BYG: An Approach to Just-in-Time Gridification of Conventional Java Applications

Cristian MATEOS [a,b,1], Alejandro ZUNINO [a,b], Marcelo CAMPO [a,b] and Ramiro TRACHSEL

[a] *ISISTAN Research Institute - UNICEN. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel./Fax: +54 (2293) 439682/439681.*

[b] *CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)*

Abstract. Grid technologies allow developers to run applications with enormous demands for resources such as processing power, data and network bandwidth. However, exploiting Grid resources demands developers to Grid-aware their applications by explicitly accessing the services of specific Grid middlewares, which involves more development effort and requires expertise on Grid programming. A number of recent research efforts known as *gridification methods* aim to avoid these problems by semi-automatically deriving the Grid-enabled version of an application based on either its source code or binary code. Most of the approaches belonging to the second group produce coarse-grained Grid applications that prevent programmers from employing tuning mechanisms such as parallelism and distribution, and are rather inflexible, since they were not designed to reuse existing Grid middleware services. We present BYG (BYtecode Gridifier), a new gridification method that allow developers to easily gridify binary Java applications and solves these issues. The paper describes the prototype implementation of BYG and some experiments showing the feasibility of the approach. Experiments show that BYG can be used to transparently gridify and efficiently execute a broad range of resource-intensive applications.

Keywords. Grid Computing, gridification, Java bytecode

Introduction

Grid Computing is a paradigm for distributed computing based on arranging geographically dispersed computational resources to execute resource-intensive applications [11]. Typically, Grid applications are intended to solve scientific or engineering problems that require by nature huge amounts of computational resources such as CPU cycles, memory, network bandwidth, data and services. Examples of such applications include protein folding, financial modeling, aerodynamic design and weather simulation. Just like an electrical power grid, computational Grids are pervasive computing environments whose goal is to provide coordinated resource sharing across different administrative domains to meet complex user demands [13].

[1] Corresponding author. E-mail: cmateos2006@gmail.com.

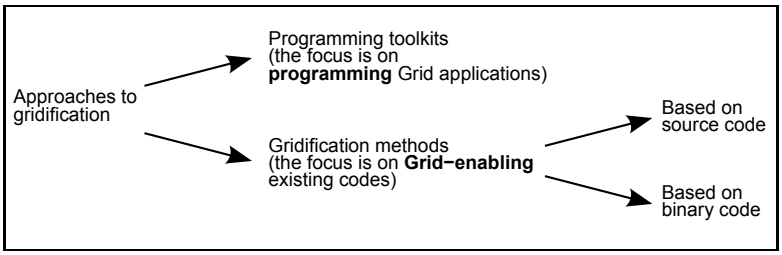


Figure 1. Gridifying applications: a simple taxonomy

Looking back into the history, the first attempts to establish Grids were focused on creating infrastructures to support CPU-intensive, large-scale applications by linking supercomputers [14]. With the inception of Internet standards in the 1990s, Internet-wide Grids became a reality. Consequently, applications that leveraged idle processors of thousands of Internet-connected PCs begun came into existence. A representative example is SETI@home [3], a desktop application by which users have their PCs process radio signals from the outer space, helping in a global search for extraterrestrial life. Similar initiatives are Folding@Home and Genome@Home [17].

Few years later, after the introduction of this form of public computing, the first Grid middlewares appeared. Examples of popular middlewares for Grid development are Globus [12], Legion [24], Condor [29] and UNICORE [10]. Roughly, the goal of these technologies is to virtualize the various resources of a Grid by means of *services* (e.g. job scheduling, load balancing, brokering, monitoring, data movement, security, etc.), and also to supply developers with rich APIs for using these services. Essentially, Grid middlewares sit in the middle between applications and Grid infrastructures by adding a software layer comprising specialized services that provide vast execution capabilities.

Besides producing more and better Grid services to applications, recent research in Grid middlewares has put emphasis on the *consumability* of the delivered services [21]. Concretely, researchers have been trying to exploit middlewares by providing programing tools, frameworks and libraries that simplify the consumption of their services from within user applications. The ultimate goal of these technologies is to come out with facilities to allow Grid application developers to benefit from middleware services with little if any implementation and configuration effort.

Works in this line of research can be grouped into programming toolkits and gridification methods [21] (see Figure 1). The idea behind programming toolkits is to provide programming APIs and constructs that abstract away the necessary details to interact with Grid middleware services [30,2,25,6]. By using these tools, Grid programming can be done at a higher level of abstraction, so that less effort and time is required when employing Grid toolkits versus directly using middleware APIs . However, as Grid toolkits are in essence programming facilities, they usually assume that developers have a solid knowledge both on Grid programming and the features offered by the particular toolkit being used.

Alternatively, the goal of gridification methods is to allow developers to incorporate or to “inject” Grid services into existing applications with little effort. In other words, these methods focus on semi-automatically (ideally, automatically) transforming existing codes to run on a Grid rather than programming Grid applications from scratch. Therefore, they are mainly intended to support users having little or even no background

on Grid technologies. Furthermore, gridification methods can accept as input either the source code of ordinary applications [4,31,22] or their compiled versions [9,15,23]. Intuitively, the first approach allows developers to have more control over the internal structure of their applications, thus very efficient Grid applications can be built. Nevertheless, the second approach allows to gridify applications even when their source code is not available, removing the need for the tedious code-compile-deploy sequence when building Grid applications. In this way, gridifying an application just involves submitting it "as is" for execution on a Grid platform.

One major problem of the current techniques for gridifying binary code is that they usually prevent the usage of tuning mechanisms suitable for exploiting Grids. After gridification, applications are essentially coarse grained, monolithic Grid-enabled codes whose structure cannot be altered to make better use of Grid resources. Specifically, most of these approaches do not prescribe mechanisms for distributing or parallelizing individual parts of an application. To sum up, although they greatly simplify gridification, employing these approaches may potentially lead to a poor usage of Grid resources. This represents a trade-off between ease of gridification versus flexibility to configure the runtime aspects of gridified applications [21]. As a consequence, there is a need for alternative gridification methods that provide a convenient balance between the effort developers have to invest when deploying and running legacy codes on a Grid and the *gridification granularity*, the size of gridified applications from a runtime perspective [21] and therefore the levels at which their components can be tuned.

We propose a new gridification method for Java applications called BYG (BYtecode Gridifier), whose utmost goal is to offer an alternative mechanism for effortlessly Grid-enabling binary codes at various granularity levels. Users indicate, via a configuration file, the portions of their applications that are subject to execution on Grid middlewares. Moreover, BYG does not reinvent the wheel by providing yet another Grid job submission system, but offers a glue between conventional binary Java applications and the execution services of existing platforms (e.g. Condor, Globus, etc.). Basically, BYG works by dynamically instrumenting the ordinary bytecode to be compliant with the application anatomy prescribed by the target middleware. Experiments show that BYG enables for easy and fine-grained gridification for a wide range of CPU-intensive applications, and effectively leverages existing Grid execution services while not incurring in much performance overheads compared to directly employing these services to gridify applications.

The rest of the paper is organized as follows. The next section discusses related works, and explains how the BYG approach improves over them. Section 2 overviews the BYG approach. Later, Section 3 describes its design and implementation. After that, Section 4 reports an experimental evaluation of BYG. Finally, Section 5 concludes the paper.

1. Related work

To date, several approaches for gridifying software have been proposed in the literature. It is worth noting that the approaches that accept as input source codes are out of the scope of this paper (see [21] for a detailed discussion on them), but we will focus on approaches aimed at gridifying binary codes. Table 1 summarizes these efforts.

Table 1. Existing approaches for gridifying binary codes

Tool	Binary code flavor	Gridification granularity	Exploitation of other Grid platforms
GEMLCA	Machine-dependent	Coarse-grained	Partially (only Globus)
GridSAM	Machine-dependent	Coarse-grained	Yes
Laskowsky et al.	Machine-independent (bytecode)	Fine-grained (thread level)	No (runs on plain JVM clusters)
LGF	Machine-dependent	Fine-grained, coarse-grained	No (uses custom RMS)
ProActive	Machine-independent (bytecode)	Coarse-grained	Yes
Satin	Machine-independent (bytecode)	Fine-grained	Partially (uses custom RMS; some integration with Globus)
Volta	Machine-independent (CIL)	Fine-grained	No (uses custom RMS)
XCAT	Machine-dependent	Coarse-grained	Yes

GEMLCA [9] lets users to deploy a legacy program as an OGSA-compliant service. The access point for a client to GEMLCA is a front-end offering operations for gridifying legacy codes, and also for invoking and checking the status of deployed Grid services. To execute the gridified codes, GEMLCA uses the Globus' GRAM job submission service. The user is responsible for specifying metadata information (parameters, executable path, etc.) and resource requirements (processors, memory, etc.) for its application in an XML configuration file. GEMLCA relies on a very nongranular execution scheme for these services (i.e. running the same binary code on one or more processors) but no internal changes are made in the gridified applications. Then, parallelism and distribution of individual portions of the application cannot be controlled in a fine-grained manner.

