Towards Autonomic Distribution of Existing Object Oriented Programs

Debzani Deb, M. Muztaba Fuad, Michael J. Oudshoorn
Department of Computer Science
Montana State University
Bozeman, MT 59717, USA
E-mail: {debzani, fuad, michael}@cs.montana.edu

Abstract

By harnessing computational power of distributed heterogeneous resources, it is possible to build a large scale integrated system so that a centralized program is partitioned and distributed to those resources in a way that results in both efficient execution of the program and maximized resource utilization. However, building such a system is a staggering challenge because of the associated complexities and required user intervention. This paper proposes an autonomic distributed architecture that statically analyzes the existing Java application, partitions it to self-managed components that handles the complexities related to distribution and coordination without user involvement. An efficient static analysis mechanism is implemented that identifies run time program instances and their dependencies in terms of a graph. It is observed that such a view of the program is essential towards self optimization and self management.

1. Introduction

For large scale, computationally intensive applications there is much to be gained from globally distributing the application across distributed and heterogeneous machines connected with diverse communication capacities and obtaining a higher throughput and performance speed-up. By harnessing the computing power and storage of these idle resources widely available within corporations, research institutes, universities etc. it is possible to build a large scale integrated computing environment. With such an infrastructure, one can effectively partition a large centralized application in terms of communicating components and distribute those components among the available resources in a way which results in the optimized execution of user program and maximized resource utilization.

However, there are many challenging aspects associated with effectively partitioning a large problem and mapping and scheduling those partitions over the heterogeneous resources across the system. Firstly, the programmer wishing to execute such an application may not have necessary skills to effectively partition and distribute the application manually. Furthermore it becomes tedious for a programmer when he needs to handle distribution aspects such as fault tolerance, load balancing and resource allocation and communication as well. Secondly, proper utilization of such a computing environment requires ways of estimating each components computation and communication needs and their dependencies so that an efficient mapping of components to resources can be achieved and the mapping’s communication cost is minimized. Thirdly, program behavior is highly dynamic and a different distribution configuration may be appropriate in different phases of the execution of a program. As a consequence, program components should be able to migrate at runtime to enhance locality and communication benefit. Finally, components are changed continuously either to expand their functionality or to improve performance and so is the infrastructure configuration in a distributed programming environment. To handle such issues, a system must be adaptive and dynamic in nature. A system that can automatically partition and distribute a program in order to maximize resource utilization and which handles all associated complexities is certainly a desirable solution.

We envision such a system as an Autonomic Computing [1] challenge where Autonomic Elements (AEs) are incorporated to handle the complexities associated with distribution, coordination and efficient execution of program components.

Fully automating the maintenance and optimization of a large computer system is a staggering challenge and the autonomic computing research community is working towards that. There are a few prototypes and research-based systems available [2, 3, 4], however, none of them share the same goal as this paper. Unity [2] provides a platform designed to help autonomic elements interact with each others and their environment. Autonomia [3] is a software development environment that provides application developers with tools for specifying and implementing autonomic requirements in a distributed application. AutoMate [4] is an execution environment for Grid based autonomic applications. AutoMate develops an autonomic composition engine to calculate a composition plan of components based on dynamically defined objectives and constraints that describe how a
given high-level task can be achieved by using available basic Grid services. The goal of our research [5, 6] is to build an autonomic distributed system so that the programmers do not have to deal with the distribution, management or optimization issues. This paper proposes an autonomic architecture to perform the distribution of user application across the available resources and provide autonomic infrastructural support to the application programmer. The goals of this system are as follows:

- **Autonomic environment:** The proposed system should be autonomic. Users should program with minimal constraints and the underlying system should transform the user code in a way that could be self managed.

- **Automatic distribution:** The proposed system should automatically determine program partitions along with their dependency, distribute them while allocating appropriate resources and coordinate them transparently at runtime. User intervention may be inevitable in certain cases, but it should be minimal.

