MANAGING DISTRIBUTED INFORMATION FOR PERFORMANCE CONTROL OF GRID-BASED APPLICATIONS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2005

Muhammad Rizwan Mian
School of Computer Science

Contents

List of Figures

List of Tables

Abstract

Distributed technologies enable distributed resources to be harnessed. The heterogeneity and dynamic behaviour of resources presents many challenges such as discovery, characterisation and monitoring of these resources. It is particularly challenging to control the application behaviour running on a grid. The ideal would be to enable such applications to be automatically adaptive by associating them with a performance control system.

Performance control systems need performance information to make effective performance control decisions. This performance information may be generated by different components of a geographically distributed application. Effective performance information management, for example the provision of rapid access to this information by a performance control system, is an interesting challenge. This challenge has been studied by developing a performance information repository, Perfrep. This repository is configured with different geographical configurations. Perfrep has been extended to use local performance control at the distributed sites. Perfrep is evaluated by comparing the effects of different repository configurations on the performance of a distributed application. All the experiments are performed over a wide area network.

This investigation empirically demonstrates that using distributed performance information management to facilitate local performance control can significantly outperform centralised performance information management.

This work is particularly suitable for those systems that generate plentiful performance information at the distributed sites, and require the performance information both locally and on a remote site. In addition, this work is particularly valuable to those systems that require minimal overhead in managing such information.

The work presented is directly applicable to the PERCO prototype.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

"In the name of Allah, the most merciful and beneficent". All praises to Him who has blessed me with uncountable favours.

I would like to express my gratitude to my supervisors, Prof. John Gurd and Dr. Ken Mayes.

My sincerest gratitude to all the people at Transitive Ltd., especially Dr. Martyn Spinks for their positive attitude for my decision to pursue post graduate work.

I wish to thank my friend Dr. Ian Rogers for helping me find a template for thesis in Microsoft Word.

I would like to extend my gratitude to the School of Computer Science, The University of Manchester and the Overseas Research Studentship awards for their financial assistance.

My love for my parents who have brought me up with affection and care.

My sincerest thanks to Mr. Graham Gough and Ms. Alex Walker for their mentorship throughout the years. Lastly, my appreciation for my peers, relatives, colleagues etc., the people around me for support.

The Author

The author graduated from the University of Manchester in 2003 with a Bsc (Hons) in Computer Science with Industrial Experience. During his undergraduate studies, he has won many academic achievements including best penultimate year student in Computer Science. He has experience in working for a telecommunication giant, Cisco, as a software engineer for a year. Furthermore, he had been employed for two years as a software engineer by Transitive, which develops dynamic binary translators.

1

Introduction

This chapter sets the context for the problem under consideration by discussing grids, performance control systems and distributed monitoring systems in Section 1.1. The requirements of a performance information repository are discussed in Section 1.2. The research aims for the investigation are specified in Section 1.3. The chapter concludes by presenting the contribution and an outline of the results in Section 1.4.

1.1. Problem Description

This study aims to investigate information repositories of performance control systems in a grid. Firstly, the term grid is introduced, followed by performance control systems and information repositories.

A grid is a system that coordinates distributed resources including general-purpose protocols and interfaces to deliver nontrivial qualities of service (Foster and Kesselman 2004). A grid middleware provides mechanisms that facilitate operation of a grid. The key to grid is the adoption of common standards. Computational grids provide resources that can be combined together to execute large distributed applications and are becoming a dominant platform for high-performance and CPU-intensive applications (Berman et al. 2003). The target domains for a grid include science, business and industry. A grid offers ample computing cycles that make a new model of utility-computing possible, namely provision of CPU cycles, by analogy with the provision of electricity supply. That is, the consumers of computing may be able to buy computing cycles in the same way as the consumers of electricity can buy electricity. The consumer of computing would have options of choosing from various

suppliers at various prices and the possibility of buying whatever is desired.

An application that requires a lot of computing cycles is an example of a grid application. Typically, a grid application has the following characteristics:

1. A grid application can be computer-intensive and memory-intensive and may require access to many resources.

2. A grid application is distributable so that it can be executed on the grid resources concurrently.

3. A grid application is scalable so that the application can be executed on numerous grid resources.

(Foster and Kesselman 2004) give examples of possible grid applications. These include *in silico* experiments in *myGrid* project. An *in silico* experiment uses computer-based information repositories and computational analysis to test a hypothesis, derive a summary, search for patterns, or demonstrate a known fact.

Another example application is collaborative science for astrophysicists. Astrophysicists use computational tools to understand processes in the universe, from localised processes (star formation, solar physics, supernovae etc.) to collections of all these (the formation and interactions of galaxies). Consequently, the computational problems are immense and increasing beyond the capacity of a single supercomputer.

Yet another example is *The NEESgrid Earthquake Engineering Collaboratory*. NEESgrid is a grid based system

that supports a wide range of activities undertaken by engineers and researchers engaged in improving the performance of buildings and other structures when subjected to the effects of earthquakes (Foster and Kesselman 2004).

There are two obstacles to the development and deployment of applications on a grid: heterogeneity and variability of resources. The heterogeneity implies variation in static characteristics of available resources. The variability implies dynamic variation in the available resources due to, for example, contention with other users. The variability of resources also includes variable bandwidth between resources, their availability, their reliability, and so on. Thus, grid applications must be able to exploit heterogeneous resources whilst adapting to fluctuations in available performance (Berman et al. 2003).

Currently, developing applications for a grid is complex. This is evident from the fact that current grid application development requires detailed knowledge of the grid middleware and the infrastructure. *Independently parallel problems* are the simplest class of applications of the computational grids. These applications divide a problem into sub-problems, so that each separate sub-problem can be sent via Internet to different computing resources, following a *scatter-gather* model. Each of these resources work independently and without communication with the other nodes to derive its results. SETI@home and Folding@home are good examples of such applications. Issues such as failed resources are dealt with by brute force techniques, i.e. by re-running the failed job from scratch. The current grid

middleware for this class of applications cannot readily handle *connected* parallel applications, in which components of the parallel applications may require inter-component communication (Kaufman et al. 2003). A thorough discussion of problems related with current grid middleware is given in (Chin and Coveney 2004). The implementation challenges that need to be addressed include accounting, security, quality of service, scalability, reliability, robustness and usability. There exist investigations (Berman et al. 2001) to make it easier for "ordinary users" to develop, execute, and tune applications on a grid. It is worth emphasizing that this work aims to address a complete grid system. This includes grid middleware all the way up to grid application development.

A component is a unit of computation with well-defined functionality. Components can be composed together to create component-based application. A component can be individually deployed on to a hardware platform. Generally, component-based applications are distributed applications, and are good candidates for computational grids.

Computational steering is the process of directing the execution of an application on the computational grid. There are two kinds of computational steering: *application steering* and *performance steering* (Vetter and Schwan 1999).

1. Researchers use application steering to investigate phenomena during the execution of their application; therefore, it is interactive. On the other hand,

2. performance steering or performance control attempts to improve the performance of an application by tuning application and resource parameters during execution.

