

PUBLICATION
INTERNE
N° 1782

PERFORMANCE AND PRACTICABILITY OF DYNAMIC
ADAPTATION FOR PARALLEL COMPUTING: AN
EXPERIENCE FEEDBACK FROM DYNACO

JÉRÉMY BUISSON , FRANÇOISE ANDRÉ AND
JEAN-LOUIS PAZAT

Performance and practicability of dynamic adaptation for parallel computing: an experience feedback from Dynaco

Jérémy Buisson^{*}, Françoise André^{**} and Jean-Louis Pazat^{***}

Systemes numériques
Projets Paris

Publication interne n° 1782 — février 2006 — 20 pages

Abstract: Emergence of grid computing and observation of its dynamic nature have led to the proposal of using techniques of dynamic adaptation within Grid applications in order to use resources better. Several projects propose architectures for dynamically adaptable parallel applications. Whereas evaluation is usually done in terms of overhead and runtime performance of the applications, the practicability of the approach from the point of view of developers has often been forsaken. Our work focuses on providing a generic framework that helps developers in building dynamically adaptable parallel applications. This framework, which we call Dynaco, is evaluated both in terms of achieved performance from the point of view of the execution, and in terms of practicability from the point of view of developers.

Key-words: Dynamic adaptation, parallel computing, software engineering

(Résumé : tsvp)

* jeremy.buisson@irisa.fr
** francoise.andre@irisa.fr
*** jean-louis.pazat@irisa.fr

Etude de performance et d'utilisabilité de l'adaptation dynamique pour les programmes parallèles: retour d'expérience de Dynaco

Résumé : L'émergence des Grilles de calcul et l'observation de leur nature dynamique ont conduit à proposer l'utilisation des techniques d'adaptation dynamique afin de permettre aux applications pour Grilles de mieux utiliser les ressources. Plusieurs projets ont proposé des architectures d'applications parallèles adaptables dynamiquement. Cependant, alors que leur évaluation est généralement faite en termes de surcoût et de performance à l'exécution des applications, l'aspect pratique de cette approche du point de vue des développeurs n'est pas étudié. Dans nos travaux, nous étudions comment fournir aux développeurs un canevas générique qui les aide à construire des applications parallèles adaptables dynamiquement. Dans cet article, ce canevas que nous avons baptisé Dynaco, nous l'évaluons à la fois en termes de performances obtenues à l'exécution et en termes de travail imposé aux développeurs.

Mots clés : Adaptation dynamique, programmation parallèle, ingénierie logicielle

1 Introduction

Since the concept of Grid computing has emerged, finding the right approach for developing Grid applications is still an open question. The challenge consists in making applications able to benefit from the huge amount of resources without burdening the developer. Observing the fact that Grid computing environments are dynamic, it has been proposed that applications should be able to adapt themselves dynamically to environmental changes. This technique is dynamic adaptation. In the context of Grid computing, changes concern mostly processor and network availability because of resource sharing between applications, administrative tasks and failures. Making applications able to modify themselves according to their actual execution environment guarantees that they will always behave in the best possible way given the resources allocated to them, as far as the expertise of the developer allows it and the resource management system is able to change resource allocation during the execution of applications.

Informal arguments such as the ones given above show well that dynamic adaptation may be a valuable technique for programming applications in the context of Grid computing. Those arguments have been confirmed by various experimental measures [1, 6, 16]. Measures show that dynamic adaptation can be implemented with negligible overhead while reducing the overall execution time of parallel applications if applications last long enough to balance the specific cost of the adaptation. On the other side, only few projects have studied dynamic adaptation from the point of view of developers. Even if proposals try to separate the dynamic adaptation functionality from applicative code, no experience gives feedback of the impact on the workload of developers. What is the amount of work to make an application able to adapt dynamically? What expertise is required? What knowledge about the application itself is necessary? Those practical questions remain unanswered, although they are essential for convincing developers of adopting dynamic adaptation. This paper aims at proposing a generic framework for dynamic adaptation that takes into account performances and practicability, thus also addressing those questions.

In section 2, we present Dynaco, our general framework for dynamic adaptation that is being used for the experiments. Section 3 exemplifies how this framework can be used to make parallel applications adaptable to the actual number of processors. Some performance evaluation is given. Section 4 describes the method we propose for designing dynamically adaptable components. Section 5 evaluates and discusses the effective work required for the cited examples. Section 6 presents related works. Section 7 concludes this paper and gives future directions of our work.

2 Model of dynamic adaptation

Developers need a model in order to ease the design and development of adaptable components. We use here the word “component” abusively to denote the entity that is made adaptable. It could be a complete application not using any component technology as in the

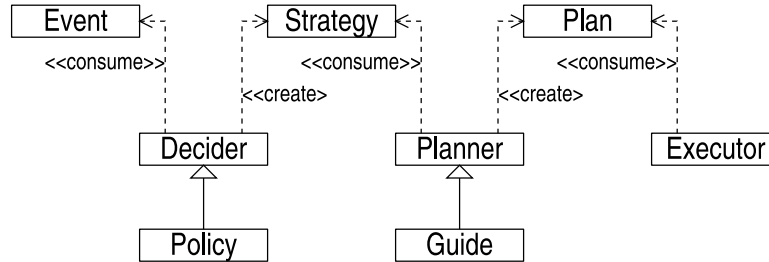


Figure 1: Model of the adaptation process

experiments we have made, a component with the traditional meaning of component-based design, a service, and so on.

