ICENI: Optimisation of Component Applications within a Grid Environment

Nathalie Furmento, Anthony Mayer, Stephen McGough,
Steven Newhouse, Tony Field and John Darlington

London e-Science Centre,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ, UK
lesc@ic.ac.uk http://www.lesc.ic.ac.uk/

Abstract

Effective exploitation of Computational Grids can only be achieved when applications are fully integrated with the Grid middleware and the underlying computational resources. Fundamental to this exploitation is information. Information about the structure and behaviour of the application, the capability of the computational and networking resources, and the availability and access to these resources by an individual, a group or an organisation.

In this paper we describe ICENI (Imperial College e-Science Networked Infrastructure), a Grid middleware framework developed within the London e-Science Centre. ICENI is a platform independent framework that uses open and extensible XML derived protocols, within a framework built using Java and Jini, to explore effective application execution upon distributed federated resources. We match a high-level application specification, defined as a network of components, to an optimal combination of the currently available component implementations within our Grid environment, by utilising a system of composite performance modelling. We demonstrate the effectiveness of this architecture through high-level specification and solution of a set of linear equations by automatic and optimal resource and implementation selection.

1 Introduction

Computational Grids, federations of geographically distributed heterogeneous hardware and software resources, are emerging in academia, between national research laboratories and within commercial organisations. Eventually, these computing resources will become ubiquitous and appear transparent to the user, delivering computational power to applications in the same manner as

electrical energy is distributed to appliances through national power grids [1]. The goal of utilising the potential of the Grid is orthogonal to the development of the Grid resources in themselves. One such area of applied Grid computing is *e-science*, the exploitation of advanced information technology for purposes of scientific computation. It is with this goal, the enabling of novel scientific applications via Grid technologies, that we have developed a Grid middleware system, ICENI (Imperial College e-Science Networked Infrastructure). As Grid middleware, ICENI's role is to provide applications with access to the underlying hardware and software resources, in an intelligent fashion which masks the heterogeneous nature of the resources.

In order to support e-science, ICENI provides:

- A **Secure Execution Environment** that is scalable in terms of the number of users and resources.
- **Virtual organisations**, formed through the federation of real resources. Resource owners will only contribute their resources to these federations if they are able to ensure access to their own local community [2].
- **Information** relating to the Grid's resources, the application's performance and behaviour and the user's and resource provider's requirements.
- **Effective resource exploitation** by exploiting information relating to the structure, behaviour and performance of the application, in order to map the application efficiently to the available resources.

ICENI provides this middleware added value in an extensible fashion, utilising common protocols (such as a realisation of the XML markup language) so that it may be used in a complementary fashion with existing tools and applications.

2 Background

In order to develop a system that supports e-science activities, it is necessary to examine the software engineering systems used to develop high performance and scientific computing. In particular the role of abstraction and encapsulation within a component based design paradigm, which we will show leads to an architecture that is eminently suitable to development for the Grid.

2.1 Abstraction within Scientific Computing

When applied computational scientists develop a program to solve a particular problem, they make use of domain-specific knowledge in the construction of the

application [3]. However, once the program is written the abstraction of this knowledge expressed in a high-level language (typically C++ or FORTRAN) is lost and only the low-level machine code remains.

Where the programming language utilises high-level abstractions as programming constructs (such as in object-oriented and functional languages [4,5]), this knowledge may be retained and used in compile-time optimisation. Following compilation, this high-level knowledge is again lost in the translation to low-level high performance executables.

In addition to the end-user's knowledge regarding the application, a developer's knowledge of the performance characteristics and behaviour of a low-level library is also lost once it has been linked into an application. Though dynamically linked libraries are not bound until run-time, there is usually little or no meta-data describing these libraries that would enable intelligent scheduling, especially within the context of the Computational Grid.

Retaining this high-level knowledge (application behaviour, properties, and low-level performance characteristics) is essential to optimising the implementation at run-time [6].

2.2 Component Architectures

Component based design is a well established software engineering pattern. A component is "a unit of independent deployment, with contractually specified interfaces and explicit context dependencies" [7]. Component based design replicates the assembly line process of mechanical engineering within software engineering, by encapsulating software within "black boxes" that enable separate development of an individual component without causing disruption to those components that are dependent upon it.

Encapsulation is provided by a strict separation between the interface and the implementation; the component design pattern requires that any context dependencies are made explicit within the interface. Thus any change to a component implementation that involves changing its dependencies or potential relationships to other components necessitates a change in the interface. All interactions between components occur through methods specified in the interface. Independent software development that does not change the interface can therefore have no effect on dependent components. This feature of the methodology is especially useful for development on the Grid.

The interface also acts as an abstraction of the component, expressing the fundamental aspects of the component's interaction with external services and other components. As such it can represent the component's essential nature

inside a larger programming environment, maintaining any high-level information regarding that component throughout its lifetime. This is another useful property of component based design that we can lever to enable utlisation of a Grid environment.

3 ICENI

To support our Grid related research activities into application composition and resource exploitation, we have designed a set of integrated XML derived protocols and produced a prototype implementation of ICENI, the Imperial College e-Science Networked Infrastructure. ICENI utilises a component based design pattern for applications that are to be deployed upon the Grid, and a system of federated computational communities to support the provision of software, hardware and network resources with heterogenous access policies.

3.1 The role of ICENI

The Grid community, and in particular the Global Grid Forum, have adopted a layered view for the Grid consisting of a Grid Fabric (the hardware, networking and protocols) as the base, enabled by a higher level layer of Grid Services such as those within the Globus toolkit [8]. Above this are application toolkits which utilise the Grid Services directly. ICENI is introduced at a different conceptual layer - it is a middleware framework lying above both the Grid Fabric and Grid Services, but below the level of individual applications. This relationship is indicated in Figure 1.

