Just-in-Time Gridification of Compiled Java Applications

Cristian Mateos[12], Alejandro Zunino[12], Marcelo Campo[12], and Nicolás Gomez[1]

[1] ISISTAN - UNICEN. Campus Universitario, Tandil (7000), Buenos Aires, Argentina
[2] CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas

Abstract. Grids allow users to build applications with enormous demands for computational resources. Intuitively, this involves more development effort and expertise on Grid programming. The so-called *gridification tools* address these problems by deriving the Grid-enabled version of an application from its binary code. Unfortunately, most of such tools produce coarse-grained Grid applications that prevent developers from employing common tuning mechanisms such as parallelism and distribution, and are inflexible, since they were not thought to reuse existing Grid middleware services. We propose a new gridification tool for binary Java applications that solves these issues called BYG (BYtecode Gridifier). Results suggest that BYG can be employed to easily gridify and efficiently execute a broad range of computing intensive applications.

1 Introduction

Grid Computing is based on arranging dispersed computational resources to run computing intensive applications [1]. Grid applications solve problems that require by nature huge amounts of computational resources. The first Grids were basically LAN and MAN networks composed of supercomputers. With the inception of Internet standards, Internet Grids became a reality, and eventually, the first Grid middlewares appeared. The goal of Grid middlewares is to virtualize the various resources of a Grid through *services* (e.g. job scheduling, load balancing, data movement, etc.), and to supply developers with rich APIs for using these services.

Recent research in Grid middlewares has put emphasis on the *consumability* of the delivered services by looking for better programing tools and frameworks to simplify the consumption of Grid services from within user applications [2]. These efforts are grouped into programming toolkits and gridification tools (GTs) [2]. The former provide APIs that abstract away the details to interact with Grid middleware services. Grid programming is done at a higher level of abstraction, so that less effort and time are required compared to directly using middleware APIs. However, as Grid toolkits are in essence programming facilities, they assume that developers have knowledge both on Grid programming and the toolkit being used.

Alternatively, GTs seek to “inject” Grid services into existing applications without much effort from the developer. GTs focus on transforming existing codes to run on a Grid rather than programming them to do so. GTs accept as input either the source code of ordinary applications, their compiled versions or both. The first approach

gives developers more control over the internal structure of their applications, thus very efficient Grid applications can be built. Nevertheless, the second approach allows gridifying applications by simply executing them "as is" on a Grid platform. Finally, the third approach are methods in which gridification happens at both the source code and compiled code levels. Programmers are supplied with markers to annotate their source code, which are processed upon compilation to introduce the Grid behavior.

One major problem of the current binary code gridifiers is that they usually prevent the usage of tuning mechanisms suitable for exploiting Grids. After gridification, applications are essentially coarse grained Grid-enabled codes that cannot be altered to make better use of Grid resources, such as distributing or parallelizing individual parts of an application. This represents a trade-off between ease of gridification versus flexibility to configure the runtime aspects of gridified applications [2]. Therefore, there is a need for GTs that provide a convenient balance between gridification effort and the *gridification granularity*, this is, the size of the execution units of a gridified application at runtime [2] and thus the levels at which their components can be tuned.

We propose a new GT for Java called BYG (BYtecode Gridifier) [3], which solves these issues. Users indicate, via a configuration file, the components of their applications that are subject to execution on Grid middlewares. BYG is not a Grid job submission system *per se*, but offers a glue between conventional applications and the execution services of platforms such as Condor-G [4] instead. BYG instruments ordinary bytecodes to be compliant with the application anatomy prescribed by the target middleware. In this paper, we complement the preliminary experiments reported in [3] by evaluating with BYG both in a local and a wide-area Grid. Results show that BYG enables for easy, fine-grained gridification and effectively leverages existing Grid middlewares without incurring in much performance overheads compared to manually employing these middlewares.

The next section discusses related works, and how BYG improves over them. Section 3 overviews the BYG approach. Section 4 explains how to use of BYG from a practical perspective. Section 5 describes the bindings of BYG to the Satin [5] Grid middleware. Section 6 evaluates BYG. Section 7 concludes the paper.

2 Background

To date, several approaches for gridifying software have been proposed. We will focus on those aimed at gridifying binary codes and/or not demanding much source code modifications from the developer. Table 1 summarizes these efforts. See [2] for a wider discussion on GTs. In general, approaches that allow users to submit their compiled codes "as is" for execution on a Grid are based on machine-dependent binary codes, but there are tools that operate on the binary output of languages like Java and .NET. Column 3 details the granularities of the execution units of Grid-enabled applications [2]. Coarse, medium and fine granularities means that these units are derived from the entire application code, some application components, or some component operations. Most of the approaches rely on a fixed granularity. Finally, column 4 indicates whether each tool provides integration with existing Grid platforms.