- **Resource Utilization:** The system should work towards utilizing available resources most efficiently. Resources are highly dynamic both in activity and availability. Therefore, the system should be able to monitor itself and take the allocation and migration decisions based on not only the application characteristics and requirements but also various runtime factors such as user demands, machine failure, connectivity change, workload etc.

The proposed system is work-in-progress, currently being studied as a number of PhD projects. This paper gives a general overview of the system architecture and more specifically concentrates on identifying program partitions and dependability among them. Section 2 briefly describes the system architecture, Section 3 elaborates the process of obtaining the object dependency graph and Section 4 briefly describes its usefulness.

### 2. System architecture

The autonomic distributed system proposed in this paper targets existing (sequential or concurrent) Java programs, consisting of independent or communicating objects, and automatically generates a self managed distributed version of that program. The availability of the source code can not always be assumed, so the proposed system performs analysis and transformations at byte code level. Figure 1 shows the system’s flow of operation. At first, a code analyzer statically inspects the user supplied byte code to derive an object interaction graph. Based on this graph, the partitioner then generates several partitions (consists of a single object or grasps of objects) along with the distribution policies. A mapping framework then estimates the cost of various mappings and based on multiple optimization policies selects the optimal one.

Partitions are initially placed based on the resource (CPU, memory, communication bandwidth etc.) requirement of the objects and their interactions, various system information collected via monitoring services such as resource availability, workload, usage pattern etc. or any user supplied policy.

During code deployment, an autonomic transformer injects the autonomic and distribution primitives into user code so that the resultant self managed partitions can execute on different nodes in a distributed fashion. The transformed program is based on self-contained concurrent objects communicating through any standard communication protocol and incorporates salient features from existing middleware technologies. After initial placement, the underlying autonomic system gains the control of the objects and provides autonomicity to the user program afterwards. Each AE also monitors the actual program execution and the system itself. Based on these observations, the autonomic system then adjusts the static parameters such as resource consumption, amount of communications to their run time values and if needed dynamically repartitions the graph.

In the proposed system, every distributed site is managed by an AE which encapsulates the program partition allocated to that site as its managed element and interacts with the environment by using standard autonomic metaphors. Besides this basic functionality, some of the AEs in the system are given some higher level administrative authority. These managerial AEs may manage system registry and policy depository or may act as the user interface for program partitioning and transformation, monitoring or the source or destination of program input and output. AEs also set up a mutual service relationship to interact with each other so that information can be shared among them.
To use the autonomic resources, a potential user must first register his computer with the autonomic system through some user portals. Once registered, an AE is initiated on that machine and configures itself properly with all the necessary system data and policy information and consequently makes its services available to other AEs. A user may deregister their machine at any time and consequently the AE running on that machine will delegate its current managed element to other available AE without the loss of useful computation.

3. Dependency graph construction

We observe that analyzing and representing software in terms of its internal dependencies is important in order to provide the self managing capabilities because this is actually the system’s view of the run time structure of a program. In our approach, the underlying system controls the partitioning and analyzes the dependencies and therefore is informative about everything. Once partitioned, AEs are responsible for taking any optimization and migration decisions for the managed element and a program’s total view in terms of a graph is helpful to make such initiatives. For instance, at runtime, due to load or some other factors an AE may decides to migrate its managed element to a less loaded machine and there may be several machines available to handle that load. As each AE has the complete structure of the object dependency graph and the information about which partitions are managed by which AEs (obtained via monitoring services and interaction among AEs), the optimal replacement AE can be found which is in closest proximity with other AEs managing other partitions that have active communication with the partition to be migrated. Therefore a dependency graph is not only important for initial object placement, but also affects the runtime decisions, made by the system, towards system wide maximized resource utilization policy or any other user policy.