The motivation of this positioning between the Grid Services and the Application Toolkits is to provide a transparent method of enabling efficient matching between applications and the Grid Fabric. ICENI deploys applications and monitors the Grid by building upon existing services to provide higher-level abstractions within the middleware. We 'add value' to the lower-level Grid Services through a standard system of structured information that allows us to match applications with resources and services, to maximise utilisation of the resource. By encapsulating the applications (in a component based manner) the burden of modification for specific Grid services is shifted from the application designer to the middleware itself.

Central to our Grid architecture is the notion of a public *Computational Community*. This represents a virtual organisation (a federation of individuals and organisations with a common goal) to which real organisations can contribute their resources. These resources may include hardware resources, software re-

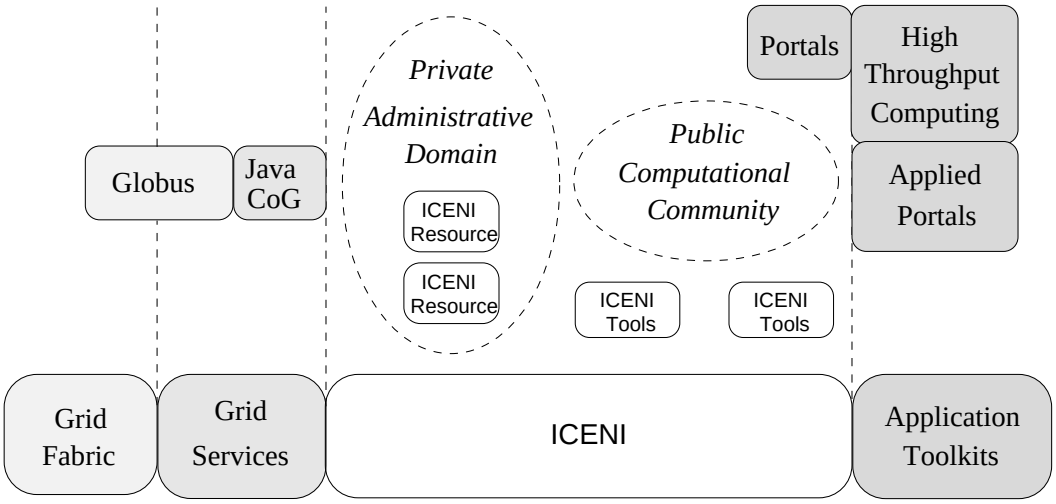

Fig. 1. Overview of the ICENI architecture within a Grid environment

sources, or data resources. These are made available to all users, satisfying access control restrictions specified by the resource provider through the *Computational Community*. Information relating to the resources in the *Computational Community* (such as the current resource load, operating system, access and usage policies, etc.) are accessed through a published API or a GUI by the user.

3.2 ICENI Architecture

ICENI consists of a number of tools and systems, unified by the use of a common mediating language. This is the Component - eXtensible Markup Language, or CXML, which is a realisation of the XML meta-language [9]. This language provides a means by which the diverse units of the framework communicate, and is used to describe the meta-data for the abstract components, the component implementations, resources and applications. All meta-data stored or used within the system is specified in CXML. The information that is used, and the CXML documents used to annotate it, is described in Section 4.

The main systems within ICENI are as follows:

Resource Manager Resources within an organisation are managed through a private *Administrative Domain* by the *Resource Manager*. It allows the local administrator to alter the configuration of the resource meta-information, such as its dynamic and static attributes (discussed in Section 4.5). This information service is critical to resource selection, and updates can be pulled on demand to the services in the *Computational Community*.

Policy Manager The *Policy Manager* tool allows the administrator to de-

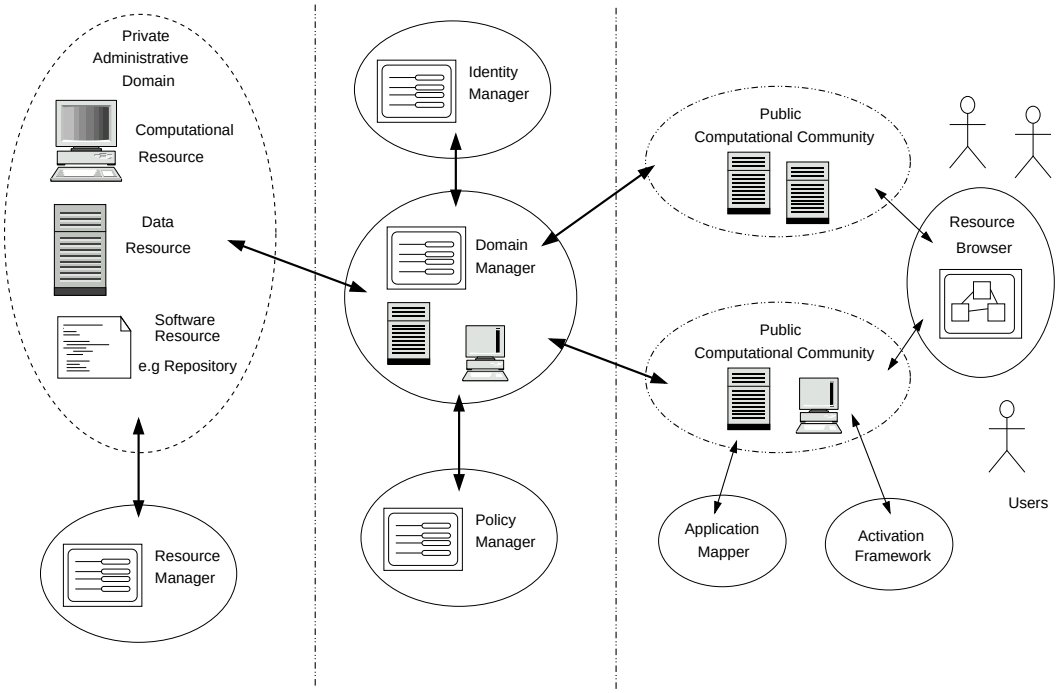

Fig. 2. ICENI Architecture

fine how the resources within the private *Administrative Domain* are published for access by users within the public *Computational Community*. The usage and access control polices specified within the *Policy Manager* are implemented by the *Domain Manager*.