2.1 Overall description of the model

The overall model of Dynaco is depicted on figure 1. This model exhibits the functional decomposition we propose for the adaptation process. Major entities *decider*, *planner* and *executor* implement the main functions of dynamic adaptation. They are organized as a pipeline. The *decider* reacts on environmental changes received as events and produces a strategy for dynamic adaptation. The *planner* derives a list of actions from the strategy in order to achieve the different steps of the process of adaptation. In the end, the *executor* implements the different steps of adaptation to modify the component.

This model is general to the extent that it is not tight to any particular component or any class of components. Section 2.2 focuses on parallel components, whereas our model could be applied to non-parallel components.

Decision-making The decision-making process of dynamic adaptation has to be specialized by a *policy*. This policy indicates how the decisions should be made. It specifies which strategies should be used in order to modify the component according to the actual observation of the execution environment. While being specific to the application domain, the policy should be expressed in a way such that the decision-making engine can be generic.

Events consumed by the decider may have various origins. Some of them can be generated by probes that monitor the execution platform; others may be created by the adaptable component itself. In either case, the entities generating events are called *monitors*. The initiator of the observation may be the monitor (push model), the decider (pull model) or both.

Adaptation planning Planning the adaptation means to create a program that achieves the strategy previously decided. This program, called the adaptation plan, consists in a collection of actions that have to be performed and ordered by some control flow. The *planner* is specialized to the actual adaptable component thanks to a planification *guide*.

Component adaptation Endly, once the plan has been created, it has to be executed by the *executor*. This executor can be seen as a virtual machine implementing the control flow instructions that orders actions within the adaptation plan. It schedules the execution of the actions, then executes this schedule. To do so, the *executor* relies on *adaptation points* that indicate states of the component at which actions can be executed. Adaptation points are annotations located in the source code of the component.

The exact meaning of “can be executed” in the definition of adaptation points depends on the component and on the action. For instance, if the action redistributes tasks, the state of the component is constrained by the integrity (executed or not executed) of the tasks that should be redistributed; if the action checkpoints the component for a later restart, the state of the component should satisfy a consistency criterion such as the one of the global states [7].

2.2 Case of parallel components

The case of parallel components, that is to say a component implemented by a parallel code, does not impact the overall structure of the model. It introduces specific synchronization constraints as adaptation points denote global states within the parallel execution of the component. In order to assist the executor in its task of scheduling actions, a *coordinator* entity is introduced that is able to choose an adaptation point in the case of a parallel component. We have discussed in [4] consistency criteria that can be used to identify global adaptation points from the combination of points locally placed in each process of the component. The basis for a distributed algorithm is proposed in [5] that chooses the next global adaptation point in the execution.

2.3 Realization as a framework

Implementations that take the form of frameworks can be derived from the proposed model. Such a framework provides a basis for the implementation of adaptability, while ensuring that adaptability is not tangled within applicative code. Separation of concerns is thus enforced.

We have prototyped Dynaco as a framework in the context of the Fractal [3] component model. This model splits components in two parts: the *content* implements the functionalities expected from the component; the *membrane* is a placeholder for non-functional services that control the behavior of the component.

As a non-functional service, Dynaco lays in the membrane of dynamically adaptable components. Figure 2 shows the structure of an adaptable component using Fractal formalism. In the membrane of the component, around the content, lays the framework. Components of the framework are gathered within a composite called the adaptation manager. Actions are implemented by modification controllers (mc), which are given a direct access to the content of their controlled components. The *executor* is bound to those controllers: it is thus able to execute actions following plans received from the *planner*. Endly, the *decider* exposes one server interface and one client interface to the outside of the adaptable component. Those

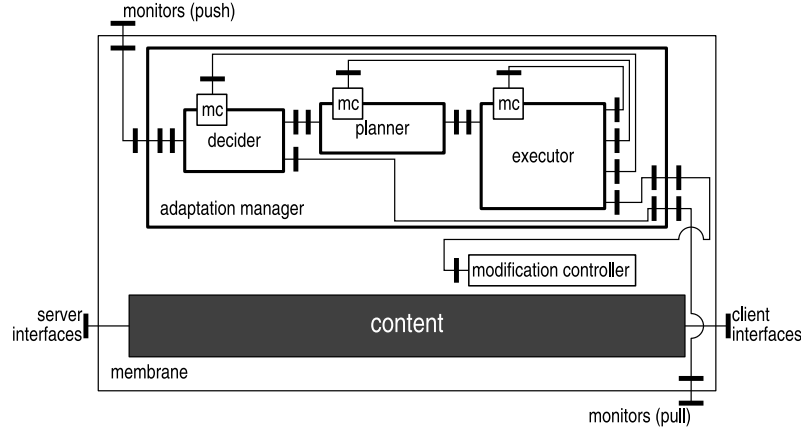


Figure 2: Structure of an adaptable component using our framework

interfaces implement the connection to monitors respectively with the push model (server interface) and the pull model (client interface). Specialization entities (*policy* and *guide*) are not shown.

