Automatic Distribution of Java Byte-Code Based on Dependence Analysis

Roxana E. Diaconescu, Lei Wang, Michael Franz
University of California, Irvine
Department of Computer Science
{roxana, wangglei, franz}@ics.uci.edu

ABSTRACT

One way to relieve resources when executing a program on constrained devices is to migrate parts of it to other machines in a distributed system. Ideally, a system can automatically decide where to place parts of a program to satisfy resource constrains (CPU, memory bandwidth, battery power, etc.). We describe a compiler and virtual machine infrastructure as the context for research in automatic program partitioning and optimization for distributed execution. We define program *partitioning* as the process of decomposing a program into multiple tasks. The main motivation for our design is to enable experimenting with optimizing program execution on resource-constrained devices with respect to memory consumption, CPU time, battery lifetime and communication.

With our approach, we represent a program as a graph of class instances and their interactions (dependences). Then, we augment the graph with weights that represent resource consumption. A general graph partitioning algorithm assigns nodes to abstract processors (partitions) according to the resource constraints. We then schedule the partitions on available actual processors for execution. We report results on the class instance graph construction. The sizes of the graphs and the analysis times indicate that it is feasible to use this representation for realistic applications. Moreover, the representation is the key to a flexible distribution model for Java byte-code.

1. INTRODUCTION

We present an experimental platform for research in compilation and distributed virtual machine technologies. Our infrastructure is a unifying framework that allows experimentation with various partitioning strategies and scheduling techniques in order to understand how the multiple optimization targets interplay. The main goal is to experiment with various (partitioning and mapping) strategies that are known to optimize execution with respect to resource consumption. Commonly employed optimizing (distributed) execution strategies include offloading parts of the computation to address power consumption, offloading code that is not executed at run-time to reduce code size, slowing down CPU to save power, etc. These techniques are most of the time manually performed or applied in isolation.

The main idea in our approach is to study how some of the techniques affect the others and to come up with a strategy that best exploits the available resources. For instance, off-loading computation may increase communication which, in turn, may lead to increased activity to system buffers and thus, increased power consumption. It is important to be able to model the program in terms of its interacting entities and their resource consumption to understand its run-time behavior. While static analysis can give a first approximation of the run-time behavior, we rely on dynamic profiling and adaptive repartitioning and mapping to reschedule tasks to computational nodes based on actual execution information.

A benefit of such a unifying experimental infrastructure is that it lets the compiler and the run-time system to perform partitioning and mapping automatically. Our system is intended to be open and general, so that it is easy to incorporate new strategies. Thus, we can plug in any partitioning strategy to our partitioning interface, without changing the system.

We build our infrastructure on an existing compiler, Joeq [34]. We use Joeq front-end to transform the byte-code into an intermediate representation suitable for our analyses and transformations. However, our approach is not limited to handling Java byte-code. Joeq is designed to incorporate other inputs, such as ELF binaries and SUIF files.

Our infrastructure depicted in Figure 1 consists of the following components:

- A compiler analysis framework to statically approximate the object[1] dependence graph of a program.
- A strategy to statically approximate the resource consumption for each object and for each object interaction.
- A general graph partitioning framework to uniquely assign a node (object) to a partition and compute the cost of doing so on the entire graph (taking into consideration multiple optimization criteria).
- A code generator that takes into account the partitioning information and generates communication respecting consistency and correctness constraints.
- A run-time system (or distributed virtual machine) that allows adaptive partitioning as well as scheduling and migration of program partitions according to the available resources with minimum impact on the observed output.

We will describe the infrastructure in Figure 1 in detail in Section 2.

[1] We use the term object to denote a class instance. Throughout the paper we use both, objects, and class instances interchangeably.

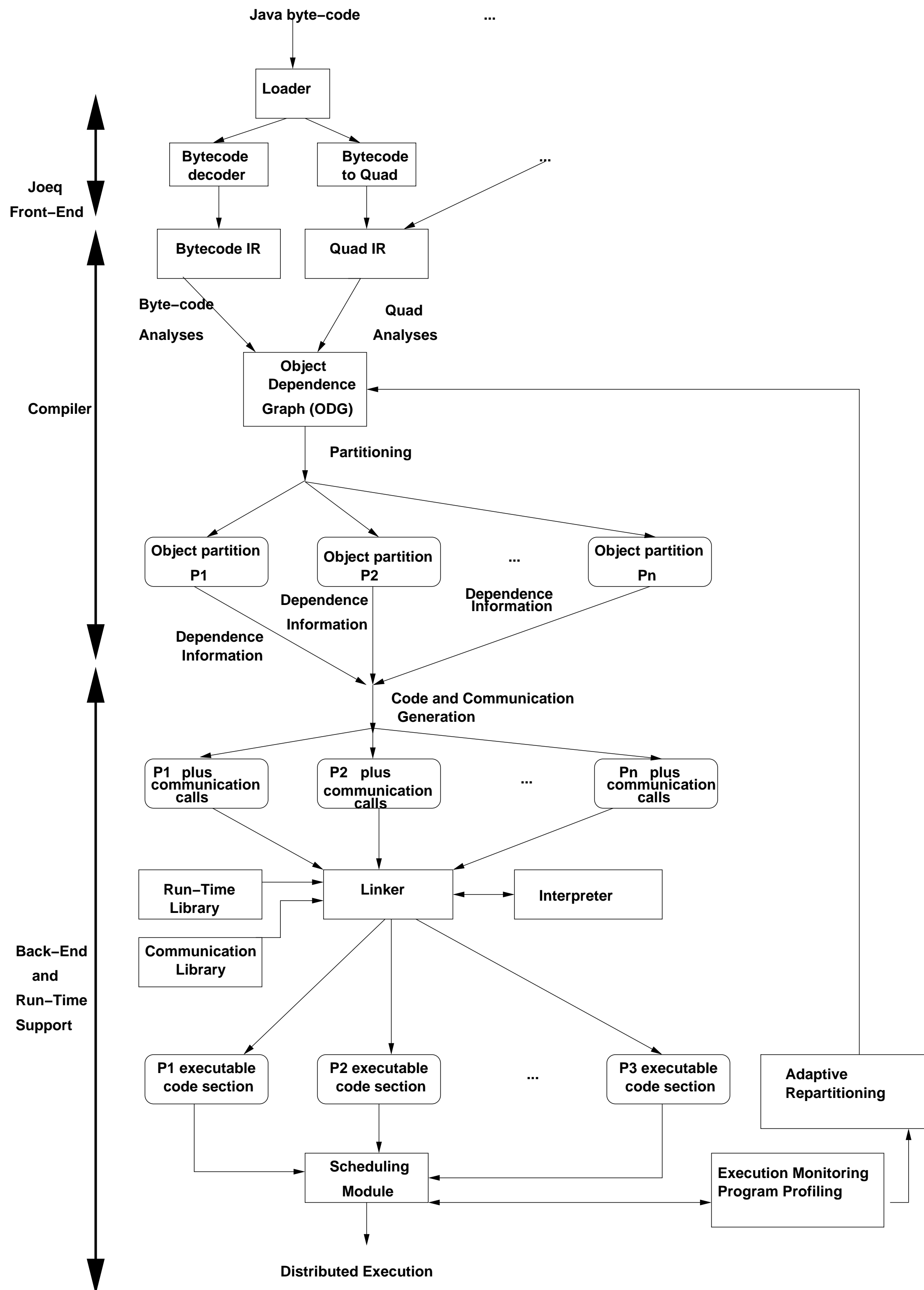

Figure 1: The distributed virtual machine and compiler infrastructure.