Consequently, performance control is autonomous. As noted above, component-based applications can be executed in a computational grid. The heterogeneous nodes in a computational grid use grid middleware to inter-operate and execute the application. Such middleware, in its simplest form, consists of security, resource management and information services. A resource scheduler is central to such middleware. A resource scheduler is responsible for allocating grid resources to applications submitted for execution.

This thesis deals with work related to performance control.

It is worth re-iterating that the performance control systems automatically tune the performance of an application, at *run time*. The alternative approach is *compile time* optimisation based on static data. Static and dynamic approaches are not mutually exclusive. Previous work in this area has noted that "...adaptive run-time scheduling outperforms compile-time scheduling" (Berman et al. 2003). The performance behaviour of an application instance may be hard to predict at compile time. There are many factors which may lead to this difficulty. These factors may vary from different data that the application is processing, to different characteristics in the execution environment. The ability to observe application behaviour is at the heart of performance control.

1.1.1. Performance control systems

Typically, the basic operation of a performance control system can be described in three stages:

1. Instrumentation: instrument application code with calls to performance control system. Such an instrumentation can be manual or automatic. Instrumentation provides *sensors*.

2. Monitoring: Sensors associated with an application observe and report on performance information metrics. This information is transported to the performance control system.

3. Optimisation: The performance control system modifies the behaviour of the application based on the performance information. The performance control system uses *actuators* to tune the behaviour of the application.

The optimisation is based on the monitored information. This optimisation leads to new performance information. The new performance information may lead to a better optimisation, and so on. This is *closed-loop* control.

This thesis deals with an aspect of the monitoring process -- distributed data storage architecture.

The monitoring of a geographically distributed component-based application is critical to accurate performance control. Each distributed component of this application needs to be monitored. This need for monitoring presents a challenge to ensure the availability of remotely monitored information to a performance control system. This challenge is further

complicated when distributed components exist in a dynamic environment, such as a grid.

1.1.2. Distributed monitoring systems

Distributed monitoring systems monitor the state of the distributed application and/or infrastructure. These systems are general purpose. The monitored information may not necessarily be performance information. The clients of a distributed monitoring systems are diverse in nature. A client may be a visualisation tool, or a scheduler looking for a particular resource. In fact, a client of a monitoring system could also be an optimisation component of a performance control system, that is a performance *controller* or *steerer*. Therefore, a distributed monitoring system must provide performance information to enable performance control.

1.1.3. Similarity between performance control systems and distributed monitoring systems

Performance control systems and distributed monitoring systems both need to monitor and manage performance information and performance history in some repository. There is a need for information storage underpinning both these systems.

Both of these systems may be viewed as a producer-consumer architecture. Typically, *producers*, implemented as sensors, are entities that produce information. Such producers list the service they provide in a directory service. *Consumers*, also known as clients, of performance information then query the directory service to find a suitable producer of information in which they are interested. Thereafter, the information is shipped from the

producer to the consumer directly, with or without processing. A producer can also be an actuator; it monitors data but can be instructed to also modify the behaviour of the execution entity. Typically, the benefits of such an architecture include flexibility, which allows the producer and the consumer of the data to be bound at run-time. Another benefit is generality, that is, the consumers of the data could be anything from a performance controller to a visualisation engine. The producer-consumer architecture does not impose additional software pre-requisites on a grid resource. This is due to the fact that performance information is shipped from producer to consumer, and therefore there is no need for a local repository or database engine at the producer.

However, there are potential problems with such a producer-consumer architecture. Performance control systems need fast access to performance information to make performance control decisions and to maintain performance. Performance information is shipped from producer to consumer, in this case a performance controller. Typically, this performance controller is not on the same computing resource as the producers, and is therefore a remote controller. This configuration dictates that performance control of a component cannot be performed in isolation, as there is no local performance control. Furthermore, this architecture requires that performance information needs to be frequently shipped from producer to consumer. If the network is slow then the transfer latency of performance information could be high. This could be so high that the remotely executing components wait most of the time for the performance control decisions.

It is obvious that there is a need to investigate an optimal storage for a performance control system in a grid environment. The scope of this research is to study various alternatives for such storage, including how such storage can facilitate local or component-level performance control.

1.2. Requirements for a performance information repository

The following is a list of the requirements underlying a performance information repository needed for performance control. This list was extracted by performing a review of the published literature.

1. **Rapid availability of monitoring information:** Performance control systems control an application at run-time by tuning performance-related factors of the application. Accurate performance control requires performance information about the application (Mayes et al. 2003) and rapid access to such information is critical for performance control. Performance control systems thus must have fast access to the latest performance information.

2. **Non-intrusive:** There is a trade-off between intrusiveness and measurement accuracy. The management of performance information and the repository system must minimally impact the resources being monitored (Wolski et al. 1999). In other words, the resources (both time and space) should be used mainly for the application itself and should spend as little time as possible in managing the performance information repository. At the very least, the consumption of

resources by sensors should be low relative to the consumption of resources by the application.

3. **Minimal software demands on distributed resources:** The performance information repository should not put unnecessary software demands on a performance control system. For example, a performance information repository could require a database to be installed on every remote resource. This may be possible for a limited number of resources that participate in the same resource pool for a grid. However, a grid is highly dynamic. Resources, participating in the grid, may vary in number. Consequently, the software demands should be the bare minimum required by a resource to participate in the grid.

4. **Optimised for performance information:** There are two types of performance information: dynamic and static. Therefore, the performance information repository must have two parts: dynamic and static. The performance information generated at runtime, i.e. dynamic, has unique characteristics: dynamic performance information is relevant for a short period of time; is updated more than it is queried; and is stochastic (Tierney et al. 2002). The dynamic part must be optimised for these characteristics. The static part is discussed later (point 6).

5. **Adaptive:** The performance information repository must be able to adapt to the heterogeneous nature of a grid. The adaptivity has two aspects: infrastructure and robustness. Infrastructural adaptivity implies that the performance information repository must be responsive

to changes in the infrastructure; e.g. a performance generating entity may be re-deployed to a different resource by some scheduler. Robustness implies that the performance information repository must be able to recover from a crash. Some systems are constructed under the assumption that failures are the norm (Czajkowski et al. 2001).

6. **Post-mortem analysis:** The run-time or dynamic performance information gathered for a particular application execution instance could be post-processed to produce static performance information (Mayes et al. 2003). Such information would be used for future runs of the application. The performance information repository must be able provide post-processing or post-mortem analysis, and storage of static performance information into the static part of the performance information repository.

7. **Relevant information in repository:** The performance information repository should only hold the information necessary for performance control of the application currently under consideration. For example, the execution times of the components of an application may suffice for the needs of the performance control system. In such a case, other information, such as memory usage, may be irrelevant. If never required at run-time or for post-mortem analysis, it is unnecessary to store irrelevant information. Storing irrelevant information will only add to the run-time overhead of managing it. Therefore, there is a need for a performance control system to specify the particular performance

information that is relevant to it. This could be used to tailor the storage of performance information in the performance information repository.