GridSAM [23] allows users to publish legacy applications as Web Services. GridSAM then treats these services as a number of separate components, which can be composed via a workflow description document that is processed and executed according to resource requirements on top of other Grid platforms. Unlike the aforementioned tools, GridSAM does not in itself provide the functionality of a Resource Management System (RMS), but instead acts as a common interface to existing Grid job execution services. The same principle is followed by [1], but it is even more focused on integrating different Grid job submission and storage services rather than facilitating the construction of Grid applications.

XCAT [15] supports distributed execution of component-based applications on top of existing Grid platforms (preferably Globus), linking components to concrete platform-level execution services. In addition, application components can also represent legacy binary programs. XCAT allows developers to build complex applications by programmatically assembling service components and legacy components. Though this task can be carried out with little coding effort, it still requires programming and therefore requires knowledge on the XCAT API. As both GridSAM and XCAT treats input legacy codes as black boxes, these tools share some of the limitations of GEMLCA with respect to granularity of gridified applications. Finally, LGF [5] is an execution and monitoring framework that allows users to deploy legacy applications as Web Services. Central to

its design is a two-layered architecture in which the adaptation service layer is heavily decoupled from the legacy back end layer. With LGF, it is possible to monitor the performance of gridified applications at the service, legacy application and code region level. However, the framework is not designed to take advantage of existing Grid execution services such as those provided by Condor and Globus.

With respect to tools that gridify machine-independent binary codes, [18] proposes a method for transparent execution of multi-threaded Java applications on clusters of JVMs deployed on desktop Grids. First, the tool derives graphs from the compiled bytecode of an application by using representative sets of input data. Roughly, the graphs account for data and control dependencies within the application. Then, a scheduling heuristic is applied to place certain mutually exclusive execution paths extracted from the graphs among the nodes of a JVM cluster. In opposition, BYG aims to gridify single-threaded Java applications by leveraging existing execution services suitable for exploiting Internet-wide Grids. ProActive [4] is another Java platform for parallel distributed computing that provides *technical services*, a flexible support that allow developers to address non-functional concerns (e.g. load balancing and fault tolerance) by plugging external configuration to applications at deployment time. ProActive allows users to deploy ordinary classes as mobile entities on a Grid without code modification by exposing their operations through a number of protocols (RMI, Web Services, etc.). ProActive features integration with a wide variety of Grid schedulers. Unfortunately, creating computations based on a subset of the methods of an ordinary class unavoidably requires to manually use the ProActive API within the source code of the input application.

In addition, there are other tools that follow a *hybrid* approach to gridification of binary codes, in which developers are actively involved in the process of altering an application to gridify it. Satin [31] is a Java framework for gridifying and parallelizing divide and conquer applications. The user is responsible for indicating the points in the application code in which a fork (i.e. a recursive call that is to be handled in parallel) or a join (i.e. to wait for child computations) should take place. Then, Satin instruments the compiled code so as to transparently handle the execution of parallel tasks in a Grid. Similarly, Volta [19] recompiles executables .NET applications on the basis of declarative developer annotations, inserting remoting and synchronization primitives to transparently transform applications into a distributed form. As recompiling operates at the CIL (.NET Common Intermediate Language) level, Volta is compatible with a broad variety of .NET programming languages. However, the weak point of these tools is that they require modifications to the source code of applications prior to actually Grid-enabling their compiled counterpart.

While the above approaches are targeted at supporting users with little knowledge and expertise on Grid technologies, some of them (Satin, Volta) are some way off from being true binary code gridifiers, as they require code modifications on the input applications. Furthermore, some of the approaches (GEMLCA, GridSAM, ProActive, XCAT) offer a poor balance to the "ease of gridification versus tunability" trade-off, since they completely avoid the requirement of code modification, but gridification results in coarse-grained Grid applications that cannot be modified for reconfiguration or paralellization purposes. Finally, only a small number of the analyzed approaches are designed to exploit the execution services of other Grid platforms. However, a recent trend in Grid Computing, as evidenced by broadly adopted Grid standards such as OGSA and WSRF [8], is to promote interoperability and therefore integration among Grid tools and

middlewares. In this sense, Grid middleware integration is rapidly becoming the rule and not the exception.

2. BYG

To address the above problems, we propose BYG (Bytecode Gridifier), a new gridification method that aims at dealing with the above trade-off by letting developers to introduce tuning into their applications with little effort, and minimizing the configuration and deployment effort that is necessary to put a Grid application to work. Furthermore, the approach offers mechanisms to easily produce efficient Grid-aware applications, but it does not seek to provide yet another runtime system or middleware for supporting application execution. Instead, our research aims at leveraging the services of existing Grid platforms through the use of *connectors*. Basically, a connector materializes the protocol to access the various execution services provided by a specific Grid platform. Connectors are non-invasively injected into the application binary code in order to dynamically delegate the execution of certain parts of the application to a Grid platform. In addition, connectors are responsible for transparently adapting the binary codes to take advantage of the API library provided by the target platform. Roughly, the mapping of which parts of an application whose execution is delegated to particular Grid services is specified by means of user-supplied configuration external to the application.

BYG specifically targets component-based applications implemented in Java. On one hand, we chose Java as it is broadly adopted by developers. Besides, the JDK provides many features and libraries that facilitate the construction of distributed execution environments, such as sockets, object serialization, extensible class loading, reflection, amongst others. On the other hand, component-based programming is commonplace in Java development, which is evidenced by the high popularity of several component programming models such as JavaBeans[2], EJB[3] and Dependency Injection [16]. For these two reasons, our tool could benefit a large amount of today's applications.

Component-based development emphasizes on building applications in which functionality is split into a number of logical components with well-defined interfaces. Every component is designed to hide their associated implementation, to not share state, and to communicate with other components via message exchange [27]. In the end, application components only know each other's interfaces and are self-contained, which yields as a result reusable and decoupled building blocks where interfaces are abstracted away from implementation (i.e. each component is materialized through one or more classes plus an interface) and any kind of interaction that involves tightly-coupled communication or state sharing is disallowed, such as invoking component operations by passing arguments by reference.

Figure 2 depicts an overview of BYG. Conceptually, BYG takes as input the compiled version of an ordinary component-based Java application, and dynamically transforms it so as to run some component operations on different Grid middlewares. In this sense, BYG can be seen both as a competitor of existing approaches and a complement to them. The developer is responsible for indicating which operations should be handled by external services, and for each one of them which specific middleware should be used.

[2] JavaBeans `http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html`
[3] Enterprise JavaBeans `http://java.sun.com/products/ejb`

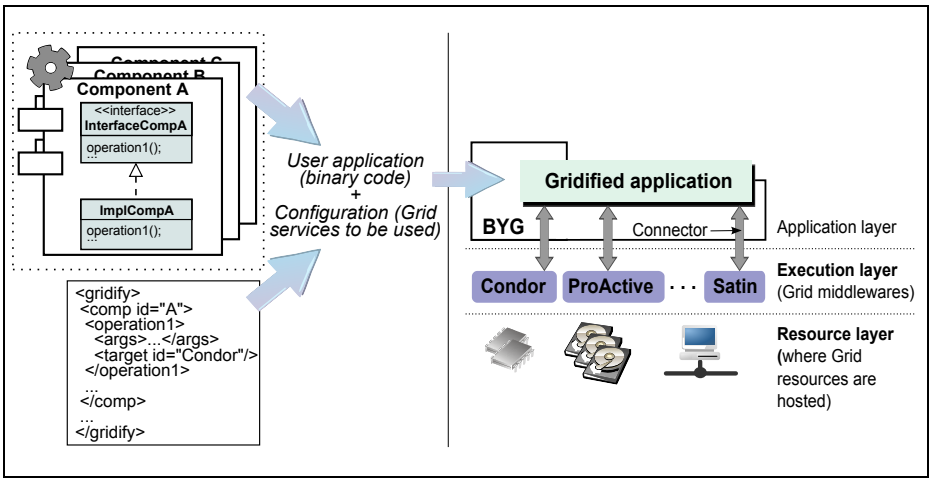


Figure 2. Overview of BYG

The runtime support of BYG is in charge of processing the developer configuration, intercepting all invocations to this kind of operations (in this case operation1), and delegating their execution to the associated target Grid middleware (in this case Condor). Some evident benefits of this approach are:

Gridification effort Gridification requires less effort. The programmer just specifies which parts of its application should be executed on a Grid environment, but without explicitly coding the application to do so. Besides, gridification is possible even when the source code of an application is not available. In addition, the Grid services associated to an application can be straightforwardly detached by simply modifying its associated configuration.

Flexibility Depending on the nature of each component operation, a different Grid execution service could be used. For example, all invocations on an embarrassingly parallel operation may be sent to a Grid platform able to execute them concurrently, thus improving performance and scalability. Similarly, all mission critical computations may be submitted to another Grid platform providing fault tolerance capabilities.

Tunability Unlike related tools, in which applications are mostly executed on a black box fashion, BYG allows developers to fine-tune the execution of their applications by submitting calls to component operations in parallel to different Grid middlewares. More details on this support can be found in Section 3.1.