Several distributed virtual machine infrastructures are based on the Distributed Shared Memory paradigm. This clearly increases usability. While achieving the same transparency effect, our infrastructure differs in goals and realization from DSM.

The program partitioning technique may be used to split a program in smaller parts and schedule them for execution based on resource availability. Throughout this paper we use the term subprogram to denote a part of the program consisting of a subset of class instances and the computation associated with them. We use the term partition to denote an abstract processor holding only a subset of class instances from the original class instance set of the original program.

Conceptually, each object in an object-oriented program encapsulates data and code. At run-time, this translates into memory usage, CPU time, communication load or battery consumption. Each object in a program consumes resources at run-time. In many situations it would be profitable to be able to play with various execution strategies (interleaving, concurrency, etc.) for parts of the programs to find optimization opportunities.

Object-oriented programs naturally lead to partitioning at object granularity. That is, object-oriented programs provide natural encapsulation of the resource usage with an object. Also, since each object denotes a program variable, it makes sense to partition objects, or program variables, rather than types, or object classes. This choice of granularity also increases the flexibility of choosing the grain of the partitioning strategy. That is, depending on the actual environment characteristics, we can aggregate the objects in a partition in *coarse-grained* or *fine-grained* collections as needed.

This paper makes the following contributions:

- **Infrastructure:** We describe a compiler and run-time infrastructure as the context for our research in program partitioning and scheduling to optimize execution with respect to resource consumption. Our infrastructure is a unifying framework that allows experimentation with flexible distributed execution strategies such as: concurrent processing, synchronous and asynchronous client/server processing and loosely coupled distributed execution (processes loosely synchronizing via message exchange).
- **Analysis:** An improved implementation of an object dependence graph construction algorithm by Andre Spiegel [27]. We improve the original analysis in the following aspects:
  - We analyze programs at byte-code level instead of source-code level.
  - We use an intermediate representation and a reduced set of methods to perform our analyses at method granularity. The original implementation syntactically infers the analysis objects (types, methods, class instances).
- **Results:** We present results of the comparison of three implementations for the dependence analysis algorithm: the original implementation and our improved implementation in two versions, working on two internal representations (byte-code instructions and a register-based representation). The results provide useful insight in program analysis for distributed execution and underline directions for future improvements.

The remainder of the paper is organized as follows. Section 2 gives an overview of the system. Section 3 explains in detail the analysis framework, the intermediate representation and the key structures used to compute the object dependence graph. It also discusses some challenges, our solution to overcome them and some future improvements. Section 4 discusses implementation details. Section 5 presents the results of the class instance graph construction. Section 6 reviews related approaches. Section 7 concludes the paper outlining the future steps to improve on our dependence analysis framework.

2. SYSTEM OVERVIEW

Figure 1 describes the main building blocks of our system:

- **Front-End.** We use Joeq front-end to parse the byte-code and transform the program in the intermediate representation. Any front-end can be added to transform programs from other (binary) representations into the internal Joeq representation. For example, Joeq is designed to handle SUIF files and binary object files (ELF). Our techniques can handle programs written in other languages than Java. However, for the time being, we do not handle pointers (in the C/C++ sense).
- **Compiler.** Joeq provides us with two intermediate representations: byte-code and *quad*. The latter is a quadruple style IR which resembles register-based representations [3]. Joeq compiler offers a wealth of analyses and optimizations. We have extended the compiler with our analysis framework to statically approximate the object dependence graph and resource consumption. This includes the partitioning transformation which results in subsets of class instances and the dependences across partitions.
- **Back-End and Run-Time Support.** The code generation phase includes communication generation for object dependences across different partitions. The run-time support phase includes a scheduling module which performs the mapping of partitions to computation units. The scheduler starts with a static mapping of the partitions to abstract computation nodes. Then, it uses the run-time profiling data to adjust the static parameters (resource consumption) to their run-time values. The adaptive repartitioning module applies the run-time class instance access patterns and resource estimation to the object dependence graph to dynamically repartition the graph.

We can use any virtual machine (modified to support communication) to interpret and run the subprograms, or we can generate code directly for specific platforms. For initial experimentation with homogeneous computation nodes, we plan to install virtual machines on each node. On small, resource-constrained devices we plan to experiment with custom minimalistic virtual machines or binaries compiled directly for the given platform. The communication protocol takes care of the data format and protocol agreements across platforms. We use point-to-point message exchange for efficient communication (with MPI [26]).

The compiler and run-time infrastructure resides on a single, powerful machine which will, in turn, control scheduling on other computation nodes (master/workers model). The master node probes the client machines for dynamic profiling and repartitioning. The

client machines running the subprograms cooperate by exchanging messages (streamed parts of object states) to carry out execution. The subprograms may be running on actual resource constrained devices, or can be off-loaded to other powerful workstations in the systems, etc., depending on the concrete execution platform and scheduling strategy (i.e. ranging from one-to-one client/server distributed computing model, to many-to-many multiple clients/multiple servers models).

The hardest aspect of our infrastructure is the object dependence graph construction. In previous work, we have experimented with general graph partitioning and mapping and we have explored the issues of communication generation, partial replication, distributed consistency and synchronization [8]. Parting from general graph representation and partitioning, these aspects are similar in our present work. However, in our previous work, we only looked at data parallel applications and specific constructs (large sets) for partitioning. Constructing the dependence graph for specific structures (e.g. linear arrays, nested sets) is easier than for user defined, heap allocated structures. In the following sections we will focus on object dependence graph construction from Java byte-code.

3. OBJECT DEPENDENCE GRAPH CONSTRUCTION

Our graph construction algorithm builds on an existing object graph analysis algorithm described in [27]. We preserve some of the semantic of the original algorithm, while changing significantly the algorithmic aspects. The most significant differences are:

- The original analysis targets one distribution paradigm: synchronous remote method call. Our analysis targets a flexible distributed model that includes exploiting concurrency and asynchronous communication. Thus, the existing algorithm it is not sufficient for our proposes. Nevertheless, it gave us extremely useful insight into the problem of object graph construction for program distribution.
- The original algorithm analyzes Java source code. Our algorithm starts from Java byte-code, or other low-level binary representations (SUIF files, ELF object files). Thus, we are not tied with a specific language syntax or source code availability. Also, we can easily handle java library classes that are always accessible in byte-code format.
- The original algorithm relies on syntactically identifying the type closure of the program, as well as many of the analysis objects. That is, it infers the type set of the program by adding types for each syntactic encounter of a reference in the source code. We use state-of-the-art compiler analyses and a full-fledged intermediate representation to account for preciseness, generality and extendibility when computing the class and instance graphs. With our infrastructure we can reuse most of the analyses (e.g. type analysis, call graph construction, class relation construction, etc.) when experimenting with various object dependence graph construction algorithms.
- The original approach is limited in handling libraries and some dynamic aspects (e.g. exception handling).