Actions are not restricted to apply to the main function of the component: even entities of the framework have a modification controller. In addition, modification controllers are able to modify themselves. As they are collections of methods, the only modifications that apply to them consist in adding and removing methods. This ability ensures that the adaptation mechanism can modify the whole component, including its own adaptability.

3 Examples of use of the framework

In order to illustrate the proposed framework, two examples are described. The two examples consist in making existing applications able to adapt to the actual number of available processors in the execution environment.

3.1 FFT benchmark

This experiment targets the standard NAS Parallel Benchmark [12] FFT application. This component is a benchmark that computes the fast fourier transform of a matrix. This function is computed several times in order to evaluate the performance of the executing architecture. The transform itself is split into six computation steps interleaved with some transpositions of the matrix. Communications are performed thanks to MPI-1 standard functions.

3.1.1 Adaptation points

For the FFT benchmark, we placed an adaptation point in the main loop of the code. We also added an adaptation point before each computation step and transposition operations of the algorithm. This fine-grained placement of adaptation points increases the frequency, at the cost of raising difficulty for implementing the actions.

3.1.2 Decision policy

In this experiment, the only goal of the adaptation consists in making the component use as many processors as possible in the execution environment. As reducing the overall execution time is not the primary goal of this experiment, no performance model is required to prevent process spawning when the cost of communications rises.

Given this goal, the only significant events are processor appearance and disappearance. For the sake of simplicity, it is assumed that when an appearance event is received, the processors are already available; whereas disappearance events are received before processors are effectively reclaimed. This assumption makes the adaptation mechanism unable to implement fault tolerance. Nevertheless, it matches significant cases such as foreseeing resource reallocations and maintenance operations.

Strategies that should be achieved are the following ones: if some processors appear, then one process should be spawned on each of these processors; if some processors disappear, then the processes they host should terminate.

3.1.3 Planification guide

Summarized from the above policy, the only two strategies are spawning and terminating processes. Adaptation plans should be generated as follows.

Spawning processes In order to spawn processes on new processors, firstly, those processors have to be prepared in such a way that the creation of processes for the component becomes possible. Secondly the new processes have to be created. Those new processes must be connected to the previously existing ones in order to set up communication paths. Then the matrices should be redistributed over the new collection of processes. Endly, the state of the newly created processes needs to be initialized to match the next compute phase of previously existing processes.

Terminating processes When terminating some processes, firstly, matrices must be redistributed in such a way that terminating processes does not hold any data. Secondly, terminating processes must be disconnected from the other processes in order to make future collective operations not expect any message from the terminating processes. Then the requested processes can terminate their execution. Endly, reclaimed processors should be cleaned up by removing what resulted from the preparation phase.

3.1.4 Actions

The two plans exhibit six different actions. A detailed description of those actions follows.

Preparation of new processors Preparing new processors means to set up everything that is required in order to be able to create some processes for the component. This means that files of the component should be accessible from those processors. In case of a networked file system, nothing is necessary; otherwise, the files should be copied using any available technology. In addition, the MPI implementation must be made able to reach the new processors. Depending on the MPI implementation, this step may require to start daemons on the new processors.

Creation and connection of processes As the applicative component makes use of MPI-1 functions, MPI functions should be used at least to connect the newly created processes to the previously existing ones. This action can be implemented thanks to the `MPI_Comm_spawn` function that creates and connects new processes in one operation (this function belongs to the MPI-2 standard, which extends MPI-1). Alternatively, processes could be created independently of MPI then connected thanks to the `MPI_Comm_join` function. With either solution, care should be taken to connect each process individually in order to be able to disconnect each process independently of the others.

Initialization of newly created processes As several adaptation points are placed within the applicative code, the initialization action must ensure that newly created processes begin their execution from the adaptation point at where previously executing processes achieve the adaptation. To do so, the source code of the component must be modified in order to be able to skip the execution of the pieces of code preceding the target adaptation point.

Redistribution of the matrix In the case of the FFT benchmark, no redistribution function is available in the original source code. The redistribution itself is done by a collective all-to-all communication operation in which the collection of sending processes differs from the collection of receiving processes. This action is used in two cases: when some processes have been created in order to give them a part of the matrix and before some processes are terminated in order not to lose any part of the matrix.

Disconnection and termination of processes Implementing the action of disconnecting and terminating processes is straightforward. Disconnection can be done thanks to the `MPI_Comm_disconnect` function of the MPI-2 standard. Then the original termination code of the component can be used, as no more communication with other processes can occur.

Cleaning up of processors The content of the cleaning up action depends on what the preparation action does: if some files are copied, they should be deleted; if some daemons are started, they should be terminated.

3.2 N-body simulator

This experiment consists in making the existing real-world Gadget 2 [14] simulator able to adapt itself. Again, the adaptation is done with regard to the actual number of processors.