Service availability The developer is asked to specify which Grid execution service should be employed to execute a component operation. However, at the time of executing the application, the service might not be available (e.g. the Condor cluster that was supposed to run the application is temporarily down). Then, BYG could choose an alternative service to still allow the user to run the application.

BYG does not aim to automatically gridify any kind of application and exploit any kind of Grid middleware. First, our approach is specifically designed to gridify component-based applications written in Java. This ensures that application components are heavily

decoupled and do not share state, thus component operations can be run in a different memory address space without worrying about which component issued the invocation or how the operation arguments/results are interchanged. Second, depending on the particular connectors being used, operations must adhere to certain extra coding conventions. Nevertheless, following good object-oriented practices such as employing proper method modularization, placing the result of calls on local variables, and avoiding parameter passing by reference is usually enough to prepare an application to use various connectors.

We have developed a proof-of-concept implementation of our approach, which is described in Section 3. It is basically implemented by means of the java.lang.instrument package of the JDK, which provides facilities for transforming class files at class load time. Based on this support, the prototype works by instrumenting bytecodes to delegate the execution of certain component operations to external Grid execution services, this is, offered by existing Grid middlewares. BYG-enabling an already compiled application only requires the user to (a) configure an XML file[4] that instructs BYG how to map component operations to Grid execution services, and (b) add a JVM argument to the bootstrap script that initiates the user application. The implementation of this prototype as well as the applications used in the experiments described later in this paper are available upon request.

At present, our tool provides a connector for accessing the services of the Satin platform [31], which provides support for executing and parallelizing divide and conquer Java computations on LANs and WANs. Basically, this connector automatically generates a Satin application based on the bytecode of an ordinary application component. Furthermore, the development of connectors for Condor and ProActive [4] is underway. This will enable developers to take advantage of useful Grid functionalities not present in Satin such as monitoring and administrating running computations. This integration is in principle viable from a technical point of view since ProActive is also implemented in Java, and there are successful experiences on integrating Java and Condor clusters [28]. It is worth noting that both BYG and its current implementation are strongly inspired by concepts and ideas derived from previous research in the context of the JGRIM project [22], a method for Grid-enabling Java source code.

3. Implementation

The first step to put any conventional application to execute on a Grid with BYG is to create the corresponding XML configuration file. This XML file specifies relevant parameters that BYG needs to Grid-enable the application, such as the components to be gridified, and the binding information that depends on the Grid middleware(s) to which BYG will delegate the execution of these components, namely the entry point of the middleware(s) and the job submission protocol used in each case.

A user application may have many parts suitable for execution on a Grid. In this sense, BYG allows applications to be gridified at a fine-grained level, this is, any component operation can be configured by the user to be submitted onto a Grid middleware. To this end, the user must provide:

[4] We are currently working on graphical tools to further simplify configuration

1. The list of operations or Java methods (owner class, method name and parameter types) to be gridified. Roughly, the owner class allows BYG to unambiguously identify methods with the same signature but implemented by different classes.
2. The connector to be used (and consequently the Grid execution service). As we suggested in the previous section, this decision will in general depend on the nature of the operation being gridified.
3. The job submission protocol over those provided by the connector(s) being employed. For example, Condor provides a remote job submission mechanism based on raw sockets, but it also offers a Web Service submission interface. Consequently, the user is responsible for selecting the specific job submission protocol when more than one choice is offered by the target middleware.

```
1  <connectors xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2              xsi:noNamespaceSchemaLocation="bygConfiguration.xsd">
3    <connector name="example">
4      <middleware name="satin">
5        <property name="protocol">raw_sockets</property>
6        <property name="host">192.168.19.79</property>
7        <property name="port">5432</property>
8      </middleware>
9      <classes>
10       <class name="Fibonacci">
11         <methods>
12           <method name="fib">
13             <parameter type="long"/>
14           </method>
15         </methods>
16       </class>
17       ...
18     </classes>
19   </connector>
20   ...
21 </connectors>
```

The above configuration tells BYG to execute via the services of the Satin platform (lines 4-8) a specific method from the Fibonacci class (lines 10-16). The name of the class must be fully-qualified. Of course, it is possible to define more than one connector within a configuration file, and associate several classes and methods to them. In its current shape, BYG requires developers to deal with conflicting connectors, this is, to avoid associating the same class method to more than one connector.

To inject connector code, BYG takes advantage of the java.lang.instrument package, a feature starting at Java 5 that defines an API for modifying bytecodes by transforming their associated class files at load time. The instrumentation package is intended to be extended through special libraries called Java agents. A Java agent is a pluggable user library that runs embedded in the JVM and customizes the class loading process. An agent works in front of the main application method, executes on the same JVM as the application, is loaded by the same system class loader, and is governed by the same security policy and context. Basically, the kernel of the mechanism for dynamically injecting connectors in BYG is implemented as a Java agent.

Figure 3 illustrates the differences –from a runtime perspective– between executing an ordinary Java application in the usual way (left) versus executing it by taking advan-

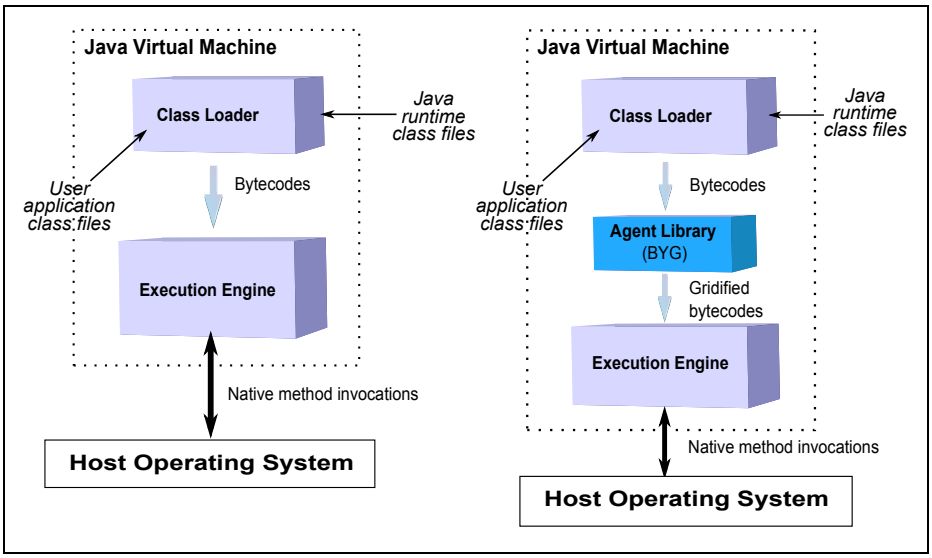


Figure 3. Rewriting bytecodes on the fly: The BYG agent

tage of the BYG agent (right). The agent library is simply a JAR (Java ARchive) file that contains a special bootstrap class with a *premain* method, which is from where the class transformations are triggered. This method is invoked by the JVM every time the application requests to load a class, and its signature is as follows:

```
public static void premain(String agentArguments,
                           java.lang.instrument.Instrumentation instr);
```

The first parameter is a string representing the command line arguments passed to agent, while the second parameter is a Java object maintained by the JVM that provides services to register/deregister class transformers, obtaining the classes already loaded into the JVM, etc. Basically, the above method is the hook by which BYG configures and injects connector code into classes according to the information supplied by the user in the configuration file. To BYG-enable an application (i.e. to use the BYG agent), the startup command that executes the main application class must look like:

```
java -javaagent:bygAgent.jar=<config-file> <Main-Class> ...
```

When the user application starts, the agent reads the configuration file and extracts both the methods to gridify and the connectors to use. Based on this information, the BYG agent dynamically instruments the bytecodes of the classes owning these methods as these classes load. Furthermore, instrumenting an individual class method implies:

1. To rewrite its body in order to include the necessary instructions to launch the execution of its bytecode on a specific Grid middleware. The injected "stub" will employ the middleware-related information extracted from the connector associated to the method (i.e. the protocol, host and port properties) to transparently send an adapted version of the original method's body for execution via an external Grid service interface every time this method is called by the application. The

stub is created and injected by using Javassist[5], a high-level library for modifying and creating classes and methods at runtime.

2. To adapt its original bytecode to take advantage of the middleware the method was connected to. In other words, this adaptation involves to prepare the method as well as the structure of its owner class to the code anatomy prescribed by the target Grid middleware. For example, some platforms require applications to extend from middleware-specific API classes, use certain API calls to carry out object distribution and parallelism, make all objects to be serializables, and so on.

The transformations described in the above steps strongly depend on the Grid middleware selected for execution. In the next subsection, we focus on explaining these mechanisms in the context of Satin, for which the current version of BYG provides a connector.

