Computing and Informatics, Vol. 30, 2011, 1259–1285

A NOVEL MECHANISM FOR GRIDIFICATION OF COMPILED JAVA APPLICATIONS

Cristian Mateos, Alejandro Zunino
Ramiro Trachsel, Marcelo Campo

ISISTAN – UNICEN. Tandil (B7001BBO), Buenos Aires, Argentina
Also Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
e-mail: `cmateos@exa.unicen.edu.ar`

Communicated by Jacek Kitowski

Abstract. Exploiting Grids intuitively requires developers to alter their applications, which calls for expertise on Grid programming. Gridification tools address this problem by semi-automatically making user applications to be Grid-aware. However, most of these tools produce monolithic Grid applications in which common tuning mechanisms (e.g. parallelism) are not applicable, and do not reuse existing Grid middleware services. We propose BYG (BYtecode Gridifier), a gridification tool that relies on novel bytecode rewriting techniques to parallelize and easily execute existing applications via Grid middlewares. Experiments performed by using several computing intensive applications on a cluster and a simulated wide-area Grid suggest that our techniques are effective while staying competitive compared to programmatically using such services for gridifying applications.

Keywords: Grid computing, gridification, parallelism, Grid middlewares, Satin, resource-intensive applications, Java bytecode

1 INTRODUCTION

Grids arrange geographically dispersed computational resources to execute computing intensive applications [7]. Grid resources are managed through Grid middlewares such as Globus, Condor-G [21] and CrossBroker [6], which virtualize resources by means of *services* (e.g. job scheduling) and supply developers with APIs for using them. Recently, research in Grid middlewares has focused on simplifying the consumption of the offered services from within applications. Works in this line are grouped into programming toolkits and gridification tools [14]. Programming toolkits provide high-level APIs that abstract away the details to interact with such services, thus less effort is required compared to directly using Grid middleware APIs, or low-level Grid libraries such as MPI [5, 28]. However, these toolkits still require solid knowledge on Grid programming.

Gridification tools incorporate Grid services into existing applications by semi-automatically transforming codes to run on a Grid; thus they better support users having little background on Grid technologies. Gridification tools can accept as input the source code of an application, its compiled version or both. The first approach gives developers more control over their codes to build efficient Grid applications. The second approach allows submitting codes "as is" for execution on a Grid platform. In the third approach gridification happens at both the source code and compiled code levels. Typically, programmers are supplied with directives to annotate their source codes, which are then processed to incorporate the Grid behavior. From now on, applications employed to feed a gridification tool will be referred as "Grid-unaware" applications, whereas applications produced from using such a tool will be referred as "Grid-aware" or "gridified" applications.

Unfortunately, current techniques for gridifying binary codes prevent the usage of common Grid tuning mechanisms. Then, gridified applications are monolithic, coarse grained Grid-aware codes that cannot be altered to better exploit Grid resources. Most of these approaches do not provide mechanisms for distributing or parallelizing individual parts of an application. Although they simplify gridification, they usually lead to a poor usage of Grid resources [14]. Therefore, there is a need for alternative gridification tools that provide a convenient balance between the effort necessary to deploy and run codes on a Grid, and the *gridification granularity* [14] or the levels at which the application can be tuned. We propose BYG (BYtecode Gridifier), a Java tool that allows binary codes to be gridified at various granularity levels. Users can configure the components[1] of their applications that are subject to execution on Grid middlewares. BYG does not provide yet another Grid resource manager, but offers a glue between gridified applications and the execution services of existing Grid platforms.

Preliminary experiments conducted on a small cluster [16] showed the feasibility of BYG. Here, we provide a deeper explanation of BYG concepts and report experiences with BYG in a larger cluster and a wide-area Grid to run classic computing intensive codes. Results show that BYG does not incur in much performance overheads compared to manually programming Grid applications and proved to be very competitive. The next section discusses relevant related works. Section 3 overviews BYG. Section 4 explains the integration of BYG with Condor-G and Satin, two representative Grid middlewares. Section 5 reports the experiments. Section 6 concludes the paper.

[1] From now on, the term should be understood in the context of component-based software.

2 BACKGROUND

Several approaches for gridifying software can be found in the literature. We will focus on the efforts aimed at making gridification as invisible to the user as possible, i.e. those works that transparently incorporate Grid behavior into the compiled versions of Grid-unaware applications. Table 1 summarizes these approaches. The first column indicates the type of binary code accepted as input by each tool. In general, tools that only allow users to submit their compiled codes "as is" to a Grid are based on machine-dependent binary codes. The second column details the supported gridification granularities. "Coarse", "medium" and "fine" means that the Grid execution units associated to an application are derived from either its entire code, some of its objects or components, and some of its methods or operations, respectively. The third column indicates whether each tool provides integration with the execution services of existing Grid middlewares and Resource Management Systems (RMS). The following subsections discuss the tools presented in Table 1.

Tool	**Binary code flavor**	**Gridification granularity**	**Grid platform reuse**
GEMLCA	Machine-dependent	Coarse	Partially (only Globus)
GridSAM	Machine-dependent	Coarse	Yes
GRID superscalar	Machine-dependent	Coarse	Yes
DG-ADAJ	Machine-independent (bytecode)	Fine	No (custom RMS)
LGF	Machine-dependent	Fine, coarse	No (custom RMS)
J-Orchestra	Machine-independent (bytecode)	Medium	No (RMI messaging)
ProActive	Machine-independent (bytecode)	Medium	Yes
Satin	Machine-independent (bytecode)	Fine	Partially (only Globus)
Volta	Machine-independent (CIL)	Fine	No (custom RMS)
XCAT	Machine-dependent	Coarse	Yes

Table 1. Contemporary gridification tools based on binary codes and hybrid approaches

2.1 Machine-Dependent Binary Code Gridifiers

GEMLCA [4] lets users to deploy legacy programs as OGSA-compliant services [10] via a frontend. To execute the gridified codes, GEMLCA uses the GRAM job submission service of Globus. Upon gridification, the user specifies metadata information (parameters, executable path, etc.) and resource requirements (CPUs, memory, etc.) for his application in a configuration file. GEMLCA uses a nongranular execution scheme (i.e. running the same binary code on several processors) but no internal changes are made in the gridified applications. Then, distribution and parallelism of individual application modules cannot be controlled in a fine-grained manner. GridSAM [17] can be used to publish legacy applications as Web Services, which are then treated as separate modules that can be composed via a workflow description document that is executed on top of other Grid platforms. Unlike GEMLCA, GridSAM is not an RMS, but an interface to existing Grid job execution services.

XCAT [8] supports execution of component-based applications on existing Grid platforms by linking components to middleware-level execution services. Application components can also wrap legacy binary programs. Complex applications are built by programmatically assembling components together, which demands little coding effort from the developer but still requires knowledge on the XCAT API. As both XCAT and GridSAM treat legacy codes as black boxes, they share the limitations of GEMLCA with respect to gridification granularity. GRID superscalar [25] provides an API for programming applications composed of tasks, whose granularity is at the level of programs, which take data files as input and produce result files as output. The tool includes a special compiler that links compiled task codes together and optimizes the performance of the application by analyzing file dependencies. LGF [2] is another framework for deploying legacy applications as Web Services. Central to its design is a two-layered architecture in which the adaptation layer is heavily decoupled from the backend layer. With LGF, it is possible to monitor the performance of Grid applications at the application and code region levels. However, LGF does not take advantage of existing Grid execution services.

2.2 Machine-Independent Binary Code Gridifiers

DG-ADAJ [11] is a mechanism for transparent execution of multi-threaded Java applications on JVM (Java Virtual Machine) clusters. DG-ADAJ derives graphs from a compiled application, which account for data and control dependencies, by using representative sets of input data. Then, an heuristic is applied to place mutually exclusive execution paths from the graphs among the hosts of a cluster. DG-ADAJ promotes threads as the base programming model, which have been much criticized since threads make programming and debugging rather difficult [12]. In contrast, BYG gridifies single-threaded programs by leveraging existing execution services for exploiting Internet-wide Grids.