Tool	Binary code flavor	Gridification granularity	Exploitation of other Grid platforms
GEMLCA [6]	Machine-dependent	Coarse	Partially (only Globus)
GridSAM [7]	Machine-dependent	Coarse	Yes
DG-ADAJ [8]	Machine-independent (bytecode)	Fine	No (custom scheduler)
LGF [9]	Machine-dependent	Fine, coarse	No (custom scheduler)
ProActive [10]	Machine-independent (bytecode)	Medium	Yes
Satin [5]	Machine-independent (bytecode)	Fine	Partially (only Globus)
Volta [11]	Machine-independent (CIL)	Fine	No (custom scheduler)
XCAT [12]	Machine-dependent	Coarse	Yes

Table 1: Approaches for gridifying applications (based on binary codes and hybrid)

GEMLCA [6] lets users to deploy a legacy program as an OGSA-compliant service. Produced services are executed via Globus GRAM. The user specifies metadata (parameters, executable, etc.) and resource requirements for his application in a configuration file. GEMLCA relies on a very nongranular execution scheme for these services (i.e. running their binary code on several processors) but no internal changes are made in the gridified applications. GridSAM [7] can be used to publish legacy applications as Web Services. These services can be composed via a workflow document that is 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 Grid scheduler, but instead acts as a common interface to existing Grid job schedulers.

XCAT [12] supports distributed execution of component-based applications on top of existing Grid platforms, linking components to platform-level execution services. Application components can also represent legacy binary programs. Complex applications are built by programmatically assembling service components and legacy components. However, this task requires coding and knowledge on the XCAT API. As both GridSAM and XCAT treats input legacy codes as black boxes, they share the limitations of GEMLCA with respect to gridification granularity. LGF [9] is a framework for deploying legacy applications as Web Services. Performance of gridified applications can be monitored at the service, legacy application and code region levels. However, LGF is not designed to leverage existing Grid execution services.

With respect to tools that gridify machine-independent binary codes, DG-ADAJ [8] derives graphs from the bytecode of a multi-threaded Java application, which account for data and control dependencies within the application, and all mutually exclusive execution paths from these graphs are executed on a Grid. In opposition, BYG targets single-threaded Java applications. ProActive [10] provides *technical services*, a support to address non-functional Grid concerns (e.g. load balancing) by plugging external configuration to applications at deployment time. ProActive allows users to deploy ordinary compiled Java classes as mobile entities on a Grid without code modification. Upon deployment, users can specify which technical services they want to use for their applications. Sadly, creating computations from a subset of the functionality of a class requires to manually employ the ProActive API within its source code.

In addition, there are other tools that follow a *hybrid* approach to gridification, in which developers have to alter an application to gridify it. Satin [5] is a framework for parallelizing divide and conquer Java applications. The user indicates in the code the points in which a fork (i.e. recursive methods) or a join (i.e. to wait for child computations) take place. Then, Satin instruments the compiled code to transparently execute the parallel tasks on a Grid. Similarly, Volta [11] recompiles executable .NET applications on the basis of declarative developer annotations, inserting primitives to transparently transform applications into their distributed form. Recompilation operates at the CIL (.NET bytecode) level. The weak point of these tools is that they require some modifications to the source code of applications prior to Grid-enabling them.

While the above approaches are targeted at users with little knowledge on Grid technologies, hybrid gridifiers are not true binary code gridifiers, as they require code modifications on the input applications. Furthermore, some of the approaches (GEMLCA, GridSAM, XCAT) offer a poor balance to the "ease of gridification versus fine tuning" trade-off [2], since they completely avoid source code modification, but gridification results in coarse-grained Grid applications that cannot be restructured for reconfiguration or paralellization purposes. Finally, only a small number of the analyzed approaches are designed to exploit external Grid execution services. However, a recent trend in Grid Computing, as evidenced by recent Grid standards such as OGSA and WSRF, is to promote interoperability and therefore integration among Grid technologies [13]. In this sense, Grid middleware integration is becoming the rule and not the exception.

3 The BYtecode Gridifier

We propose BYG (BYtecode Gridifier) [3], a GT that deals with the above trade-off by letting developers to introduce tuning into their applications while minimizing the effort necessary to put a Grid application to work. BYG does not seek to provide yet another runtime system for supporting application execution, but aims at leveraging the execution services of existing Grid platforms through *connectors*. A connector implements the protocol to access the execution services of a specific Grid platform. Connectors are non-invasively injected into the application binary code to delegate the execution of certain parts of the application to a Grid platform. The mapping of which parts are Grid-enabled is specified by means of user-supplied configuration.

BYG specifically targets component-based applications implemented in Java. We chose Java as it is broadly adopted by developers. On the other hand, component-based programming is commonplace in Java development, which is evidenced by the high popularity of component models such as JavaBeans and EJB. For these reasons, BYG could benefit a large amount of today's applications.

Component-based development splits application functionality into logical components with well-defined interfaces. Such components hide their implementation, do not share state, and communicate with other components via message exchange [14]. Components only know each other's interfaces and are self-contained, which yields as a result highly reusable and decoupled building blocks materialized through an interface and implementing classes. Besides, any kind of interaction that involves tightly-coupled