Domain Manager The public and private areas of the Grid middleware, the *Administrative Domain* and *Computational Community*, are linked through the *Domain Manager*. The *Domain Manager* publishes the resources within its private *Administrative Domain* in one or more specified public computational communities. The local administrator defines which resources are to be published in each *Computational Community* and their associated access and usage policy through the *Policy Manager*. The same resource may appear with different access and usage policies in several computational communities. This functionality is essential in building virtual organisations with preferential access levels. All access to the resources within the *Administrative Domain* pass through the *Domain Manager* where each request is authenticated through the *Identity Manager* using an X.509 public key infrastructure.

Identity Manager The authentication of the users (their identity, organisation and group memberships) is delegated by the *Domain Manager* to a trusted *Identity Manager*. The user's identity and organisation are encapsulated within an X.509 certificate [10] associated with the resource request. Thus the *Identity Manager* has two roles, to authenticate users, and to act as a certification authority, by providing the certificates used by its local community. The *Identity Manager* also maintains a list of local users which

is used to group individuals within the organisation, and to mark certificates as revoked if a user leaves an organisation.

Repository A large quantity of information is used by the various ICENI tools, including meta-data describing the component abstractions, implementations, and resources. This meta-data is stored within the Repository, which is itself a software resource.

Resource Browser Users interact with the public *Computational Community* through tools such as the *Resource Browser*. The *Resource Browser* allows the user to browse the application, job and resource meta-data (with a functionality comparable to that of a virtual machine room [11]) and allows for the construction of applications from the available component abstractions.

Activation Framework This system generates an active run-time representation from the description of the application produced by the user within the *Resource Browser* or with any other tool. This run-time representation is responsible for providing a set of potential implementation selections together with potential information regarding their performance, and to manage the resulting execution of the composite application. This is discussed in more detail in Section 5.

Application Mapper The *Application Mapper* generates an *Execution Plan* that matches the application's requirements to the resources that are currently available. This Execution Plan represents a feasible execution strategies optimised with regard to performance of the application and the user's requirements, along with potential costs imposed by the resource meta-data.

These core tools are highly extensible by developers, and may be supported or supplemented by other systems. For example tools may be used to provide the meta-data that is stored within the repository, such as XML editors to design component abstractions, or to perform performance modelling of implementation codes. This extensibility is provided by the use of CXML as the medium of communication and information storage - any supplementary tool need only consume or produce conforming CXML to be able to be integrated into the framework.

3.3 Component Technology within ICENI

In order to exploit the high-level information for scheduling in a Grid context it is necessary to adapt the component based design pattern. The encapsulation and abstraction of the component paradigm allows for a clear separation between implementation and abstraction. Thus a single abstraction (the component interface) may have multiple low-level implementations. Within the conventional component-based design paradigm, this feature is used to provide robust incremental software development over time.

Within a Grid context, this separation may be exploited to allow optimised deployment and scheduling, and enable the run-time management that is essential to Grid-based computing. If the component abstraction is realised as a platform independent entity, this entity may be paired with a platform-specific entity at run-time. We call this realisation of the component abstractions as the 'run-time representation'. In order for a component to execute and perform meaningful computation, the platform specific implementation is loaded into the run-time representation, which handles inter-component communication. This separation of the component into platform specific and independent aspects produces the following benefits:

Portability The run-time representation may be deployed onto any platform that possesses an implementation for that component.

Mobility As the run-time representation may load or unload particular implementations, it may be relocated to other hardware resources and implementations, as long as any persistent data is carried with the high-level abstraction.

High Performance As the low-level implementations are specifically designed for the platform on which they operate, they may be highly tuned, or utilise architecture specific libraries such as BLAS [12] and LAPACK [13].

The selection of the implementations for the run-time representation also provides opportunities for optimising the component application alongside its deployment. This cross-component optimisation takes place as part of the process of selecting implementations for the run-time representation. Implementations are selected at run-time, so as to take advantage of dynamic information, and are selected in the context of the application, rather than a single component, so that the complete selection of implementations is optimal. The process of cross-optimisation is described in Section 5.

4 Structured Information

Fundamental to the ability to map an application on the grid resources intelligently is structured information, which we describe in CXML through a set of 'tags' which are used to annotate data. As CXML is a realisation of XML, it may be manipulated by generic XML handling tools as well as specific ICENI systems developed using standard APIs. Detailed descriptions and examples of CXML tags may be found in a previous work on this subject [14].

The CXML documents describe information in five major categories that are received from three primary sources. Information supplied by the user, includes the User Requirements and Application Structure, while the Component Specification and Implementation Meta-Data are stored in the *Repository*, and the

Resource Information is received from the *Domain Manager*.