3.1 An Example

Figure 2 shows an example of a Java program that we use throughout the paper.

```
class Student{
  String name;
  double gpa;
  public Student(String n, double gpa ){
    name = n;
    this.gpa = gpa;
  }
  double Gpa() {return gpa;}
  public String toString(){
    return "Student " + name + " highest gpa " + gpa;
  }
}
public class StudentRegister {
  Vector register = new Vector();
  Student highestGpa;
  void add(Student s){
    register.add(s);
  }
  void maxGpa(){
    ...
    highestGpa = ...;
  }
  public Student highestGpa(){
    return highestGpa;
  }
  public static void main(String[] args) {
    StudentRegister sr = new StudentRegister();
    Student s1 = new Student ("Jens Dahl", 4.57);
    sr.addStudent(s1);
    Student s2 = new Student ("Paola Del Piero", 4.49);
    sr.addStudent(s2);
    Student s3 = new Student("Bogdan Popescu", 5.0);
    sr.addStudent(s3);
    sr.maxGpa();
    System.out.println(sr.highestGpa().toString());
  }
}
```

Figure 2: The example program used throughout the paper.

In our example, there are two classes. A `Student` class describes a student by his/her name and gpa. A `StudentRegister` class uses a `Vector` structure (from `java.util.*`) to store student elements. The `add` method receives a `Student` reference and adds it to the vector structure. The `maxGpa` method computes the maximum `gpa` between all the registered students. The `main` method creates one instance of the `StudentRegister` class and three instances of the `Student` class. Our analysis purpose is to find these instances and their relations.

The appendix lists the quad internal representation for the `Student` class.

We use rapid type analysis to compute the call graph and the program types [12, 33]. Then, for each method in the graph we compute the *class relations* by looking at field access and method call statements. As in [27], a *usage* relation between two classes occurs when one class calls methods or accesses fields of another class. *Export* or *import* relations occur when new types may propagate from one class to another through field accesses or method calls.

Figure 3 shows the class relation graph for our example. The types are annotated with the `ST_` or `DT_` prefix to indicate static or instance (dynamic) versions. The *use* relations tell that some classes occur in the context of other classes and their occurrence is noted by looking at the method calls, field accesses and allocation state-

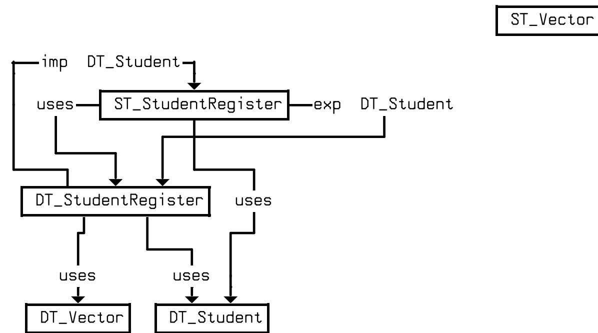

Figure 3: The class relation graph structure visualized with aiSee tool for vcg format.

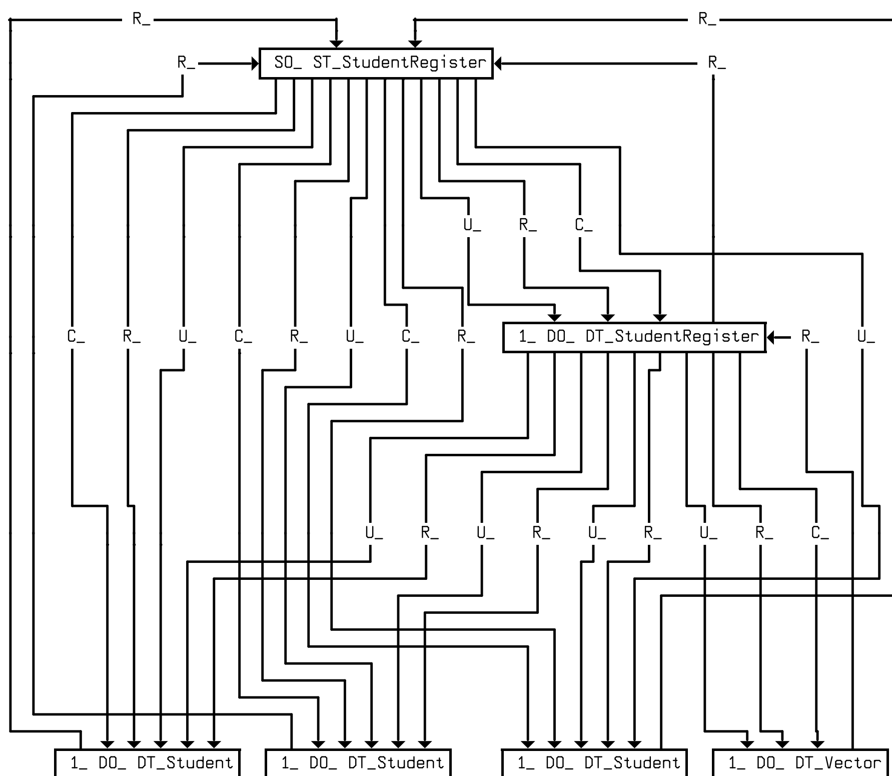

Figure 4: The class instance relation graph structure visualized with aiSee tool for vcg format.

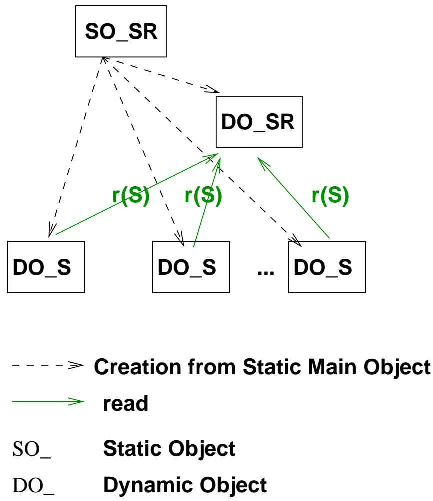

Figure 5: The simplified version of object dependence graph with *read/write* dependences.

ments. The *export* edge occurs due to the invocation of the `addStudent` method on the dynamic `StudentRegister` class with a `Student` class as parameter. The *import* edge occurs due to the `highestGpa()` invocation that returns a result of `Student` type.

Given the class relation graph, and the class instance set, we can compute the relation between class instances (run-time objects). For each allocation statement, we add *reference* relations between the instance of the class where the allocation takes place and the newly created instance. We then create new references by matching the initial class instance references against the *export* and *import* relations between the corresponding classes. We iterate through all object triples and propagate references matching against the type relations until the algorithm reaches a fix point.

Figure 4 shows the class instance graph for our example. The edges are labeled by *create*, *use*, *reference*. The class instances are prefixed by `1_` indicating single instances (a `*_` prefix indicates summary instances of zero or more – i.e. created inside a control structure). The *reference* relation is redundant and only used for intermediate processing. We can safely abandon it. The *create* relation means that an object creates another object. The creation relation between object pairs is propagated to discover new *usage* relations from the class relation graph. Therefore, after the propagation, only the *usage* relation should matter for the partitioning: if an object `a` on abstract processor `Pa` *uses* an object `b` on abstract processor `Pb`, then communication may be generated. However, the relation describes dependence too weakly for our purposes.

Figure 5 shows the *read/write* dependence relation for the same example. This is a more useful dependence graph for a flexible distribution model. It is the key to exploiting concurrency, controlling replication, consistency, and generating communication. Here, it is obvious that the only dependences introduced are *read* via the `StudentRegister.addStudent(Student s)` call. Thus, the application can be partitioned in *server/clients* style, with clients residing on different machines, or fully replicated at the server, depending on the actual running environment.