3.1. The Satin connector

Satin [31] is a Java framework that lets programmers to easily parallelize divide and conquer Java applications. Satin provides two core primitives to parallelize single-threaded conventional applications: *spawn*, to create subcomputations (i.e. divide), and *sync*, to block execution until the results of subcomputations are available. Methods considered for parallel execution are identified by means of *marker interfaces* that extend the *satin.Spawnable* interface. Furthermore, a class containing spawnable methods extends the *satin.SatinObject* class and implements the corresponding marker interface. In addition, the result of the invocation to a spawnable method must be stored on a local variable. For instance, the Satin version of the recursive solution to compute the n^{th} Fibonacci number would be:

```
public interface IFibMarker extends satin.Spawnable{
  public long fib(long n);
}
public class Fibonacci extends satin.SatinObject implements IFibMarker{
  public long fib(long n){
    if (n < 2)
      return n;
    // The next two calls are spawned according to IFibMarker
    long f1 = fib(n - 1);
    long f2 = fib(n - 2);
    // Execution suspends until f1 and f2 are both instantiated
    super.sync();
    return f1 + f2;
  }
  public static void main(String[] args){
    ...
    Fibonacci fibComp = new Fibonacci();
    long result = fibComp.fib(n);
    ...
  }
}
```

[5] Javassist `http://www.csg.is.titech.ac.jp/~chiba/javassist`

After specifying spawnable methods and inserting appropriate synchronization calls into the application source code, the developer must feed a compiled version of the application to the Satin compiler that translates, through Java bytecode instrumentation, each invocation to a spawnable method into a Satin runtime task. For example, from the code shown above, a task is generated for every single call to the fib method.

The purpose of the Satin connector is to automatically reproduce the above code structures from the compiled version of the components of an ordinary application, this is, an application which has not been coded to take advantage of the Satin API. The Satin connector generates the marker interface(s) based on the contents of the XML file associated to the application, and rewrites the bytecode of the component(s) in order to extend and implement the required API classes and interfaces. In addition, the connector automatically inserts proper calls to sync by deriving a high-level representation from the bytecode and analyzing the points where a barrier must be introduced. These tasks are explained in detail in the next subsection. To execute the Satin-enabled version of components, this connector relies on an extended Satin runtime, which is explained in subsection 3.1.3.

3.1.1. Client side processing

Besides injecting proper bytecode to transparently execute ordinary methods on Satin, the Satin connector is responsible for dynamically adapting the bytecodes of both these methods and the classes owning them to be compliant to the application anatomy described in the previous section. Thus, based on a compiled conventional class, the Satin connector carries out two main tasks:

- Marker interface generation: As explained, Satin requires the application to implement a marker interface, which explicitly list the methods that are considered by Satin for parallel execution. The Satin connector automatically builds this interface from the methods listed in the XML configuration for the class being connected to the Satin services. For creating the corresponding class file for the interface, the ASM[6] library is employed. Generated interfaces are suffixed with "_Spawnable".
- Class generation: Satin requires applications to implement a marker interface as well as to extend from SatinObject. In this sense, a clone (from now on *peer*) of the non-gridified class under consideration is created and instrumented to fulfill these requirements. Again, cloning is performed by using Javassist. Generated peers are suffixed with "_Peer".

Figure 4 depicts all the steps performed by the Satin connector to build the Satin-enabled version of an ordinary class. The connector receives as input from the BYG agent the ordinary class being gridified (actually, its bytecode) and the target method(s), and creates the corresponding marker interface and a Satin peer for it. In a subsequent step, based on an heuristic algorithm we have specially designed, the connector automatically inserts calls to the sync primitive at appropriate places of the parallel methods of the generated peer. This algorithm is explained in the next subsection. Afterwards, the peer is processed with the bytecode instrumentation tools of the Satin platform. At runtime, the final peer will be instantiated at the client side by the ordinary application and sub-

[6] ASM `http://asm.objectweb.org`

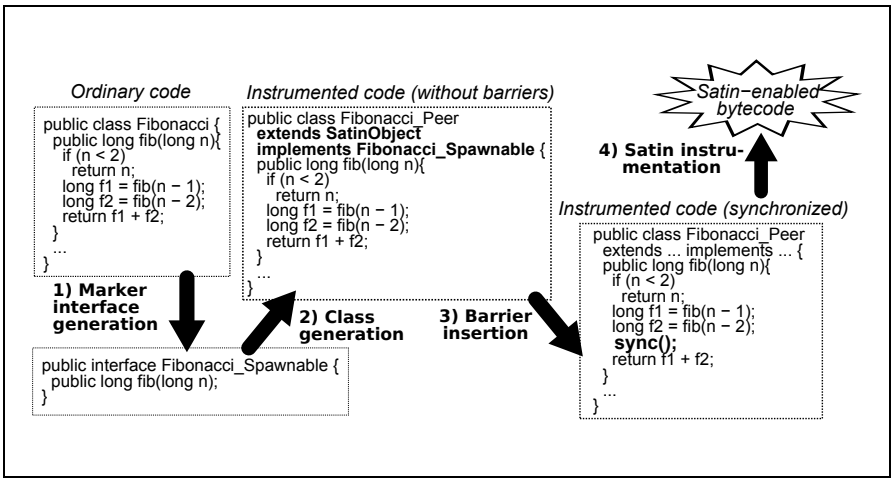


Figure 4. The Satin connector: Satin-enabling ordinary bytecode

mitted for execution through our socket-based interface to Satin, which is discussed in Section 3.1.3.

3.1.2. Barrier insertion

As explained, when programming a conventional Satin application, the results of recursive calls to parallel methods must be placed on local variables. Furthermore, before reading such variables, the developer is responsible for inserting a call to the sync primitive, which ensures that recursive results are always available before they are accessed. The step 3 in Figure 4 automatically reproduces this task, this is, the connector analyzes the bytecode generated at the previous step and then rewrites this bytecode in order to insert barriers at the right places.

Intuitively, a naive solution to this problem is to blindly place such a barrier right before any access to a local variable. However, we found that this solution generated more calls to sync than needed and negatively affected the performance of the resulting peer. As a consequence, we designed an heuristic that aims at inserting a minimal number of synchronization barriers and at the same time preserving the semantics of the original code. The algorithm works by deriving a high-level, ad-hoc representation of the bytecode instructions to carry out the analysis as much close to the source code of the application as possible. This mapping relies on the fact that there is a direct correspondence between Java source and bytecode [7,26].

Java compiles the source code of methods as a number of labels, each containing a number of bytecode instructions. Individual labels form disjoint instruction blocks where local variables are declared, calls to other methods are performed, goto-like directives to jump to other labels are included, etc. Precisely, the relationships between the different labels define the behavior in terms of control flow the program will have at runtime. Figure 5 exemplifies these notions. The instructions of the source code shown on the left are compiled into seven labels (center), and gives origin to a *block tree* comprising three nodes: one for the whole method, one representing the loop construct, and finally one for the conditional branch inside the loop. It is worth noting that any node within a block tree may have more than one child. The root of a block tree always correspond to the

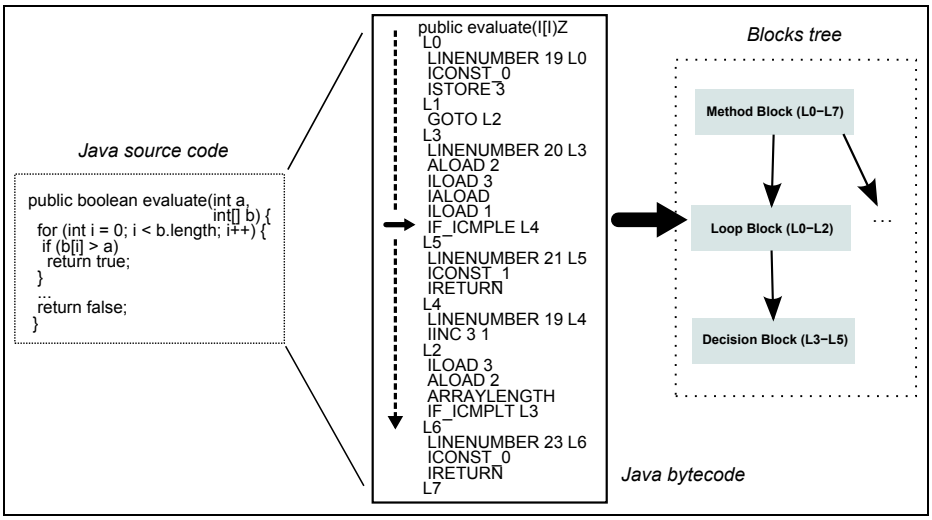


Figure 5. Deriving a block tree from bytecode

body of a method, whereas the rest of the nodes exclusively depend on the structure of the sentences within this method.

To derive the block tree of a method, the Satin connector analyzes its bytecode instructions sequentially in order to find those that provide information about the higher-level structure of the program, this is, loops, conditional branches, try/catch constructs, and so on. Specifically, in Java, these instructions are the ones that perform jumps within a method (IFEQ, IFNE, IFLT, IFGE, IFGT, IFLE, IF_ICMPEQ, IF_ICMPNE, IF_ICMPLT, IF_ICMPGE, IF_ICMPGT, IF_ICMPLE, IF_ACMPEQ, IF_ACMPNE, GOTO, JSR, IFNULL, IFNONNULL). As these instructions are spotted, the corresponding block tree is built in such a way each block has a reference to every single bytecode instruction it contains, and also a pointer to every block representing immediately inner scopes. The result of this process is a high-level, object-based view of the bytecode, which is then used to insert synchronization barriers. Some Java sentences, namely switch/case and try/catch constructs are still not recognized.