ProActive [1] provides *technical services*, a support to address non-functional Grid concerns (e.g. load balancing and fault tolerance) by plugging configuration external to applications at deployment time. ProActive allows users to gridify compiled classes as mobile entities without code modification, and features integration with a variety of Grid schedulers. However, creating computations based on a subset of the methods of a Grid-unaware class requires to manually use the ProActive API, which is precisely what gridification tools intend to avoid. Lastly, J-Orchestra [22] relies on bytecode rewriting to place application objects into distributed JVMs and to replace local method calls with remote methods calls based on RMI. Classes/objects subject to distribution are indicated through a GUI. J-Orchestra is essentially an application partitioner operating at the class granularity level. Unlike BYG, J-Orchestra takes advantage of object distribution but is not designed to exploit parallelism within classes.

2.3 Hybrid Approaches

Other tools follow a *hybrid* approach to gridification, in which developers are involved in the process of altering an application to gridify it. It is worth noting that the term "hybrid" does not refer to gridifiers able to both gridify machine-dependent and machine-independent binary codes, but refers to tools requiring modifications to the source code of input applications prior to automatically alter their compiled counterpart.

Satin [24] is a Java framework for parallelizing divide and conquer applications. The user indicates in the application code the points in which a fork or a join should take place. Then, Satin instruments the compiled code to transparently handle the execution of parallel tasks on a Grid. Similarly, Volta [13] recompiles executable .NET applications on the basis of declarative developer annotations to insert remoting and synchronization primitives so as to transform applications into their distributed form. Recompilation operates at the CIL (.NET Common Intermediate Language) level. Precisely, the weak point of these tools is that they require some modifications to the source code of applications.

3 THE BYTECODE GRIDIFIER APPROACH

While the discussed approaches are targeted at users with little expertise on Grid technologies, Satin and Volta are some way off from being true binary code gridifiers, as they impose source code modifications. GEMLCA, GridSAM, ProActive, J-Orchestra, XCAT and GRID superscalar offer a poor balance to the "ease of gridification versus fine tuning" trade-off [14], since they avoid code modification, but gridification results in coarse or medium-grained Grid execution units that cannot be restructured for parallelism. Only a small number of the approaches are designed to fully exploit the execution services of other Grid platforms. In fact, DG-ADAJ, LGF, J-Orchestra and Volta rely on custom Grid execution platforms. However, a trend in Grid Computing, as evidenced by broadly adopted standards such as OGSA and WSRF [10], is to promote interoperability among Grid tools and middlewares. Then, Grid middleware integration is now the rule and not the exception.

We propose BYG (BYtecode Gridifier), a gridification tool that avoids source code modification, and deals with the above trade-off by letting developers to tune their applications with little effort while minimizing the deployment effort to put a Grid application to work. BYG leverages the execution services of existing Grid platforms through the explicit but non-invasive use of *connectors*, which materialize the protocols to access the various services of specific Grid platforms. Connectors are non-intrusively injected directly into the binary code to delegate the execution of certain parts of the application to a Grid platform and to transparently adapt this code to take advantage of the API provided by the target platform. The mapping of which parts of an application are delegated to such services is specified by means of user-supplied configuration.

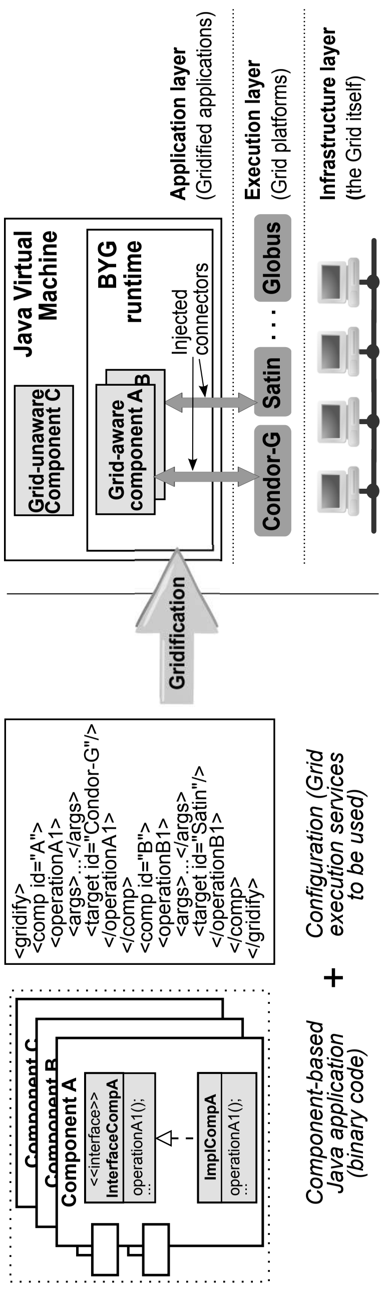
BYG targets component-based applications implemented in Java. We chose Java as it is broadly adopted by developers, and provides many features that facilitate the modification of applications at the bytecode level (e.g. extensible class loading and reflection). In addition, component-based notions are commonplace in Java, as evidenced by the high popularity of component models such as JavaBeans and EJB. For these reasons, BYG can benefit a large amount of today's Java applications.

Component-based development focuses on designing applications as a number of logical components with well-defined interfaces. Components only know each other's interfaces, and are self-contained, thus any kind of interaction involving tightly-coupled communication is disallowed (e.g. invoking component operations by passing arguments by reference). Therefore, any application component can be replaced without affecting the rest of the components. Precisely, this allows BYG to non-invasively replace a Grid-unaware component with its Grid-aware version.

Figure 1 overviews BYG. Conceptually, BYG takes an application and dynamically transforms it so as to run some component operations on different Grid platforms. In this sense, BYG can be seen both as a competitor of existing approaches and a complement to them. BYG processes the developer's configuration, intercepts all invocations to such operations (in this case *operationA1* and *operationB1*), and delegates their execution to the associated target Grid platform (in this case Condor-G and Satin, respectively) by using connectors. All in all, the benefits of this approach are:

Less gridification effort: The developer specifies which parts of its application should be executed on a Grid, but without explicitly introducing source code modifications to do so. In addition, the Grid services associated to an application can be detached by simply modifying its configuration, i.e. "ungridifying" the application.

Middleware independence: Depending on the nature of each component operation, different Grid execution services could be used. For example, all mission critical computations may be submitted to a Grid platform with fault tolerance capabilities such as Condor-G (see Section 4.1). Furthermore, all calls on



```
<gridify>
 <comp id="A">
  <operationA1>
   <args>...</args>
   <target id="Condor-G"/>
  </operationA1>
 </comp>
 <comp id="B">
  <operationB1>
   <args>...</args>
   <target id="Satin"/>
  </operationB1>
 </comp>
</gridify>
```

Fig. 1. Overview of BYG

a parallel operation may be sent to a Grid platform able to execute the operation in parallel (e.g. Satin) in order to improve performance and scalability (see Section 4.2).

Flexible fine tuning: Unlike related tools, in which applications are mostly executed in a black box fashion, developers can use BYG to fine-tune the execution of their applications by submitting calls to component operations of various sizes. Rather than following the coarse-grained gridification scheme [14] promoted by many tools, in which a whole application is submitted for execution to a Grid, BYG lets programmers to select which component operations – of potentially different granularities – are gridified. Then, the unit of computation in BYG are component operations.

Using or not a specific Grid execution service is mostly subject to availability factors, i.e. whether a Grid running the desired Grid platform is available for job submission. However, the choice of gridifying an operation depends on whether the operation is suitable for being gridified. The potential performance gains in gridifying an application depends on two design factors: the amount of data (i.e. parameters) that needs to be transferred to execute the gridified operations, and the resource requirements of the operations. BYG aims at alleviating the burden of adapting and submitting a Grid-unaware application for execution on a Grid, while these factors must be addressed early by the user.