The Gadget 2 simulator computes the evolution of a self-gravitating N-body system without collision. In addition, gas dynamics can be simulated by the mean of smoothed particle hydrodynamics. Parallelism comes from the distribution of the particles over the processes. The simulator includes an *ad-hoc* load-balancing mechanism able to redistribute particles. The whole application is structured as a main loop: each iteration performs a load-balance action, then advances the simulation for one time step. More details about applicative domains and algorithms involved in the simulator can be found in [14, 15]. Communications are performed thanks to MPI-1 standard functions.

3.2.1 Adaptation points

For the sake of simplifying the implementation of actions, only one adaptation point is placed in the applicative code. This point lays at the beginning of the main loop of the simulator. Indeed, at that place, all particles are at the same time step of the simulation. In addition, this place ensures that any adaptation is followed by a load-balancing action.

3.2.2 Decision policy and planification guide

As this component is made adaptable to the same kind of environmental changes than those of the FFT benchmark component, the decision policy is the same for the two components. It consists in the following statements: if some processors appear, then one process should be spawned on each of these processors; if some processors disappear, then the processes they host should terminate.

The adaptation plans that should be generated are also similar in the two examples. The main difference is that in the case of the Gadget 2 simulator, particles should be redistributed; whereas in the case of the FFT benchmark, matrices should be redistributed.

3.2.3 Actions

Most of the actions are the same as the ones of the FFT benchmark component. They will not be detailed any more in this case. The main differences lay in the initialization and redistribution actions.

Initialization of newly created processes With the Gadget 2 simulator, the initialization phase is a collective operation over the whole set of processes: one of the processes reads

the initial conditions of the simulation, then it broadcasts it to the others to make them initialize their internal state. The initialization phase terminates by an initial distribution of particles. This means that in order to initialize the newly created processes, the previously existing ones should perform a reinitialization phase. This reinitialization is simpler than a complete initialization as initial conditions have already been read and the internal state of previously existing processes is already ready for the simulation.

Eviction of particles from terminating processes Before a process can terminate, it must ensure that it does not host any particle. Otherwise, hosted particles would be lost. Implementations this action can benefit from the original load-balancing mechanism: cheating this mechanism by masking terminating processes makes the action of evicting particles as simple as a redistribution, i.e. a function call.

3.3 Performance evaluation

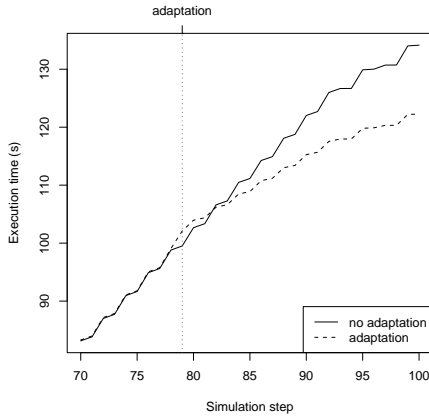


Figure 3: Execution time of the adaptable Gadget 2 simulator

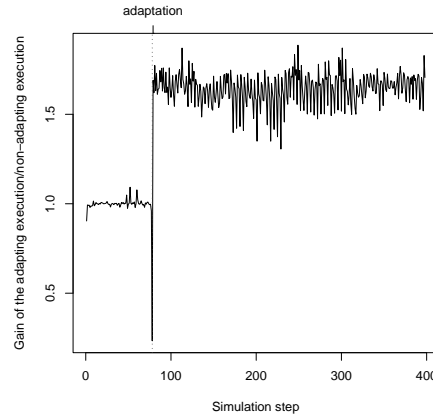


Figure 4: Evolution of the gain provided by the adaptation of Gadget 2

The overhead of using the framework is due to calls that must be inserted within the source code of the component. Those calls are executed even if no adaptation is performed. Because of the underlying algorithms [5], those calls have to be inserted before and after each control structure (loop, condition, function) and at each adaptation point. Measures show that the mean execution time of those functions ranges from $10\mu s$ to $46\mu s$. In the case of the FFT benchmark, the whole overhead is under 0.05% of the execution time of the component; it is under 0.02% in the case of the Gadget 2 simulator. The overhead appears to be negligible. Overhead measures of our framework in the case of the FFT benchmark is further detailed in [6].