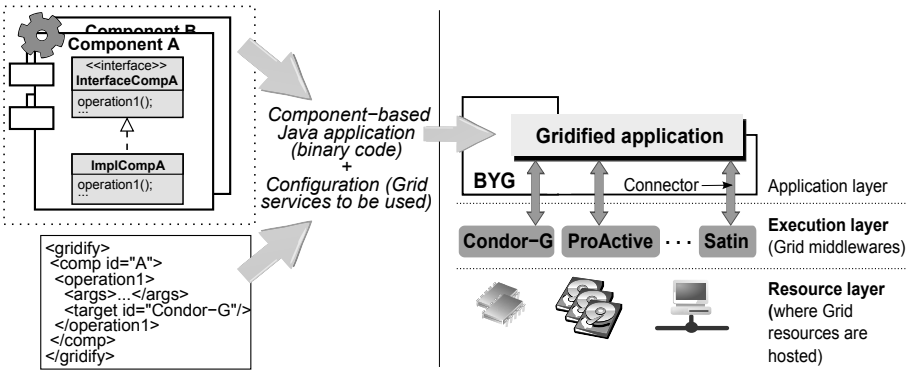


Fig. 1: Architecture of BYG

communication or state sharing is disallowed, such as invoking component operations by passing arguments by reference. This allows any component to be replaced without affecting the rest of the components of the application.

Figure 1 depicts the architecture of BYG. Conceptually, BYG takes as input the bytecode of an ordinary component-based Java application, and dynamically transforms it so as to run some component operations on different Grid middlewares. The developer must indicate through a configuration file which operations should be run on a Grid and which Grid middlewares should be used. Then, BYG processes the developer configuration, intercepts all invocations to this kind of operations (in this case operation1), and delegates their execution to the target middleware (in this case Condor-G) by means of an appropriate connector. All in all, the benefits of this approach are:

Gridification effort Gridification requires less effort. The programmer specifies which parts of its application should be executed on a Grid environment, but without modifying the source code of the application. Besides, the connectors of an application can be easily detached by modifying its configuration.

Flexibility Depending on the nature of a component operation, a different Grid execution service could be used. For example, an embarrassingly parallel operation may be sent to a Grid platform able to execute it in parallel. Section 5 explain this capability in the context of Satin. Similarly, all mission critical computations may be submitted to a platform providing fault tolerance, such as Condor-G.

Fine tuning Unlike related tools, in which applications are mostly executed on 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 to different Grid middlewares. Rather than following the coarse-grained gridification scheme [2] promoted by many existing methods, BYG lets programmers to select which component operations –of potentially different granularities– are gridified.

Service availability The developer is asked to specify which Grid execution service should be used to execute a component operation. However, at the time of executing the application, the service might not be available (e.g. the Condor-G infrastructure

that was supposed to run the application is temporarily down). The developer could then seamlessly configure a connector for another middleware.

When configuring connectors, using or not a specific Grid execution service (e.g. Condor-G, Satin, etc.) is mostly subject to availability factors, this is, whether a Grid running the desired Grid platform is available. Furthermore, the choice of gridifying an individual operation depends on whether the operation is suitable for execution on a Grid. The potential performance gains in gridifying an application are subject to two user design factors, namely the amount of data (i.e. parameters) to be passed on to the gridified operations, and the computational requirements of such operations. In this sense, BYG alleviates the burden of adapting and submitting an ordinary application for execution on a Grid, while these factors must be addressed early by the user.

BYG does not aim to automatically gridify any kind of application or exploit any kind of Grid middleware. Architectonically, BYG provides a tier that mediates between an ordinary 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 exploit any Grid middleware exposing a job submission interface for Java applications.

BYG is designed to gridify *component-based* applications. This ensures that components are heavily decoupled, so that they can be seamlessly run in a different memory address space. Depending on the connectors being used, operations must adhere to certain coding conventions at development time. Nevertheless, following good development practices such as using proper method modularization, placing the result of calls on local variables, and avoiding parameter passing by reference is usually enough to prepare an ordinary application to use various connectors. For instance, these two latter in conjunction allows BYG to automatically spot the points in a component's bytecode representing calls to operations or accesses to their results. This in turn lets BYG to rewrite this bytecode to exploit common Grid middleware features such as asynchrony and parallelism at the operation level. In addition, good method modularization allows for tuning of components at different granularities.

There are two types of component-based applications for which BYG has limitations. Applications in which components have a high degree of interdependency, such as workflow applications, may not be suitable for BYG. Particularly, workflow applications are a collection of tasks (components) with transitions between them. The more the number of transitions, the more the communication between components at runtime. Then, remotely executing some of these components may increment the communication cost and render gridification counterproductive. Another limitation concerns applications that comprise data components that have a semantic based on their localization (e.g. files) whose execution environment cannot be easily changed. To address this problem, transparent proxies [10] to the data components can be used.

We have developed a proof-of-concept implementation of BYG, which works by instrumenting bytecodes to delegate the execution of certain component operations to external Grid execution services. BYG-enabling an application only requires the user to (a) configure an XML file that instructs BYG how to map component operations to such services, and (b) add a JVM argument to the command that initiates the application. At present, BYG supplies a connector for accessing the services of Satin [5]. Furthermore,

the development of connectors for Condor-G and ProActive is underway. This will enable developers to take advantage of features not present in Satin such as monitoring of running computations. It is worth noting that BYG is strongly inspired by the JGRIM project [15], a tool for Grid-enabling Java source code.