Architectonically, BYG provides a tier that mediates between a Grid-unaware application (the client side) and Grid middlewares (the server side). Gridified components are run at the server side by means of connectors, whereas non-gridified components remain at the client side. In principle, BYG can take advantage of any Grid middleware exposing a well-defined remote job submission interface for Java. Depending on the case, an additional software layer on top of the target middleware may be required though. Even when BYG can be used with different Grid middlewares, the approach is not capable of accessing any middleware in a fully transparent way, because some Grid platforms cannot be easily exploited in a client-server fashion. In the following sections we will illustrate this requirement in the context of our bindings to Satin.

As mentioned, BYG is designed to gridify *component-based* applications. In general, this ensures that application components are highly decoupled, so that component operations can be run in a different memory address space without worrying about which component issued the invocation and how the operation arguments and their results are interchanged. Depending on the particular connectors being used, operations must adhere to certain coding conventions at development time. 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 suffices for the effective use of the various connectors. For instance, these two latter ones allow BYG to spot the points in a component's bytecode representing calls to operations and access to their results, which in turn let BYG to rewrite the bytecode of the component to exploit asynchrony and parallelism at the operation level. Moreover, good method modularization allows for flexible tuning of components at different granularity levels.

There are two types of component-based applications for which BYG has limitations. Applications in which components have a high degree of interdependency (e.g. workflows), may not be suitable for BYG. Particularly, a workflow is a collection of tasks (or components) with transitions between them. The higher the number of transitions, the higher the amount of communication between components at runtime. Then, remotely executing some of these components may increase communication costs and thus render gridification counterproductive. For similar applications, developers should analyze the amount of communication before deciding whether to use BYG or not. Another limitation may arise with applications comprising data components (e.g. files), which have a semantic based on their localization and whose execution environment cannot be changed. A common approach to address this problem are proxies [1], so that remote (or gridified) components can transparently access the data components, or any other non-gridified components. For instance, J-Orchestra [22] supports proxies by modifying bytecodes to allow partitioned remote objects to transparently communicate between each other.

We have developed a proof-of-concept implementation of BYG, which is described in the next section[2]. The prototype can gridify applications compliant to the JavaBeans specification, a set of conventions for componentizing Java classes that is very popular in the software industry. Gridifying a compiled application with BYG only requires to

1. configure an XML file that instructs BYG how to map component operations to Grid execution services, and
2. add a JVM argument to the bootstrap script that initiates the user application.

Currently, BYG supplies connectors for Condor-G [21] and Satin [24].

4 PUTTING BYG TO WORK

The first step to gridify an application with BYG is creating an XML configuration file, which specifies the components to be gridified, and the binding information (or *entry point*) that depends on the Grid middleware(s) to which BYG will delegate the execution of these components. Consequently, the user has to know some details of the Grid host that plays the role of job executor of each middleware. Broadly, entry points are a common abstraction of the middleware-level frontend components that reside on a specific host of a Grid and accept jobs for execution. An application may have many parts suitable for execution on a Grid. To this end, the user must provide

[2] BYG is available at `http://www.exa.unicen.edu.ar/~cmateos/files/BYG-1.0.zip`.

1. the list of Java methods (owner class and signature) to be gridified,
2. the connectors to be used, and thus the Grid execution service, and
3. the job submission protocol over those provided by the connectors being employed

for example, Condor-G offers a remote job submission mechanism based on raw sockets and a Web Service submission interface.

```
1  <connectors>
2    <connector name=”example”>
3      <middleware name=”satin”>
4        <property name=”protocol”>raw_sockets</property>
5        <property name=”host”>satin_server_ip</property>
6        <property name=”port”>satin_server_port</property>
7      </middleware>
8      <classes>
9        <class name=”Fib”>
10         <method name=”fib”>
11           <parameter type=”long”/>
12         </method>
13       </class>
14     </classes>
15   </connector>
16 </connectors>
```

The above configuration tells BYG to execute a method from the *Fib* class (lines 9–13) via the services of Satin (lines 3–7). Lines 3–7 are the binding information necessary to submit this method for execution to a Grid running Satin. Particularly, *host* and *port* are the contact information of the entry point to Satin. It is possible to define more than one connector within a configuration file, and associate several methods to them.

BYG uses the *java.lang.instrument* package, a built-in Java API for modifying bytecodes intended to be extended through special libraries called Java agents, which run embedded in the JVM and customize the class loading process. The kernel of the mechanism for dynamically injecting connector code into application classes in BYG is implemented as a Java agent. To gridify an application, its startup command must look like: *java -javaagent:bygAgent.jar=config-file main-class* ... At runtime, the BYG agent processes the user's configuration file and instruments the bytecodes of the affected methods as classes load. Instrumenting an individual class method implies:

1. Rewriting its body to include instructions for launching the execution of its bytecode on a specific Grid middleware. The injected "stub" employs the corresponding *protocol*, *host* and *port* properties to send an adapted version of the original method body for execution to a Grid every time this method is called by the application.

2. Adapting its original bytecode (if applicable) to take advantage of the middleware the method has been connected to, which involves preparing the method and 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 exploit parallelism, and so on.

Grid middlewares with coarse gridification granularities like Condor-G do not provide mechanisms to express independent computations within a method. Then, gridifying an operation with BYG/Condor-G does not require to perform step 2. However, middlewares with finer granularities such as Satin do have API primitives to express parallelism. BYG exploits such primitives by generating *peers*, which are components whose bytecode is derived from the components being gridified but rewritten so that they exploit the underlying paralellization constructs. In summary, building a connector requires transferring code and parameter values to execute methods within a Grid, and designing the bytecode rewriter according to targeted middlewares. The first task mostly requires engineering efforts. For middlewares not relying on coarse-grained execution models, the second task raises some difficult issues as it is necessary to transparently modify the input bytecode to exploit the underlying parallel primitives. The next two subsections discuss the Condor-G and Satin connectors, respectively, emphasizing on how they address these aspects.

4.1 The Condor-G Connector

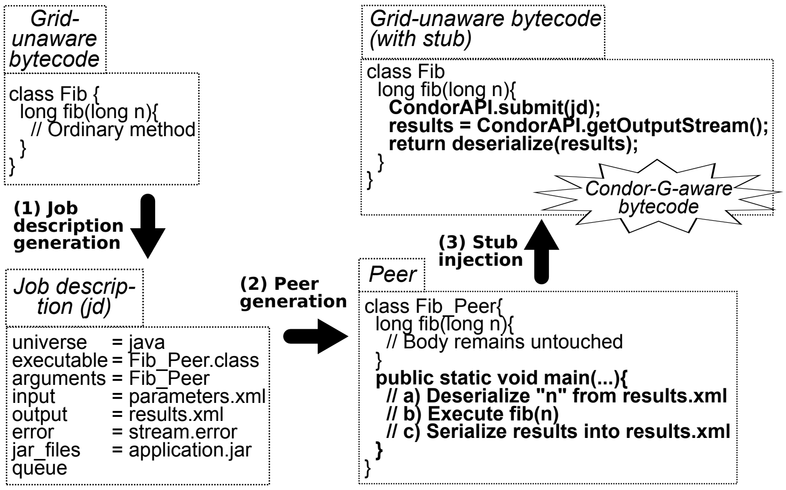
Condor-G [21] is a popular Grid middleware that provides a powerful task broker for machine-dependent executables. Condor-G also offers a subsystem to create jobs from compiled Java programs. Based on a Java interface to this middleware [18], we have built a Condor-G connector that wraps Grid-unaware components as Java applications and submits them for execution to Condor-G. This interface is an API with functionality for talking to the *master*, i.e. the entry point to Condor-G. To submit a Java class to Condor-G, it is necessary to create a description file with the fully-qualified class name, the input, output and error stream files for the corresponding Java process, and the JAR files of the application. For instance, to submit the *Example* class, this file would have the following directives:

```
universe = java, executable = Example.class,
arguments = Example, input = stream.input,
output = stream.output, error = stream.error,
jar_files = application.jar, queue
```

Example must contain a main method with the computing intensive code. In this case, Condor-G will transfer *stream.input* and *application.jar* from the host where the job was submitted to the host where it will be executed. Condor-G executes *Example* and then transfers *stream.output* and *stream.error* back to the submitting host. Lastly, the *queue* command indicates Condor-G to execute the job only once.