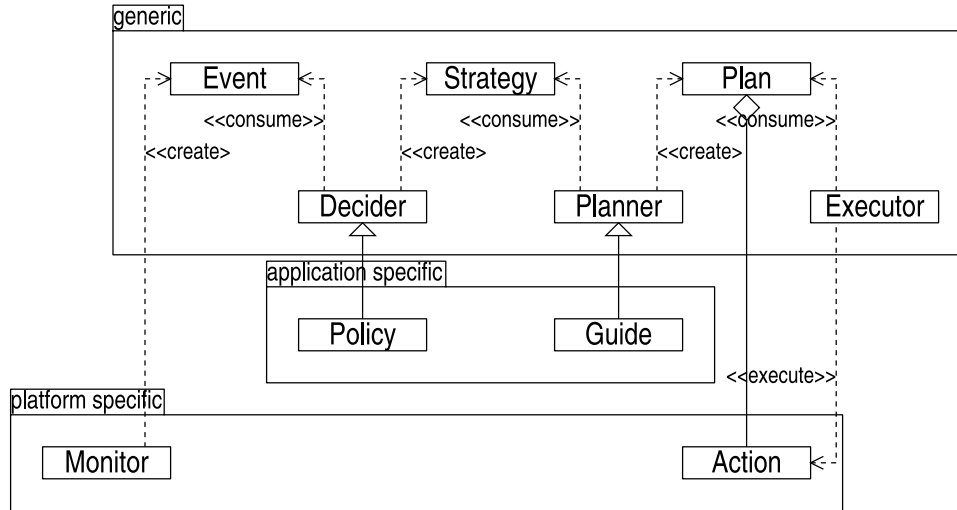


Figure 5: Structural decomposition of the framework

The execution of the applicative component benefits from the adaptation. Figure 3 shows the progression of the execution time of the Gadget 2 simulator when it adapts to the number of processors. In this experiment, the number of processors has been increased from 2 to 4 at timestep 79. This curve shows that the adaptation as a specific cost that can be balanced if the component continues its execution for long enough. Figure 4 shows how the gain provided by the adaptation evolves as the component executes. The gain is the ratio between the timestep duration for the adapting execution (2 to 4 processors) and the same duration for the non-adapting execution (2 processors). Before the adaptation, the gain oscillates around 1 as the two executions use the same resources. At the time of the adaptation, the gain falls: it reflects the specific cost of the adaptation. Then, the gain increases as the adaptation makes the simulator execute faster.

4 Guidelines for designing adaptable components

In order to integrate adaptability in software design and development cycles, we introduce a new role in those cycles: the adaptation expert who is responsible for making a component able to adapt. To do so, the expert should rely on methods and frameworks provided by the model of dynamic adaptation. This role captures exactly the work needed to make a component able to adapt itself.

The design method we propose describes the tasks expected from the expert. It relies on a structural decomposition of the framework for dynamic adaptation depicted in figure 5. Entities spread over 3 genericity levels: generic, specific to the application component or specific to the implementation platform. This decomposition comes from the expertise that

is required for each of the specialization entities (*policy*, *guide*, *actions*, *adaptation points* and *monitors*).

4.1 Required expertise

The expert is expected to have good knowledge about the applicative component. In addition, he must be skilled in the area of dynamic adaptation and in the paradigms, such as parallelism, used by the applicative component. Depending on the considered specialization entity, the expertise expected from the expert focuses on different aspects of that knowledge.

Policy and monitors The policy specializes the *decider* in order to specify how the decisions should be made for the actual component. In order to write it, the expert has to identify the goal of the adaptation. For instance, depending on whether the user expects the component to execute as fast as possible, at a given speed or not exceeding a given cost, ways to react to environmental changes differ.

Given the goal, the expert needs to model the behavior of the component with regard to that goal. This step includes the definition of a performance model if the execution speed is considered; a cost model if resource consumption is considered; an error propagation model if numerical precision is considered; and so on. This modelization step identifies the characteristics of the execution environment that may impact the behavior of the component with regard to the identified goal. Only changes of those characteristics are significant events to the adaptation: only them needs to be monitored. At the end of that step, the expert has identified the *monitors* that are required.

Endly, for each of the identified significant events, the expert defines which strategy should be performed in order to respect the goal given to the adaptation. Whatever the technology that implements the decision engine of the *decider*, the *policy* consists in a specification of this association of strategies to events.

Guide In order to specialize the *planner* to the actual component, the expert writes a guide that describes how plans are built. The purpose of a *plan* is to compose *actions* in order to achieve a *strategy* that has been decided.

In addition to knowledge about actions and strategies, writing the planification guide requires some expertise about the implementation of the component. For example, depending on the implementation, some synchronization of actions may be needed: if the component uses message passing, some of the actions such as state extraction may require that there is no on-fly message; if the component implements a parallel algorithm, some actions may require some consistency criterion over the state of the component to be satisfied; and so on.

The planification *guide* is a specification of the association of *plans*, as collections of *actions* with control flow, to *strategies*. It allows to capture the dependency to the component implementation outside the *planner*.

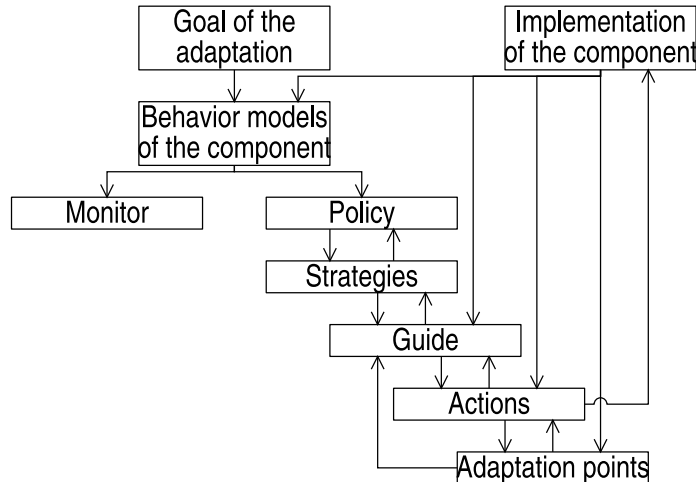


Figure 6: Dependencies between steps and entities of the design method

Actions and adaptation points In order to implement the *actions*, the expert should rely on its knowledge of the implementation of the component. The same applies to the placement of *adaptation points*, which are markers in the source code of the component.