3.2 Algorithm Overview

The algorithm consists of two main phases. The first phase of the algorithm derives the class relations in a program based on field access, method invocation and array access statements. These relations are captured in a graph. A node exists for each class in the program found by the RTA. An edge connects two classes according to their relations (usage, import, export). The class instance relations are also represented in a graph. A node in the graph represents a new class instance. An edge connects two nodes if one node creates another node either explicitly, via an allocation statement, or implicitly, through superclass relation, constructor invocation or *this* pointer. The algorithm maps the class instance graph onto the class graph to infer the usage relation between class instances.

The structure of the algorithm in our approach is:

1. Class hierarchy and type analysis to construct the call graph and find the types of the program (enhanced RTA).
2. For each field access and method invocation:
   (a) Let A be the context class (containing the statement).
   (b) Let B be the declaring class for the field or method.
   (c) Let C (flow) be one of the following:
      i. The returned type of the method.
      ii. The type of the parameters.
      iii. The type of the field.
   (d) Then add the following dependences:
      i. Usage(A, B).
      ii. Export(A, B, D) for field store and method parameters.
      iii. Import(A, B, D) for field load and return type.
3. For each allocation statement:
   (a) Create a class instance for the corresponding type.
      i. *Single instance* for control independent statements.
      ii. *Summary instances* for control dependent statements.
   (b) Add recursively all super-type instances.
   (c) Add recursively the (implicit and explicit) nested allocations.
   (d) Add a reference from the creating instance to the created one.
4. Propagate (a,b), (a,d) class instance relations to (A, B, D) class relations to find relations between class instances (b, d).

In the first step, we construct the call graph of the application and for each method we add the classes corresponding to allocation statements (*new*).

In the second step, the algorithm approximates the relation between classes in a program. The intuition behind class relations is to approximate relations that may exist between the corresponding class instances. For every class that appears in the context of another class, a *usage* edge exists between the two classes (the edge is created at the end of step 2 of the algorithm). We say that the enclosing class *uses* the class to which it refers. Then, we propagate references between

classes by capturing data flow from fields to method bodies and from method bodies to fields in the given context (class). That is, an *export edge* between two classes indicates that a new class is propagated from one class to another. Therefore, at each point, our analysis keeps track of the current method (method context) and the current class (class context) in which a statement occurs.

In the third step, the algorithm approximates the set of class instances existing in a program. Class instances are created by allocation statements only. For each allocation we add recursively the allocated super-type, as well as the types allocated implicitly through explicit constructor invocations. For each allocation we keep a pointer *back* to its enclosing allocation and a pointer to the actual constructor which is called by the creating allocation. For each class instance that creates another class instance we add a reference edge.

In the fourth step, the algorithm propagates references between class instances by matching the relations between class instances against the type relations. The algorithm constructs the usage edges between class instances based on their reference relation and the corresponding usage relation between their types.

4. ALGORITHM IMPLEMENTATION

4.1 Implementation Details

We have implemented two versions of the algorithm: one that uses the quad internal representation, another that works directly on Java byte-code. Experimenting with both quad and byte-code internal representations gives useful insight on how the internal representation helps with various analyses. A register-like representation is more suitable for static analysis and optimizations of programs.

4.1.1 Call Graph Construction

Since we use Joeq's front-end and quad representation along with the control flow graph structure, our implementation is fairly small (about 5000 lines of code for the quad version of the algorithm).

Our enhanced RTA is different from a standard RTA algorithm. The standard algorithm concentrates on method invocations and does not consider field accesses and methods that are not invoked (`main, <clinit>`). Therefore, we process field accesses and special methods as well and add new classes to the set of classes in the original RTA.

4.1.2 Class Graph Construction

The implementation of the class relation computation is an iterative, propagation-based style algorithm. We perform our analyses on the call graph. We start with the *main* method and the corresponding class. For each reachable method, we traverse the control flow graph and detect the statements of interest : field accesses, method calls, allocation statements and array accesses. We adopt Joeq's *visitor pattern* (see [11] for a detailed discussion of the visitor pattern) to compute the relation between classes. Joeq offers numerous visitor interfaces to visit byte-code, quads, types, methods, classes, basic blocks and so forth. A compiler developer only needs to implement the visitor interfaces with the concrete actions to be performed for the structure (byte-code or quad instructions, types, etc.) of interest.

The control flow graph for a method consists of basic blocks and each basic blocks is a list of quads. A quad is a quadruple consisting of an operation and up to three operands. Almost all the byte-code instructions have a correspondent in the quad format. A pass computing the class relations traverses the quads for each basic block in the control flow graph. The pass is both a *type visitor* and a *quad visitor* and it receives a pointer to the currently analyzed method. Thus, we are able to reconstruct high-level information from the low-level quad representation. That is, we can precisely infer all the information needed by the analysis when constructing the type relation: the context class and the context method and the type of elements in the statement. By context class we mean the actual class where the currently analyzed method occurs. By context method we mean the actual method where the statement occurs. Once we have computed all the type relations as in the second step of the algorithm, we propagate the relations to all the subtypes.

4.1.3 Class Instance Graph Construction

4.1.3.1 Quad Implementation

We compute all the *allocations* in the program in the beginning of the analysis. An *allocation* object contains the following information: the calling context as the current method where the allocation statements occurs, the class of the allocated object, the invoked constructor, the quad location of the allocation within the calling method, a pointer back to the enclosing allocation if the current allocation is nested inside another allocation. We use hash-based data structures to keep the lists of program classes and allocation objects for the duration of the analysis.

The class instance graph is also a propagation style iterative algorithm. We keep two separate instances for the static and dynamic part of a class. A *static* class instance consists of all class variables and methods. A *dynamic* class instance consists of all the instance class variables and methods. The algorithm first adds all the static class instances to the class instance set of the program. Then, for each static class instance it adds all the class instances instantiated in the corresponding class context (through the reachable methods). We distinguish between class instances created by allocation statements depending on the control flow:

- A *single class instance* is created for a *control flow independent* allocation statement. A single class instance represents exactly one instance of a class, since the corresponding allocation statement executes precisely once.

- A *summary class instance* is created for a *control flow dependent* allocation statement. A summary class instance represents zero or more instances of a class, depending on the actual flow of control at run-time.

We uniquely identify program locations in our analysis. That is, each statement in a program can be identified by a quad and a method location. Each quad in a method has a unique identifier. Each method in the call graph can be identified by the corresponding invocation quad. Therefore, a statement in a program can be uniquely identified by its corresponding quad and the context method (the method where the statement occurs).

To account for control independence, we first compute the dominators for each method. Then, for each program location consisting of a quad and a method, we check if the quad is in a basic block that dominates the exit. If so, then the quad is control flow independent relatively to the method it belongs to (i.e. *intra-procedurally*).

Then, for all the callers of the method, we check if the corresponding program location is control independent by checking the quad location and all the callers recursively (i.e. *inter-procedurally*). Since we keep the *actual* calling context for allocation statements, the control independence check is more precise for the allocation statements. That is, we store the quad, the actual method and the actual constructor for each allocation statement in an initial traversal of the call graph. Thus, the single and summary class instance approximations are more precise. This implementation leads to more accurate results than both, the byte-code and original, Pangaea [28] implementation. Both of the latter approaches fail to identify control independence inter-procedurally, leading to a larger set of summary class instances and a smaller set of single class instances (i.e. more imprecise result).

