterminadas.

Após a análise, é verificado se alguma classe não primitiva está ausente no grafo de dependências. Se existir alguma classe nesta situação, esta é inserida no grafo como um nodo independente. Este caso representa a condição onde uma classe é totalmente independente das demais classes definidas pelo programador. Além disso, se a classe é totalmente independente, isto significa que esta não foi utilizada no processamento da aplicação analisada.

A análise de dependências gera um grafo de dependências entre as classes definidas no programa.

Por sua vez, o grafo de invocação dos métodos pode ser obtido em decorrência da análise de dependências. Neste grafo, os nodos representam os métodos invocados, e as arestas, a relação de invocação entre eles.

Tanto a análise de dependências entre as classes declaradas em um programa como a representação gráfica das invocações entre as mesmas, podem auxiliar no processo de escalonamento dos conjuntos de objetos em uma arquitetura distribuída.

III. Implementação

A implementação do DEPAnalyzer conta com a utilização do sistema JavaCC [ENS 00]. Este sistema é um gerador de analisadores léxico-sintático (*parsers*). O JavaCC é utilizado para leitura do código fonte Java e para coletar as informações apresentadas na figura 2.

O algoritmo de análise de dependências está sendo desenvolvido em Java. Este algoritmo fornece como resultado as informações de dependências de forma textual. Estas informações textuais, após manipuladas, serão utilizadas no processo de escalonamento.

Por sua vez, para gerar os grafos de dependências característicos da aplicação, o DEPAnalyzer utilizará uma ferramenta de visualização gráfica, a qual recebe informações textuais e gera a correspondente representação gráfica das mesmas.

IV. Trabalhos Relacionados

De forma análoga ao DEPAnalyzer, existem alguns trabalhos que realizam a análise estática das invocações ocorridas em programas Java com o objetivo de otimizar o desempenho destes programas.

Aroson foca-se em uma ferramenta para a paralelização de programas escritos em um subconjunto da linguagem Java [ARO 01]. A meta é paralelizar os métodos em programas seqüenciais livres de efeitos colaterais. A ferramenta gera um grafo, o qual representa os comandos do programa e as dependências de dados entre estes. Além disso, há uma preocupação em estimar os custos de processamento e de comunicação das tarefas.

Por sua vez, o DEPAnalyzer visa somente as dependências de invocações entre as classes de um programa; no entanto, este também realiza sua análise sob programas Java seqüenciais.

Agrawal apresenta um algoritmo de *demand-drive* que visa obter o conjunto de definição para um ponto do programa. Esta proposta tem como primeiro passo a análise

da hierarquia das classes. Inicialmente, assume-se um grafo de invocações conservativo, o qual é baseado nas informações obtidas através da análise da hierarquia das classes. Posteriormente, gera-se um grafo parcial através do grafo conservativo. Depois, realiza-se a propagação do fluxo dos dados para melhorar o grafo gerado, a partir da utilização das informações de tipos [AGR 99].

Em [DEW 01] é proposto um *framework* de análise estática visando a aproximação das informações dinâmicas de tipos para as expressões de invocações de métodos. Assim, a análise pode auxiliar na criação de grafos de fluxo de invocações mais precisos.

Um dos problemas encontrados na análise estática de programas orientados a objetos são as características dinâmicas da linguagem. Em [AGE 95] pode-se encontrar uma importante contribuição relacionada à inferência de tipos em programas orientados a objetos. A complexidade deste tipo de análise fica bem caracterizada neste trabalho.

O DEPAnalyzer, para realizar a análise de dependências necessita da identificação dos métodos invocados. A presença do polimorfismo tipo reescrita (existência de dois métodos, da mesma hierarquia, com o mesmo nome e a mesma assinatura), dificulta a identificação dos métodos invocados. Este aspecto descarta, a princípio, a análise destes programas por parte do DEPAnalyzer.

V. Conclusão

A análise realizada pelo DEPAnalyzer gera informações de dependência entre os conjuntos de objetos de um programa seqüencial. No escalonamento de objetos distribuídos as informações sobre quem depende de quem auxilia na distribuição física dos conjuntos de objetos entre os nodos de uma arquitetura. Esta distribuição pode gerar um aumento no desempenho do processamento, pois os grupos de objetos estarão cada um utilizando os recursos de um nodo de processamento, o que pode acelerar o processo da computação como um todo. Para que ocorra este aumento no processamento, deve-se levar em consideração o nível de comunicação entre os objetos distribuídos. Os custos decorrentes de elevados níveis de comunicação podem comprometer os ganhos decorrentes da exploração do paralelismo. A detecção de uma elevada dependência entre os objetos sugere o mapeamento dos mesmos em um mesmo nodo processador, ou em nodos interligados por canais rápidos de comunicação.