The implementation of actions can affect the implementation of the component itself. The following examples of section 5 will exhibit this point: components will be modified in order to allow the actions to change the communicator object used for communications.

4.2 Design method

The several steps listed above are not totally ordered. Furthermore, cycles may appear between them. Figure 6 summarizes the dependencies between steps of the design method.

As shown, there are dependency cycles between several steps. Indeed, writing the policy allows the expert to determine strategies used by the adaptation mechanism; on the other side, available strategies are the building blocks for writing the policy. Similarly, the planification guide defines available strategies, whereas the collection of used strategies specifies the minimal support expected from the guide. The same kind of cycle applies to other steps as shown by the figure.

4.3 Structural decomposition

Orthogonally to the functional decomposition described in section 2, the framework shows that the different entities involved in dynamic adaptation spread over several levels of genericity.

The most generic level corresponds to entities that could potentially be reused for making adaptable any kind of component. At this level lays the entities encapsulating the major functionalities of the framework and the data flowing between those entities. It includes the *decider*, *planner* and *executor* phases, and the *event*, *strategy* and *plan* data types.

The second level, the application specific level, encompasses entities that depend on the applicative domain while being independent of its implementations. This level contains entities that specialize the framework to the component, including *policy* and *guide*.

Endly, the platform specific level indicates entities that are strongly dependent on the implementation of the component and its execution platform. This level contains *monitors* for observing the environment and *actions* that modify the component.

5 Evaluation of the work of the adaptation expert

Evaluating the method in terms of work hours and lines of code proves the feasibility of the approach. This evaluation is done for the two examples described in section 3. As those examples rely on previously existing applications, the evaluation takes into account the fact that the adaptation expert has no *a priori* knowledge about the component.

5.1 FFT benchmark

As this experiment is the first we have made, the time required for realizing the adaptation can be hardly evaluated. Indeed, this experiment superimposes with some finalization and debugging of the framework itself. Nevertheless, it can be estimated that this experiment required nearly 40 hours of full time work.

In order to make the application adaptable, the expert requires some knowledge about the application itself. The original version of the FFT benchmark consists in 2100 lines of Fortran 77 code. It is structured as a main loop containing one call to the FFT function. This function contains the sequence of calls for the six computation steps and transposition operations. This knowledge is necessary to insert calls to the framework at adaptation points and upon control structures. Those calls add 50 new lines of Fortran 77 code tangled within applicative code. In addition, the expert must provide a description of those adaptation points and control structures. This description is made of 125 lines of C++ code.

Realizing the adaptation to the number of processes requires to indirect references to the `MPI_COMM_WORLD` constant, which identifies the set of processes launched with the same command than the local one, in order to make the application able to change its communicator object at runtime. This modification impacts 15 lines of existing Fortran 77 code.

The expert must also provide functions that implement the several actions such as matrix redistribution, process creation and connection, and process disconnection and termination. The redistribution functions require the addition of 750 lines of Fortran 77 code. Functions that create and connect new processes require 250 lines of C++ code; functions for disconnecting and terminating processes require 300 lines of C++ code. Those modifications are

not tangled with applicative code. One can notice that existing libraries can be used to implement those functions.

As newly created processes must start their execution at the adaptation point at which previously existing processes suspend to perform the adaptation, the adaptation expert must realize a mechanism that allows processes to skip parts of their code. This mechanism can be implemented by inserting conditional instructions that discard the execution of the following code block if the target adaptation point has not been reached. This modification requires the addition of 60 lines of Fortran 77 code tangled within applicative code.

In order to integrate the framework within the component, an initialization phase is required before the component starts its execution. This phase sets up the adaptation framework. It is done thanks to 100 lines of C++ code. In order to take this modification into account, 5 lines of Fortran 77 code are modified.

Endly, the expert is expected to provide a decision policy and a planification guide. With our current framework implementation, those two entities are mixed. They are made of nearly 100 lines of Java code.

To summarize this experiment, the implementation of dynamic adaptability coarsely requires the addition of 810 lines of Fortran 77, the addition of 775 lines of C++, the addition of 100 lines of Java and the modification of 20 lines of Fortran 77 code. In the end, nearly 45% of the adaptable version implements adaptability, less than 8% of which is tangled within applicative code.

5.2 N-body simulator

Making the Gadget 2 simulator able to adapt to the actual number of available processors requires approximatively 25 hours of full time work. Initially, Gadget 2 consists in 17000 lines of C code. It is structured as an initialization phase, followed by a main loop. Each iteration of the loop makes the simulation progress for one time step: it invokes a load-balancing mechanism, then computes the new state of the simulated system.

The adaptable version of the simulator contains only one adaptation point at the beginning of the loop. As we have proposed in [17] a tool for automatically inserting calls upon control structures and generating the description of adaptation points and control structures, only 1 tangled line of C++ needs to be inserted.

As in the case of the FFT benchmark, adaptability to the number of available processors requires to indirect references to the `MPI_COMM_WORLD` constant. This modification impacts 164 lines of C code.