The Condor-G connector builds, from the bytecode of a Grid-unaware component operation, a class file (the *peer*) and its description file. These files represent a Condor-G job, which are submitted to Condor-G by the connector through the Condor-G API. Upon job completion, the results of the operation and potential errors are passed back by BYG to the Grid-unaware component that originally called the operation. The input and output streams are used to transfer the operation parameters and return value, respectively.

Figure 2 illustrates how the connector creates a Condor-G job from a Grid-unaware component operation. Based on the bytecode of the Grid-unaware class and the method to gridify, the connector generates a job description (step 1) and a peer (step 2). The peer includes a copy of the methods of the input component plus the static main method that is invoked by Condor-G upon job submission. As depicted, the generated description file points to two XML files, which are used by the peer at runtime to read the parameters of the invocation to *fib* issued at the client side, and to return the results of the computations performed at the (Condor-G) server side. To this end, we use XStream [23], a library to serialize/deserialize any Java object to/from XML. Stub injection is performed at step 3, in which the Grid-unaware operation is rewritten to transparently submit its gridified counterpart to Condor-G. The stub uses the entry point information from the user-supplied configuration to contact the Condor-G master. Bytecode instrumentation is performed by using ASM [19].



```
class Fib {
  long fib(long n){
    // Ordinary method
  }
}
```

```
universe    = java
executable  = Fib_Peer.class
arguments   = Fib_Peer
input       = parameters.xml
output      = results.xml
error       = stream.error
jar_files   = application.jar
queue
```

```
class Fib_Peer{
  long fib(long n){
    // Body remains untouched
  }
  public static void main(...){
    // a) Deserialize "n" from results.xml
    // b) Execute fib(n)
    // c) Serialize results into results.xml
  }
}
```

```
class Fib
  long fib(long n){
    CondorAPI.submit(jd);
    results = CondorAPI.getOutputStream();
    return deserialize(results);
  }
}
```

Fig. 2. Submitting Grid-unaware classes to Condor-G through the Condor-G connector

Unlike the Condor-G connector, which provides means to run Grid-unaware operations by using a coarse granularity execution model (i.e. a single operation call is encapsulated into just one Grid job), the Satin connector modifies the bytecode of such operations to exploit parallelism at a finer level of granularity (i.e. a single call may dynamically generate more than one Grid job). The next subsection explains the Satin connector.

4.2 The Satin Connector

Satin [24] allows parallelizing divide and conquer Java codes by providing two primitives: *spawn*, to create subcomputations, and *sync*, to block execution until subcomputations are finished. Methods considered for parallel execution are identified through *marker interfaces*. In addition, the result of the invocations to such methods must be stored in local variables. Below is the Satin code to compute the n^{th} Fibonacci number:

```
interface FibMarker extends satin.Spawnable{
  long fib(long n);
}
class Fib extends satin.SatinObject implements FibMarker{
  long fib(long n){
    if (n < 2) return n;
    long f1 = fib(n - 1); // Spawned according to IFibMarker
    long f2 = fib(n - 2); // Spawned according to IFibMarker
    sync(); // Blocks until f1 and f2 are instantiated
    return f1 + f2;
  }
}
```

After specifying spawnable methods and inserting appropriate synchronization calls into the application, the developer must use the Satin compiler, which translates each invocation to a spawnable method into a Satin runtime task. In our example, a task is generated for every call to *fib*. The Satin connector automatically reproduces these manual tasks from the compiled version of components which have not been coded to use the Satin API. The connector generates the marker interface(s) and rewrites the bytecode of the component(s) to follow the anatomy of Satin applications (next subsection). The connector also inserts proper calls to *sync* by analyzing where a barrier must be introduced (subsection 4.2.2). To execute Satin-aware components, the connector relies on an extended Satin runtime [16], which provides a remote, client-server interface to the execution services of Satin.

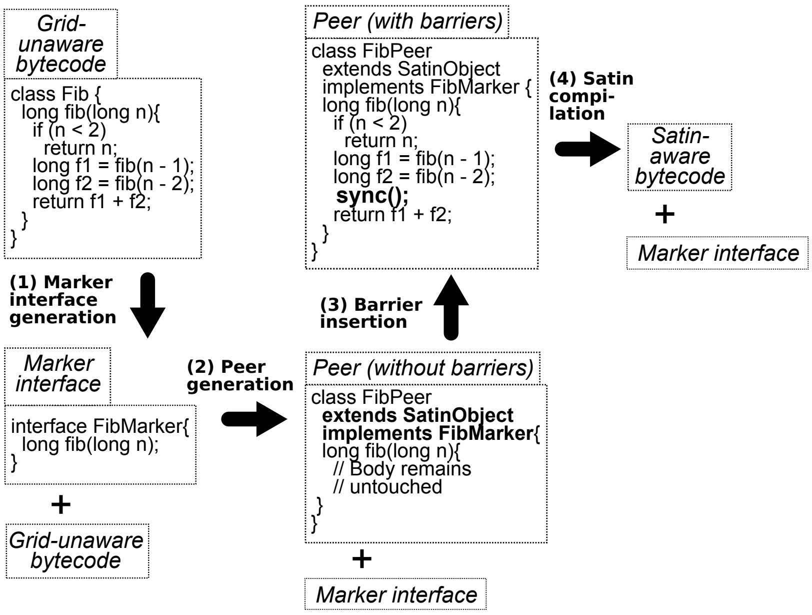
4.2.1 Dynamic Bytecode Instrumentation

Beside injecting proper glue bytecode to execute Grid-unaware 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 with the application anatomy described in the previous section. To this end, the Satin connector carries out two main tasks:

Marker interface generation: Satin requires applications to implement a marker interface, which explicitly lists the methods considered by Satin for parallel execution. The connector builds this interface from the methods listed in the XML configuration for the class being connected to the Satin services by using ASM.

Peer generation: Satin requires applications to implement a marker interface and to extend from *SatinObject*. In this sense, a clone or peer of the non-gridified class under consideration is created and instrumented to fulfill these requirements by using ASM. In a subsequent step, the peer is further rewritten to use the Satin *sync* primitive.

Figure 3 depicts the steps performed by the Satin connector to build the Satin-aware version of a class. Based on the bytecode of the class being gridified and the target method(s), the connector creates the corresponding marker interface (step 1) and a Satin peer for it (step 2). Then, based on a special algorithm, the connector automatically inserts calls to the *sync* primitive into the generated peer (step 3). Afterwards, the peer is processed with the Satin compiler itself (step 4). At runtime, the final peer is instantiated at the client side by the connector and submitted for execution to the abovementioned extended Satin runtime.



```
class Fib {
  long fib(long n){
    if (n < 2)
      return n;
    long f1 = fib(n - 1);
    long f2 = fib(n - 2);
    return f1 + f2;
  }
}
```

```
class FibPeer
  extends SatinObject
  implements FibMarker {
  long fib(long n){
    if (n < 2)
      return n;
    long f1 = fib(n - 1);
    long f2 = fib(n - 2);
    sync();
    return f1 + f2;
  }
}
```

```
interface FibMarker{
  long fib(long n);
}
```

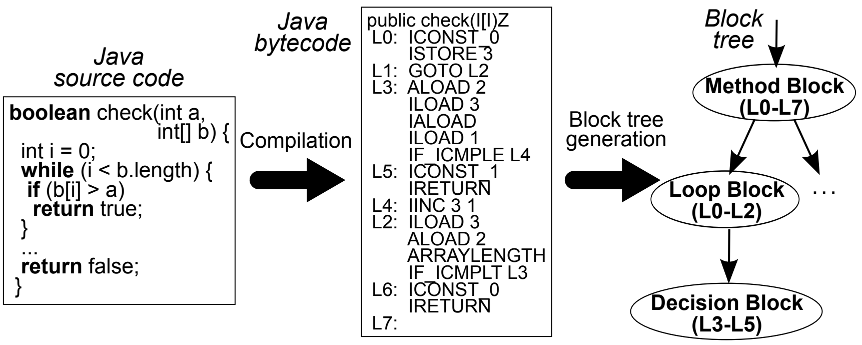
```
class FibPeer
  extends SatinObject
  implements FibMarker{
  long fib(long n){
    // Body remains
    // untouched
  }
}
```

Fig. 3. Gridifying Grid-unaware through the Satin connector

4.2.2 Automatic Insertion of Synchronization Barriers

When programming with Satin, the results of calls to parallel methods must be placed on local variables. Before reading such variables, the developer must insert calls to *sync*, which ensures that such results are available before they are accessed. Step 3 (see Figure 3) automatically reproduces this task by analyzing the bytecode generated at step 2 by using an algorithm 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 to carry out the analysis as close to the source code of the application as possible. This mapping is possible since there is a direct correspondence between Java source and bytecode [3].