8. **Security:** Performance information must be handled securely end-to-end (Tierney et al. 2002). It is necessary to ensure the integrity and privacy of data. This is because performance information can give insight into a grid application. Consider a financial simulation that is highly sensitive to a bank. Any insight into such an application may be exploited. The security should deal with the generation of performance information at distributed resources, the transfer and the storage of performance information on trusted recipients. Storage must also be secure for static performance information.

1.3. Overview of the thesis

The aim of the research reported here is to investigate different methods of managing geographically distributed performance information and providing it to the performance control system. In addition, the research considers the presence of component-level performance control on computing resources which are executing the application being tuned. This implies that information repositories local to the component-level performance controller may be required.

Firstly, a set of requirements have been identified (see Section 1.2) which are necessary for a performance information repository to effectively manage performance information. Security is acknowledged as an important repository requirement by many systems. However, it is not

implemented in practice, in general. Therefore, this requirement is not discussed further.

The rest of thesis is structured as follows.

Chapter 2 presents background on the existing systems that may have a performance information repository component as part of their architecture, or present useful ideas for a performance information repository. The suitability of a conventional database as a performance information repository is also discussed.

An emulator, Perfrep, based on the PERCO design, is used to study effective performance information management in a distributed environment. In Chapter 3, the PERCO design is introduced and described to set the context for Perfrep. The roles of various building-block of the PERCO design are specified. Different levels of run-time performance control are introduced. Then, implementation details of all the Perfrep configurations are presented. Possible implementation issues with these configurations are highlighted.

In Chapter 4, the Perfrep configurations described in earlier chapters are evaluated. In particular, two requirements: **rapid availability** and **non-intrusive** are empirically studied in a WAN environment. The rest of the requirements are discussed but are not empirically studied due to limited time available for this investigation. The Perfrep configurations are evaluated against the requirements laid out in Chapter 1.

Chapter 5 concludes by discussing the benefits, limitations, and suitability of Perfrep. Possible future work is presented.

1.4. Contributions

This is the first study that presents an implementation of managing performance information at distributed resources that execute components of a component-based application. In addition, this investigation empirically demonstrates some effects of distributed performance information management and component-level performance control at the distributed resources.

2

Background

This chapter is divided into two parts. The first part discusses existing systems: (a) that may have a performance information repository component as part of their architecture, or (b) that present useful ideas for a performance information repository. The second part discusses the suitability of conventional databases that can be used for implementing a performance information repository. For each system, a brief description of its functionality is presented, followed by any performance information management issues addressed by the system. Typically, performance information management is a "building block" in the overall system. Therefore, it is not necessarily discussed in detail in the published literature. These systems are reviewed in the context of the requirements introduced in Chapter 1. The compatibility of these systems with the requirements falls into five categories:

1. Fully met: the system provides an implementation that is fully compliant with the requirement, or the system provides a strong indication that the implementation fully meets the requirement.

2. Partially met: the system implementation has the foundation that can be easily extended to meet the requirement.

3. Not met: the system implementation does not meet the requirement and does not have the foundation to meet the requirement.

4. Not discussed: the cited literature does not discuss the information repository or its implementation; therefore the requirement is probably not met.

5. Not clear: it is unclear if the system meets the requirement. The issues with managing information are discussed. However, it is difficult to make a judgement that the system meets or does not meet the requirement based on the discussion provided.

6. Not applicable: the requirement is not applicable to the system.

The chapter concludes by presenting a table summarising the likely ability of each system to meet these requirements.

Typically, the following four types of systems have an information repository:

1. Performance Control Systems
2. Distributed Monitoring Systems
3. Performance Modelling and Analysis systems
4. Web-services

This study aims to investigate aspects of performance information repositories needed for performance control. Therefore, it is useful to highlight where performance control fits in the bigger picture of running component-based applications on a grid. Generally, there are three activities supported by systems that are intended to enable a comprehensive approach to running multi-component applications on a computing grid (Mayes 2005). These are:

1. application programme development,
2. application execution, and
3. run-time control of the application.

Most performance control systems support, at least, the third activity. Distributed monitoring systems, in general, facilitate or supplement the second activity. Performance modelling and analysis systems are aimed at developing models for optimised execution. These systems are normally used for static or load time analysis of the distributed application. Therefore, these systems can be classified as post application development but pre-execution. Performance

control systems, distributed monitoring systems, and performance modelling and analysis systems are aimed at distributed systems. These systems can be and are being turned into web-services. Once these systems are available for web-services they would probably still support the same activity as they currently support in conventional distributed environments.

2.1. Performance control systems

Performance control systems perform runtime tuning of an application. Performance control systems have been introduced in Section 1.1. The basic operation of a typical performance control system can be divided into three stages:

1. Instrumentation: application code is instrumented, manually or automatically, with calls to the performance control system. Instrumentation can also provide *sensors*.

2. Monitoring: The sensors observe and report on performance information metrics. This information is transported to the performance control system, following a producer-consumer architecture (see Section 1.1.3).

3. Optimization: The performance control system tunes the performance of the application. The tuning is based on performance prediction. There is a great range of performance prediction methods. Generally, these methods can be divided into two categories (Mayes et al. 2003):

    a. Heuristics-based methods: these methods apply heuristics to performance history in order to select changes that may improve application performance. The performance prediction is implicit in the selection decision based on heuristics. Such heuristics-based decisions may be more

appropriate where a performance model for the whole application is not available.

b. Performance model-based methods: an analytic performance model is derived which can be parameterised by application and system characteristics. This model is used to generate performance predictions for specific values of application and system parameters.

Using either of these categories of methods, the performance control system instructs *actuators* to modify the behaviour of the application.

Typically, a performance control system is based on closed-loop control. Pertinent performance information lies at the heart of fast and accurate performance control. Therefore, the provision of fast access to performance information is critical to a performance control system. Some of existing performance control systems are now discussed.

2.1.1. PERCO

PERCO is a performance control system for controlling a connected application or an application consisting of interacting coupled components.

The PERCO design (Mayes et al. 2003) makes a two-level distinction for controlling an application consisting of components. Local or component-level performance control, using a component performance steerer (CPS), is concerned with the tuning of a component only by adjusting the control parameters of a component at run-time. Global or application-level performance control, using an application performance steerer (APS), is concerned with re-deployment of components on the underlying computing resources in

order to achieve better performance of the overall application at run-time. As there can be many components in an application, each component is associated with a local CPS. Each component reports to its local CPS for component-level performance control. In contrast, there is only one APS to which all the CPSs report for application-level performance control. The APS and the CPSs are deployed on different computing resources. A detailed discussion of PERCO is presented in Chapter 3.

The PERCO prototype (Mayes et al. 2004) requires manual instrumentation of the application components. The PERCO prototype implements the second and the third operational stages of a typical performance control system.

The following is a review of performance information management of the PERCO prototype in the context of the requirements laid out in Chapter 1.