4 Putting the BYtecode Gridifier to work

To Grid-enable a conventional application with BYG, users must create a configuration file, which specifies the components to be gridified and the binding information that depends on the Grid middleware(s) selected for execution. Both the components and the bindings are application-specific. Consequently, the user has to know the machine that plays the role of job executor (or *entry point*) of each middleware. Specifically, to gridify an application with BYG, the user must provide:

- The list of operations or Java methods (owner class and signature) to be gridified.
- The connector to be used (and consequently the Grid execution service). Connectors are provided to developers by means of different class libraries.
- The desired job submission protocol. For example, Condor-G provides a socket-based job submission mechanism and a Web Service submission interface.

```
<configuration>
  <connector name="example">
    <middleware name="satin">
      <property name="protocol">raw_sockets</property>
      <property name="address">satin_ip:satin_port</property>
    </middleware>
    <operations>
      <class name="Fibonacci">
         <method name="fib"><param type="long"/></method>
      </class>
    </operations>
  </connector>
</configuration>
```

For example, the above configuration gridifies a method from the Fibonacci class through Satin. To inject connector code, BYG relies on the support for agents provided by Java. An agent is a pluggable user Java library that customizes the class loading process, for example, by performing bytecode transformations. When a gridified application starts, the BYG agent extracts from the configuration the methods to gridify and the connectors to use, and instruments the bytecodes of these methods as their owner classes load by using ASM [16]. Instrumenting a method implies:

1. Rewriting its body to include the instructions (or "stub") for launching it on a Grid. The stub uses the corresponding entry point information to transparently send the adapted version of the method for execution via the configured connector.
2. Adapting its bytecode and the structure of its owner class to the bytecode anatomy prescribed by the target Grid middleware (e.g. Satin). For example, some platforms require applications to extend or to implement specific API classes, use certain API calls to carry out distribution and parallelism, and so on.

Step 2 strongly depends on the middleware selected for execution. Middlewares relying on a coarse-grained execution model (e.g. Condor-G) do not provide mechanisms to express independent computations inside methods. This is, gridifying a method with BYG/Condor-G does not lead to rewrite its bytecode. Moreover, middlewares relying on a finer execution model (e.g. Satin, ProActive) offer programmers APIs to express parallelism. Particularly, Satin provides the sync primitive to block an application until child computations are finished. To transparently exploit such APIs, BYG automatically generates components called "peers" whose bytecode is derived from ordinary components such as Fibonacci and rewritten so they use these synchronization constructs. The next section explains this mechanism in the context of our connector to Satin.

5 The Satin connector

Satin [5] is a Java framework to parallelize divide and conquer applications. Satin provides 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*, and spawn results must be stored in local variables. For example, the Satin version for computing the n^{th} Fibonacci number is:

```
interface IFibMarker extends satin.Spawnable{
  public long fib(long n);
}
class Fibonacci extends satin.SatinObject implements IFibMarker{
  public 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
    super.sync(); // To suspend until f1 and f2 are instantiated
    return f1 + f2;
  }
}
```

After specifying the marker interface and inserting appropriate synchronization calls into the application code, the developer must feed a compiled version of the application to the Satin compiler that translates each invocation to a spawnable method (i.e. a call to fib) into a Satin runtime task. Then, the Satin connector automatically reproduces these tasks from a compiled application that has not been coded to use the Satin API. The connector generates the marker interface based on the BYG configuration of the application, and rewrites the bytecode of the component(s) to extend/implement the necessary Satin API classes/interfaces. In addition, the connector inserts proper calls to sync by deriving a high-level representation from the bytecode and analyzing the points where barriers are needed. To execute the Satin-enabled version of components, BYG relies on an extended Satin runtime. For details on this runtime, see [3, 17].

Besides injecting instructions to transparently execute ordinary methods on Satin (step 1 of Section 4), the Satin connector dynamically adapts the bytecodes of both these methods and their owner classes to be compliant with the Satin application anatomy (step 2 of Section 4). Basically, the second task further divides into:

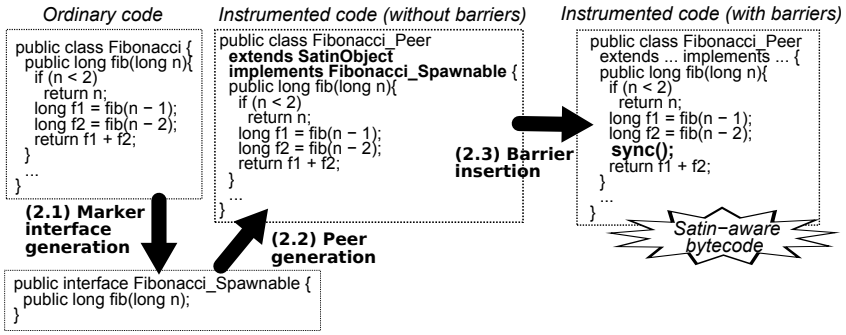


Fig. 2: The Satin connector: Satin-enabling ordinary bytecode

- Marker interface generation (step 2.1): As explained, Satin requires applications to include a marker interface, which lists the methods considered for parallel execution. The Satin connector builds this interface from the method signatures listed in the XML configuration for the class being "satinified".
- Peer generation (step 2.2): Satin applications must implement a marker interface and to extend from SatinObject. A clone (from now on *peer*) of the non-gridified class under consideration is created and instrumented to fulfill these requirements.
- Barrier insertion (step 2.3): Based on an heuristic algorithm, the connector inserts calls to the Satin sync primitive in the spawnable methods of the previous peer.