Precisely, the algorithm for inserting barriers works by walking through the instructions of a method and detecting the points in which a local variable is either *defined* or *used* by a sentence. A variable is defined when the result of a spawned computation is assigned to it. On the contrary, a local variable is used when its value is read. Since to work properly Satin allows variables to be read provided a sync has been previously issued, our algorithm operates by modifying the bytecode so as to ensure a call to sync is done between the definition and use of any local variable, for any execution path between these two points. Moreover, as sync suspends execution until *all* the subcomputations associated to defined variables have finished, our algorithm employs the aforementioned tree structure to as to keep the correctness of the program while minimizing the inserted calls to sync. Any local variable that does not represent results from parallel computations is naturally ignored by the algorithm.

Algorithm 1 summarizes the process of identifying the points (*syncPoints*) of a method's bytecode (*instr*) where a barrier must be inserted. When analyzing a method, the algorithm maintains a list of the spawnable variables and their associated state per

block, which in turn maintain a private hashing structure that maps a variable with its current state. Possible states are SAFE (up to the current instruction the variable is safe to use, this is, a synchronization point is not needed) and UNSAFE (unsafe to use; a barrier from where the variable is defined is needed). The algorithm takes into account the scope at which spawnable variables are defined and used, this is to say, it computes the state of each variable according to the state it has within the (scope) node of the tree where the variable is read and the state of the same variable within the ancestors of that node. As the reader can see, the algorithm is based on several important helper functions, whose purpose is described below:

deriveBlockTree(instr) Builds the blocks tree corresponding to the input bytecode instructions list *instr*.

isSpawnableVariable(anInstruction) Checks whether the instruction *anInstruction* has a reference to a local spawnable variable. In such a case, the variable code within the method is returned. Local variables are identified in Java bytecode as *$i*, where *i* represents the index of the variable within the method (arguments are codified in a similar way).

getContainerBlock(anInstruction) Returns the block from the tree where some given instruction *anInstruction* belongs. An instruction always belongs to one block only, this is, if a parent block B_P has a child block B_c, the instructions of this latter do not belong to B_P.

beingDefined(varCode) Checks whether a variable (i.e. with code *varCode*) is being assigned a spawnable call. Actually, in bytecode terms, assigning the result of a spawnable call to a local variable involves several instructions forming a recognizable pattern. Under the current implementation of BYG, *instr[i]* potentially corresponds to the first instruction of such a pattern, thus this function analyzes whether the pattern occurs by also taking into account the subsequent instructions.

beingUsed(varCode) Analogous to beingDefined, but checks whether the current instruction reads a spawnable variable. Here, the bytecode pattern associated to using a variable is easier to recognize than the one associated to defining a variable.

getFirstState(varCode,block) Traverses the block tree upwards starting from a given *block* looking for the occurrence of a variable *varCode* in any of the hashing structures of these blocks. When the variable is first found, the function returns the state it has in the block it was encountered.

syncVariablesInBlock(block) Sets to SAFE the state of all spawnable variables contained in *block* (encountered up to the current analyzed bytecode instruction) as well as the ancestors of *block*. The resulting pairs <varCode,SAFE> are just put into the hashing structure associated to *block*.

desyncVariableUpToRoot(varCode,block) Sets the state of a specific variable to UNSAFE from a given block up to the root block. This means that the variable becomes UNSAFE in *block* as well as all its ancestor blocks.

Algorithm 1 Identification of synchronization points

```
1:  procedure IDENTIFYSYNCPOINTS(instr)          ▷ Receives bytecode instructions
2:      tree ← DERIVEBLOCKTREE(instr)
3:      syncPoints ← CREATEEMPTYLIST             ▷ List of synchronization points
4:      for i ← 1, LENGTH(instr) do
5:          if varCode ← ISSPAWNABLEVARIABLE(instr[i]) then
6:              actualBlock ← GETCONTAINERBLOCK(tree,instr[i])
7:              if BEINGUSED(varCode,instr[i]) = true then
8:                  if GETFIRSTSTATE(varCode,actualBlock) = UNSAFE then
9:                      SYNCVARIABLESINBLOCK(actualBlock)
10:                     ADDELEMENT(syncPoints, instr[i])
11:                 else
12:                     DONOTHING
13:                 end if
14:             else if BEINGDEFINED(varCode,instr[i]) then
15:                 DESYNCVARIABLEUPTOROOT(varCode, actualBlock)
16:             end if
17:         end if
18:     end for
19:     return syncPoints
20: end procedure
```

To better illustrate how the algorithm works, let us apply it on a simple recursive method whose code is shown below. Basically, the method contains one non-spawnable variable and two different spawnable variables. For convenience, the points in which synchronization barriers are needed have been explicitly indicated in the code. Figure 6 depicts the state of these spawnable variables within the nodes of the corresponding block tree as the analysis progresses. Finally, for simplicity, we will perform the analysis not on the bytecode of the method but on its source code.

```
1  public String spawnableMethod(int a) { // Block 1
2    String nonSpawnableVar = "not a spawnable variable";
3    String spawnableVarA = spawnableMethod(a/2);
4    if (!nonSpawnableVar.equals("some string")) { // Block 1.1
5      String spawnableVarB = spawnableMethod(a/3);
6      if (a > 0) { // Block 1.1.1
7        ...
8        // A call to sync() should be placed here
9        System.out.println(spawnableVarB);
10       spawnableVarA = spawnableMethod(a/2);
11     }
12   }
13   if (nonSpawnableVar.equals("some other string")) { // Block 1.2
14     ...
15     // Another call to sync() should be placed here
16     System.out.println(spawnableVarA);
17   }
18   ...
19 }
```

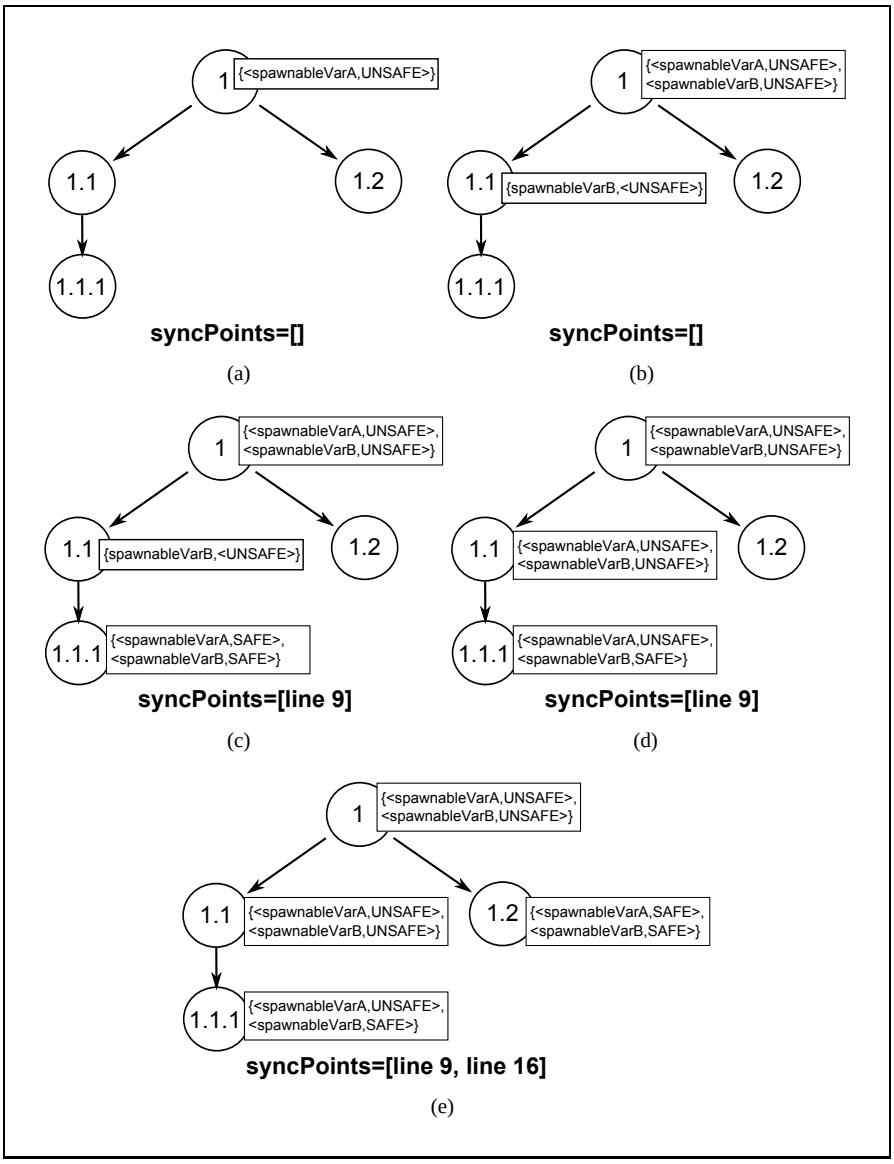


Figure 6. Block trees of the example method in the different steps of the algorithm

The algorithm iterates the instructions of the method up to line 2, where a reference to spawnableVarA is found. As the variable is being defined, it becomes UNSAFE in the current (root) block (see Figure 6 (a)). At line 5, the spawnableVarB is defined within block 1.1, which makes the variable UNSAFE in blocks 1.1 as well as its ancestor block 1 (see Figure 6 (b)). At line 9, spawnableVarB is used within block 1.1.1. Its first occurrence is encountered in the parent of block 1.1.1 as UNSAFE. Consequently, all spawnable variables in the current block's hash table (none) as well as the ones encountered on the ancestors of block 1.1.1 (i.e. spawnableVarA and spawnableVarB) are set to SAFE in block 1.1.1, and a barrier is scheduled for insertion at line 9 (Figure 6 (c)).