Rapid availability: The PERCO design allows for a central repository for an APS and local repositories for the CPSs. However, the PERCO prototype implements a central repository that is only associated with the APS. In the prototype, APS-CPS communications are used to send performance information generated at a component to be stored in the central repository at the APS. The PERCO prototype has been so far used only with applications whose components are incapable of component-level performance control. Thus, in the PERCO prototype, there is no component-level performance control. Consequently, only the APS requires access to the performance information for application-level performance control. Therefore, this central repository configuration is adequate for serving the APS in the prototype.

However, there may be problems associated with a centralised repository configuration if CPSs require rapid access to performance information. This would be the case, for example, where the CPS is undertaking component-level performance control. In this situation, the CPS could benefit from access to locally stored performance information. The current PERCO prototype configuration dictates that performance information be fetched from the centralised repository. However, it is anticipated that local repositories can be introduced easily in CPS to meet this requirement. Therefore, this requirement is only partially met.

Non-intrusive. The performance information is piggybacked onto the status information from a CPS to the APS and does not introduce additional intrusiveness, thereby meeting this requirement. However, the single communication channel for passing both status and performance information breaks the good software engineering practice of separation-of-concerns.

Relevant information: With the centralised repository configuration there is an issue about how much performance information from a CPS should be sent to the APS for storage. Some of this information might be irrelevant for application-level performance control, but exists in the central repository for completeness, in the current prototype (thereby violating this requirement).

Optimised: The performance information repository is currently implemented with a memory cache, for short term storage. Long term storage uses BerkeleyDB (Olson et al. 1999), with B-Trees as the storage data structure. A tree data structure is not optimal for insertion. The out-of-date data is not removed at present. The memory cache is implemented using a shared memory segment between the APS processes. However, the cost of accessing the cache

(updates and retrieval) is unclear. Whether this requirement is met is thus unclear.

Minimal demands: In the PERCO prototype, the CPS and the component are compiled into the same binary image. In addition, a CPS does not have a local repository. This setup indicates that there are minimal software demands on the distributed computing resources, thus satisfying this requirement.

Post-mortem analysis: Currently, the information repository stores performance information across multiple executions. However, the current PERCO prototype does not perform any post-mortem analysis on this information for the generation of static performance information. It is envisaged that the PERCO prototype can be easily extended to perform post-mortem analysis, thereby partially meeting this requirement.

Adaptive: If a component is re-deployed or migrated, the performance information repository can deal with this, indicating infrastructural adaptivity. However, fault-tolerance was considered to be outside the scope of the initial PERCO prototype. Therefore, this requirement is only partially met.

2.1.2. Autopilot

Autopilot is an infrastructure for dynamic tuning of application performance on heterogeneous computational grids (Ribler et al. 1998).

Autopilot either assumes presence of sensors and actuators in the application, or requires manual instrumentation i.e. an application developer places sensors and actuators in the source code, specifying the program variables that sensors should monitor, and the control points that can be modified by actuators (Vetter and Reed 2000).

The Autopilot toolkit implements the second and third stages of a performance control system i.e. execution and control. Autopilot follows the typical producer-consumer architecture. A directory service is present that lists the sensors available. The *clients* can use this directory service to find sensors with certain characteristics. The sensors extract qualitative and quantitative performance information from the executing application and forward the information to the registered clients. Actuators can be used to modify the behaviour of the application. Autopilot provides a fuzzy logic engine that accepts sensor inputs, "fuzzifies" the values for rule application, computes the relative truth of each rule, and "defuzzifies" the rule consequent to activating remote actuators. An Autopilot manager coordinates connections between sensors, actuators and clients. It is unclear from the literature (Ribler et al. 1998; Vetter and Reed 2000) whether the fuzzy logic engine is implemented as part of the Autopilot manager.

The software components of the Autopilot toolkit have been implemented in a policy-independent and platform-independent infrastructure. This independence is based on the Nexus communication substrate (Foster et al. 1996), which provides a global address space that encompasses all processes executing on a network.

Repository requirements review

The following is a review of the performance information management capabilities of Autopilot in the context of the requirements laid out in Chapter 1.

Minimal demands: The policy and platform independence are based on the Nexus communication substrate. This implies minimal software prerequisites on grid resources. This requirement is thus met.

Non-intrusive: Autopilot uses sensors which extract performance information from an executing application. Sensors are sophisticated; e.g. they can be activated or deactivated depending on application demand. This implies intrusiveness may be relative to application demand, hence meeting the requirement.

Relevant information: Another feature of Autopilot is called *Data Reduction*. This feature allows data transformation for data reduction at a sensor site before transfer. An example transformation might be to compute an average performance data value. Such a feature has two possible implications: reducing network traffic and converting performance information into more relevant performance information before storage, hence meeting this requirement.

Rapid availability: The fuzzy logic engine tunes an application. The consequences of stale data are recognised by Autopilot. The published experiments demonstrate that the researchers of Autopilot are aware of communication overhead. These statements strongly indicate implicit fulfilment of this requirement.

Optimised: The cited Autopilot literature does not discuss how performance information is stored, so this requirement is probably not met.

Adaptive: Autopilot advocates that the performance control systems should apply real-time adaptive control techniques to adapt dynamically to changing application resource demands and system resource availability. Autopilot provides a directory service. The directory service is used in conjunction with dynamic binding of producers and consumers of performance information, and reflects adaptivity to infrastructure dynamics. For example, in the case of a temporary network failure, a consumer may be able to re-establish communication with a producer when the network eventually

becomes available. The implementation, and therefore the robustness, of the information repository is not discussed in the available literature. So, infrastructural adaptivity is present, but lack of a discussion of the robustness of an information repository implies that this requirement is only partially satisfied.

Post-mortem analysis: Once a client finds a suitable sensor through the directory, performance information is directly transferred from sensor to consumer. It might be possible to use a client to store data across multiple executions of the application. Alternatively, the Autopilot manager might be extended to store data and perform post-mortem analysis. It seems that either of these alternatives requires a simple extension. Therefore, Autopilot only partially meets this requirement.

2.1.3. AppLeS

The AppLeS (Application Level Scheduling) project suggests a methodology, application software, and software environments for adaptive scheduling and development of applications in heterogeneous and multi-user grid environments (Berman et al. 2003).

An application that needs to be performance controlled is manually instrumented so that it can be dynamically scheduled by an AppLeS scheduling agent. Each application is provided with a customised scheduling agent for dynamic monitoring and scheduling of the application.

The AppLeS scheduling agent follows six steps in a loop, thereby forming a closed loop control. These steps include Resource Discovery, Resource Selection, Schedule Generation, Schedule Selection, Application Execution and Schedule Adaptation.

Generally, a performance control system may assume that resources are allocated, by some "external resource scheduler", to execute the application. In such a case, the performance control system is only concerned with improving application performance using previously allocated resources. Hence, such a performance control system does not need to discover resources. However, AppLeS includes its own scheduler instead of assuming the presence of an external scheduler: the Resource Discovery and Resource Selection stages in the AppLeS methodology involve discovering and selecting the resources that can be used to execute the application. In addition, the Resource Selection stage employs performance predictions of dynamically changing system variables.