Figure 2 depicts the steps performed by the Satin connector to build the Satin-enabled version of an ordinary class. The connector builds the corresponding marker interface and a Satin peer from the class being gridified. In a subsequent step, the connector automatically inserts Satin synchronization into the peer by using an heuristic algorithm, which is briefly explained in the next subsection (for more details on it see [3]). At runtime, the final peer is instantiated at the client side by the ordinary application and submitted for execution to the abovementioned extended Satin runtime.

5.1 Barrier insertion

We designed an heuristic that processes the bytecode generated at step 2.2 to insert a minimal number of sync barriers while preserving the semantics of the original code. The dependency analysis is carried out on a high-level representation derived from this bytecode. This mapping relies on the fact that there is a direct correspondence between Java source and bytecode [18].

Java compiles method code as labels containing a number of bytecode instructions. Individual labels form disjoint instruction blocks where local variables are declared, jump to other labels are included, etc. The relationships between the different labels define the control flow of a method. Figure 3 exemplifies these notions. The source code (left) is compiled to six labels (center), which originate a *block tree* (right) comprising three nodes (i.e. the whole method, the loop and the conditional branch). To derive the block tree of a method, the connector analyzes its bytecode instructions

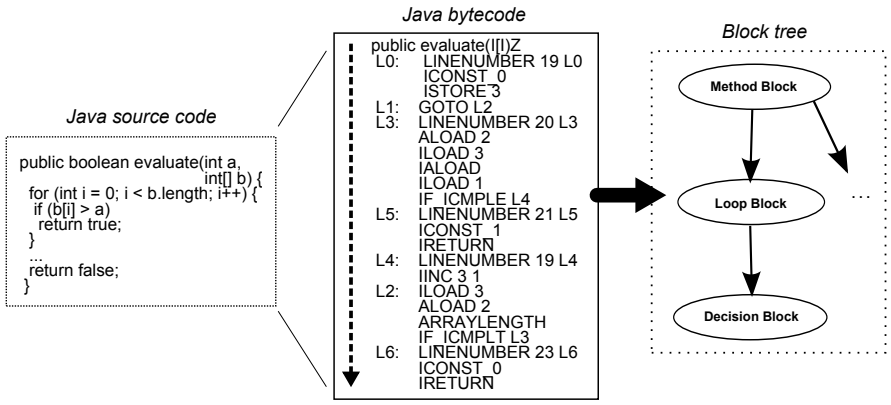


Fig. 3: Deriving a block tree from bytecode

sequentially to gather information about the higher-level control sentences. In Java, the useful instructions are those that lead to jumps within a method (e.g. IF_ICMPLE, IF_ICMPLT, GOTO, etc.). As these instructions are spotted, the corresponding blocks are built so that each block has a reference to every single bytecode instruction it contains, and a pointer to every block representing immediately inner scopes.

The algorithm for inserting barriers works by iterating the instructions of a method and detecting the points in which a local variable is either *defined* or *used* by a sentence (see Algorithm 1). A variable is defined when the result of a spawned computation is assigned to it, whereas it is used when its value is read. To work properly Satin methods can read such variables provided a sync has been previously issued. Therefore, our algorithm operates by modifying the bytecode 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. As sync suspends the execution of the method until *all* subcomputations associated to defined variables have finished, our algorithm uses an heuristic to keep the correctness of the program while minimizing the inserted calls to sync.

The algorithm maintains a map of the spawnable variables and their associated state per block. Possible states are SAFE (up to the current instruction the variable is safe to use; a synchronization point is not needed) and UNSAFE (a barrier from where the variable is defined is needed). Moreover, the state of each variable is computed 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. The algorithm is based on several helper functions:

deriveBlockTree(instr) Builds a block tree from the bytecode instructions list *instr*.

isSpawnableVariable(anInstruction) Checks whether *anInstruction* points to a local spawnable variable. In such a case, the variable code within the method is returned.

getContainerBlock(anInstruction) Gets the block from the tree where *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.

Algorithm 1 Identification of synchronization points

```
procedure IDENTIFYSYNCPOINTS(instr)                    ▷ Receives bytecode instructions
    tree ← DERIVEBLOCKTREE(instr)
    syncPoints ← CREATEEMPTYLIST                       ▷ List of synchronization points
    for i ← 1, LENGTH(instr) do
        if varCode ← ISSPAWNABLEVARIABLE(instr[i]) then
            currentBlock ← GETCONTAINERBLOCK(tree,instr[i])
            if BEINGUSED(varCode,instr[i]) = true then
                if GETFIRSTSTATE(varCode,currentBlock) = UNSAFE then
                    SYNCVARIABLESINBLOCK(currentBlock)
                    ADDELEMENT(syncPoints, instr[i])
                end if
            else if BEINGDEFINED(varCode,instr[i]) then
                DESYNCVARIABLEUPTOROOT(varCode, currentBlock)
            end if
        end if
    end for
    return syncPoints
end procedure
```

beingDefined(varCode,anInstruction) Checks whether the *varCode* variable is assigned a spawnable call. In bytecode terms, assigning the result of a spawnable call to a local variable forms a recognizable pattern, potentially starting at *anInstruction*. Analogously, beingUsed checks whether a local variable is read.

getFirstState(varCode,block) Traverses the block tree starting from a given *block* upwards looking for the occurrence of *varCode* in any of the variable mappings of these blocks, and return its state when the variable is first found.