Then, for each dynamic class instance we add single and summary class instances depending on the corresponding allocation statements.

For each newly added class instance we add a *reference* from the creating to the created class instance. At the end of this step (3), we obtain an initial class instance graph approximation consisting of class instances and their references.

We first map this initial graph onto the corresponding class graph to propagate references as in step 4 of the algorithm. We use the class instance graph and the class graph to propagate *usage* relations.

4.1.3.2 Byte-code Implementation

In the byte-code analysis implementation, we start the class instance graph construction by identifying the control flow independence of allocations. An allocation is control independent if it meets the following constraints:

1. The allocation appears in an initializing method that is guaranteed to execute exactly once when the type is instantiated. We treat *static* types and *instance* types differently.
   - Static type: The class initializer `<clinit>` is the initializing method for static types. `<clinit>` will execute exactly once when the class is loaded at runtime. A special initializing method is the program's entry point, the `main` method.
   - Instance type: The initializing methods for instance type include the `<init>` constructors and the `run()` method for a thread instance type. They both execute once when a Java class is instantiated.
2. The allocation only executes exactly once when the enclosing method is invoked.

The byte-code analysis uses a simple approach to check the control flow independent region for each initializing method. As we have noted before, this approach leads to a less precise result than the quad implementation.

- Construct Control Flow Graph (CFG) for the method. Set the initial *independent* attribute for each basic block to `true`.
- Traverse the method byte-code. Whenever a branch statement appears, set the *independent* attribute to `false` for all the basic blocks in between the current statement and the target. The byte-code branch statements include `if<cond>, if_Xcmp<cond>, goto, tableswitch`, and `lookupswitch`.

The output is one *independent* attribute for each basic block with `true` being the control flow independent code segment.

The byte-code implementation to locate allocations is not straightforward. That is, allocations appear in byte-code sequence pattern `new, dup, ..., invoke <init>`[2]. We use a stack to model a very simple Pushdown Automaton (PDA) to match this pattern.

4.2 Implementation Issues

Generally, Java program analysis typically encounters the following particularities:

Arrays. We model arrays as first class entities with elements of the same type. Array access statements have direct correspondents in the quad representation and therefore we can identify the type of array elements. We treat array accesses as instance field accesses (e.g. a[i] is treated as access to instance field i with the same type as a[i]). Then, it is straightforward to proceed with type propagation as for the load and store instructions in step 2 of the algorithm. In the future, we plan to treat arrays as collections of objects that can be themselves partitioned.

Exception handling. We treat exceptions as regular class instances. That is, we treat exceptions as return statements for the purpose of class relation construction and propagation in step 2 of the algorithm. For each basic block, there is a list of thrown exceptions associated with it. In step 3 we treat exception class instances as regular class instances.

Class libraries. Our analysis separates between program and library classes. Libraries become a problem when propagating relations through super-classes. All the parts of our analysis can handle classes from the standard libraries and their instances. We only include the library classes that are directly referred in the code (first-level library classes) in the graph representation. We do not include their super-classes. That is because libraries increase the size of the call graph significantly and therefore, the sizes of the class relation and class instance graphs. To reduce the size of these graphs we intend to associate all classes and method calls from libraries with external sets of classes and methods. Then, we treat differently the propagation via method calls and superclass relation to the program defined classes and external classes.

Reflection and Dynamic Loading. Handling reflection and dynamic loading statically is difficult. Since our focus is to approximate the class instance relation graph statically, we do not address these issues in our present implementation.

Specifically, we address some particular situations in our algorithm implementation as follows:

[2]The pattern `new, ..., invoke <init>` also exists. But such an allocation does not give out a reference. So we do not locate such allocations. Also, for simplicity, we assume `new` and `dup` always go together.

- We filter out `java.lang.System`, `java.lang.String`, `java.lang.StringBuffer`, and `java.io.PrintStream`. (e.g. in our example in Figure 2 `StringBuffer` and `System` would have been introduced by the call `System.out.println`).
- We do not add export and import edges to or from the library classes, since they do not introduce new references during propagation. We add export and import edges with library classes as data flow, because they may introduce new reference propagations.
- We should not introduce any relation in the case of `this` accesses. We can identify the `this` pointer from the `quad` instructions (a special register stores it). However, in very few situations, Joeq uses normal, temporary registers to hold the `this` pointer. In such situations we cannot identify it.
- We do not add *usage* edges for the special `<init>` invocation, since the invocation belongs to an allocation and we treat them separately.
- We keep distinct instances of summary class instances created by distinct allocation statements inside the loop (even when they are of the same type). This treatment allows to exploit and handle concurrency correctly when distinguishing between objects accessed in different instances of the loop.

5. RESULTS

We measure the following parameters for our graph construction algorithms for each of the applications:

- The characterization of the application (#classes, #methods, LOC, byte-code size).
- The time to construct the call graph.
- The number of nodes (classes) in the class relation graph.
- The number of edges (relations) in the class relation graph (the number of use, export and import).
- The time to construct the class relation graph.
- The number of nodes (class instances) in the class instance graph.
- The number of edges in the class instance graph (create, use, reference).
- The time to construct the class instance graph.

5.1 Benchmarks

Table 1 describes the programs that we used to evaluate our algorithm implementation. The first two programs are just simple examples that we wrote, while the remaining applications belong to the Java Grande benchmark [24]. Due to space constraints, we only selected (randomly) two applications from each of the three sections of the Java Grande applications set.

Table 1 shows, for each program, the number of classes it contains, the number of methods, the number of lines of code and the actual byte-code size. For the call graph, we show the number of classes found by the RTA analysis and the number of reachable methods. Our algorithm analyzes only the reachable methods and their actual context classes. The original algorithm analyzes all the classes syntactically referenced in the main program, along with their methods and fields, irrespective of their actual use.

5.2 Graph Sizes

Table 2 shows the sizes for all the graphs in the program. We construct the object dependence graph statically and therefore we can perform an initial static partitioning of the program on this graph. We use partitioning algorithms that can handle graphs of order of million of nodes in seconds [1]. Thus, we expect the initial static partitioning to be fast. However, the size of the graph is important for both the dynamic profiling and adaptive repartitioning.

The results in Table 2 show the number of nodes and edges for the class and class instance graphs in the three implementations: Pangaea original implementation [28], and our implementations in both quad and byte-code versions. The sizes of the class relation graphs in Pangaea are sometimes bigger. This is because, in our versions of the algorithm, we treat libraries differently. On the one hand, Pangaea adds library classes super-classes or super-interfaces to the class set of a program, while we do not. On the other hand, Pangaea adds classes by symbolic analysis at the source code level, while we only add classes that are reachable via method invocation or field access.

Our class instance graphs are sometimes bigger than Pangaea. This is partially due to the difference in the class relation graphs sizes. Another reason is that Pangaea does not distinguish between class instances allocated in different methods or in different statements of a loop. Since we record the context method of an allocation, we create separate class instances for different methods and hence, the larger number of class instances. Pangaea also fails to identify some single class instances due to its symbolic analysis. For instance, Pangaea does not identify initializations like `int[] data = {1, 2, 3}`. Both Pangaea and the byte-code implementation are limited in correctly identifying the control dependent and control independent statements. Thus, the quad implementation is the most precise when finding the instances of the classes created just once (single class instances).