At the Schedule Generation stage, AppLeS scheduling agent uses a performance model to determine a set of candidate deployments for the application on potential resources. The optimal deployment for the application is chosen at the Schedule Selection stage.

The application is deployed and executed with the chosen schedule at the Application Execution stage.

In the Schedule Adaptation stage, the AppLeS agent accounts for changes in resource availability by looping back to the Resource Discovery stage.

Therefore, the optimisation stage of a general performance control system is implemented in the Schedule Generation, Schedule Selection, Schedule Adaptation and (partially) Resource Selection stages.

AppLeS has been used for an application, SARA (Caltech 2005), that provides access to satellite images in distributed repositories (Berman et al. 2003). The Simple AppLeS SARA (Su et al. 1999) demonstrates the AppLeS Resource Selection stage. The Simple

AppLeS SARA uses monitoring services provided by a general purpose distributed monitoring system, the Network Weather Service (NWS) (Wolski 1998; Wolski et al. 1999). NWS is discussed in Section 2.2.3. The monitoring stage of a general performance control system is mapped onto the Resource Selection stage.

Originally, any application that needed to be tuned with AppLeS needed to be instrumented ad hoc by hand. However, to facilitate the instrumentation process, AppLeS provides templates to integrate a new application into AppLeS to form a new self-scheduling application. One of these templates is APST (AppLeS Parameter Sweep Template) (Berman et al. 2003).

The APST software is implemented as two distinct processes: a daemon and a client. The responsibility of the daemon is to deploy and monitor the applications. The client is a portal that can be used periodically, either interactively or from scripts, to interact with the daemon for submitting requests for computation and checking application progress. The AppLeS scheduler, being a central component of the daemon, makes all distribution decisions mapping resources to application tasks and data. This AppLeS scheduler has three components: a data manager, a computer manager and a meta-data manager. The data manager and computer manager are used to launch and monitor data transfers and computations. They also record the history of resource performance and provide the scheduler with that history information. The third component, the metadata manager, uses information services actively to obtain published information about available resources (e.g. CPU speed information

from MDS[1]). A predictor uses information from these three components to compute forecasts.

Repository requirements review

The following is a review of the performance information management capabilities of APST in the context of the requirements laid out in Chapter 1.

Rapid availability: It is unclear if the scheduler and the predictor, in particular, are invoked frequently in order to impose a requirement of rapid availability of information to these components. Therefore, this requirement may not be applicable.

Optimised: The information storage in the data manager and the compute manager is not discussed in detail. Therefore, this requirement is probably not met.

Non-intrusive, minimal demands and relevant information: The monitoring implications, intrusiveness or software requirements on the system being monitored are not addressed, along with relevancy of performance information. Therefore, these requirements are probably not met.

Post-mortem analysis: If the meta-data manager can be used to manage static performance information, then this requirement may be achieved.

Adaptive: APST implements resilience by using a check-pointing mechanism to easily recover with minimal loss for the application. This implies that the performance information may be resilient too. The grid resources are accessed via deployed middleware, so the

---

[1] Monitoring and Discovery System, a grid information service.

infrastructural adaptivity is unclear. Therefore, APST implicitly provides performance information resilience; but there is no discussion of infrastructural adaptivity thereby partially satisfying this requirement.

2.2. Distributed monitoring Systems

Distributed monitoring systems were introduced in Section 1.1.2. Such systems monitor the state of the distributed application and/or infrastructure. They tend to be general purpose. For example, they provide information about the state of resources. A typical client of a distributed monitoring system may use this information for visualising the state of a distributed application. These systems are not aimed at a particular use, such as performance control, where there may be stringent constraints on the rapid availability of the information.

As discussed in Section 1.1.3, performance control systems and distributed monitoring systems, generally, follow a producer-consumer architecture. A producer, usually a sensor, registers with a directory service. The consumers consults the directory service for a suitable producer. The directory service passes the handle to the relevant producer, if there is one. Thereafter, the data is directly transferred from producer to consumer.

2.2.1. R-Grid Monitoring Architecture (R-GMA)

The R-Grid Monitoring Architecture (R-GMA) (Cooke et al. 2003) is a general purpose monitoring system based on the Grid Monitoring Architecture (a producer-consumer architecture). Grid Monitoring Architecture (GMA) (Tierney et al. 2002) outlines one possible approach towards monitoring resources and applications in a distributed environment. GMA is an architectural description, not an

implementation. GMA aims to serve a wide range of systems that may require monitoring information, and hence is general purpose. This is reflected in the implementation of R-GMA. Typical uses identified for R-GMA include a visualisation tool used to monitor the status of a compute job. Such a tool may require very coarse-grained and high-level information. In contrast, a performance control system typically requires detailed performance information in order to make accurate decisions. Scalability and performance have also been identified as requirements for R-GMA.

R-GMA implements a global schema as a vocabulary for the interaction between producers and consumers. A global schema may assume complete knowledge of the system. It is argued (Czajkowski et al. 2001) that users should not be provided, in general, with a consistent view of global state. Moreover, it is unclear how such a schema would dynamically cope with new monitoring data types that have not been encountered before. Nevertheless, a question arises whether a global schema is an unnecessarily overweight to handle simple performance information.

Repository requirements review

The following is a review of the performance information management capabilities of R-GMA in the context of the requirements laid out in Chapter 1.

Rapid availability: R-GMA aims to service queries with different temporal characteristics. However, it is unclear if such temporal characteristics can serve rapid access to performance information by clients --- it is unclear whether the requirement of rapid availability is met.

Adaptive: As part of the scalability requirement, it is identified that the R-GMA monitoring system should scale to handle the large

amounts of data published and still return answers in a timely manner. The system should also be robust to failure of any of its components. In addition, there should not be any central control since resources will be contributed by organisations that are independent of each other. All these characteristics point to adaptivity of the system, hence satisfying this requirement.

Post-mortem analysis: The requirements identified for R-GMA distinguish between dynamic and static data pools. The static data pools may be extended to be used for static meta-data, partially satisfying this requirement.

Optimised: It is not discussed whether the dynamic component is optimised for characteristics of the information stored, in particular for performance information. Therefore, this requirement is probably not met.

Relevant information: Storing only relevant information in the repository is not discussed.

Non-intrusive and minimal demands: The intrusiveness and software demands placed on a producer are not discussed, indicating that these requirements are probably not met.

2.2.2. Monitoring & Discovery System (MDS)

Grids enable large-scale sharing of resources within formal or informal consortia of individuals and/or institutions (virtual organisations). These settings present a challenge for the discovery, characterization and monitoring of resources, services and computations. “Hence, information services are a vital part of any Grid software infrastructure, providing fundamental mechanisms for discovery and monitoring, and thus for planning and adapting application behaviour” (Czajkowski et al. 2001). Czajkowski et al.

(2001) present an information services architecture that addresses performance, security, scalability and robustness requirements. This architecture has been implemented as MDS-2[1].