4.1 User Requirements

When users assemble and submit an application specific to the resources within a computational community, they may specify their own expectations of, for example, the quality of service they expect to receive. This user oriented metadata on the job execution, places constraints on the decision making process that are taken into account by the ICENI system, in particular the *Application Mapper*. In the current implementation of ICENI User Requirements are limited to two particular constraints: *Minimise Execution Time*, and *Minimise Execution Cost*, where the cost model is determined by the resource managers on a resource by resource basis (see Section 4.5). The system of CXML is extensible, and it is intended that user requirements may be extended in the future to allow further quantitative constraints.

4.2 Application Structure

The application structure is described by an Application Description Document. This CXML document is produced by the end-user, and represents an application consisting of a number of specified component *instances* together with the *connectors* which relate the instances to each other. The component instances are occurances of the component *types* provided in the repository (see Section 4.3 below), where exposed data types may have been customised by the end-user.

We have developed a simple visual programming environment that enables users to define their component application, from the components available within local or remote component repositories, using a 'drag-and-drop' paradigm. The application description is placed into an ICENI public *Computational Community* with additional information from the user describing how the application should be executed (e.g. as quickly as possible or as cheaply as possible). Higher-level services within the ICENI framework (in particular the *Application Mapper*) analyse the application description and the state of the available resources to effectively map the component network onto the distributed resources.

4.3 Component Specification

The component types that are available to the end-user are specified in CXML, and stored within the repository as CXML documents. These documents specify the component's interface, and its methods' behaviour. A component type may have methods which are placed in groups known as *Outports*, which are typed collections. *Inports* are typed references to the outports of other components. It is this port mechanism that allows the composition of component instances into applications. Exposed data elements are also indicated within the interface specification and may be customised at design time.

In order to perform optimisation upon an application, it is necessary to analyse its behaviour, and to the end the high-level component specification includes a description of the contextual dependencies of a component's methods. That is, each method has an attached *behaviour* which indicates any inter-component calls that are made as result of calling that method, but in order to preserve encapsulation no behaviour internal to the component implementation is specified. A method's behaviour is a documentation of what subsequent component-level calls are made by the implementation, and in what particular order and fashion (e.g. concurrently, sequentially etc). This information is provided by the author of the component, whose role is to specify the components' explicit context dependencies, through both interface and behaviour.

4.4 Implementation Meta-Data

Each component type specified in Section 4.3 may have multiple implementations. Each implementation is stored within a Repository, together with meta-data that describes the implementation. This meta-data is provided by the implementation developer, who may use any tools they see fit to formulate the information.

Each implementation is platform specific (though that platform may be a virtual one in the case of Java implementations), and is indicated within the meta-data. Each method the component implementation offers has an attached performance model, possibly parameterised by some component-level property (such as problem size), and specific to the hardware resource that the implementation may be deployed upon. Given the limited domain of the implementation, it is possible to develop these performance models purely empirically, using black box techniques. Additional implementation specific binding code used to load the implementation into the run-time representation is also included here.

Implementation meta-data is used to both pair an abstract component inter-

face with an available implementation (and hence a hardware resource upon which it may be deployed) but the performance model is used to guide the choice of implementations.

4.5 Resource Policy

All resources possess two types of attributes, static and dynamic. Static attributes, such as the machine name and operating system, are generally specified on installation and remain unchanged thereafter. Dynamic attributes, such as the machine load or the available software libraries, will change at irregular intervals.

Local access and usage policy are specified through the *Policy Manager*, and are then implemented as attributes by the local *Domain Manager*. A simplified access control policy has been implemented that accepts or rejects a request on the basis of the user, group or organisation [14]. Different policies can be specified for different periods of the day or week.

Resource meta-data may also include variable costing models which are used in the process of implementation selection by the *Application Mapper*. This is described in Section 5.2.

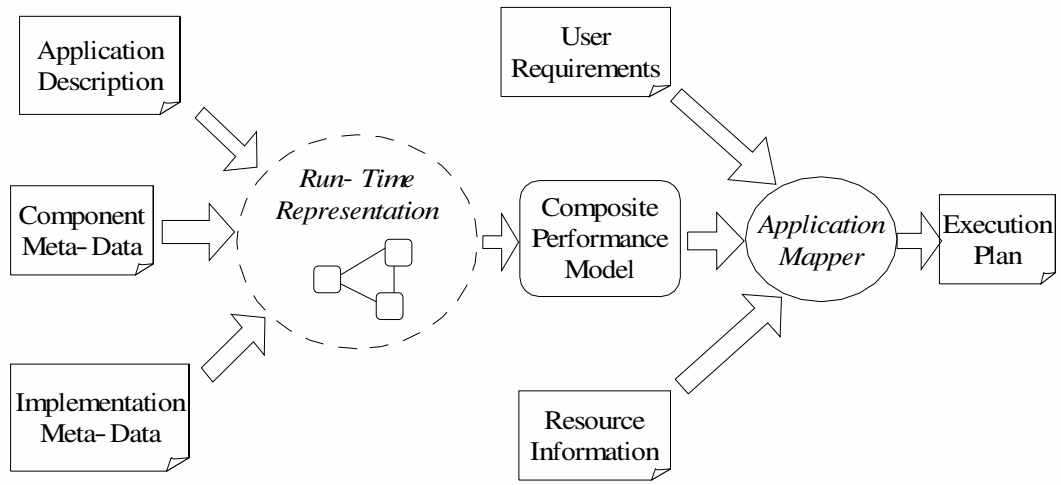

Fig. 3. Information Flow through the Optimisation Process

Figure 3 shows how the information described in Section 4 is used to inform the deployment and execution of the component based application. This information is used to enable the automatic choice of implementation for each

component in the application that possesses multiple implementations. The optimisation is thus a decision making process, consisting of a two key phases. Firstly, the creation of a composite performance models for the application with its various implementation selections, and secondly using those models to select the best set of implementations over the currently available resources.