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 only put into the variable map 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 the variable becomes UNSAFE in *block* as well as all its ancestor blocks.

To illustrate the algorithm, let us apply it to the recursive method shown below, which contains one non-spawnable variable and two spawnable variables (varA and varB):

```
1  public String spawnableMethod() { // Block 1
2    ...
3    boolean nonSpawnableVar = (Math.random() > 0.5) ? true:false;
4    String varA = spawnableMethod();
5    if (!nonSpawnableVar) { // Block 1.1
6      String varB = spawnableMethod();
7      if (Math.random() > 0.5) { // Block 1.1.1
8        // A call to sync() should be placed here
9        System.out.println(varB);
10       varA = spawnableMethod();
```

```
11      }
12    }
13    if (nonSpawnableVar) { // Block 1.2
14      // Another call to sync() should be placed here
15      System.out.println(varA);
16    }
17    ...
18  }
```

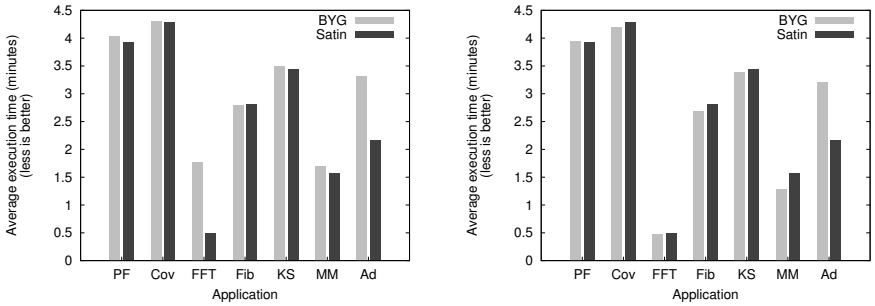
The algorithm iterates the instructions up to line 4, in which varA is defined. Hence, varA becomes UNSAFE in block 1. At line 6, varB is defined within block 1.1, which makes it UNSAFE in blocks 1.1 and its ancestor block 1. At line 9, varB is used within block 1.1.1. Its first occurrence is encountered in the parent of block 1.1.1 as UNSAFE. All spawnable variables in the maps of block 1.1.1 (none) and its ancestors (varA and varB) are set to SAFE in block 1.1.1, and a barrier is scheduled for insertion right before line 9. At line 10 another definition of varA is found, which makes the variable UNSAFE in blocks 1.1.1, 1.1 and 1. At line 15, varA 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 right before line 15. Once the input bytecode has been modified to include barriers at the spotted points (*syncPoints*) and processed with the Satin compiler, the bytecode is ready for execution on Satin.

6 Experimental results

We compared the performance of using Satin versus the BYG/Satin connector by running seven classic divide and conquer applications on a local and a simulated wide-area Grid (the source codes were obtained from the Satin project):

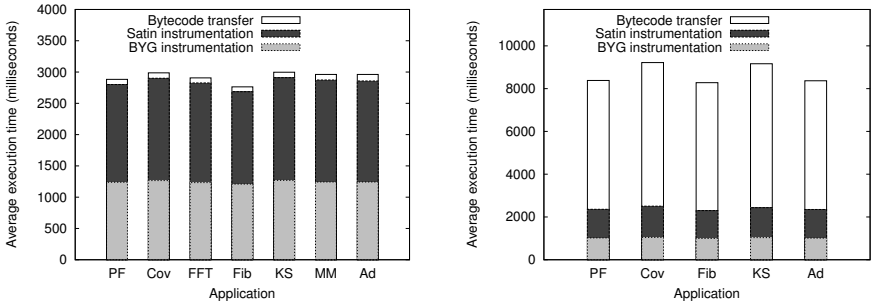
- *PF* (prime factorization): Splits an integer I into its prime factors.
- *Cov* (the set covering problem): Finds a minimal number of subsets from a list of sets L which covers all elements within L.
- *FFT* (Fast Fourier transform): Approximates a continuous function by a sum of sinusoids over a finite number of points sampled over a regular interval.
- *KS* (the knapsack problem): Finds a set of items, each with a weight W and a value V, so that the total value is maximal, while not exceeding a fixed limit.
- *Fib*: Computes the Fibonacci number for a given integer.
- *MM*: Implements the popular Strassen's algorithm for matrix multiplication.
- *Ad* (adaptive numerical integration): Approximates a function $f(x)$ within an interval (a,b) by replacing its curve by a straight line from $(a, f(a))$ to $(b, f(b))$.