Such information services find uses in a variety of grid scenarios and hence seem to be general purpose. For performance control, an application adaptation agent is most relevant. It monitors both a running application and the external resource availability, and performs application tuning or rescheduling.

Repository requirements review

The following is a review of the information management capabilities of MDS in the context of the requirements laid out in Chapter 1.

Rapid availability: Requirements have been defined for the grid information service. It is mentioned that any information delivered to a client might be "old". A producer is required to model the confidence of its information, for example via timestamps or time-to-live. Such notions may help a performance control system to decide if the performance information is sufficiently recent to base a performance control decision on it. However, it is implicit that there are times when this information is too dated to be any useful for a performance control system. In addition, there seems to be no attempt to ensure rapid availability of information. Therefore, this requirement is only partially met.

Non-intrusive and minimal demands: Another requirement for a system like MDS is that it needs to manage failures well. Individual entities and the networks providing access to these entities may fail. Therefore, information services are expected to be robust. One of the

---

[1] MDS exists as part of the Globus toolkit.

strategies identified for this is that the information service should be decentralised, with providers possibly located on the entities they describe -- the intrusiveness and software demands of doing this are not discussed.

Adaptive: As described above, an information service component must be constructed under the assumption that failures are the norm, not the exception. These strategies are indicative of infrastructural adaptiveness and robustness, thus satisfying this requirement.

Relevant information: The notion of information relevance has been discussed above by providing a model of confidence. However, this is not discussed in detail. Hence, this requirement is classified as partially met.

Optimised and post-mortem analysis: There is no discussion on how information may be stored and whether the storage would be optimised for storing performance information**.** There is also no discussion on whether experience of executing on a particular system is processed into static performance information for the future**.**

2.2.3. Network Weather Service (NWS)

The Network Weather Service (NWS) aims to provide accurate forecasts of dynamically changing performance characteristics from a distributed set of computing resources (Wolski et al. 1999).

The NWS consists of four different component processes: a *Persistent State* process, a *Name Server* process, a *Sensor* process and a *Forecaster* process. There can be many instances of these processes to serve many purposes, for example, to adapt to lossy network connections. A Sensor process collects performance measurements from a specified resource. There are different kind of Sensor

processes taking different measurements e.g. CPU Sensor process, Network sensor process etc. A Name Server process provides a directory capability used for binding process and data names with low-level contact information (e.g. TCP/IP port number, address pairs). A Forecaster process predicts deliverable performance during a specified time frame for a specified resource. The most relevant NWS component in this context is the Persistent State process which manages measurement to/from persistent storage. For robustness, all NWS processes are stateless.

The NWS processes can be distributed across the distributed resources. The Sensor processes perform monitoring of both network and processor characteristics and send this information to the Persistent State processes.

Repository requirements review

The following is a review of the performance information management capabilities of NWS in the context of the requirements laid out in Chapter 1.

Non-intrusive: NWS must minimally load the resource it is monitoring. This point led to the requirement of non-intrusiveness in Chapter 1. It is clear that there is a trade-off between the intrusiveness of a monitoring technique and the measurement accuracy achieved. NWS Sensors use heuristics adaptively to adjust the frequency of active probes. This requirement is met.

Post-mortem analysis: NWS claims that the utility of (dynamic) performance information is valid for a short period of time. Therefore, NWS itself does not maintain any performance information indefinitely. It is assumed that permanent archiving of data happens outside NWS. This indicates post-mortem analysis can be added to NWS to satisfy this requirement. This indicates that this

requirement is only partially satisfied in the sense that mechanisms are there that could implement it.

Adaptive: NWS should be accessible from all potential execution sites within a distributed resource set. In addition, it aims to monitor and forecast the performance of all the available resources. This goal implies that NWS must respond to the dynamic nature of distributed resources, i.e. infrastructural adaptivity. Since NWS does not maintain performance information indefinitely itself, there is no discussion of robustness of managing performance information. Infrastructural adaptivity but no discussion on robustness implies that this requirement is at least partially satisfied.

Rapid availability: A Forecaster process can be used as a proxy to a control system. Therefore, a Forecaster process must require rapid access to performance information. It is unclear if the Forecaster, Persistent State and Sensor processes are distributed to minimise the latency of performance information availability to the Forecaster Process. In addition, data archive is performed outside the NWS. For increasingly accurate and detailed predictions, the prediction algorithm may require a longer history than provided by the Persistent State processes. Such history can only by provided by added clients that provide a data archive. The meeting of the requirement for rapid availability is unclear.

Optimised: A Persistent State process manages performance information in a circular queue in a file. It is unclear whether such a file is optimal to store performance information.

Relevant information: A Sensor is configured with the location of the Persistent State process to store information. The Sensors contact Persistent State processes to store the information. Since there are different kind of Sensor processes, Persistent State processes can be

used to store information from a particular type of Sensor. This increases relevancy of information. A Forecaster process requests the relevant information from a Persistent State process for a forecast. This requirement is fully met.

Minimal demands: The CPU Sensor process uses Unix system utilities *uptime* and *vmstat* for performance information. This implies that NWS demands these utilities be present on the system being monitored. These system utilities may not be present on some platforms, such as Microsoft Windows. Hence, this requirement is not met.

2.3. Performance modelling and analysis systems

Typically, performance modelling and analysis systems are used for static or load time analysis of the distributed application. For example, a performance modelling system might be used to find an optimal deployment of an application on the grid resources before execution of the application commences. Such deployment may be based on static analysis of the application requirements against the abilities of the grid resources. In such a case, rapid availability of performance information, if generated at all, is not as critical as in a performance control system, where the performance depends on fast-yet-accurate re-deployments.

Examples of performance modelling and analysis systems include PACE (Cao et al. 2000) and POEMS (Adve et al. 2000). In the introduction to this chapter, three activities were identified for a comprehensive approach to running a component-based application on a computing grid; these are: application program development, execution, and run-time control.

PACE generates a performance model which can be used on-the-fly for selection of system and implementation. This system has been used in the context of grid resource scheduling (Mayes 2005).

POEMS supports all activities of the comprehensive approach to running a multi-component application on a computing gird. It is a language-driven and compiler-driven system (Mayes 2005).

A compiler/interpreter analogy is relevant for typical performance modelling and analysis systems, which are aimed at pre-execution. Performance modelling systems can be thought of as analogous to compilers, where the time taken to compile the application does not matter as long as it is not embarrassingly large. On the other hand, performance control system can be thought of as an interpreter that is sensitive to the time spent on optimising the application code. Spending too much time in optimising may imply not spending enough time in the execution of the application, and hence the overall application execution time does not improve.

Therefore, it seems that the requirement of **rapid availability** is not applicable, or at least not stringent, in performance modelling and analysis systems. Also, many of the requirements set forth in Chapter 1 **(non-intrusive, minimal demands and adaptive)** may not be applicable for static analysis.