5.1 Composite Performance Modelling

In order to make a decision regarding an optimal implementation selection, it is first necessary to quantitatively analyse the performance of the application given each selection of implementations. It is necessary that this performance analysis be *composite*, as while a particular implementation choice may provide the best performance for a given application's method, it may inhibit other components to the detriment of the complete application. To this end the information regarding the composition of the component instances (the application's structure) and the behaviour of the individual component methods is combined within the run-time representation. This process is described below.

a. Activation of Run-Time Representation When deployed onto a resource the CXML application description is converted into a *run-time representation*, by the Activation Framework. The run-time representation consists of a network of Java Proxy Objects (JPOs), corresponding to the component instances in the application description document [14]. This abstraction of the component application enables the dynamic selection of implementations, cross-component optimisation and implementation independent management.

Each JPO provides a black box abstraction of the component's implementation and acts as the run time interface for the component. Any interaction between components occurs at the level of the JPO. This provides a means to trap method calls and select the appropriate implementation when required.

b. Simulation of Behaviour In order to build the model, the composite application's behaviour is simulated by the run-time representation. As such the application structure is fully utilised. Those calls which occur at the component level (those represented by methods within the component meta-data) are simulated, while the full processes of each component implementation are bypassed. This simulation utilises those elements of component level behaviour described using the `<behaviour>` elements of the component meta-data to construct a call path, beginning with a specified `<start>` method. This simulated execution utilises the state information stored within the run-time representation where necessary (which is then restored before actual execution) allowing predictive branching and esti-

mates of loop iterations.

c. Model Construction Once the path has been constructed, a recursive function representing the composite performance model may be derived from traversing the path, and evaluating the function according to the behavioural characteristics of each method call. Where a call refers to an implementation, it is temporarily replaced with a variable. These implementation variables form the bases of the recursive function. The composite performance model is thus implementation independent at this stage: all implementation configurations possess the same composite behaviour.

d. Model Evaluation The composite performance model function is then evaluated with the performance meta-data provided for each potential implementation selection. Each variable is replaced with the value provided by evaluating the respective implementation's atomic performance model with the application and hardware parameters. In general, if there are n different components with multiple implementations, and hence n variables in the performance function, and c_n choices for each component, the total number of implementation selections is $\prod_1^n c_n$.

Thus each possible selection of implementations is provided with a measure of the estimated execution time of the application with that particular implementation selection.

5.2 Comparison of Composite Models

Given the set of evaluated composite performance models, it is the responsibility of the Application Mapper to choose the desired implementation selection. This decision is made using the constraints provided by the information on the user requirements.

In the current implementation of the execution environment there are only two possible options for user requirement information, *Minimise Execution Time* and *Minimise Execution Cost*. For the former, the implementation selection with the smallest estimated execution time is chosen. For the latter, a *Cost Model* must be provided. This is a function of the hardware resources and is stored as resource meta-data. It represents an arbitrary value per processor per unit time utilised by the application, allowing for the Application Mapper to choose between potentially less valuable resources that may take a long time to complete the application, or those which are more greatly contested but are likely to be occupied for less time.

The Application Mapper produces an Execution Plan (a CXML document), which specifies the implementation selection and the resources upon which they are to be deployed. The Activation Framework, which has built the Run-

Time Representation in order to simulate the application behaviour, then deploys this Run-Time Representation onto the specified resources, where the application's components and the selected implementations are executed.

6 Evaluation & Results

In order to illustrate this system, a simple example application that solves a system of linear equations was deployed within the system. The solution of linear systems arises frequently within scientific computing problems. When such an application is considered as a composition, it consists of three component instances. A linear equation generator creates a system of linear equations. It has two possible implementations (C and Java). The core of the system is a linear equation solver, which may take a number of different implementations on different platforms. Additionally a display component outputs the results - this component also makes the initial calls to the solver. While this application is unlikely to be deployed in practice, the solver could well form the basis of of larger and more complex scientific applications. The connection of the component instances is shown in Figure 4.

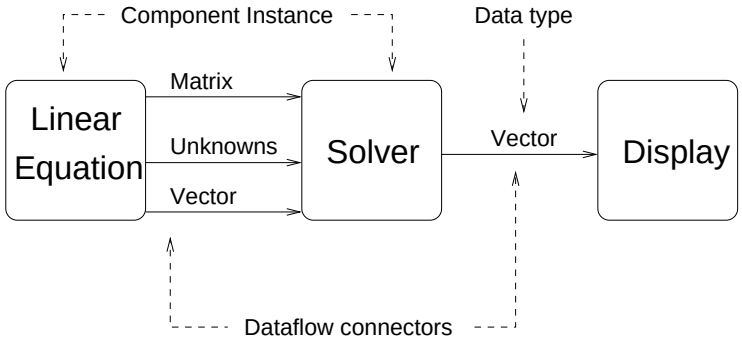

Fig. 4. Linear Equation Solver

The testbed grid environment on which this application was deployed consisted of three different machines, *AP3000*, *atlas*, and *hotol*. The descriptions of these machines' processors and interconnects are given in Table 1. The linear equation generator's implementations are both specific to a single processor Linux machine, and therefore run on *hotol*. As *atlas* and *AP3000* are parallel machines, they may be considered as multiple resources depending upon the number of processors employed. For example, *atlas* may be considered as 3 separate resources, with 1, 4 or 9 processors deployed respectively.

The source component generates random unsymmetric set of real linear equations, for a given number of degrees of freedom, through either a C or Java implementation. The solver component has a wide choice of implementations, including the direct LU factorisation algorithm, and the iterative biconjugate

gradient method. These are implemented on a range of different machines, as listed in Table 1, utilising high performance parallel libraries where available. The performance models for the generation of the linear equations and the solution of those systems of equations are published in a technical report [15].