First, we set up a local Grid of 8 machines connected through a 100 Mbps LAN and equipped with Java 5 and Satin 2.1 (machines had quite different hardware and software). We chose application parameters to produce moderately long-running computations. All BYG variants were launched from the same machine. Figure 4 (a) depicts the average execution time for 25 executions. Figure 4 (b) compares the portion of the time spent by BYG applications executing under the Satin runtime versus the time



(a) Overall execution time (b) Execution time within the Satin runtime

Fig. 4: Test applications: performance results (local Grid)



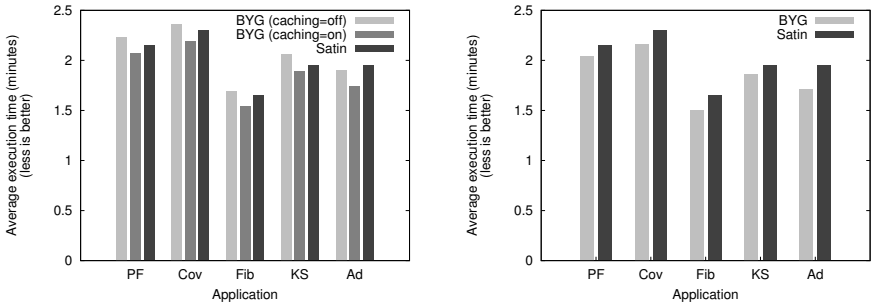
(a) Local Grid (b) Wide-area Grid

Fig. 5: Bytecode instrumentation and transfer

it took to run them with Satin only. Except for *FFT* and *Ad*, BYG did not performed much worse than Satin, even when BYG adds a software layer on top of Satin.

Figure 4 (b) shows that for 5 out of 7 applications BYG introduced gains with respect to Satin. Similar effects were observed with our extended Satin runtime in real wide-area Grids [19]. This may result confusing since the Satin connector uses Satin as the underlying scheduler, but by adding technological noise that intuitively should translate into performance overhead. However, the bytecode interpreted by the Satin runtime in either cases is subject to different execution conditions. When running a pure Satin application, the application itself performs a handshaking process among the machines to request its execution, whereas under BYG the Satin-enabled version of the application being run is sent to an already initiated Satin runtime.

The ordinary applications were implemented as a bootstrap class that invoked the computing intensive component. The bootstrap class of *FFT* passed as an argument to this component a large array, thus sending the computation for execution to our Satin runtime required to send this data as well, which resulted in a significant performance overhead. In Satin *FFT*, the invocation was far more cheaper as it is performed locally. Broadly, the cause of this problem is that physically distributing the components of the



(a) Overall execution time (b) Execution time within the Satin runtime

Fig. 6: Test applications: performance results (wide-area Grid)

application may make the interactions between these components more expensive. To mitigate this problem, we could provide a programmatic support to decide at runtime whether gridifying an operation may be beneficial. For *Ad*, the source of overhead was the time it took to run its Grid-enabled bytecode under Satin (Figure 4 (b)). At present, the bytecode rewriter of BYG is not optimized and thus may cause applications to have more Satin sync primitives than needed, which may harm the performance of applications as the cost of invoking sync is high. Basically, Satin does not consider the case when sync is unintentionally called by a programmer within a program.

Figure 5 (a) shows the average *gridification time* (25 executions), which includes the time required to instrument the ordinary bytecode to inject middleware bridging and synchronization instructions (T_B), to instrument via the Satin rewriter the previous bytecode (T_S), and to transfer the application jar files to the machines (T_T). Interestingly, gridification time was below 3 seconds, and T_B remained almost constant while T_S appeared to be more affected by the bytecode sizes. When building a pure Satin application, T_S is not present as recompilation is performed not at runtime but deployment time. However, programmers must manually build their applications with Satin. Finally, as the experiments were run on a local Grid, the file transfer times were negligible. This overhead is not present in Satin either, as it does not support automatic transfer of application classes.

Furthermore, we set up a wide-area Grid of 3 local clusters C_1, C_2, and C_3 comprising 4, 5 and 6 machines, respectively. The clusters were connected through WAN links simulated by using WANem 2.0 [20]. Each link had a bandwidth of 1.5 Mbps, a latency of 200 ms, and a jitter of 10 ms. The BYG applications were launched from cluster C_1. Both versions of the test applications were configured to use the Cluster-aware Random Stealing (CRS) [5] algorithm provided by Satin, which implements a steal-based task scheduler for wide-area Grids that prioritizes local steals over wide-area steals.

Figure 6 (a) depicts the average execution time (40 executions) of the applications. The computation to communication ratio of *FFT* and *MM* in this setting was very small, which severely harmed CRS. Specifically, for these two applications the percentage of task steals (i.e. the average amount of successful steals over the amount of steal

attempts) was below 1%, whereas for the rest of the applications this percentage was in the acceptable range of 20-25%. We therefore decided to left *FFT* and *MM* out of the analysis. Figure 6 (b) shows the time spent by BYG applications executing under Satin versus the time required to run these applications natively with Satin. In all cases, deviations were around 12%, which is partially due to the random nature of CRS.

For the BYG applications, we obtained two variants by disabling and enabling *caching* (see Figure 6 (a)). Caching allows Grid nodes to maintain a local copy of a gridified application by instructing BYG to avoid rewriting and transferring the application every time it is run. Without caching, BYG added for 4 of the 5 test applications an execution overhead in the range 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. The average gridification time for 40 executions is illustrated in Figure 5 (b). When using caching, BYG performed better than Satin for all applications, experiencing average performance gains of up to 10%. This result is very encouraging, since it implies that transparently exploiting Satin and supporting automatic deployment of binary Java codes when gridifying applications does not lead to losing performance.