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 containing local variable declarations, method calls, goto-like directives to jump to other labels, etc. The relationships between labels define the control flow of a method. For example, the source code of Figure 4 (left) is compiled into seven labels (center), which in turn give origin to a *block tree* (right) comprising three nodes, namely the whole method, the loop construct and the conditional branch inside the loop. The root of a block tree always corresponds to the body of a method, whereas the rest of the nodes exclusively depend on the structure of the sentences within this method.



```
boolean check(int a,
              int[] b) {
  int i = 0;
  while (i < b.length) {
   if (b[i] > a)
    return true;
  }
  ...
  return false;
}
```

```
public check(I[I)Z
 L0:  ICONST_0
      ISTORE 3
 L1:  GOTO L2
 L3:  ALOAD 2
      ILOAD 3
      IALOAD
      ILOAD 1
      IF_ICMPLE L4
 L5:  ICONST_1
      IRETURN
 L4:  IINC 3 1
 L2:  ILOAD 3
      ALOAD 2
      ARRAYLENGTH
      IF_ICMPLT L3
 L6:  ICONST_0
      IRETURN
 L7:
```

Fig. 4. From Java bytecode to block trees

To derive the block tree of a method, the Satin connector analyzes its bytecode instructions sequentially to find those that provide information about control sentences. Specifically, these instructions are the ones that lead to jumps within a method (e.g. IFEQ, IFLT, GOTO, IFNULL, etc). As these instructions are spotted, the corresponding block tree is built. Each block has a reference to every single bytecode instruction it contains and a pointer to every block representing immediately inner scopes. This high-level view of the bytecode is then used by the algorithm for inserting synchronization barriers.

The algorithm works by walking through the instructions of a method to detect 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, and is used when its value is read. To work properly, Satin methods can read

```
procedure IDENTIFYSYNCPOINTS(instrList)        ▷ Bytecode instructions of the input method
    tree ← DERIVEBLOCKTREE(instrList)
    syncPoints ← CREATEEMPTYLIST        ▷ List of resulting synchronization points
    for $i \leftarrow 1$, LENGTH(instrList) do
        if varCode ← ISSPAWNABLEVAR(instrList[i]) then
            currentBlock ← GETBLOCK(tree,instrList[i])
            if BEINGUSED(varCode,instrList[i]) = true then
                if GETFIRSTSTATE(varCode,currentBlock) = $UNSAFE$ then
                    SYNCVARSINBLOCK(currentBlock)
                    ADDELEMENT(syncPoints, instrList[i])
                end if
            else if BEINGDEFINED(varCode,instrList[i]) then
                DESYNCVARUPTOROOT(varCode, currentBlock)
            end if
        end if
    end for
    return syncPoints
end procedure
```

Algorithm 1: Identifying synchronization points

such variables provided a *sync* has been previously issued. Our algorithm thus modifies the bytecode so as to ensure a call to *sync* is done between the definition and use of a local variable, for any execution path between these two points. Moreover, as *sync* suspends the execution of the method until *all* subcomputations associated to defined variables have finished, the algorithm heuristically minimizes the calls to *sync* while keeping the correctness of the method. Algorithm 1 illustrates the barrier insertion algorithm. Its helper functions are listed in Table 2.

The algorithm maintains a map with the spawnable variables per block, and their associated state, i.e. SAFE (up to the current instruction the variable is safe to use; a barrier is not needed) or UNSAFE (a barrier from where the variable is defined is potentially needed). The algorithm takes into account the scope at which spawnable variables are defined and used, by computing the state of each variable according to the state it has within the (scope) node of the tree where the variable is read and its state within the ancestors of that node. Once modified to include barriers at the spotted points (*syncPoints*) and processed with the Satin compiler, the input bytecode is ready for execution, i.e. inserting calls to *sync* at these points guarantees the operational semantics of Satin. Nevertheless, the algorithm minimizes the inserted barriers to gain efficiency by further reducing *syncPoints*. For example, moving such barriers out of loops results in less calls to *sync*.

Signature	**Purpose**
deriveBlockTree (instrList)	Builds the block tree from the instructions list *instrList*.
isSpawnableVar (anInstr)	Checks whether *anInstr* references a spawnable variable, and returns the variable code within the method. Local variables are identified in Java bytecode as *$i*, where *i* represents the index of the variable within the method.
getBlock (anInstr)	Returns the block from the tree where *anInstr* belongs. Instructions belong to one block only; if B_P has a child B_c, an instruction of B_c does not belong to B_P.
beingDefined (varCode, anInstr)	Checks whether the *varCode* variable is being assigned a spawnable call. Assigning the result of a spawnable call to a local variable forms a recognizable bytecode pattern. The function analyzes whether the pattern occurs by also considering the subsequent instructions. Analogously, *beingUsed* checks whether such a variable is read.
getFirstState (varCode, block)	Traverses the block tree starting from *block* upwards looking for the occurrence of a variable *varCode* in the variable maps of these blocks. The function returns the state it has in the block it was first encountered.
syncVarsInBlock (block)	Sets to SAFE the state of all spawnable variables in *block* (up to the current analyzed instruction) as well as the ancestors of *block*. The resulting pairs [varCode,SAFE] are put into the map of *block* only.
desyncVarUpToRoot (varCode, block)	Sets the state of a specific variable to UNSAFE from a given block up to the root block. The variable becomes UNSAFE in *block* and all its ancestors.

Table 2. Helper functions used by the barrier insertion algorithm

5 EVALUATION

We performed experiments to assess the performance benefits and potential overheads associated to using BYG. We compared Satin versus our BYG/Satin connector by running seven classic divide and conquer applications, namely *PF* (prime factorization), *Cov* (the set covering problem), *KS* (the knapsack problem), *FFT* (Fast Fourier transform), *Fib* (Fibonacci), *MM* (Strassen's matrix multiplication) and *Ad* (adaptive numerical integration). We focused on the Satin connector because it is much more sophisticated than the Condor-G connector in terms of the client side bytecode rewriting technique, the bytecode transfer mechanism, and the BYG-supplied runtime that wraps the services of Satin.

PF, *Cov* and *KS* are NP problems, whereas the rest of the applications are benchmarks commonly employed to evaluate Grid frameworks. Due to their diversity in terms of functionality and resource demands, the applications provided the basis for a significant performance evaluation. For the sake of fairness, the codes were obtained from the Satin project [24]). The BYG variants were obtained by removing from the original Satin codes any sentence related to parallelism to derive their sequential counterparts. The applications were executed on a cluster (Section 5.1), and a wide-area Grid (Section 5.2), which was established by using WANem [20], a software for emulating WAN conditions over a LAN.

The gridification model promoted by BYG is basically inspired by previous research in the context of our JGRIM project [15], an approach for Grid-enabling component-based applications from their source code. Like JGRIM, BYG assumes that input applications comprise one or more components heavily decoupled from each other. This is precisely what allows BYG to submit a subset of the components for execution to a Grid.