The results we have presented so far show that the graph size is well under the realistic sizes (order of millions of nodes) that graph partitioning algorithms can handle. This is an encouraging result. We plan on improving on the algorithm so that it identifies or classifies class instances more precisely (e.g we plan on treating arrays as collections of objects that can be partitioned) and therefore we expect the graph sizes to increase. However, we believe that the sizes of the graphs for realistic applications will stay within manageable limits.

5.3 Graph Construction Times

Table 3 shows the times associated with the construction of each graph for the analyzed program. These times only influence the static partitioning process that can be performed ahead of time for Java byte-code. As explained in the previous sections, these times have a greater impact on the dynamic aspects of partitioning.

The results in Table 3 show that the time to construct the call graph is the greatest of all (Pangaea does not construct a call graph). The times for the class and class instance graph construction are smaller and sometimes, a lot smaller than in the Pangaea implementation (e.g. almost twice smaller in the quad implementation). This is because, once the call graph constructed, we reduce significantly the size of the analyzed sets. Furthermore, only the class instance

[3]Pangaea does not use a call graph. Instead, it uses standard syntactic type inference.

Table 1: The benchmark characterization.

program	#classes	#methods	LOC	byte-code size
test	12	27	108	4,382B
StudentRegister	2	7	37	2,032B
Create	14	28	672	13,222B
Method	6	35	431	10,133B
FFT	5	40	321	11,833B
HeapSort	5	36	246	9,547B
MolDyn	7	42	503	16,555B
MonteCarlo	17	192	1043	41,891B
Average	8.5	51	420	13,699B

Table 2: The sizes for the graphs in the program.

version	program	Call Graph (methods)		Class Relation Graph				Class Instance Graph			
		Reachable	Analyzed	Nodes	Use	Imp	Exp	Nodes	Create	Use	Ref
quad	test	21	17	10	7	1	1	12	11	7	28
	StudentRegister	438	8	5	4	1	1	6	5	8	13
	Create	454	28	17	6	0	0	210	208	6	418
	Method	452	30	12	10	0	0	9	6	10	16
	FFT	471	28	17	11	0	0	14	9	10	24
	HeapSort	467	27	13	13	0	0	11	8	7	18
	MolDyn	550	33	12	15	0	0	9	7	9	16
	MonteCarlo	697	103	40	53	2	7	110	106	43	225
bytecode	test	21	17	13	12	2	1	13	10	12	19
	StudentRegister	3243	8	7	5	1	1	6	5	8	8
	Create	3261	29	22	6	0	0	291	289	6	291
	Method	3263	33	14	13	0	0	11	6	13	13
	FFT	3267	30	20	18	1	2	14	9	23	24
	HeapSort	3261	28	13	15	0	0	9	7	9	9
	MolDyn	3276	35	21	35	48	6	16	10	29	33
	MonteCarlo	3468	112	55	130	5	9	128	114	238	289
pangaea	test			16	15	3	3	13	10	12	19
	StudentRegister			13	5	3	2	6	5	8	11
	Create			28	6	1	1	22	20	6	24
	Method			19	11	1	1	9	5	11	12
	FFT			28	21	2	3	14	8	23	25
	HeapSort			18	10	1	1	7	5	7	8
	MolDyn			27	27	9	5	13	8	18	21
	MonteCarlo			73	118	14	13	60	46	159	190

Table 3: The times (in milli-seconds) for the construction of the graphs.

version	program	Call Graph/Type Closure[3]	Class Relation Graph	Class Instance Graph	Total
quad	test	440	24	52	516
	StudentRegister	2147	26	25	2198
	Create	2127	44	1829	4000
	Method	2022	42	43	2107
	FFT	2017	49	35	2101
	HeapSort	2023	35	33	2091
	MolDyn	2206	45	82	2333
	MonteCarlo	2607	202	185	2994
	Average	1948.6	58.3	285.5	2292.5
bytecode	test	287	813	58	1158
	StudentRegister	339	787	50	1176
	Create	541	1307	319	2167
	Method	432	993	55	1480
	FFT	507	1026	59	1592
	HeapSort	483	897	54	1434
	MolDyn	940	1098	66	1774
	MonteCarlo	940	1412	160	2512
	Average	517	1041	103	1662
pangaea	test	189	58	85	332
	StudentRegister	278	54	79	411
	Create	718	117	375	1210
	Method	505	101	111	717
	FFT	653	81	177	911
	HeapSort	541	76	90	707
	MolDyn	714	130	164	1008
	MonteCarlo	1680	230	465	2375
	Average	660	106	193	959

graph construction time will have impact on dynamic profiling and repartitioning. The delay introduced should be small enough for our purposes and definitely smaller than in a Pangaea like implementation. Also, byte-code analysis (for the byte-code version) is more time consuming since it requires stack operation emulation (e.g. to identify the actual parameter passing between caller and callee).

In the quad implementation, there is an unusual large time associated with the class instance graph construction for the `Create` benchmark program. This is because both Pangaea and the byte-code implementation fail to identify an inter-procedural control-independent call site and thus, conservatively summarize instances created through this site.

6. RELATED WORK

6.1 Program Partitioning for Scientific Applications

The problem of program partitioning has been explored by many researchers with the hope that would significantly increase the optimization gain. Parallel compilation (automatic concurrency support) is one research area that has investigated the partitioning problem mainly for scientific programs typically targeting a significant reduction in CPU or memory consumption [15, 19, 4, 13, 38, 18, 35, 30, 7, 16, 21, 9, 36, 2]. There are two main differences between partitioning for scientific applications and our work. Most of the previous work focuses on array partitioning, or loop iteration partitioning in scientific applications. We address general program partitioning, where all the objects in a program are of interest. Second, the main objective for partitioning in scientific application is to speedup execution, either on distributed or on shared memory machines. Our partitioning objective is to be able to come up with execution strategies that can accommodate programs in resource-constrained environments. This does not necessarily mean CPU, but also battery, memory, storage space etc. Our general techniques should however be applicable to special situation like partitioning arrays or loop iterations in scientific applications.

Typical analyses for array-based languages are data and control dependence for arrays accesses that are affine expressions of loop indexes. Such analyses are able to infer the dependences between elements of linear structures (i.e. multidimensional arrays) and statically compute the location of the elements on a processor. The assignments of elements to processors is also linear (block, cyclic). Therefore the compiler can statically generate communication based on data dependences and elements location.

We also aim at being able to statically determine dependences between program variables. In our case, instead of arrays and affine array accesses, we deal with heap allocated objects. We also assign statically each object to a partition using multi-objective, general graph partitioning algorithms [25]. Instead of computing the location, we keep track of abstract locations based on processor and object identifiers. Therefore, we can also generate communication statically. However, this is only a *best initial guess* for partitioning the given program. We intend to enhance Joeq run-time to monitor the program and use both static and dynamic techniques to recompute the object dependence graph based on actual execution knowledge and possibly repartition the application dynamically.

6.2 Distributed Shared Memory

Another approach to transparently schedule applications onto multiple resources is the distributed shared memory approach [20, 17]. The main problem in this approach is to ensure consistency between address spaces. Since the approach is at the memory level, each read or write to memory has to be synchronized. Our approach is closer to the application level, in that we keep track of a finite number of objects and their interactions for a given program. Then, we only need to consider consistency problems when distributing those objects. This paper did not address further analyses to optimize communication or execution scheduling.

The Orca language [5, 6] is a pioneering effort on object-based program execution in distributed environments. Hawk [14] runtime system supports *partitioned objects* for distributed applications. Such efforts are very inspiring for work on program partitioning, scheduling and execution consistency.