Um futuro aperfeiçoamento deste trabalho consiste em inferir informações referentes aos métodos de forma a determinar seu comportamento, isto é, se é um método de leitura ou um método de escrita. Este tipo de informação visa auxiliar, dentre outros trabalhos, no processo de replicação de objetos do ReMMoS [FER 01]. Este consiste em um ambiente para suporte a replicação de objetos distribuídos móveis. Para que as réplicas sejam criadas, é necessário que o ReMMoS tenha a informação do tipo de comportamento do método. Desta forma, a cada acesso remoto ao objeto, o sistema de replicação controla se o acesso é de leitura ou de escrita. Esta informação é usada também para atualizar e descartar as réplicas, e deve ser obtida por um analisador através da análise dos métodos. Outro aspecto importante é o tratamento do polimorfismo tipo reescrita. Acredita-se que, para realizar uma análise precisa desta característica, deva-se ter uma análise estática acompanhada de uma análise dinâmica. Isto deve-se ao fato destas informações serem de natureza dinâmica, o que dificulta uma precisão sobre as versões dos métodos que estão sendo invocados a nível estático.

Referências Bibliográficas

[AGE 95] AGESEN, O. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism European Conference on Object-Oriented Programming. In: ECOOP: 9th European Conference on Object-Oriented Programming. August, 1995. Disponível em: http://citeseer.nj.nec.com/agesen95cartesian.html. Acessado em 18 de junho de 2001.

[AGR 99] AGRAWAL, G. Simultaneous Demand-Drive Data-flow and Call Graph Analysis, Procedings of International Conference on Software Maintainance. September, 1999. Disponível em: http://citeseer.nj.nec.com/417649.html. Acessado em 17 de maio de 2001.

[ARO 01] ARONSSON, Peter; FRITZSON, Peter. Static Scheduling of Sequential Java Programs for Multi-Processors. In: JOSES: Java Optimization Strategies for Embedded Systems. April, 2001. Disponível em: http://i44w3.info.uni-karlsruhe.de/~josesworkshop/. Acessado em 07 de maio de 2001.

[AZE 99] AZEVEDO, S. C.; BARBOSA, J. L. V.; GEYER, F. R. Automatização da Análise Global no modelo GRANLOG. In: XXV Conferencia Latino Americana de Informática, Asunción-Paraguai. *Anais*.. 1999, v. 1, p. 601-612.

[COR 97] CORTESI, et all. Complementation in Abstract Interpretation. *ACM Transactions on Programming Languages and Systems*, Vol.19, No.1, 1997, pp.7-47.

[DAM 97] DAMS, D.; GERTH, R. Abstract Interpretation of Reactives Systems. *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 2, 1997, pp.253-291.

[DEW 01] DEWES, Holger. PROBST, Christian. Static Method Call in Java. In: JOSES: Java Optimization Strategies for Embedded Systems. April, 2001. Disponível em: http://i44w3.info.uni-karlsruhe.de/~josesworkshop/ Acessado em 6 de maio de 2001.

[ENS 00] ENSELING, Oliver. *Build your own languages with JavaCC*, December 2000. Disponível em: http://www.javaworld.com/jw-12-2000/jw-1229-cooltools.html Acessado em 17 de junho de 2001.

[FER 01] FERRARI, Débora N. *Um Modelo de Replicação em Ambientes que Suportam Mobilidade.* PPGCC/UFRGS, 2001. Dissertação de Mestrado

[JAV 01] JavaParty - A distributed companion to Java. Disponível em http://wwwipd.ira.uka.de/JavaParty/. Acessado em 06/05/2001.

[MAN 01] MANTA - Fasta Parallel Java. Disponível em http://www.cs.vu.nl/~rob/manta/index.html. Acessado em 06/05/2001.

[SUN 01] SUN Microsystems. The Java Hotspot Performance Engine Architecture. Disponível em: http://java.sun.com/products/hotspot/whitepaper.html. Acessado em 6 de maio de 2001.

[TOP 01] TOP 500 Supercomputer site. http://www.top500.org/. Acessado em 6 de maio de 2001.

[TYM 98] TYMA, P. Why are we using Java again? *Communications of the ACM*, New York, v.41, n.6, p.38-41, june 1998.