Moreover, at line 10 another definition of spawnableVarA is found, which makes the variable UNSAFE in blocks 1.1.1, 1.1 and 1 (Figure 6 (d)). The last relevant line is 16, in which spawnableVarA is being used in block 1.2. According to its parent block 1, the first state of this variable is UNSAFE. This causes the algorithm to set to SAFE all variables found in the variable maps of blocks 1.2 and 1, and to schedule another barrier for line 16 (Figure 6 (e)).

For space reasons, there are many aspects regarding optimizations of the algorithm which are not covered here. Once the input bytecode has been added barriers at the obtained synchronization points (*syncPoints*) identified by the algorithm and instrumented with the Satin class rewriting tools, the bytecode is ready for execution on Satin. Put differently, inserting calls to sync at these points guarantees the operational semantics of Satin. The next subsection describes the server-side support used to execute such a code.

3.1.3. The Satin server

Deploying and running a pure Satin application requires to carry out a number of manual configuration steps. First, the application bytecode has to be copied to the Grid hosts that will participate in the execution of the application. Second, each node must be explicitly assigned a numeric global identifier, the unique identifier of the application, and the address and port of the so-called *nameserver*, which is usually one of these machines. Nameservers provide runtime information about a particular run, such as determining the applications that are being executed, finding hosts participating in a run, providing address and port information, and so on. Finally, the application must be launched by manually initiating it in every host. Then, hosts coordinate themselves to cooperatively execute the Satin application.

However, this mechanism is literally too manual, as it demands users to be excessively involved in the deployment, configuration and even the execution of applications. Besides, the mechanism is inflexible, since the application code that is executed on the Satin platform is determined statically. After launching, applications execute their associated main method that invokes the actual divide and conquer spawnable code, and then die. In consequence, it is not possible to dynamically parametrize Satin with the code to be executed.

To allow Satin connectors to take advantage of the execution services of the Satin platform under a client-server scheme, we developed a Satin server component, which is materialized as a pure Satin application –this is, compliant to the Satin application structure– that is able to execute other Satin applications. A *Satin network* is statically established by simultaneously configuring and starting the Satin server on one or more hosts, which ensures that the Satin runtime is up and waiting for incoming application execution requests at any time. A network is identified by the port on which it listens for requests. A request comprises three elements: a method signature, invocation arguments, and a target Java object that represents an instance of the Satin application on which the method must be executed. Instances of these Java objects are precisely the spawnable objects that are created from the process explained in the previous subsection. The Satin connector interacts with a Satin server to send a spawnable object for execution to a Satin network and wait for the results. Prior to this, the connector communicates with the Host Information Server (HIS) to distribute the application bytecode as well as the necessary third-party libraries from the client node to the hosts of the corresponding Satin network. Basically, the HIS is a centralized component that maintains information

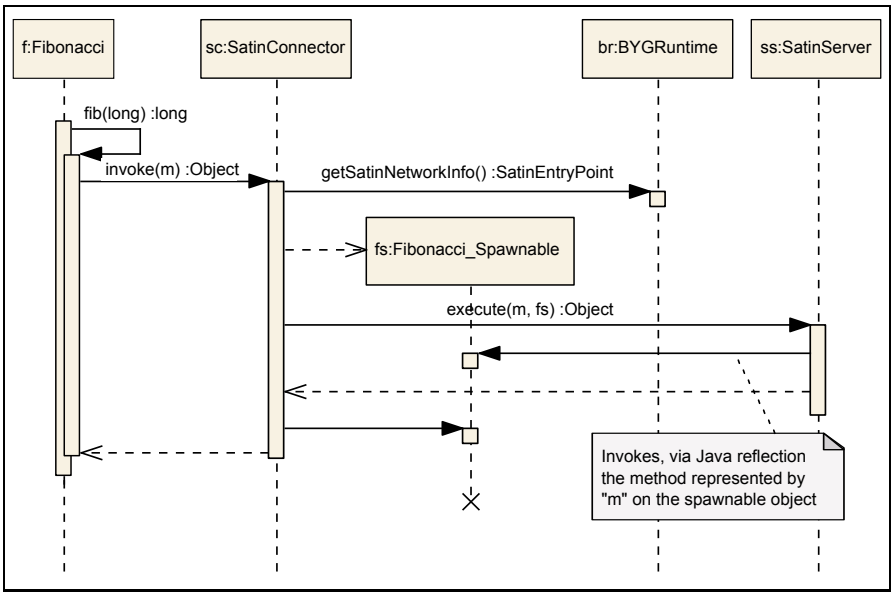


Figure 7. Execution of divide and conquer methods as Satin applications

about the nodes of a Satin network (address, ports, etc.) and provides transparent code transfer capabilities.

Figure 7 shows some of the components that are involved in the execution of the divide and conquer fib operation discussed earlier in this section. For simplicity, we have omitted the interaction with the HIS. When the operation is first invoked, BYG dynamically creates an instance of a Satin application based on the bytecode of the component implementing the method (Fibonacci). The resulting spawnable object and the information for executing fib (i.e. method signature and arguments) is sent to a Satin network, whose parameters are obtained via configuration. Eventually, the computation finishes and the Satin server delivers the result back to the connector, which in turn passes it to the ordinary component.

The BYG runtime is statically supplied with the address of the Satin node that is contacted by connectors to run spawnable objects, and the specific port (*execPort*) where the Satin server application running in that network is listening. Basically, the entry point to a Satin network is the server instance listening on nameserver:execPort. Moreover, several logical Satin networks can be established on top of a number of physical nodes. This is, an individual host can belong to one or more Satin networks, playing the role of either a slave or a master (i.e. nameserver) machine within a single network. These kind of networks are useful for administration purposes, such as logically arranging machines with similar processing capabilities or operating system. The parameters that must be supplied to configure a host as a node of a Satin network are the IP address and the port to which the network's nameserver is bound. Figure 8 exemplifies this support.

As depicted, a simple Grid composed of two Satin networks A (with hosts H_1, H_2 and H_3) and B (with hosts H_3 and H_4) have been configured. The Satin nameserver of A and B are hosts H_1 and H_4, respectively. In consequence, two instances of the Satin server will be run, waiting for incoming execution requests on $H_1 : 10000$ and $H_4 : 10000$. In this way, spawns generated by applications received through the former/latter entry point

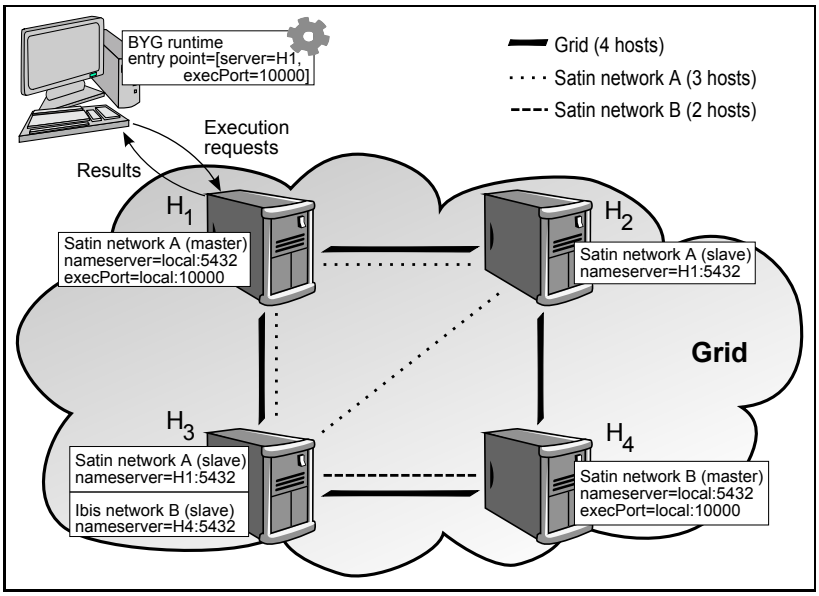


Figure 8. Satin connectors and Satin networks

will be executed on the machines of the Satin network *A*/*B*. By default, Satin connectors send all execution requests to a specific Satin entry point (H_1 : 10000 in our example), but it is also possible to override this information by altering the XML file that configures the Satin connectors for an application.