To construct such a graph, we must determine two pieces of information, namely: 1) the resources (i.e. computation time, memory, disk space, network etc.) consumed by each object and 2) the dependencies (directionality and weight) among the objects which is caused by the interactions among them. Therefore it is necessary to construct a weighted directed graph $G = (V, E)$, where each node $v \in V$ represents an object and the edge $(u, v) \in E$ resembles the communication between objects $u$ and $v$. The computational weight $w(v)$ represents the amount of computation takes place at object $v$ and the communication weight $w(u, v)$ captures the amount of communication between objects $u$ and $v$. Constructing such a dependency graph statically is a challenging task. There are several Java automatic partitioning tools [7, 8], are around, however, they only detects interaction at class level, therefore performs partitioning at class granularity and limits the opportunity to exploit object level concurrency. To our knowledge, Spiegel’s Pangaea [9] is the only system that performs analysis at the object level. Spiegel’s algorithm statically analyzes the Java source code and produces a directed graph where each node and edge represents run time instances and relations (create, use and reference) among them respectively. We use a modified and extended version of this algorithm in our system to deduce object dependency graph.

While Spiegel’s algorithm provides important insight about object dependency graph construction, it is not sufficient for our purpose. For instance, the original algorithm simply produces a directed graph. In contrast, we are interested on a weighted directed graph to effectively extract the computation and communication requirements of objects as well as the relations among them. Moreover, instead of having a general use relation, our target is to further categorize it as read-only and write based on whether the data members of an object are simply accessed or modified during use.

There are also significant differences in algorithmic aspects and implementation strategies. The original implementation assumes the presence of source code, while we are performing the analysis at the byte code level. One additional advantage of our approach is that it allows us to access system classes which are accessible at byte code level. Spiegel’s algorithm finds the set of types, objects, methods etc. by examining the code at the syntactic level. In contrast, our study uses standard compiler analysis and an efficient intermediate representation (IR). Another study [10] also implemented a modified version of Spiegel’s algorithm at the byte code level. However, like [9], they also deduced a dependency graph without computation and communication significances and used a different IR. Determining the type of a run time object is critical in Java due to polymorphism, inheritance and dynamic binding. Spiegel’s algorithm did not use any standard type inference mechanism to resolve dynamic dispatch and therefore the set of types each reference variable may point to at runtime includes all subtypes. Consequently the resultant object graph has unnecessary edges. In contrast, we use standard techniques like call graph and points-to analysis to resolve dynamic dispatch. Consequently, our analysis produces a more compact graph which is less expensive to perform further analysis on and in that way larger applications can be handled. In the following sections, we first briefly describe the original algorithm and then we discuss the issues related to our implementation. Readers are advised to read [9] for extensive description of the original algorithm.

3.1 Original algorithm

Spiegel defines set of Java types as classes (non primitive), interfaces and arrays used in the program. He
The original algorithm works as follows: at first, the set of types the program consists of is computed by syntactically identifying the type closure of the program. Then the class relation graph (CRG) is constructed which captures relationships at the type level. Whenever, a statement belongs to the context of type $A$ calls a method or accesses a field of type $B$, an usage edge $(A,B)_u$ is added to the CRG. Data flow information such as Export or Import relations take place when new types propagate from one type to another through field access or method calls. An export edge $(A,B,C)_e$ is added between type $A$ and $B$ when type $A$ owns a reference of type $C$ and passes it to type $B$. Similarly an import edge $(A,B,C)_i$ is added when type $A$ owns a reference of type $B$, from which it receives a reference of type $C$.

The algorithm then computes the object graph (OG) consisting of run time class instances along with create and reference relations among them. For each allocation (new) statement, create and reference edges are added between the class instances where the allocation takes place and the newly created instance. After the object population has been computed, the algorithm then iterates over all object triplets and use the data flow information from CRG to propagate references within OG until a fixed point is reached. Finally, the algorithm adds a usage edge between objects in OG if they already have a reference relation in OG and their corresponding types have usage relation in CRG.