System	Processor	Interconnect	Processors	Language	Solution
hotol (x86 Linux)	AMD 900Mhz	N/A	1	Java	LU
					BCG
				C	LU
					BCG
				LAPACK	LU
atlas	Alpha 667Mhz	Quadrics	1,4,9	ScaLAPACK	LU
					BCG
AP3000	UltraSPARC 300Mhz	Fujitsu	4,9,16	ScaLAPACK	LU
					BCG

Table 1
Available solver implementations

Three different cost models are considered. The basic model (Cost Model A) has each resource possessing unit cost per unit time per processor. This model was used as a baseline for experiments. The Cost Models B and C are shown in Table 2, and represent a richer pattern, where the Linux processor is cheaper than the *AP3000* which is in turn cheaper than *atlas*. The differences between Model B and C may represent night- and daytime usages.

System / Cost Model	A	B	C
Linux PC (*Hotol*)	1	0.8	0.8
Alpha Cluster (*Atlas*)	1	4.25	4.75
UltraSPARC (*AP3000*)	1	1	1

Table 2
Cost Models: Cost per Processor per Unit Time

Figure 5 shows the predicted overall execution time of the application varied with the problem size. Two possible user requirements are shown for comparison, that of Minimum Execution Time (with the solid line) and Minimum Cost (Model A). The Minimum Execution Time is achieved by selecting the LU solver over 9 processors on *atlas* for small problems and then switching to the biconjugate gradient method for larger problems. Where the selection criteria is Minimum Cost, again *atlas* is chosen (unsurprisingly, given that Cost Model A makes no distinction between processor costs, but does penalise the slower solution). However as the problem size increases 1, 4 then 9 proces-

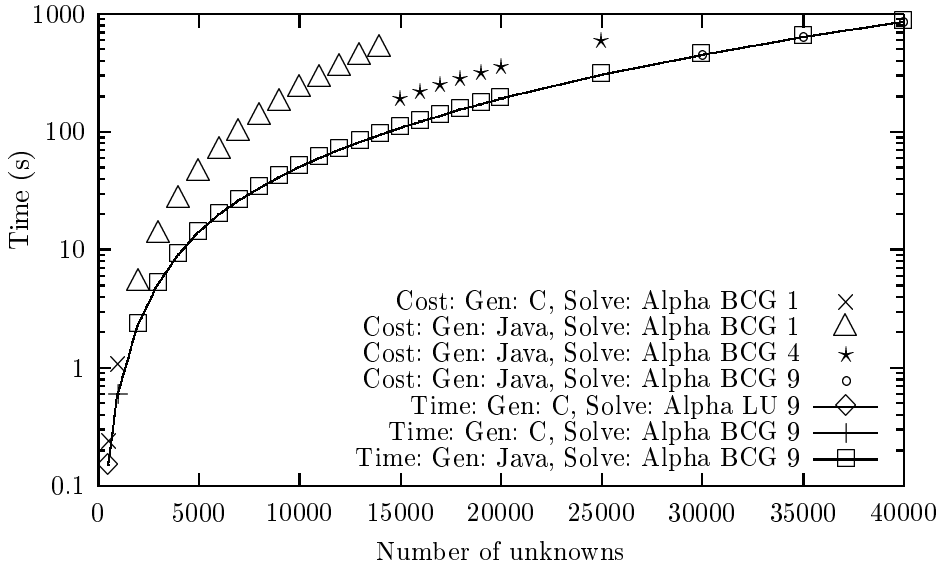

Fig. 5. Estimated Execution Time of Composite Application

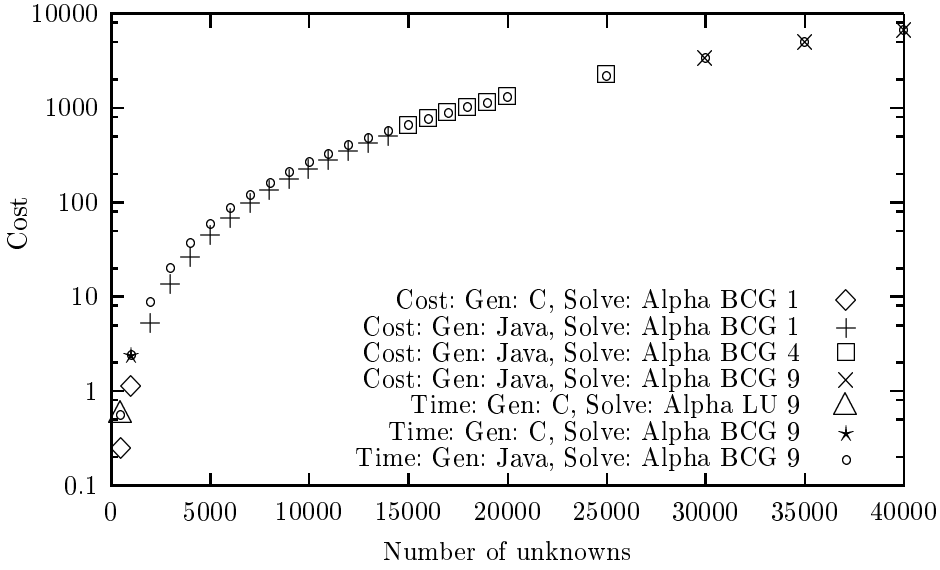

Fig. 6. Computational Resource Cost (Cost Model A)

sors are chosen. The figure indicates the improved execution time gained from adopting the Minimum Execution Time requirement.