Unlike the codes employed for evaluation, most real-life applications comprise several components with complex dependency graphs. Basically, the more coupling among components, the more overhead upon the gridification of an individual component, since it is necessary to deploy the executable code of the component being gridified as well as that of its related components. Moreover, since Satin does not support automatic code deployment, using test applications with complex dependency graphs when comparing against BYG – which does support such mechanism – would have been resulted in an unfair evaluation.

In this way, the above test applications allowed us to better measure the performance resulted from applying our automatic parallelization techniques versus the one obtained via manual gridification. Besides, we also provide an assessment of the overhead necessary to submit the software artifacts associated to individual components to a Grid. We have nevertheless recently investigated the effects of gridifying real-life component-based codes with more complex dependency graphs in the performance and network usage of gridified applications [26], which resulted in competitive values for our techniques and hence provided positive evidence regarding their applicability for such kind of codes.

5.1 Experiments on a Cluster

We used a cluster of 15 single core hosts with 3 MHz CPUs and 1.5 GB of RAM running Mandriva Linux 2009.0, Java 5 and Satin 2.1, and a 100 Mbps LAN. We chose application parameters to produce moderately long-running executions. Figure 5 a) shows the average execution time for 25 runs of these applications. The bars corresponding to BYG also include the time required to initiate our extended Satin runtime. Figure 5 b) compares the time spent by BYG applications executing under Satin versus the time it took to run the test applications natively with Satin. Standard deviations were below 5 %, which is an acceptable noise level considering that Satin – and therefore our Satin connector – relies on a *random* task scheduler [24]. Except for *FFT*, BYG performed competitively, even when BYG is based on automatic parallelism and adds a software layer on top of Satin.

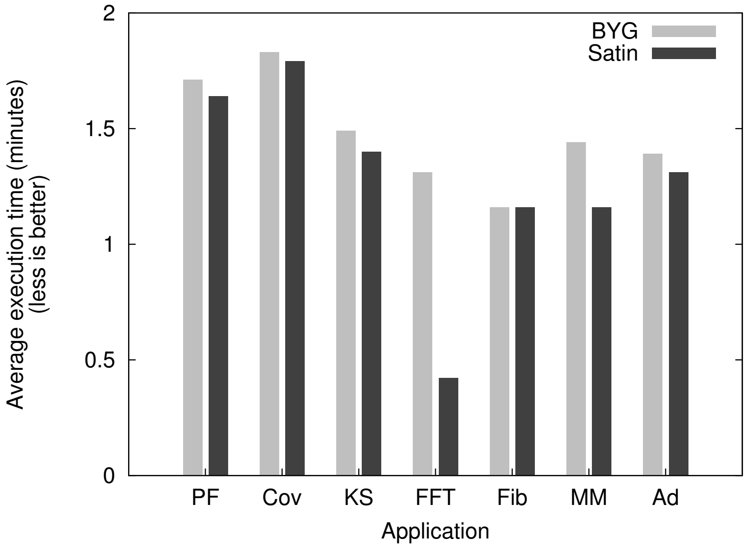
The Grid-unaware version of the applications were implemented as a main class that invoked the actual computing intensive component. For *FFT*, this class passed as an argument to this latter a large array. Then, sending the computation for execution to our Satin server required to send this data as well. In contrast, in Satin *FFT*, this bootstrap invocation was far cheaper as it is performed locally. Broadly, the cause of this problem is that distributing application components across Grid hosts can lead to costly component interactions. To mitigate this problem, BYG could offer to developers a rule-based support for dynamically deciding whether it is convenient to gridify an operation or not (e.g. when the size of the arguments is below some threshold).

Figure 5 b) shows that for 3 out of 7 of the applications (*Cov*, *Fib*, *MM*) BYG introduced gains of up to 7 % with respect to Satin. In principle, this may seem confusing since the BYG connector uses Satin as the underlying support for execution. Similar gains were obtained with our synchronization techniques and our extended Satin runtime in an heterogeneous cluster [16] and a real wide-area Grid [15]. This is explained in part by the random nature of the Satin scheduler, but the main reason is that the bytecode interpreted by the Satin runtime in either case is subject to different execution conditions. First, the pure Satin versions of the applications were parallelized by hand, while the BYG counterparts were parallelized via our synchronization heuristic, which may introduce differences in the number of calls to *sync* or the places in which these calls are located in the gridified codes. Second, the execution of a pure Satin application is directly handled via the Satin platform, whereas our Satin connectors submit applications to an extended Satin runtime, i.e. a thin software layer on top of Satin able to execute Grid-aware codes in a client-server fashion. Note that, despite the obtained results, our goal is not to outperform existing Grid middlewares, but simplifying their usage without incurring in excessive performance penalties. This experiment shows that BYG facilitates the construction of Satin applications, while delivering competitive performance.

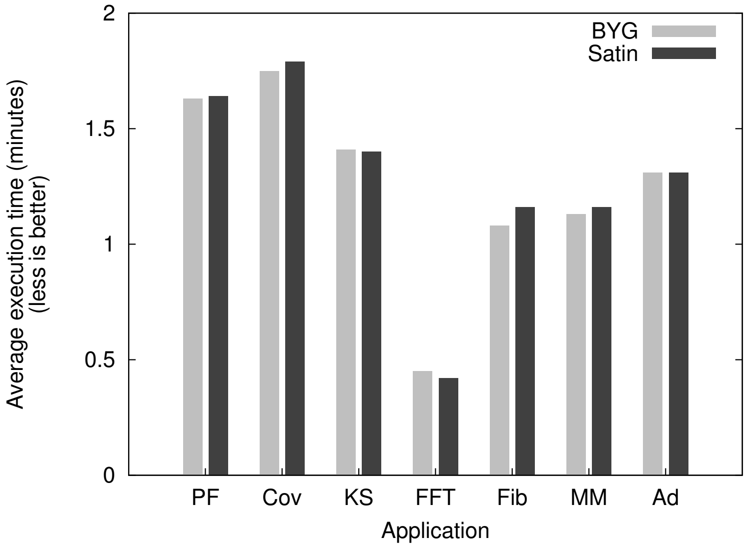
Figure 6 a) shows the average *gridification time* for 25 executions, which includes the times required to

1. instrument the Grid-unaware bytecode to inject middleware bridging instructions and synchronization barriers,
2. process the bytecode resulting from the previous step with the Satin compiler, and
3. build and transfer the executable files to the Grid hosts.

Gridification time was around 2.5 seconds. Note that the time complexity of the algorithm for inserting synchronization is not $O(1)$. Nevertheless, Figure 6 a) shows that the time (1) remains almost constant, which proves that the performance of the bytecode instrumentation techniques of BYG was not severely affected by the control structure of the test applications. The time (2) appears to be slightly more



a)



b)

Fig. 5. Test applications: performance results (cluster): a) Overall execution time, b) Execution time within the Satin runtime

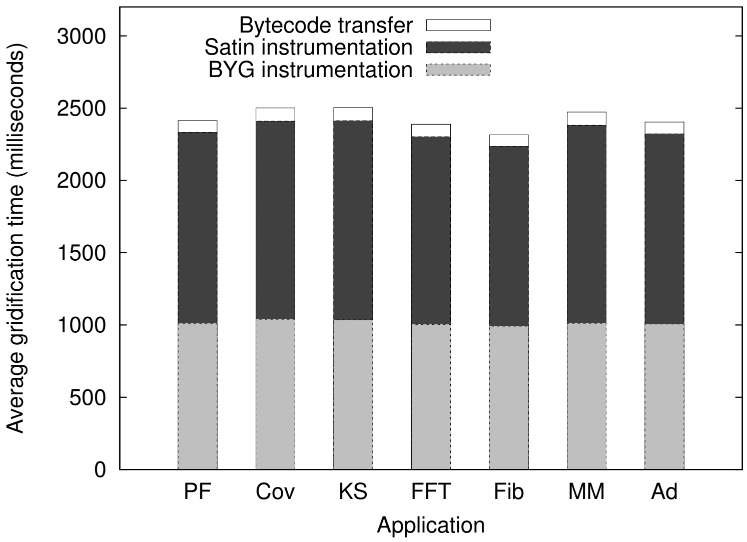
affected by the bytecode size of the gridified methods, since the Satin compiler performs an analysis over the entire declaring class. With native Satin applications, this overhead is not present since (3) is performed not at runtime but during deployment. However, the programmer must manually build its application with Satin. Finally, since the experiments involved a cluster, the time (3) was negligible. Again, this overhead is not present in Satin, as it does not support automatic transfer of application classes.