6.3 Java Program Partitioning

Several algorithms analyze source code to build the graph representation for a partitioning framework. The graph representation abstracts different program aspects. For instance, [29] uses the *behavioral edges* to represent dependence information and the interactions of the communication and computation. The idea is to find the dominant and sensitive program components and then apply an iterative scheduling that tries to optimize their execution. Another approach [37] is to find optimal schedules for special DAGs (fork, join, coarse-grain trees, some fine-grain trees). In [32] the control flow graph structure contains information on merge nodes, distribution flow edges and relationship edges. The relationship edges use reaching definition data flow information. The partitioning is dynamic and targets array structures.

Recently, partitioning has gained interest for Java community as a means to design distributed virtual machines for automatic program distribution. JavaParty [23] extends Java with *remote objects*. The idea is to provide location transparency in a distributed memory environment. We achieve the transparency effect without extending Java syntax. However, we do not give the user any control over distribution. This is because our research infrastructure is aimed at system designers who want to experiment with various optimization techniques and have all the control at the system level. In our scenario, the user would only run an application on a (possibly mobile) device (in a network) and would not be aware of how the application is actually executed.

An approach closer in goals with ours is [22]. The idea is to transparently off-load portions of service to relieve memory and processing constraints on resource-constraint devices. The partitioning is dynamic, based on application monitoring and class interaction. The granularity is at class level. The main problem is the handling of object references. In this approach each JVM maps all other JVMs references, and thus it results in a *replicate all* approach. Our approach is static, and it considers also class instance interaction.

An approach similar to the distributed shared memory paradigm is to implement a distributed JVM as global object space [10]. This approach virtualizes a single Java object heap across machine boundaries, implicitly ensuring location transparency. There are two types of objects. *Node local* objects are reachable from a node, while *distributed* objects are reachable from multiple nodes. Then, *reachability analysis* is performed on an object connectivity graph to detect the *local* and *escaping* objects. This approach assumes that objects can reside anywhere and thus can escape the local address space. We control the partitioning and analyze the dependences and thus, we know exactly where each object resides.

J-orchestra [31] transforms Java byte-code into distributed Java applications. This is also an abstract shared memory implementation, consisting of two steps. First, it classifies objects as *anchored* and *mobile*. Second, it converts all references into *indirect* references. The communication middleware is RMI and the approach is quite expensive.

Pangaea [28] is a system that can distribute Java programs using arbitrary middleware (Java RMI, CORBA) to invoke objects remotelly. The system is based on the original algorithm by Spiegel that we also use. Pangaea's input is a centralized Java source-code program. The result is a distributed program underlining the synchronous remote method invocation distributed computing paradigm. Our approach starts from Java byte-code and targets a flexible distribution model (i.e. allows to exploit concurrency and asynchronous communication) to distribute the program.

7. CONCLUSION AND FUTURE WORK

This paper has presented a dependence analysis in the context of a compiler and run-time virtual machine infrastructure for automatic distributed program execution. The goal of our analysis is to facilitate research in automatic program distribution by providing a unifying framework for general program partitioning and scheduling. The motivation of our work is flexibility to experiment with various partitioning strategy to respond to various optimization targets in heterogeneous environments.

We have presented an improved implementation of an existing static algorithm [27] to construct the object dependence graph for an application. Our implementation of the algorithm constructs the object dependence graph directly from Java byte-code. Also, we target a flexible distribution model that allows to exploit concurrency and asynchronous communication between subprograms. The original analysis is designed to handle Java source code and use the synchronous remote method invocation distribution paradigm. Our analysis implementation differs significantly from the original implementation in the realization of the various phases of the algorithm. We have shown that the preciseness of the algorithm is increased in our implementation. Also, we have shown that the high cost associated with the call graph construction time is a small price to pay for the increased preciseness and reduced time for the later phases of the algorithm.

We have also shown that for the partitioning and scheduling purposes the relations between the class instance graph are too weak. We are working on an improved version of the algorithm that eliminates some of the redundant relations and translates all the class instance interactions into *read* and *write* interactions. Such classification is the key to efficient communication generation and partial replication.

So far, we have presented an all static approach to automatic program distribution. However, the static partitioning is only an initial approximation. In the future, we plan to adaptively partition the graph and reschedule the program for distribution based on dynamic profiling of actual flow of control, class instance access patterns and resource consumption.

8. ACKNOWLEDGEMENTS

Parts of this effort are sponsored by the National Science Foundation under ITR grant CCR-0205712 and by the Office of Naval Research under grant N00014-01-1-0854.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of the National Science foundation (NSF), the Office of Naval Research (ONR), or any other agency of the U.S. Government.

We thank Andre Spiegel for handing his code to us and enabling the comparison of the three implementations.

9. REFERENCES

[1] Metis family of multilevel partitioning algorithms. Available at: http://www-users.cs.umn.edu/ karypis/metis/.

[2] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. An overview of a compiler for scalable parallel machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 253–272, Portland, OR, August 1993. ACM press.

[3] C. S. Ananian. Available at: www.flex-compiler.csail.mit.edu/ Harpoon/quads/quads.pdf, October 1998.

[4] C. Ancourt and F. Irigoin. Automatic Code Distribution. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[5] H. E. Bal and M. F. Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In A. Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, volume 28 of *SIGPLAN Notices*, pages 162–177, New York, NY, 1993. ACM Press.

[6] H. E. Bal, M. F. Kaashoek, A. S. Tanenbaum, and J. Jansen. Replication Techniques for Speeding up Parallel Applications on Distributed Systems. *Concurrency Practice and Experience*, 4(5):337–355, August 1992.

[7] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 213–223, Williamsburg, Virginia, United States, 1991. ACM Press.

[8] R. E. Diaconescu. *Object Based Concurrency for Data Parallel Applications: Programmability and Effectiveness*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, August 2002. [NTNU 2002:830, IDI Report 9/02, ISBN 82-471-5483-8, ISSN 0809-103X].

[9] G. Edjlali, G. Agrawal, A. Sussman, J. Humphries, and J. Saltz. Compiler and runtime support for programming in adaptive parallel environments. *Scientific Programming*, pages 215–227, January 1997.

[10] W. Fang, C.-L. Wang, and F. Lau. Efficient global object space support for distributed jvm on cluster. In *International Conference on Parallel Processing*, Vancouver, Canada, August 2002.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing. Addison-Wesley, 18th printing edition, September 1999.

[12] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *ACM Conference on ObjectOriented Programming Systems, Languages, and Applications (OOPSLA).*, pages 108–124, 1997.

[13] M. Gupta and P. Banerjee. Paradigm: a compiler for automatic data distribution on multicomputers. In *Proceedings of the 7th international conference on Supercomputing*, pages 87–96. ACM Press, 1993.

[14] S. B. Hassen, I. Athanasiu, and H. E. Bal. A flexible operation execution model for shared distributed objects. In *Proceedings of the OOPSLA'96 Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 30–50. ACM, October 1996.

[15] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, 1992.

[16] D. E. Hudak and S. G. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 4th international conference on Supercomputing*, pages 187–200. ACM Press, 1990.

[17] L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):498–507, 1999.

[18] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proceedings of the 1995 conference on Supercomputing (CD-ROM)*, page 76. ACM Press, 1995.

[19] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[20] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 229–239, New York, NY, 1986. ACM Press.