### 3.2 Our implementation

Our static analysis of Java byte code is built on top of the Jimple [11] representation, which is part of the Soot [11] framework. The Soot framework is a set of Java APIs for manipulating and optimizing Java byte code. We analyze complete application, therefore by using Soot we first read all class files that are required by the application starting with the main method and recursively loading all class files used in each newly loaded class. As each class is read, it is converted into Jimple IR, suitable for our analysis and transformations. Jimple is a typed, stackless and compact three-address code representation of byte code. Jimple only involves 19 kinds of instruction and as a result is much easier to manipulate compared to stack oriented bytecode representation that involves 201 different instructions. For clarity, we explain different steps of our implementation by using an example program. Figure 2 shows the Java source code of the example program. In this program, Course c2 is instantiated only if the user supplied boolean variable math is true i.e. course Math100 is compulsory for each student. Class St_Student is not part of the original application and is produced by our system during a preprocessing stage as a result of separating static and dynamic types. In this case all the static members of class St_Student were actually part of class Student in the original program.

Soot also provides the necessary support for generating a call graph and performing points-to analysis based on Jimple code. In Java, all instance methods are invoked using virtual call. The actual method invoked depends on the run-time type of the object receiving the method call and often termed as receiver object. The call graph approximates the set of target methods that could possibly be invoked at each call site (method invocation) in the program. On the other hand, points-to analysis

---

**Figure 2: Example Java source code**

public class Example { public static void main(String[] argvs) { Student s1 = new Student("James", 76); Student s2 = new Student("Jill", 57); Course c1 = new Course("CS150"); c1.addStudent(s1); System.out.println(c1.getcName()+":" + c1.findhGpa()); if(math) { Course c2 = new Course("Math100"); for(int i = 0; i<St_Student.getAllStudent().size(); i++) { c2.addStudent((Student)St_Student.getAllStudent().elementAt(i)); } System.out.println(c2.getcName()+" ":" + c2.findhGpa()); } } }

public class Course { private String courseName; Vector registeredStudent; public Course(String n) { courseName = n; registeredStudent = new Vector(); } public String getcName() { return courseName; } void addStudent(Student s){ registeredStudent.add(s); } public Student findhGpa(){ Student hGpaHolder; ...; return hGpaHolder; } }

public class Student { private String studentName; private double gpa; public Student(String name, double gpa) { studentName = name; this.gpa = gpa; St_Student.addToAllStudent(this); } public String toString() { ... }

public class St_Student { static Vector allStudent = new Vector(); static void addToallStudent(Student s){allStudent.add(s); } static Vector getAllStudent() { return allStudent; } }

```java
public static void main(String[] argvs) { Student s1 = new Student("James", 76); Student s2 = new Student("Jill", 57); Course c1 = new Course("CS150"); c1.addStudent(s1); System.out.println(c1.getcName()+":" + c1.findhGpa()); if(math) { Course c2 = new Course("Math100"); for(int i = 0; i<St_Student.getAllStudent().size(); i++) { c2.addStudent((Student)St_Student.getAllStudent().elementAt(i)); } System.out.println(c2.getcName()+" ":" + c2.findhGpa()); } } }
```

---

0-7695-2653-5/06/$20.00 (c) 2006 IEEE
makes the call graph more compact and precise by limiting the number of target methods invoked in a call site. We use Soot’s SPARK [12] points-to analysis engine that computes the set of run-time types pointed-to by each program variable. Using the set of receiver objects at each call site, for each type, the methods that will be invoked actually are identified.

We observe that, once we obtain the call graph, it captures all program type interactions if we consider that the only way object a of type A can access an object b of type B is by invoking a method defined in type B. Identifying interactions that occurs whenever an object a directly accesses a data field of b requires additional passes through the Jimple code and manipulation of some other data structures beside the call graph which is time consuming and introduces additional complexity. We therefore, during preprocessing, generate additional accessor and mutator methods at byte code level that allow the load and store accesses to data fields via getXXX() and setXXX() methods and replace all direct field accesses in the original program by respective method call statements. Thus we perform our analyses at the level of method granularity which is another major distinction from Spiegel’s approach. One important additional advantage of this transformation is that, not only methods, but also fields can be then accessed remotely by a RPC style mechanism as implemented in JavaParty [13]. However, this approach can not be applied to system classes as they are not modifiable.