There may be no intrusiveness involved on the resources that execute the distributed application, since the analysis can be performed prior to application execution. Therefore, non-intrusiveness is not relevant here. Again, as the static analysis is performed prior to the application execution, there are not any run-time software demands on the grid resources. Minimal demands is not thus applicable here. However, if the static analysis is based on the performance

information of the previous executions of the application, then intrusiveness or minimal demands may become relevant.

Assuming the requirement of rapid availability is not stringent, then providing data structures that are optimised for performance information (optimised) may not be as beneficial as in a performance control system.

Since many requirements highlighted in Chapter 1 may not be relevant or applicable, the information repositories used by performance modelling and analysis systems are not further discussed.

2.4. Web-services

Performance control systems, monitoring systems, and performance modelling and analysis systems are aimed at distributed systems. These systems can be and nowadays are being turned into web-services. Once these systems are available as web-services they will probably still support the same kind of activities as they support in distributed environments.

There are performance control systems for web-services (Levy et al. 2003; Chung and Hollingsworth 2004).

Example uses of monitoring information include upholding the commitments and contracts of various entities participating in web-services (Sahai et al. 2002), and performance studies for cluster and grid architectures (Prodan and Fahringer 2003).

There also exists an attempt to decentralise web-service workflow in order to minimize the communication cost and maximize throughput (Nanda et al. 2004). This work is based on partitioning a BPEL program represented as a program dependence graph. BPEL (Business Process Execution Language) is a high-level distributed

programming language for creating combined web-services. A program dependence graph consists of control dependence and data dependence edges superimposed on the same set of nodes of control flow graph.

Since all three kinds of system (performance control systems, monitoring systems, and performance modelling and analysis systems) can exist in web-services, it is difficult to generalise how web-services meet the requirements set forth in Chapter 1. This is because some of these systems, being performance control systems, would demonstrate the need for rapid availability of performance information. On the other hand, performance modelling and analysis systems do not require rapid access to performance information for pre-execution analysis of an application. Therefore, it is unfair to say that web-services, in general, do require rapid access to performance information. The comparison of any web-service based system against the requirements laid in Chapter 1 is on a category basis. There may, at least, be three categories:

1. performance control systems for web-services,
2. monitoring systems for web-services, and
3. performance modelling and analysis for web-services.

The following example is used to study an information repository in an existing monitoring system in web-services. A comparison is made between such a repository and the requirements in Chapter 1.

Generalising these categories and reviewing other systems in these categories are not covered here.

2.4.1. A web-service based experiment management system for a grid (ZENTURIO)

ZENTURIO (Prodan and Fahringer 2003) is an experiment management system for performance and parameter studies as well as software testing for cluster and grid architectures.

ZEN, a directive-based language, is used to instrument arbitrary files and specify arbitrary application parameters. An Experiment Generator web-service parses application files, containing ZEN directives, to generate appropriate application codes based on the semantics of the directives encountered. Another web-service, the Experiment Executor, retrieves a set of experiments and compiles, executes and monitors them on the target machine.

Repository requirements review

The following is a review of the performance information management capabilities of ZENTURIO in the context of the requirements laid out in Chapter 1.

Non-intrusive: The annotation introduces additional code. When the annotated code executes at run-time, there must be some intrusiveness introduced by additional code. There is no discussion of the intrusiveness of this additional code on the application execution**.**

Minimal Demands: The software demands that this annotation puts on application codes are not discussed.

Post-mortem analysis: At the completion of experiments, the resulting files and performance information are stored in an information repository for post-mortem multi-experiment performance and parameter studies.

Rapid availability: As the performance information is only stored at the completion of the experiment, it seems that rapid access to this

information repository by any component (in this case the Application Data Visualiser) is not required; therefore this requirement may not be applicable.

Optimised: It seems that the monitored data is stored by the Experiment Executor (EE) web-service – also a monitor – during the experiment, since the information repository is updated at the end of the experiment. Since the requirement for rapid availability may not be applicable, the need for optimised dynamic performance information storage is questionable. Therefore, this requirement is classified as not applicable.

Relevant information: There is a notion of Filter that can be used to select a subset of messages. This may be used to be selective of the information stored.

Adaptive: Since the EE is a web-service, the adaptability of the storage may be at least equal to the adaptability of the web-service, even though this issue is not discussed. Web-services are designed to be adaptive. Therefore, ZENTURIO is classified as partially meeting this requirement, at worst.

2.5. Databases

In this section, the general relational database is studied for its suitability as a performance information repository by comparing against the requirements set forth in Chapter 1.

Such a database could be used in two configurations: a central database; or a set of databases located at various sites communicating with each other with user-specified wrappers.

Distributed databases – databases that implement their own (system) wrappers – are discussed later in this section.

2.5.1. Central database requirements review

The following is a review of the performance information management capabilities of a central database in the context of the requirements laid out in Chapter 1.

Adaptive: A central database would neither cope with dynamic creation and removal of geographically distributed streams nor coordinate the communication of data from sources to clients. This central configuration is potentially a single point of failure as all information sources and clients of the system would need to interact with it (Cooke et al. 2003).

However, many relational databases implement atomic, consistent, isolated and durable (ACID) properties that are more than enough to guarantee fault tolerance.

Therefore, a central database can be robust, but may not be adaptive to infrastructural dynamics, thus partially satisfying the adaptive requirement.

Rapid availability: A central database can take a long time to update from a remote site. Equally, if an entity, say a performance controller, that requires rapid access to performance information is located on a remote resource, then the information needs to be transported across the network, adding latency. This is a clear violation of rapid availability.

Non-intrusive: In addition, the central configuration may be intrusive to the sites generating performance information, as these sites might spend most of the time updating the central database.

Rapid availability and non-intrusiveness of a central database are studied by experiment in Chapter 4**.**

Relevant information: However, if the remote sites are intelligent enough to selectively send relevant data to the performance information repository, this could increase execution intrusiveness on the resource but might reduce communication latency as less information is sent. This is especially true if the remote site generates performance information frequently. Whether a performance information repository implemented as a central database meets this requirement depends on the particular implementation; therefore whether this requirement is met is classified as unclear.

Optimised: In addition, traditional relational databases are optimised for queries rather than updates. They would not be optimised for dynamic performance information, which is updated more than it is queried, thereby violating this requirement.

Minimal demands: The software prerequisite on a remote site generating performance information depends on where the intelligence of interacting with the database lies. Assuming such an interface lies with the database, for example using a functional or SQL interface, then remote sites simply stream performance information using a transport mechanism. This configuration assumes minimal software prerequisites on the remote sites generating the performance information. That is, mechanisms to deliver performance information to the central database.

Post-mortem analysis: A central database could be interfaced with a data warehouse to store experience of multiple executions to satisfy this requirement.

2.5.2. Distributed databases with user wrappers

The work reported in this thesis investigates the possibility of using distributed databases with user wrappers. Remote sites that

generate performance information can have local databases, implemented as for example a cache or a file, serving local data requests. The central database can be updated with information from local databases based on user-defined wrappers. The distributed nature of performance information management is explicit and handled by these wrappers. Such user-defined wrappers potentially allow the most flexible policies for updating central database. Some such configurations for managing performance information are suggested in Chapter 3. A review of these configurations against the requirements set forth in Chapter 1 is undertaken in Chapter 4.