Conversely Figure 6 illustrates how much additional cost is incurred by adopting the Minimum Execution Time strategy, as compared with adopting the Minimum Cost approach. This requirement forces a balanced approach to optimal resource selection with additional computational resources being used as the problem size increases.

The use of the more realistic Cost Models B and C demonstrate the effect of using different models within the Minimum Cost strategy. Figure 7 indicates

the predicted execution time (again with the selection for Minimum Execution Time for comparison), while Figure 8 shows the cost incurred with both models, as against Minimum Execution Time. The key to the symbols is given in Table 3.

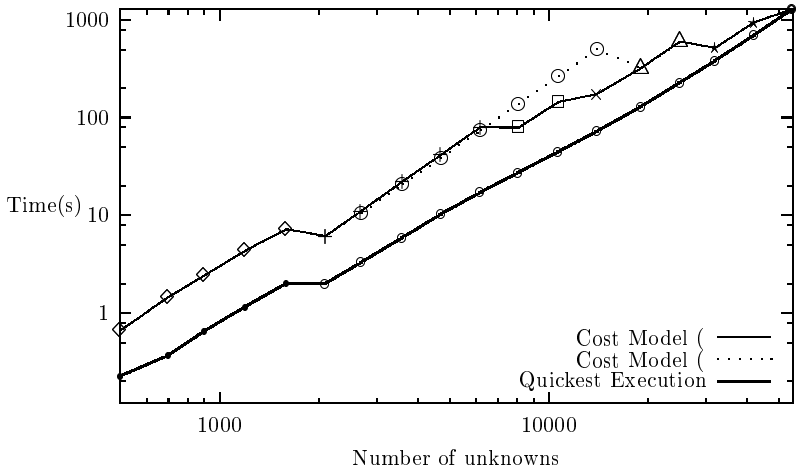

Fig. 7. Execution Time for the Composite Model with Different Number of Unknowns

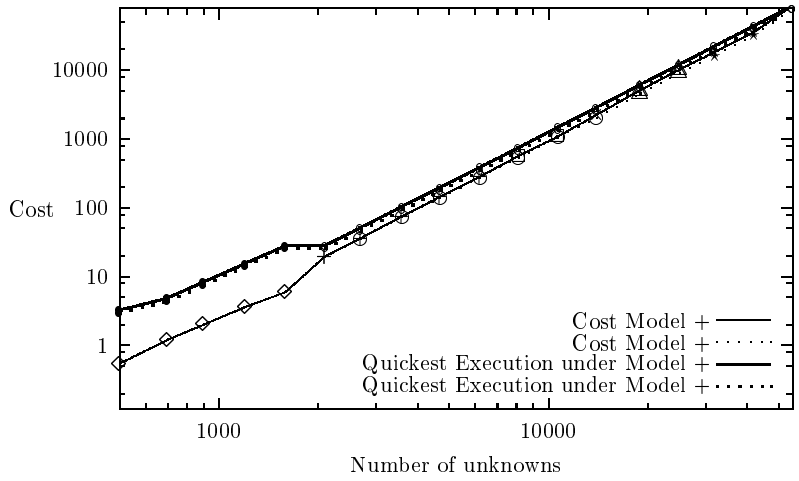

Fig. 8. Execution Cost for the Composite Model with Different Number of Unknowns

It can be seen that different resource costs (an inevitable consequence of any demand driven computational economy) result in the selection of different implementations on different resources if the overall costs are to be minimised. The graphs show that minimising cost produces significantly longer (on average 5 times) execution times while minimising execution time increases the resource cost (on average 1.6 times) with the current cost models.

Symbol	Source Implementation	Solver Implementation	Processors
◇	C	Hotol, BCG	1
+	Java	AP3000, BCG	4
□	Java	AP3000, BCG	9
×	Java	AP3000, BCG	16
◯	Java	Atlas, BCG	1
△	Java	Atlas, BCG	4
⋆	Java	Atlas, BCG	9
○	Java	Atlas, BCG	16
•	C	Atlas, BCG	16

Table 3
Key to Graphs

7 Related Work

The Globus toolkit [8] has become the de facto standard framework for service provision within the Grid and e-Science communities for the construction of *virtual organisations* for collaborative resource sharing. ICENI complements this work by building on top of Globus' established execution and security infrastructures by interfacing through the Java CoG system [16] to explore what information is needed to effectively exploit the resources within the *Computational Community*.

ICENI exploits Jini's [17] leasing mechanism to enable the soft-state registration of transient grid resources, represented by Java objects, to connect and re-connect over time while allowing unexpected failures to be handled gracefully. We use Jini's look-up server to provide dynamic registration of Grid objects in the public computational communities and private Administrative Domains. Within Globus, the federation of resources is achieved, for example, through the use of an LDAP server [18]. The initial stages of this work were described in [19] and are expanded in this paper. Other Java computing infrastructures such as Javelin [20], Popcorn [21] (using computational economics) and those based on Jini [22] have not addressed the usage and access policies which are essential in building virtual organisations.

There are many established general purpose component based programming systems including JavaBeans [23], which is based upon the Java language, CORBA [24], which provides language interoperability and Microsoft's DCOM [25] and .NET [26]. These systems provide software bindings between components, but do not maintain any form of high-level information or meta-data which we feel is essential within dynamic execution environments, such as the Grid. CXML is not a component definition language in itself, rather it is a meta-language that contains sufficient information to generate a CORBA or Java Beans binding.