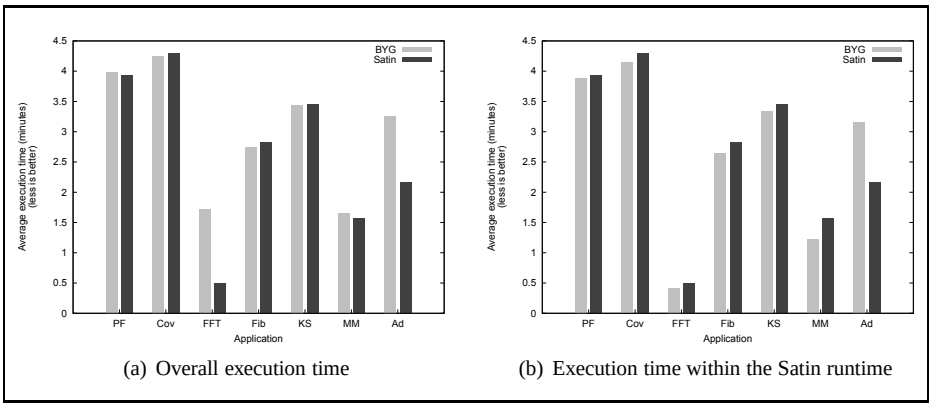
4. Experimental results

This section presents some experiments that were carried out to provide evidence about the practical soundness of BYG. The goal of the experiments was to quantify the performance benefits and potential overheads associated to employing BYG when exploiting existing middleware-level execution services. To this end, we compared the performance in terms of execution time of using Satin versus BYG/Satin connectors by running some classic divide and conquer applications. When using our tool, we also analyzed the time taken to carry out the corresponding administrative tasks before actually executing the divide and conquer codes, namely, instrumenting and then sending the gridified bytecodes to the hosts participating in the experiments.

The evaluation involved the execution of seven different applications: prime factorization (*PF*), the set covering problem (*Cov*), fast Fourier transform (*FFT*), the knapsack problem (*KS*), Fibonacci series (*Fib*), matrix multiplication (*MM*) and adaptive numerical integration (*Ad*). To this end, we set up a cluster composed of 8 machines connected through a 100 Mpbs LAN. Table 2 shows the characteristics of the machines of our experimental setting. To run the applications, we used JDK 5 and Satin 2.1. We chose application parameters that produced moderately long-running computations. All tests associated to the BYG variants of the applications were launched from machine E. The HIS was run on machine H. Figure 9 (a) depicts the overall average execution time for 25 runs of these applications. As a complement, Figure 9 (b) compares the portion of the time

Table 2. Hardware/software specification of the machines of our experimental testbed

Machine	CPU	Memory (MB)	Operating system
A	Intel(R) Pentium(R) 4 2.00 GHz.	1.024	Ubuntu Linux 7.04
B	Intel(R) Celeron(R) 2.40 GHz.	1.024	Mandriva Linux 2007.0
C	Intel(R) Celeron(R) 2.40 GHz.	1.024	Mandriva Linux 2007.0
D	Intel(R) Pentium(R) 4 2.80 GHz.	768	Mandriva Linux 2007.0
E	Intel(R) Pentium(R) 4 2.80 GHz.	1.024	Mandriva Linux 2007.0
F	Intel(R) Pentium(R) 4 2.80 GHz.	768	Mandriva Linux 2007.0
G	Intel(R) Xeon(TM) Dual Core 2.66 GHz.	1.024	CentOS 4.2
H	Pentium III (Coppermine) 852 Mhz.	256	Red Hat Linux 9



(a) Overall execution time

(b) Execution time within the Satin runtime

Figure 9. Test applications: performance results

spent by BYG applications executing under Satin (i.e. within the Satin network) versus the time it took to run these applications natively with Satin. In all cases, deviations were below 7%. Despite being an acceptable noise level when experimenting on wide area Grids, note that this percentage is rather high for a LAN-based cluster. The cause of this effect is that Satin –and therefore our Satin connector– relies on a task scheduler that is based on a set of *random* task stealing algorithms [31]. All in all, except for *FFT* and *Ad*, BYG did not performed much worse than Satin, even when BYG adds a software layer on top of Satin.

The ordinary version of the applications were implemented as a bootstrap class that invoked the actual CPU-intensive computation, which was implemented by another class. In particular, the bootstrap class of *FFT* passed as an argument to the main computation a very large array of data. Consequently, upon gridification, sending the computation for execution to a Satin network required to send this data as well, which resulted in a significant performance overhead. In contrast, in Satin *FFT*, the invocation was far more cheaper as it is performed locally. In a broad sense, the cause of this problem is that distributing the components of the application among different machines and therefore different address spaces can potentially cause the interactions between these components to become much more expensive.

To mitigate this problem, a mechanism for deciding whether it is convenient to submit an operation for execution by means of its associated connector or not could be employed. For instance, we could provide a programmatic or rule-based support to allow developers to express heuristics to indicate the cases in which gridifying an operation may be beneficial (e.g. when the size of the arguments is below some threshold). In addition, complex heuristics for automatically computing the potential gains of gridifying code could be incorporated, for example by taking into account environmental conditions and using user-provided performance models.

From Figure 9 (b) it can be seen that for all the test applications BYG introduced performance gains (of up to 21% for the case of *MM*) with respect to Satin. Similar effects were observed when experimenting with our Satin server and networks in Internet-wide Grids [20]. This fact may result confusing since the BYG connector used in the experiments employs Satin as the underlying support for application execution, but by adding technological noise such as custom Java streams and class reflection, which intuitively should translate into performance overhead. However, the bytecode that is interpreted by the Satin runtime in either cases is subject to different execution conditions. On one hand, when running a pure Satin application, the Satin runtime performs a handshaking process among its hosts to start and cooperatively execute the application. On the other hand, when employing our Satin connectors, the Satin-enabled version of the application being executed is sent by BYG to an already deployed Satin network, which is running a pure Satin application that is able to execute other Satin applications.

With respect to *Ad*, the source of overhead was in the time it took to execute its Grid-enabled bytecode under Satin (see Figure 9 (b)). As mentioned in Section 3.1, at present, our bytecode analysis techniques and our barrier insertion scheme present certain limitations that will be addressed in future implementations of BYG. Particularly, the bytecode rewriting process of BYG may cause applications to have more Satin sync primitives than needed, which harms the performance of applications as the cost of invoking this primitive is rather high. This may also reveal a limitation of the implementation of the Satin sync primitive, which is somewhat expensive and does not consider the case when it is unintentionally called by a programmer more than once after doing spawns.

Figure 10 shows the average *gridification* time (25 executions), which includes (a) the time it took to analyze and instrument the ordinary bytecode in order to inject middleware bridging instructions and synchronization barriers, (b) the time it took to the Satin compiler to instrument the bytecode resulting from the previous step, and (c) the time it took to build and transfer the application jar files to the machines involved in the computations. The file sizes were approximately 14.3 KB (*PF*), 21 KB (*Cov*), 19.1 KB (*FFT*), 12.6 KB (*Fib*), 21.4 KB (*KS*), 20.6 KB (*MM*) and 15.2 KB (*Ad*). In all cases, gridification time was around 3 seconds. It can be observed that (a) remains almost constant, which shows that the performance of the bytecode instrumentation techniques of BYG, at least for these applications, was not affected by the size (in number of bytecode instructions) of the class methods that were configured to be passed on to Satin via connectors. On the other hand, as depicted in the figure, the time required by the Satin compiler to instrument the applications appears to be slightly more affected by the binary size of those methods, since this compiler performs an analysis over the entire class being gridified. When building a pure Satin application, this overhead is not present since compilation is performed offline. However, the programmer must manually build its application with Satin. Finally, since the experiments were run on a LAN, the time required to transfer

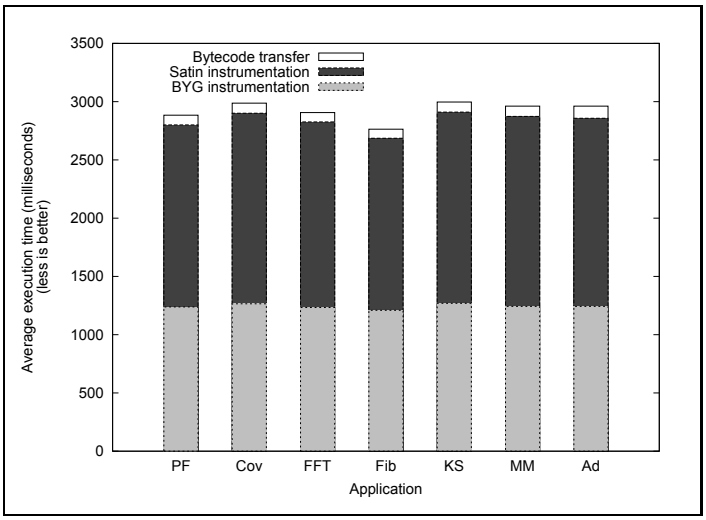


Figure 10. Test applications: bytecode instrumentation and transfer

these files were negligible. Again, this overhead is not present in Satin, as it does not support automatic transfer of classes to the Grid hosts that execute an application. To alleviate the negative effects that would result from employing our bytecode transfer mechanism in wide-area networks, a file caching technique could be used.