5.2 Experiments on a Wide-Area Grid

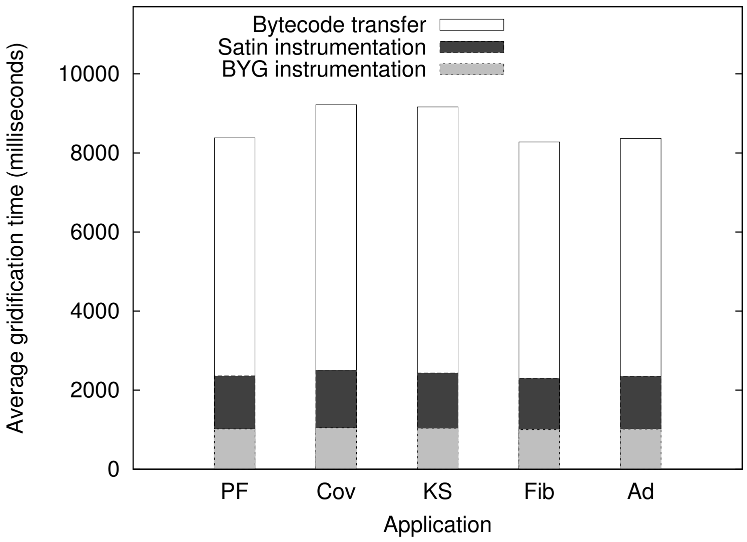
We then simulated a wide-area Grid of 3 clusters comprising 4, 5 and 6 hosts, respectively, and WAN links with a bandwidth of 1.5 Mbps, a latency of 200 ms and a jitter of 10 ms. The BYG variants of the applications were launched from the smallest cluster. All test applications were configured to use the Cluster-aware Random Stealing (CRS) [24] scheduler of Satin. With this strategy, when a host becomes idle, it attempts to steal an unfinished task from remote or local hosts, but intra-cluster steals have a greater priority than inter-cluster steals, which saves bandwidth and minimizes latency.

Figure 7 a) depicts the average execution time for 40 runs of the applications. The computation to network usage ratio of *FFT* and *MM* in this setting was extremely small and thus harmed CRS. Specifically, for *FFT* and *MM*, the average amount of successful steals over the amount of total steal attempts was below 1 %, whereas for the rest of the applications this percentage was in the acceptable range of 20–25 %. In consequence, we decided to left *FFT* and *MM* out of the analysis. Figure 7 b) shows the time spent by BYG applications executing under Satin versus the time required to run the pure Satin variants. Standard deviations were around 12 %, which was due to the randomness of the Satin scheduler, plus the fact that we introduced jitter.

For the BYG applications, we obtained two variants by disabling/enabling *caching* (see Figure 7 a)). When enabled, caching allows Grid hosts to maintain a local copy of a gridified application by preventing BYG from rewriting and transferring the application every time it is run, which avoids blindly distributing bytecode through WAN links without checking whether the application actually changed at the client side. Without caching, BYG added for 4 of the 5 test applications a performance overhead in terms of execution time of 2–6 %, whereas for *Ad* it introduced a performance gain of 2 %. This is acceptable, considering both the advantages and the administrative costs inherent to supporting automatic instrumentation and deployment of bytecodes. Figure 6 b) illustrates the average gridification time for 40 runs. On the other hand, with caching, BYG performed better than Satin for all applications, experiencing average performance gains of up to 10 % and 5.8 %. This is very encouraging as well, since it implies that transparently exploiting the Satin scheduler and supporting automatic deployment of binary Java codes when gridifying applications does not lead to a performance decrease. All in all, these results show that the BYG applications also performed well in the WAN setting,

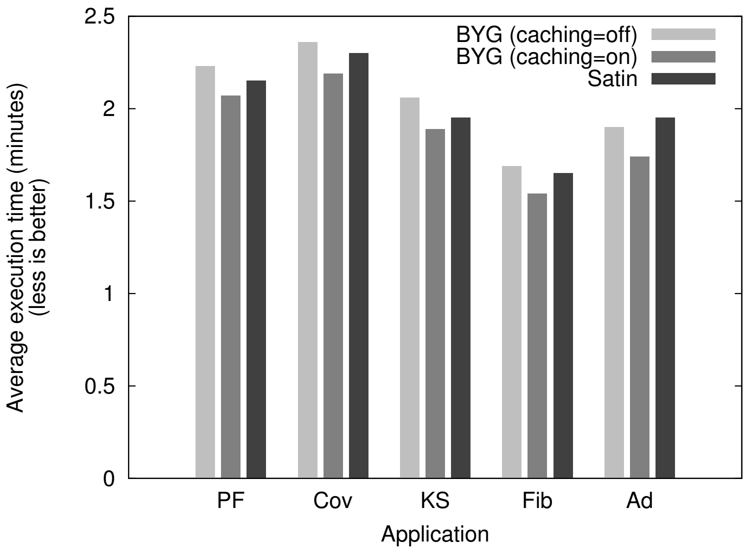


a)

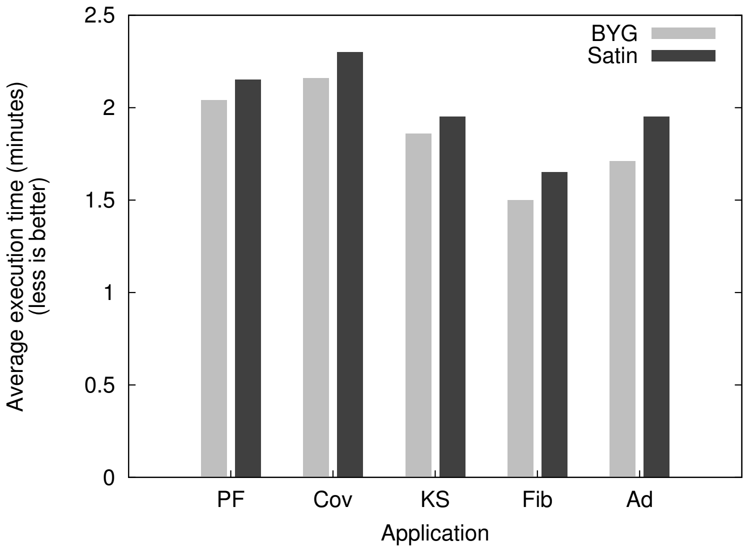


b)

Fig. 6. Test applications: bytecode instrumentation and transfer: a) Cluster, b) Wide-area Grid



a)



b)

Fig. 7. Test applications: performance results (wide-area Grid): a) Overall execution time, b) Execution time within the Satin runtime

and are consistent with the goal of our synchronization heuristic, which is to mimic a human parallel programmer.

6 CONCLUSIONS AND FUTURE WORK

We have presented BYG, an approach to simplify the execution of compiled Grid-unaware applications on Grids, which lets developers to selectively gridify the components of existing applications. We have materialized BYG as a tool for gridifying component-based Java applications, thus we reasonably expect it will be a benefit for a large number of users.

Experiments show that using BYG does not imply resigning performance. We evaluated BYG by running several computing intensive codes through Satin connectors and pure Satin in a cluster and a wide-area Grid. In the cluster, BYG performed competitively, whereas in the latter setting BYG introduced significant performance improvements. These are interesting results considering that the only tasks necessary to gridify an application is to supply some configuration, which reduces gridification effort. Although the evaluation conceived Satin and BYG as competitors, both tools are complementary: BYG promotes separation between application logic and the external Grid services used for execution. Then, BYG is a binary code gridifier rather than a Grid platform *per se*.