7 Conclusions and future work

We presented BYG, a new approach to simplify the execution of conventional applications on Grids. Its goal is to let developers gridify the binary code of existing applications and at the same to select which components of the compiled code should run on a Grid and how. The approach targets component-based Java applications. We can thus reasonably expect the tool will benefit a large number of today's applications.

Experimental results suggest that using BYG does not imply resigning performance. We evaluated BYG by running several computing intensive applications on a local and a wide-area Grid. Most of the BYG versions performed similar to their Satin counterparts. We believe this is an interesting result considering that, to gridify an application, only some configuration have to be provided. However, we will conduct experiments with other applications and other Grid settings. We are currently extending the Satin connector to recognize more high-level Java sentences (e.g. try/catch) and to refine the heuristic for inserting Satin barriers, and implementing connectors for other Grid middlewares. For example, we have developed a prototype connector for Condor-G [4], which is based on a Java interface to Condor-G clusters [21]. We are also integrating BYG with GMAC [22], a P2P protocol that offers Internet multicast services, to allow applications to discover entry point information dinamically. Finally, we are adding a programmatic support to specify rules to decide at runtime whether to run ordinary components via Grid services or execute them unmodified instead.

References

1. Foster, I.: The Grid: Computing without Bounds. Scientific American **288**(4) (2003) 78–85
2. Mateos, C., Zunino, A., Campo, M.: A Survey on Approaches to Gridification. Software: Practice and Experience **38**(5) (2008) 523–556

3. Mateos, C., Zunino, A., Campo, M., Trachsel, R.: BYG: An Approach to Just-in-Time Gridification of Conventional Java Applications. Advances in Parallel Computing. In: Parallel Programming, Models and Applications in Grid and P2P Systems. IOS Press, Amsterdam, The Netherlands (2009)
4. Thain, D., Tannenbaum, T., Livny, M.: Condor and the Grid. In Berman, F., Fox, G., Hey, A., eds.: Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons Inc., New York, NY, USA (2003) 299–335
5. Wrzesinska, G., van Nieuwport, R., Maassen, J., Kielmann, T., Bal, H.: Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments. International Journal of High Performance Computing Applications **20**(1) (2006) 103–114
6. Delaittre, T., Kiss, T., Goyeneche, A., Terstyanszky, G., Winter, S., Kacsuk, P.: GEMLCA: Running Legacy Code Applications as Grid Services. Journal of Grid Computing **3**(1-2) (2005) 75–90
7. McGough, S., Lee, W., Das, S.: A Standards Based Approach to Enabling Legacy Applications on the Grid. Future Generation Computer Systems **24**(7) (2008) 731–743
8. Laskowski, E., Tudruja, M., Olejnik, R., Toursel, B.: Byte-code Scheduling of Java Programs with Branches for Desktop Grid. Future Generation Computer Systems **23**(8) (2007) 977–982
9. Bartosz Baliś, M.B., Wegiel, M.: LGF: A Flexible Framework for Exposing Legacy Codes as Services. Future Generation Computer Systems **24**(7) (2008) 711–719
10. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, Composing, Deploying on the Grid. In: Grid Computing: Software Environments and Tools. Springer, Berlin, Heidelberg, and New York (2006) 205–229
11. Manolescu, D., Beckman, B., Livshits, B.: Volta: Developing distributed applications by recompiling. IEEE Software **25**(5) (2008) 53–59
12. Gannon, D., Krishnan, S., Fang, L., Kandaswamy, G., Simmhan, Y., Slominski, A.: On Building Parallel and Grid Applications: Component Technology and Distributed Services. Cluster Computing **8**(4) (2005) 271–277
13. Grimshaw, A., Morgan, M., Merrill, D., Kishimoto, H., Savva, A., Snelling, D., Smith, C., Berry, D.: An Open Grid Services Architecture Primer. Computer **42**(2) (2009) 27–34
14. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Boston, MA, USA (2002)
15. Mateos, C., Zunino, A., Campo, M.: JGRIM: An Approach for Easy Gridification of Applications. Future Generation Computer Systems **24**(2) (2008) 99–118
16. ObjectWeb Consortium: ASM. http://asm.objectweb.org
17. Mateos, C.: An Approach to Ease the Gridification of Conventional Applications. PhD thesis, UNCPBA, Tandil, Buenos Aires, Argentina (February 2008)
18. Cierniak, M., Li, W.: Optimizing Java Bytecodes. Concurrency: Practice and Experience **9**(6) (1997) 427–444
19. Mateos, C., Zunino, A., Campo, M.: Grid-Enabling Applications with JGRIM. International Journal of Grid and High Performance Computing **1**(3) (2009) 52–72
20. TATA Consultancy Services: WANem. http://wanem.sourceforge.net
21. Nakada, H.: Condor Java API. http://staff.aist.go.jp/hide-nakada/condor_java_api
22. Gotthelf, P., Zunino, A., Mateos, C., Campo, M.: GMAC: An Overlay Multicast Network for Mobile Agent Platforms. Journal of Parallel and Distributed Computing **68**(8) (2008) 1081–1096