3.2.1. Method database. An important structure we utilize in our implementation is the Method Database (MD) that records information about each method and helps us determining the computational and communicational weights. The MD is constructed by inspecting the Jimple representation of each method used in the program. Each such method is analyzed to obtain information such as 1) whether it involves reading/writing of the fields of the class containing the method 2) amount of resource (CPU, memory) consumed by the method and 3) amount of communication needed to invoke the method. To avoid ambiguity due to polymorphism and method overloading, each method in the MD is represented by a unique method signature as follows:

\(<\text{className}: \text{returnType methodName}(\text{paramType}_1, \text{paramType}_2, \ldots, \text{paramType}_n)\>

Each method is categorized as read/write by recursively examining the load/store field accesses in method’s context. Approximating the computation time for each method is difficult mainly because of the presence of conditional statements and dependability on both input parameters and hardware platforms. In this study, the computational weight of each method is estimated in terms of number of Jimple instructions needed to be executed (ins_cost). Figure 3 shows a Java method and its Jimple representation, where categorization of each Jimple statement is also included. Each of 19 different kinds of Jimple instruction is then weighted according to the associated cost and the final estimate for a method is generated based on these weights. In the case of control structures, the system assumes that each branch of an if-else is taken 50% of the time and loops are executed for a configurable number of times [14]. The architecture-specific execution costs associated with ins_cost can then be computed by arch_cost, provided in units of msec per Jimple instruction. To obtain the arch_cost, the Jimple code is benchmarked on each type of resource. The computation time obtained in this way is roughly approximated and parameters can be tuned over time to provide better approximation. The memory requirement of each method is estimated by the memory consumed by data members of the respective class, method parameters and local variables.

3.2.2. Finding set of types. To find the set of types we first build the call graph of the application using Soot framework. The set of types of an application is exactly the set of types of method parameters and local variables.

3.2.3. Construction of class relation graph (CRG). The CRG is a directed graph \(G_c = (V_c, E_c)\) where \(V_c\) is the set of types that we computed in the previous step and \(E_c\) is the set of usage, export or import edges among types as explained in Section 3.1. The CRG construction algorithm explores every node (method) in the call graph and works as follows:

For each method invocation:

1. Let A be the class containing the method call statement, B be the receiver class of the method, \(P\) be the list of types of method parameters and \(R\) be the return type.

\[
\text{Figure 3: Java method and Jimple code}
\]
2. If \( A \neq B \), then add the following edges between type \( A \) and \( B \):
   a. An usage edge \( (A,B)_u \)
   b. An export edge \( (A,B,P[])_e \)
   c. An import edge \( (A,B,R)_i \)

Figure 4 shows the CRG deduced from the example program. For better visualization, we summarize all usage, export and import edges between type \( A \) and \( B \) into a single edge as \( (A,B,E[[]]) \), where list \( E[] \) contains a set of types exported from type \( A \) to \( B \) and list \( I[] \) contains a set of types imported from type \( B \) to \( A \). It should be noted that, the Java runtime system classes that are also invoked by the references to classes System.out and String are omitted in the final CRG.

Each edge \( (A,B) \) in CRG also contain a list of methods through which type \( A \) and \( B \) interacts. With this list and information contained about each method in MD, the edge \( (A,B) \) in the CRG is then further categorized as read-only/write. For a method \( M \) appears in directed edge \( (A,B) \) in CRG, the estimated computation weight and memory usage of \( M \) from MD are added as \( w(B) \) (computation weight of node \( B \)). The communication weight \( w(A,B) \) depends both on how many times \( A \)'s enclosing statements invokes a method of \( B \) and number of bytes required to represent the method parameter and return type. Final values of \( w(B) \) and \( w(A,B) \) are calculated by adding the weights caused by all methods appearing in edge \( (A,B) \). For example edge \( (Example, Course) \) in Figure 4 involves four methods such as
   - Course: void <init>(java.lang.String)
   - Course: void addStudent(Student)
   - Course: java.lang.String getCourseName()
   - Course: Student findGpa()

Some of these methods modify data members of type Course, so the edge \( (Example, Course) \) is categorized as write. The weight \( w(v) \), where \( v \) is an object of type Course, is then the summation of computation performed by the above methods.