[21] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24:445–475, 1998.

[22] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, T. Giuli, and X. Gu. Towards a distributed platform for resource-constrained devices. In *Proceedings of 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*. IEEE, July.

[23] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[24] J. G. B. Project. Java Grande Forum Benchmark Suite - Version 2.0. Edinburgh Parallel Computing Centre, The University of Edinburgh, 1999. email: epcc-javagrande@epcc.ed.ac.uk.

[25] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. In J. Dongarra et al., editors, CRPC Parallel Computing Handbook. Morgan Kaufmann, 2000 (in press), 2000.

[26] M. Snir, S. Otto, S. H. Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, second edition, September 1998.

[27] A. Spiegel. Object graph analysis. Technical Report B-99-11, FreieUniversitat Berlin, 1999.

[28] A. Spiegel. PANGAEA: An automatic distribution front-end for JAVA. In *IPPS/SPDP Workshops*, pages 93–99, 1999.

[29] R. Subramanian and S. Pande. A framework for performance-based program partitioning. pages 151–169, 2001.

[30] A. Sussman. Model-driven mapping onto distributed memory parallel computers. In *Proceedings of the 1992 conference on Supercomputing '92*, pages 818–829. IEEE Computer Society Press, 1992.

[31] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP 2002 - Object-Oriented Programming: 16th European Conference Malaga, Spain*, volume 2374/2002 of *Lecture Notes in Computer Science*, pages 178–204. Springer-Verlag, June 2002.

[32] J. Tims, R. Gupta, and M. L. Soffa. Data flow analysis driven dynamic data partitioning. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 75–90, 1998.

[33] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, 2000.

[34] J. Whaley. Joeq: A virtual machine and compiler infrastructure. In Proceedings of the Workshop on Interpreters, Virtual Machines, and Emulators, San Diego, CA. Pages 58-66., 2003.

[35] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proceedings of the 6th international conference on Supercomputing*, pages 25–34. ACM Press, 1992.

[36] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6):737–753, June 1995.

[37] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trnasactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

[38] A. Zaafrani and M. R. Ito. Partitioning the global space for distributed memory systems. In *Proceedings of the 1993 conference on Supercomputing*, pages 327–336. ACM Press, 1993.

APPENDIX

This section lists the Joeq quad representation for the `Student` class in our example. The print pass inherits from three Joeq *visitors*: a quad visitor, a basic block visitor and a control flow graph visitor. That is, it first visits each method in the class, then it visits its control flow graph, and for each basic block, it lists its quads.

In the listing, each method shows the basic blocks as well as the entry and exit basic blocks. We do not show the associated exception handlers for readability reasons. Each basic block identifier is unique, with zero and one designating the entry and exit basic block. Each quad lists its identifier, the operator and its operands. However, the quad identifiers hold no meaning.

For instance, the operator in quad 1 in method `Student.<init>` is `INVOKE_SPECIAL` operator, with return type void (the `_V` suffix), which may need to be loaded dynamically (the `%` symbol). Further, the `java.lang.object.<init> ()V` indicates that the current invoke needs to invoke the initialization function for the corresponding superclass (`java.lang.object`).

Each quad defines or uses a number of registers for its operands. These are divided into temopary (T_i) and non-tempotray (R_i) registers. The special register R_0 is used to hold the `this` pointer (in this case a pointer to the current class instance).

```
Student.<init> (Ljava/lang/String;D)V

BB0 (ENTRY)  (in: [] out: [BB2] )
BB2  (in: [BB0 (ENTRY)] out: [BB1 (EXIT)] )
 2   NULL_CHECK              T-1 <g>,R0 Student
 1   INVOKESPECIAL_V%             java.lang.Object.<init> ()V,(R0 Student)
 3   NULL_CHECK              T-1 <g>,R0 Student
 4   PUTFIELD_A              R0 Student,.name,R1 String,T-1 <g>
 5   NULL_CHECK              T-1 <g>,R0 Student
 6   PUTFIELD_D              R0 Student,.gpa,R2 double,T-1 <g>
 7   RETURN_V
BB1 (EXIT)  (in: [BB2] out: [] )

Student.Gpa ()D

BB0 (ENTRY)  (in: [] out: [BB2] )
BB2  (in: [BB0 (ENTRY)] out: [BB1 (EXIT)] )
 1   NULL_CHECK              T-1 <g>,R0 Student
 2   GETFIELD_D              T0 double,R0 Student,.gpa,T-1 <g>
 3   RETURN_D                T0 double
BB1 (EXIT)  (in: [BB2] out: [] )

Student.toString ()Ljava/lang/String;

BB0 (ENTRY)  (in: [] out: [BB2] )
BB2  (in: [BB0 (ENTRY)] out: [BB1 (EXIT)] )
 1   NEW                     T0 StringBuffer,java.lang.StringBuffer
 2   MOVE_A                  T1 StringBuffer,T0 StringBuffer
 4   NULL_CHECK              T-1 <g>,T1 StringBuffer
 3   INVOKESPECIAL_V%             java.lang.StringBuffer.<init> ()V,(T1 StringBuffer)
 6   MOVE_A                  T1 String,AConst: "Student "
 7   NULL_CHECK              T-1 <g>,T0 StringBuffer
 5   INVOKEVIRTUAL_A%        T0 StringBuffer,java.lang.StringBuffer.append (Ljava/lang/String;)
                             Ljava/lang/StringBuffer;, (T0 StringBuffer, T1 String)
 8   NULL_CHECK              T-1 <g>,R0 Student
 9   GETFIELD_A              T1 String,R0 Student,.name,T-1 <g>
 11  NULL_CHECK              T-1 <g>,T0 StringBuffer
 10  INVOKEVIRTUAL_A%        T0 StringBuffer,java.lang.StringBuffer.append (Ljava/lang/String;)
                                     Ljava/lang/StringBuffer;, (T0 StringBuffer, T1 String)
 13  MOVE_A                  T1 String,AConst: " highest gpa "
 14  NULL_CHECK              T-1 <g>,T0 StringBuffer
 12  INVOKEVIRTUAL_A%        T0 StringBuffer,java.lang.StringBuffer.append (Ljava/lang/String;)
                                     Ljava/lang/StringBuffer;,(T0 StringBuffer, T1 String)
 15  NULL_CHECK              T-1 <g>,R0 Student
 16  GETFIELD_D              T1 double,R0 Student,.gpa,T-1 <g>
 18  NULL_CHECK              T-1 <g>,T0 StringBuffer
 17  INVOKEVIRTUAL_A%        T0 StringBuffer,java.lang.StringBuffer.append
                                     (D)Ljava/lang/StringBuffer;,
                             (T0 StringBuffer, T1 double)
 20  MOVE_A                  T1 String,AConst: ""
 21  NULL_CHECK              T-1 <g>,T0 StringBuffer
 19  INVOKEVIRTUAL_A%        T0 StringBuffer,java.lang.StringBuffer.append (Ljava/lang/String;)
                                     Ljava/lang/StringBuffer;,(T0 StringBuffer, T1 String)
 23  NULL_CHECK              T-1 <g>,T0 StringBuffer
 22  INVOKEVIRTUAL_A%        T0 String,java.lang.StringBuffer.toString ()Ljava/lang/String;,
                                     (T0 StringBuffer)
 24  RETURN_A                T0 String
BB1 (EXIT)  (in: [BB2] out: [] )
```