In order to implement the redistribution function, the expert can rely on the existing load-balancing mechanism. Whereas it requires that the expert investigates how this mechanism works, the expert does not need to know anything about the distribution of particles over processes. Making the load-balancing mechanism able to redistribute particles to a different collection of processes requires the addition of 55 lines of C code and the modification of 15 lines of C code. This modification is tangled within applicative code.

The expert has also to implement the other actions, those that spawn and terminate processes. Those actions are made of 525 lines of C++ code not tangled with applicative code.

The initialization phase of the component has to be redesigned by the expert. Firstly, the adaptation framework has to be initialized. This can be done thanks to 320 lines of C++ code. Secondly, the initialization of the simulator itself has to be modified. Indeed, when some processes are spawned at an adaptation, previously existing processes must reinitialize. This modification requires the addition of 120 lines of C++ code and the modification of 1 line of C code. The fact that the volume is bigger than the one for the initialization of the FFT is due to a more complex code for gathering and parsing options.

Endly, the expert is expected to provide a decision policy and a planification guide. Again, they are made of nearly 100 lines of Java code.

As a summary, making the simulator adaptable requires the addition of 1020 lines of C and C++ code and 100 lines of Java code, and the modification of 180 lines. In the adaptable version of the simulator, nearly 7% of the source code is due to adaptability. The tangling level of is under 30% of the source code of adaptability.

5.3 Discussion

A first observation that comes from the two experiments that have been presented is that for similar adaptations, the footprint of adaptability in source code volume is almost independent of the application itself. As its proportion decreases when the size of the application increases, adaptability seems to scale well to real-world applications.

Secondly, whatever the application, it appears that the required knowledge of the adaptation expert focuses on the structure of the application rather than on the application itself. In particular, the expert needs to know about control structures, data structures and distribution of the application. In the described experiments, knowledge about control structures helps in placing adaptation point and modifying the initialization phase; knowledge about data structures and distribution help in implementing actions. Interestingly, the expert can choose the granularity of the structures that are considered. Thanks to this, the expert masters the trade off between frequent adaptations and simple implementations.

Thirdly, the fact that the two experiments involve similar adaptations shows that the experience gained by the expert can be reused from one application to the others. Indeed, except few details, the decision policy and the planification guide are almost the same for the two described applications. Even the implementations of actions have been reused partly or entirely. Similarly, common modifications to the applicative code can be found: one typical example is the indirection of references to `MPI_COMM_WORLD` in MPI codes. All this shows that the work of the adaptation expert, both about designing the adaptation and implementing it, could (and should) be capitalized, potentially leading to “off-the-shelf” policies, guides and actions.

Code tangling of adaptability appears to be high. Nevertheless, it can be noticed that some of the modifications that cause tangling can be largely automated. This is typically the case of indirecting references to the `MPI_COMM_WORLD` constant. Those modifications

mostly aim at increasing the parametrability of the applicative code. Thus, it could even be expected to be present in the non-adaptable version.

One can notice that we had some experience in the area of dynamic adaptation before making those experiments. Furthermore, we have a good knowledge of our framework. Nevertheless, if it has reduced the work time, it has no influence on the volume of work to be carried out. Thus, results remain relevant.

6 Related works

Several architectures and frameworks have been previously proposed to implement dynamic adaptation, not necessarily in the context of Grid computing. Those architectures usually emphasize the requirement for reflective programming support in order to implement the adaptation actions, such as in [2, 8, 9, 13]. Indeed, reflexivity provides the means to impact the target application, wherever it is implemented: in the underlying runtime environment (like Iguana/J in the case of Chisel [9]) or in the adaptation framework itself (like LEAD++ [2] and PCL [8]). In our approach, the need for reflexivity is captured within *actions*. In facts, our model does not specify how actions are implemented; nevertheless, parametrability, reflexivity and runtime code modification are the major tools.

In addition, frameworks commonly define a domain-specific language for expressing the adaptation. The purpose of that language consists in allowing the developer to specify how the application should adapt and to which environmental changes. This language can be an imperative language, such as the one of PCL [8]. Many projects such as Chisel [9] define a declarative language as a collection of event-condition-action triples (upon the event and if the condition is satisfied, perform the action). The entity described by this domain-specific language is split into the *policy* and the *guide* of our approach. In addition, our framework does not specify neither the languages for expressing them nor the technology for interpreting them. Separating the policy and the guide isolates the goal of the adaptation (into the policy) from the modifications (into the guide).

In the context of Grid computing, remarkable projects are Grid.It [1] and GrADS [16]. Grid.It is a programming and runtime environment for Grid computing using the structured parallel programming approach. The runtime environment includes facilities for dynamic adaptation. Interestingly, it takes benefit from the known structure of parallel programs to make them adaptable. In the case of GrADS, adaptability is implemented within the middleware. It allows to reschedule and migrate applications. Those approaches based on the runtime environment and/or compilers allow to make applications adaptable transparently to developers. Thus, no work is expected from them. However, the set of possible adaptation strategies is restricted by the implementors of the runtime environment and/or compilers. On the other side, with our approach the developer accepts some additional work in order to be able to customize adaptability. The fact that this work can be reused, as it has been shown in the discussion 5.3, makes it even more acceptable.