3.2.4. Construction of initial object graph (OG). OG is a weighted directed graph \( G = (V,E) \) where \( V \) is the set of run-time instances and can be of type static, concrete or indefinite as explained in section 3.1 and \( E \) is the set of edges among the nodes in \( V \) and can be of type read-only, write, create or reference. To obtain the set of objects, we first identify allocations in the program. In Jimple, this includes statements that allocate objects and arrays and that load string constants. Some example of the allocation statements in Jimple are,

\[
p = \text{new Student}, \quad q = \text{newarray(int)[12]}, \quad r = \text{“Hello”}
\]

Identifying the allocation statements is performed at the same time when method database is constructed as both requires examining each Jimple method. Each method in MD keeps track of all allocations inside it including the class of the allocated object and the type of allocation (concrete/indefinite). Static edges are also identified in CRG, where the receiver object of the method call is of type static. For instance, edges \((Example,St_Student)\) and \((Student, St_Student)\) in Figure 4 are static edges. In Jimple, static invocation is always preceded by staticinvoke as shown in Figure 3.

The OG construction algorithm utilizes these entries and works as follows:
1. For each static type, add one static object in OG.
2. Repeat the following for all static objects in OG:
   a. For a static object of type \( A \) of \( A \), extract all the allocations from all the methods in the method database where the className part of method signatures matches \( A \).
      i. Add one concrete object of respective type for each concrete allocation in \( A \).
      ii. Add one indefinite object of respective type for all allocations of that particular type inside a control dependent statement in \( A \).
   b. For each concrete object of type \( A \) added in the graph, add concrete/indefinite objects in the same way as in step 2(a).
   c. For each indefinite object of type \( A \) added in the graph, all allocation statements in \( A \) are treated as uncertain and therefore an indefinite object is added to the graph for all such allocations in \( A \).
3. For each concrete or indefinite object added to the graph, add a create and a reference edge from the parent to the newly created object.
4. For each static edge \((A,B)\), add a reference edge from all objects of type \( A \) to static object \( B \).

Figure 5 shows the object graph generated from our example program.

3.2.5. Propagating usage and reference edges. The algorithm now iterates over all triples of objects \((a,b,c)\) in OG for which reference edges \((a,b)\) and \((a,c)\), or \((b,c)\), exist in OG and matches the types of the objects against the data flow edges \((A,B,C)\) or \((A,B,C)\) in CRG, then a new reference edge \((b,c)\), or \((a,c)\), is added to the object graph as explained in Figure 6. Finally a use edge \((a,b)\) is added to the OG if \((A,B)\) exists in CRG and \((a,b)\) exists in
transforms an existing Java application to a self managed distributed one so that the programmers do not have to deal with the programming or management issues and at the same time system resources are utilized properly. To implement such an infrastructure, we realize the importance of having program’s run time structure as an object dependency graph so that decisions towards self management and self optimization can consult that graph. This paper also presents the design and implementation of an algorithm that statically analyzes the code to identify runtime instances and their interactions. We used the method database to store all information related to a method in a way so that different stages of our algorithm can utilize that structure and the total cost of static analysis is minimized.

6. References


4. Applicability of object graph

Based on the object graph, it is possible to categorize objects as 1) immutable (has no incoming write edge i.e. never changes after been created), 2) Single Read/Write (Object A is only accessed by object B and no one else), 3) Single Write Multiple Read (Object A is written by object B but read by many others) and 4) Multiple Read Multiple Write (Object A is accessed by many others). Effective distribution decisions can then be generated for each object category such as replicate objects of category (1) across the system, objects of category (2) do not need to be remotely invokable, they only need to be co-located with their accessor object, objects of category (3) need to be remotely invokable and initially co-located with their writer objects etc. The objective is to place the most communicating objects in the same machine, thereby minimizing network traffic. Such classification and distribution strategies for each object along with their computation and communication significances are extremely helpful during initial object placement and preserving maximum resource utilization.

5. Conclusion

This paper presents an autonomic architecture that transforms an existing Java application to a self managed...