5. Conclusions and future work

We have presented BYG, a new approach to simplify the execution of conventional applications on Grids. Essentially, the goal of BYG is to let developers to gridify the binary code of existing applications and at the same to select which portions of the compiled code should run on a Grid and what execution service should be used in each case. The materialization of the approach is oriented towards gridifying component-based applications implemented in Java. As a consequence, we can reasonably expect that the tool will benefit a large number of today's applications.

Experimental results suggest that employing BYG does not imply resigning performance. In contrast, BYG produces Grid-enabled bytecode that can efficiently exploit existing Grid executions services. Particularly, we evaluated our tool by running a number of CPU-intensive applications through Satin connectors and pure Satin, and most of the BYG versions performed in a very competitive way with respect to Satin. We believe this is an interesting result considering that the only tasks that are necessary to gridify an application is to edit a configuration file and to specify a JVM argument. However, despite these encouraging results, we are planning to conduct more experiments with other applications and realistic Grid settings.

It is worth emphasizing that, although the experiments conceived Satin and BYG as competitors, both tools are in some respect complementary. Basically, BYG promotes separation between application logic and the Grid services that are used to execute its associated code. Moreover, these services are provided by existing Grid middlewares. Thus, BYG represents an alternative method for gridifying binary codes rather than a

Grid execution service *per se*. In fact, BYG is currently able to leverage the execution and parallelization services of the Satin platform. Besides, we are working on incorporating more connectors to supply developers with a richer catalog of Grid execution services. For example, we are developing a Condor connector that is based on a Java interface to Condor clusters[7].

We are extending our work in several directions. First, we are addressing the limitations of the implementation of the Satin connector, this is, recognizing more high-level Java sentences (e.g. try/catch), refining the algorithm for inserting Satin synchronization barriers, and so on. Second, as mentioned above, we are implementing connectors for more Grid middlewares. Finally, we are working on incorporating a programmatic or rule-based support to allow developers to specify the set of rules that govern gridification, this is, deciding at runtime whether to run ordinary components via Grid services or execute them unmodified instead.

References

[1] M. Alef, T. Fieseler, S. Freitag, A. Garcia, C. Grimm, W. Gurich, H. Mehammed, L. Schley, and O. Schneider. Integration of Multiple Middlewares on a Single Computing Resource. *To appear in Future Generation Computer Systems, 2008*, 2008.

[2] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. V. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, Mar. 2005.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

[4] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying on the Grid, pages 205–229. Springer, Berlin, Heidelberg, and New York, Jan. 2006.

[5] M. B. Bartosz Baliś and M. Wegiel. LGF: A Flexible Framework for Exposing Legacy Codes as Services. *Future Generation Computer Systems*, 24(7):711–719, July 2008.

[6] A. L. Bazinet, D. S. Myers, J. Fuetsch, and M. P. Cummings. Grid Services Base Library: A High-Level, Procedural Application Programming Interface for Writing Globus-Based Grid Services. *Future Generation Computer Systems*, 23(3):517–522, 2007.

[7] M. Cierniak and W. Li. Optimizing Java Bytecodes. *Concurrency: Practice and Experience*, 9(6):427–444, 1997.

[8] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The WS-Resource Framework. `http://www.globus.org/wsrf`, May 2004.

[9] T. Delaittre, T. Kiss, A. Goyeneche, G. Terstyanszky, S.Winter, and P. Kacsuk. GEMLCA: Running Legacy Code Applications as Grid Services. *Journal of Grid Computing*, 3(1-2):75–90, June 2005.

[10] D. W. Erwin. UNICORE - A Grid Computing Environment. *Concurrency and Computation: Practice and Experience, Special Issue on Grid Computing Environments*, 14(13-15):1395–1410, June 2002.

[11] I. Foster. The Grid: Computing without Bounds. *Scientific American*, 288(4):78–85, Apr. 2003.

[12] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *Journal of Computer Science and Technology*, 21(4):513–520, July 2006.

[13] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*, chapter Concepts and Architecture, pages 37–63. Morgan-Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[14] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[15] D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan, and A. Slominski. On Building Parallel and Grid Applications: Component Technology and Distributed Services. *Cluster Computing*, 8(4):271–277, Oct. 2005.

[7] Condor Java API `http://staff.aist.go.jp/hide-nakada/condor_java_api/index.html`

[16] R. Johnson. J2EE Development Frameworks. *Computer*, 38(1):107–110, 2005.

[17] S. M. Larson, C. D. Snow, M. R. Shirts, and V. S. Pande. *Modern Methods in Computational Biology*, chapter Folding@Home and Genome@Home: Using Distributed Computing to Tackle Previously Intractable Problems in Computational Biology. Horizon Press, 2003.

[18] E. Laskowski, M. Tudruja, R. Olejnik, and B. Toursel. Byte-code Scheduling of Java Programs with Branches for Desktop Grid. *Future Generation Computer Systems*, 23(8):977–982, Nov. 2007.

[19] D. Manolescu, B. Beckman, and B. Livshits. Volta: Developing distributed applications by recompiling. *IEEE Software*, 25(5):53–59, 2008.

[20] C. Mateos. *An Approach to Ease the Gridification of Conventional Applications*. PhD thesis, Universidad del Centro de la Provincia de Buenos Aires, Feb. 2008. `http://www.exa.unicen.edu.ar/~cmateos/files/phdthesis.pdf`.

[21] C. Mateos, A. Zunino, and M. Campo. A Survey on Approaches to Gridification. *Software: Practice and Experience*, 38(5):523–556, Apr. 2008.

[22] C. Mateos, A. Zunino, and M. Campo. JGRIM: An Approach for Easy Gridification of Applications. *Future Generation Computer Systems*, 24(2):99–118, Feb. 2008.

[23] S. McGough, W. Lee, and S. Das. A Standards Based Approach to Enabling Legacy Applications on the Grid. *Future Generation Computer Systems*, 24(7):731–743, July 2008.

[24] A. Natrajan, M. A. Humphrey, and A. S. Grimshaw. The Legion Support for Aadvanced Parameter-Space Studies on a Grid. *Future Generation Computer Systems*, 18(8):1033–1052, 2002.

[25] A. Paventhan, K. Takeda, S. J. Cox, and D. A. Nicole. MyCoG.NET: A Multi-language CoG Toolkit. *Concurrency and Computation: Practice and Experience, Special Issue on Middleware for Grid Computing: 'A Possible Future'*, 19(14):1885–1900, 2006.

[26] T. A. Proebsting and S. A. Watterson. Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?). In *3rd USENIX Conference on Object-Oriented Technologies (COOTS '97)*, pages 185–198, Berkeley, CA, USA, 1997. USENIX Association.

[27] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Boston, MA, USA, 2002.

[28] D. Thain and M. Livny. Error scope on a computational grid: Theory and practice. In *11th IEEE Symposium on High Performance Distributed Computing (HPDC-11 '02), Edinburgh, Scotland*, pages 199–208, Washington, DC, USA, July 2002. IEEE Computer Society.

[29] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and A. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 299–335. John Wiley & Sons Inc., New York, NY, USA, Apr. 2003.

[30] G. von Laszewski, J. Gawor, P. Lane, N. Rehn, and M. Russell. Features of the Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience, Special Issue on Grid Computing Environments*, 14(13-15):1045–1055, Jan. 2003.

[31] G. Wrzesinska, R. V. van Nieuwport, J. Maassen, T. Kielmann, and H. E. Bal. Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments. *International Journal of High Performance Computing Applications*, 20(1):103–114, 2006.

About the authors

Cristian Mateos received a Ph.D. degree in Computer Science from UNICEN, Tandil, Argentina, in 2008. He is a Full Teacher Assistant at the Computer Science Department of UNICEN. His recent thesis was on a solution to ease Grid application development through non-intrusive injection of Grid services.

Alejandro Zunino received a Ph.D. degree in Computer Science from UNICEN in 2003. He is a Full Adjunct Professor at the Computer Science Department of UNICEN and a research fellow of the CONICET. He has published over 30 papers in journals and conferences.

Marcelo Campo received a Ph.D. degree in Computer Science from UFRGS, Porto Alegre, Brazil. He is a Full Associate Professor at the Computer Science Department and Head of the ISISTAN. He is also a research fellow of the CONICET. He has

over 70 papers published in conferences and journals about software engineering topics.

Ramiro Trachsel is a BSc. candidate in Systems Engineering at UNICEN. He is a Teacher Assistant at the Computer Science Department of UNICEN. His involvement in this project came from an interest in Grid architectures and technologies.