At present, we are building a connector for ProActive [1] and eventually providing integration with state-of-the-art Grid schedulers such as [27]. Second, we are incorporating a rule-based support for specifying whether to gridify Grid-unaware components or execute them unmodified instead, so as to consider heuristics for dynamically computing the potential gains of gridification, which could be fed for example with user-supplied performance models. As stated earlier, gridifying applications including components with a high degree of interdependency such as workflows with BYG may be counterproductive. Particularly, workflows play an essential role in modern distributed infrastructures like service-oriented Grids and Clouds. As a starting point, we will study the provision of such optimization support in the context of workflow applications. Third, we are integrating BYG with the GMAC [9] Java-based P2P protocol to allow applications to discover required execution services rather than relying on static entry point information. Finally, we will experiment with real-world applications and more Grid topologies to further validate BYG. We are gridifying a ray tracing application and a DNA sequence alignment code on a high-speed wide-area Grid, which is a result of a country-wide Grid initiative of the Argentinian government to connect academic clusters.

Acknowledgements

We thank the anonymous reviewers for their helpful comments to improve the quality of the paper. We also thank Cristian Clasadonte for his valuable help managing the computing infrastructure required for conducting the experiments. We acknowledge the financial support provided by ANPCyT through grants PAE-PICT 2007-02311 and PAE-PICT 2007-02312.

REFERENCES

[1] Baduel, L.—Baude, F.—Caromel, D.—Contes, A.—Huet, F.—Morel, M.—Quilici, R.: Programming, Composing, Deploying for the Grid. Grid Computing: Software environments and tools, Springer 2006, pp. 2057–229.

[2] Balíis, B.—Bubak, M.—Wegiel, M.: LGF: A Flexible Framework for Exposing Legacy Codes As Services. Futur. Gener. Comp. Syst., Vol. 24, 2008, No. 7, 2008, pp. 711–719.

[3] Cierniak, M.—Li, W.: Optimizing Java Bytecodes. Concurrency Comput.: Pract. Exp., Vol. 9, 1997, No. 6, pp. 427–444.

[4] Delaittre, T.—Kiss, T.—Goyeneche, A.—Terstyanszky, G.—Winter, S.—Kacsuk, P.: GEMLCA: Running Legacy Code Applications As Grid Services. J. Grid Comput., Vol. 3, 2005, No. 1-2, pp. 75–90.

[5] Dichev, K.—Stork, S.—Keller, R.: MPI Support on the Grid. Comput. Inform., Vol. 27, 2008, No. 2, pp. 213–222.

[6] Fernández, E.—Cencerrado, A.—Heymann, E.—Senar, M.: CrossBroker: A Grid Metascheduler for Interactive and Parallel Jobs. Comput. Inform., Vol. 27, 2008, No. 2, pp. 187–197.

[7] Foster, I.—Kesselman, C.: Concepts and Architecture. The Grid 2: Blueprint for a new computing infrastructure, Morgan-Kaufmann Publishers Inc., 2003, pp. 37–63.

[8] Gannon, D.—Krishnan, S.—Fang, L.—Kandaswamy, G.—Simmhan, Y.—Slominski, A.: On Building Parallel and Grid Applications: Component Technology and Distributed Services. Cluster Comput., Vol. 8, 2005, No. 4, pp. 271–277.

[9] Gotthelf, P.—Zunino, A.—Mateos, C.—Campo, M.: GMAC: An Overlay Multicast Network for Mobile Agent Platforms. J. Parallel Distrib. Comput., Vol. 68, 2008, No. 8, pp. 1081–1096.

[10] Grimshaw, A.—Morgan, M.—Merrill, D.—Kishimoto, H.—Savva, A.—Snelling, D.—Smith, C.—Berry, D.: An Open Grid Services Architecture primer. Computer, Vol. 42, 2009, No. 2, pp. 27–34.

[11] Laskowski, E.—Tudruj, M.—Olejnik, R.—Toursel, B.: Byte-Code Scheduling of Java Programs with Branches for Desktop Grid. Futur. Gener. Comp. Syst., Vol. 23, 2007, No. 8, pp. 977–982.

[12] Lee, E.: The Problem With Threads. Computer, Vol. 39, 2006, No. 5, pp. 33–42.

[13] Manolescu, D.—Beckman, B.—Livshits, B.: Volta: Developing Distributed Applications by Recompiling. IEEE Softw., Vol. 25, 2008, No. 5, pp. 53–59.

[14] Mateos, C.—Zunino, A.—Campo, M.: A Survey on Approaches to Gridification. Softw. Pract. Exp., Vol. 38, 2008, No. 5, pp. 523–556.

[15] Mateos, C.—Zunino, A.—Campo, M.: Grid-Enabling Applications with JGRIM. Int. J. Grid High Perform. Comput., Vol. 1, 2009, No. 3, pp. 52–72.

[16] Mateos, C.—Zunino, A.—Campo, M.—Trachsel, R.: BYG: An Approach to Just-Intime Gridification of Conventional Java Applications. Parallel programming, models and applications in Grid and P2P systems, IOS Press, 2009, pp. 232–260.

[17] McGough, S.—Lee, W.—Das, S.: A Standards Based Approach to Enabling Legacy Applications on the Grid. Futur. Gener. Comp. Syst., Vol. 24, 2008, No. 7, pp. 731–743.

[18] Nakada, H.: Condor-G Java API. Available on: `http://code.google.com/p/condor-java-api`.

[19] ObjectWeb Consortium: ASM. Available on: `http://asm.objectweb.org`.

[20] TATA Consultancy Services: WANem. Available on: `http://wanem.sourceforge.net`.

[21] Thain, D.—Tannenbaum, T.—Livny, M.: Distributed Computing in Practice: The Condor Experience. Concurrency Comput.: Pract. Exp., Vol. 17, 2005, No. 2-4, pp. 323–356.

[22] Tilevich, E.—Smaragdakis, Y.: J-Orchestra: Enhancing Java Programs with Distribution Capabilities. ACM Trans. Softw. Eng. Methodol., Vol. 19, 2009, No. 1, pp. 1–40.

[23] Walnes, J.: The XStream Project. Available on: `http://xstream.codehaus.org/index.html`.

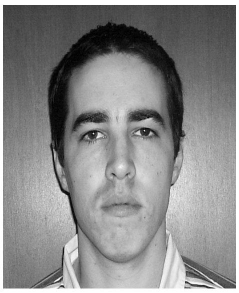
[24] Wrzesinska, G.—van Nieuwport, R.—Maassen, J.—Kielmann, T.—Bal, H.: Faulttolerant Scheduling of Fine-Grained Tasks in Grid Environments. Int. J. High Perform. Comput. Appl., Vol. 20, 2006, No. 1, pp. 103–114.

[25] Badia, R.—Labarta, J.—Sirvent, R.—Pérez, J.—Cela, J.—Grima, R.: Programming Grid Applications with GRID Superscalar. J. Grid Comput., Vol. 1, 2003, No. 2, pp. 151–170.

[26] Mateos, C.—Zunino, A.—Campo, M.: On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. Futur. Gener. Comp. Syst., Vol. 26, 2010, No. 6, pp. 797–819.

[27] Xhafa, F.—Carretero, J.—Dorronsoro, B.—Alba, E.: A Tabu Search Algorithm for Scheduling Independent Jobs in Computational Grids. Comput. Inform., Vol. 28, 2009, No. 2, pp. 237–250.

[28] Wroblewski, P.—Boryczko, K.: Parallel Simulation of a Fluid Flow by Means of the SPH Method: OpenMP vs. MPI Comparison. Comput. Inform., Vol. 28, 2009, No. 1, pp. 139–150.



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 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 50 papers in journals and conferences.

Ramiro Trachsel received a B. Sc. in Systems Engineering from UNICEN in 2009. 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.

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 80 papers published in conferences and journals about software engineering topics.