Only few studies have been conducted about the methodology of dynamic adaptation. In [11], authors list relevant questions that developers should answer to design adaptable

software. It provides a design guide line. However, this method is not evaluated in terms of required work. Furthermore, it does not provide a clear structure of the adaptable application, potentially leading to *ad-hoc* implementations and poor reusability.

In [10, 18] formalizations of dynamic adaptation are proposed that allow to use formal methods to design adaptable applications. In [18], the formalization is based on temporal logic, giving formulae for the adaptation process. Thanks to this, adaptable applications can be modeled as temporal logic formulae that can be used to verify some properties. In [10], adaptation is modeled by a graph that describes the intermediate states of the application during the adaptation. Proving that the graph is a transitional-invariant lattice is sufficient to prove that the adaptation is correct, i.e. the adaptation is executable and leaves the application in a correct state. However, those two works do not provide help in designing the adaptation itself. They only provide tools to verify it. This approach is complementary to ours, which focuses on helps for designing the adaptation while dismissing the problem of the verification.

7 Conclusion and future work

The work described in this paper gives a model of dynamic adaptation. This model is evaluated in a manner that is complementary to the usual ones focused exclusively on performance. This work shows that dynamic adaptation does not burden that much developers, breaching common *a priori* ideas. In order to strengthen our argumentation, we are conducting a third experiment with a different application and a different kind of adaptation: changing the whole implementation of the component, including the communication scheme, from C with MPI to Java with RMI, and vice versa. We expect from this experiment a basis of actions useful for implementation replacement. If in addition it appears that some actions are common with those used to change the number of used processors, it would highlight that the work done for adaptability can also be reused across different adaptation strategies.

Up to now, our work has focused on analyzing the process of dynamic adaptation in order to model it. Our work allows to give the basis for a design method that isolates distinct concerns independently of the underlying technologies. The next steps of our work will focus on the technologies that can be used for decision-making and planification. In the same time, we will investigate which the formalisms can be used to express efficiently and easily decision policies and planification guides.

Acknowledgements

Experiments described within this paper have been done with the Grid 5000 (www.grid5000.fr) French testbed.

References

- [1] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppini, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance grid programming in the grid.it project. In *Workshop on Component Models and Systems for Grid Applications*, June 2004.
- [2] N. Amano and T. Watanabe. An approach for constructing dynamically adaptable component-based software systems using LEAD++. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *OOPSLA'99 International Workshop on Reflection and Software Engineering*, pages 1–16, Nov. 1999.
- [3] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOOP02)*, Málaga, June 2002.
- [4] J. Buisson, F. André, and J.-L. Pazat. Dynamic adaptation for grid computing. In P. Sloot, A. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *Advances in Grid Computing - EGC 2005 (European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers)*, volume 3470 of *LNCS*, pages 538–547, Amsterdam, Feb. 2005. Springer-Verlag.
- [5] J. Buisson, F. André, and J.-L. Pazat. Enforcing consistency during the adaptation of a parallel component. In *The 4th International Symposium on Parallel and Distributed Computing*, July 2005.
- [6] J. Buisson, F. André, and J.-L. Pazat. A framework for dynamic adaptation of parallel components. In *ParCo 2005*, 2005.
- [7] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.
- [8] B. Ensink, J. Stanley, and V. Adve. Program control language: a programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*, 63(11):1082–1104, Nov. 2003.
- [9] J. Keeney and V. Cahill. Chisel: a policy-driven, context-aware, dynamic adaptation framework. In *4th International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, pages 3–14. IEEE, 2003.
- [10] S. S. Kulkarni and K. N. Biyani. Correctness of component-based adaptation. In I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. Wallnau, editors, *Component-Based Software Engineering (7th International Symposium)*, volume 3054 of *LNCS*, pages 48–58. Springer, May 2004.
- [11] M. McIlhagga, A. Light, and I. Wakeman. Towards a design methodology for adaptive applications. In *Mobile Computing and Networking*, pages 133–144, May 1998.
- [12] NAS. Parallel benchmark. <http://www.nas.nasa.gov/Software/NPB/>.
- [13] P.-G. Raverdy, H. L. V. Gong, and R. Lea. DART : a reflective middleware for adaptive applications. In *OOPSLA'98 Workshop #13 : Reflective programming in C++ and Java*, Oct. 1998.
- [14] V. Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 2005. submitted astro-ph/0505010.
- [15] V. Springel, N. Yoshida, and S. D. White. Gadget: a code for collisionless and gasdynamical cosmological simulations. *New Astronomy*, 6(2):79–117, Apr. 2001.
- [16] S. Vadhiyar and J. Dongarra. Self adaptability in grid computing. *Concurrency and Computation: Practice and Experience*, 17(2-4):235–257, Feb. 2005.

- [17] G. Vaysse, F. André, and J. Buisson. Using aspects for integrating a middleware for dynamic adaptation. In *The First Workshop on Aspect-Oriented Middleware Development (AOMD'05)*. ACM Press, Nov. 2005.
- [18] J. Zhang and B. H. Cheng. Specifying adaptation semantics. In *Workshop on architecting dependable systems*. ACM Press, 2005.