There are also practical examples of component based systems in use within high performance computing, including the Linear System Analyser [27], the Common Component Architecture (CCA)[28] and Ligature [29]. In particular the CCA system is designed for use in a distributed high performance environment such as the Grid, and there have been significant innovations with respect to components for parallel machines. Nevertheless, while these systems provide additional compositional meta-data beyond the basic software bindings provided by CORBA, DCOM etc, they do not exploit performance data to guide component optimisation, and implementation selection as outlined in this work is not developed fully.

8 Conclusions and Further Work

In this paper we have outlined the implementation of our approach to e-science, which we have identified as the problem of enabling the effective utilisation of Grid resources by high performance applications. We have designed an extensible system of middleware tools, ICENI, which support federated computational communities allowing secure and extensible access to the Grid services, together with a system of component based design for Grid deployed applications. ICENI tools provide optimised deployment of these application components via performance guided implementation selection. In this work we have constructed a prototype realisation of ICENI, and demonstrated the implementation decision making process for a simple example (a linear equation solver) on a number of platforms.

Our early experiences with ICENI and its associated component framework have demonstrated to us the importance of information relating to the resources within the Computational Community and of the applications and users wishing to exploit them. The use of CXML, a markup language produced and consumed by the services within the ICENI framework, as a common protocol enables the implementation of other services by other groups. It is the development of these protocols, as the composition of XML Schemas relating to resource attributes, that we see as the major strength and contribution of the ICENI framework.

The use of XML encoded interactions between elements of the ICENI framework is reflected in the emergence of web services (SOAP, WSDL and UDDI) within the Grid community. While we intend to retain Java and Jini within ICENI, we will expose its functionality through a web services gateway to allow interaction with other infrastructures.

References

[1] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, July 1998.

[2] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Intl. J. Supercomputer Applications*, 2001.

[3] B. Nardi. *A Small Matter of Programming: Perspectives on End-User Computing*. MIT Press, Cambridge, Massachusetts, 1993.

[4] J. Darlington *et al.* Parallel Programming using Skeleton Functions. volume 694 of *LNCS*, pages 146–160, 1993.

[5] J. Darlington, M. Ghanem, Y. Guo, and H. W. To. Guided Resource Organisation in Heterogeneous Parallel Computing. *Journal of High Performance Computing*, 4(10):13–23, 1997.

[6] C. Lin and S. Guyer. Optimizing the use of high performance libraries. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing*, August 2000.

[7] C. Szyperski. *Component Software: Beyond Object Oriented Programming*. Addison-Wesley, 1998.

[8] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *IPPS / SPDP'98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[9] W3 Consortium. XML: eXtensible Markup Language. `http://www.w3c.org/XML`.

[10] Sun Microsystems. X.509 certificates. `http://java.sun.com/products/jdk/1.2/docs/guide/security/cert3.html`.

[11] The Virtual Machine Room. `http://www.ncsa.uiuc.edu/SCD/Alliance/VMR/`.

[12] J. Choi, J. Dongarra, and D. Walker. Level 3 blas for distributed memory concurrent computers. In *CNRS-NSF Workshop on Environments and Tools for Parallel Scientific Computing*. Elsevier Science Publishers, 1992.

[13] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Release 1.0*. SIAM, Philadelphia, USA, 1992.

[14] N. Furmento, A. Mayer, S. McGough, S. Newhouse, and J. Darlington. A Component Framework for HPC Applications. In *Euro-Par 2001, Parallel Processing*, volume 2150 of *LNCS*, pages 540–548. Springer-Verlag, 2001.

[15] N. Furmento, A. Mayer, S. McGough, S. Newhouse, and J. Darlington. Performance Models for Linear Solvers within a Component Framework. Technical report, ICPC, 2001.

[16] G. von Laszewski, I. Foster, J. Gawor, W. Smith, and S. Tuecke. CoG Kits: A bridge between commodity distributed computing and high-performance grids. In *ACM 2000 Java Grande Conference*, 2000.

[17] Sun Microsystems. Jini(tm) network technology. `http://java.sun.com/jini/`.

[18] OpenLDAP Project. `http://www.openldap.org`.

[19] N. Furmento, S. Newhouse, and J. Darlington. Building Computational Communities for Federated Resources. In *Euro-Par 2001, Parallel Processing*, volume 2150 of *LNCS*, pages 855–863. Springer-Verlag, 2001.

[20] M. O. Neary, B. O. Christiansen, P. Cappello, and K. E. Schauser. Javelin: Parallel computing on the Internet. In *Future Generation Computer Systems*, volume 15, pages 659–674. Elsevier Science, October 1999.

[21] O. Regev and N. Nisan. The Popcorn Market - Online Markets for Computational Resources. In *The 1st Int. Conference On Information and Computation Economies. Charleston SC*, 1998.

[22] Z. Juhasz and L. Kesmarki. Jini-Based Prototype Metacomputing Framework. In *Euro-Par 2000*, pages 1171–1174, 2000.

[23] JavaSoft. JavaBeans. `http://java.sun.com/beans`, October 1996.

[24] Object Management Group. The common object request broker: Architecture and specification. formal document 97-02-25, July 1995. `http://www.omg.org`.

[25] D. Chappel. *Understanding ActiveX and OLE - A guide for Developers and Managers.* Microsoft Press, 1996.

[26] Microsoft Corporation. Microsoft .NET WWW page. `http://www.microsoft.com/net/`, 2001.

[27] R. Bramley, D. Gannon, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. The linear system analyzer. Technical Report TR-511, Computer Science Dept, Indiana University, 1998.

[28] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *In Proceedings of the 8th High Performance Distributed Computing (HPDC'99)*, 1999.

[29] K. Keahey, P. Beckman, and J. Ahrens. Ligature: Component architecture for high performance applications. *The International Journal of High Performance Computing Applications*, 14(4):347–356, Winter 2000.