2.5.3. Distributed Databases

A distributed database can be defined as a collection of multiple, logically interrelated databases distributed over a computer network (Ozsu and Valduriez 1999). A distributed database management system (DBMS) attempts to provide a fully transparent system. In other words, the role of a DBMS is to abstract away the implementation details and provide users with a unified view of the distributed database. The distributed nature of managing the performance information is implicit and handled by the DBMS.

The following reasons make a distributed database an unreasonable choice for implementing a performance information repository. It is highlighted where these reasons violate the requirements set forth in Chapter 1.

1. **Lack of control:** A DBMS provides abstraction at the expense of control over the way data is distributed and queried. The issues of replicating data and querying among interrelated databases are highlighted.

    a. Typically, a distributed database uses a DBMS policy to decide what data to replicate, and the user is not involved in

this decision. This may not be desirable, as the user might have some application knowledge that the DBMS does not have, and such replication may be irrelevant and inefficient. A DBMS may provide more control over replication policy by providing the user with an API, but this setup might be inefficient compared with user-defined wrappers.

b. A distributed database may dispatch a query to multiple sites if the data requested by query is held at multiple sites. Again, this may not be desirable, as the user might have application knowledge indicating that such distribution of a query may be irrelevant and inefficient. Again, this might be controlled if the DBMS provides the user with an API.

Both of the above points highlight that a distributed database restricts control. Less control can give rise to inefficiencies because, given application knowledge, an optimal strategy may not be used. Less control also restricts the ability to meet the requirement of **relevant information.** In practice, the control would depend on the particular implementation of a performance information repository implemented as a distributed database. Therefore, this is classified as unclear.

2. **Tightly coupled sites:** Grid computing is highly dynamic. Performance control systems may deploy application components on loosely coupled grid resources. In contrast, interrelated databases participating in the distributed database are tightly coupled, and may be less adaptive to infrastructure dynamics, thereby violating the requirement of being **adaptive**.

3. **Pre-installation:** Generally, any resource that needs to participate in the distributed database needs to be pre-installed with the database. This implies localising storage of performance

information at the resource and assumes that the resource site has an installed copy of the database. This is a substantial restriction, since the application component could be deployed for execution on any grid resource, and violates the requirement of **minimal demands**.

4. **Global Schema:** A distributed database requires a global schema to be defined. As discussed in Section 2.2.1, it is arguable that users should not be provided, in general, a consistent view of global state (Czajkowski et al. 2001).

5. **Intrusive:** Assume that a local database is installed on a participant in a distributed database, for every application component. The resulting DBMS could require significant processing power; and thus be intrusive, thereby violating the **non-intrusive** requirement.

The following is a review of a distributed database being used as a performance information repository against the requirements laid out in Chapter 1. Only requirements that have not been discussed previously are discussed here.

Rapid availability: Providing rapid access to performance information would depend on the distributed configuration of the sites and replication of data policies. It is hard to state whether a distributed database can provide rapid access without empirical analysis. Therefore, this requirement is classified as unclear.

Optimised: As discussed previously, traditional relational databases are optimised for queries rather than for update. It seems likely that distributed databases would have similar characteristics and not be optimised for dynamic performance information, which is updated more than it is queried, thus violating this requirement.

Post-mortem analysis: The distributed database could be interfaced with a data warehouse to store experience of multiple execution.

2.6. Discussion

Table 1 summarises the systems reviewed in this chapter in terms of their meeting the requirements from Chapter 1.

The information in this table is restricted to that available from published literature. Since performance information management is not necessarily discussed in detail in such papers, the table may not reflect the true capabilities of the systems included. Nonetheless, accepting the table as it stands, it is apparent that performance information management requirements are not fully addressed in current systems.

It is worth noting that monitored information needs to be managed in many systems across many domains from performance control to monitoring. However, the focus may be different. For example, rapid availability of performance information is critical in performance control systems but not so in a performance modelling system.

Generally, monitored information generated by producers is transferred to consumers located remotely. There are at least two problems associated with such a configuration.

Firstly, the transfer of information may have large communication overheads. There have thus been attempts at reducing the amount of data that needs to be transferred. For example, pre-processing and buffering can be used for such purposes. Sometimes, it is recognised that such transfers might suffer from long transfer latencies. In this case, the consumer might receive stale information. In order to

identify stale information, there have been attempts to validate information, for example, by using timestamps.

Systems	Rapid availability	Non-intrusive	Minimal demands	Optimised	Adaptive	Post-mortem analysis	Relevant information
PERCO prototype	−	√	√	?	−	−	×
Autopilot	√	√	√	/	−	−	√
AppLeS	N/A	/	/	/	−	−	/
R-GMA	?	/	/	/	√	−	/
MDS	−	/	/	/	√	/	−
NWS	?	√	×	?	−	−	√
ZENTURIO	N/A	/	/	N/A	−	√	−
Central db	×	×	√	×	−	−	?
Distributed	?	×	×	×	×	−	?

Table 1 – Systems compared against Performance information repository requirements from Chapter 1. Key: √ means requirement met, − means requirement partially met, × means requirement not met, / means not discussed in the cited literature and probably not met, ? means unclear if requirement met and N/A means not applicable.

Secondly, the consumer is remote in all of the systems reviewed here. There might be benefits in localising consumers at the producer-site. For example, part of a performance control system might be localised at the site that is executing the application component. Therefore, it

makes an interesting investigation to study localisation of consumer at the producer-site. This thesis investigates the effects of localising performance control at the producer-site. This setup is then studied with various geographical configurations for managing the performance information.

In subsequent chapters, the focus is to study the performance information repository from the view of a performance control system.

3

Perfrep – a performance information repository

The performance information repository, Perfrep, being is an emulator which represents a new implementation of the PERCO design. In Section 3.1 and Section 3.2, the PERCO design is introduced and described. The roles of the various building-blocks in the design are described in Section 3.3. The model of operation for PERCO design is presented in Section 3.4.

The description of the PERCO system is heavily based on (Mayes et al. 2003; Mayes et al. 2004)

This chapter also discusses run-time performance control. As discussed in Section 2.1.1, there is a two-level distinction made in PERCO for the performance control of an application consisting of components. Component-level performance control is concerned with the performance control of a component only by adjusting the control parameters of a component. Application-level performance control is concerned with the re-deployment of components on the given resources to achieve better performance of the overall application.

Perfrep allows different configurations of performance information repository to be investigated. An overview of these configurations is presented in Section 3.5, followed by the implementation details in Section 3.6.

3.1. PERCO

PERCO is a performance control system for controlling an application consisting of interacting coupled components. It has been developed in the RealityGrid project (RealityGrid 2005). Figure 1 depicts the design of PERCO. PERCO is used as the target performance control system for this work. That is, PERCO provides the performance control context for the performance information repository studies.

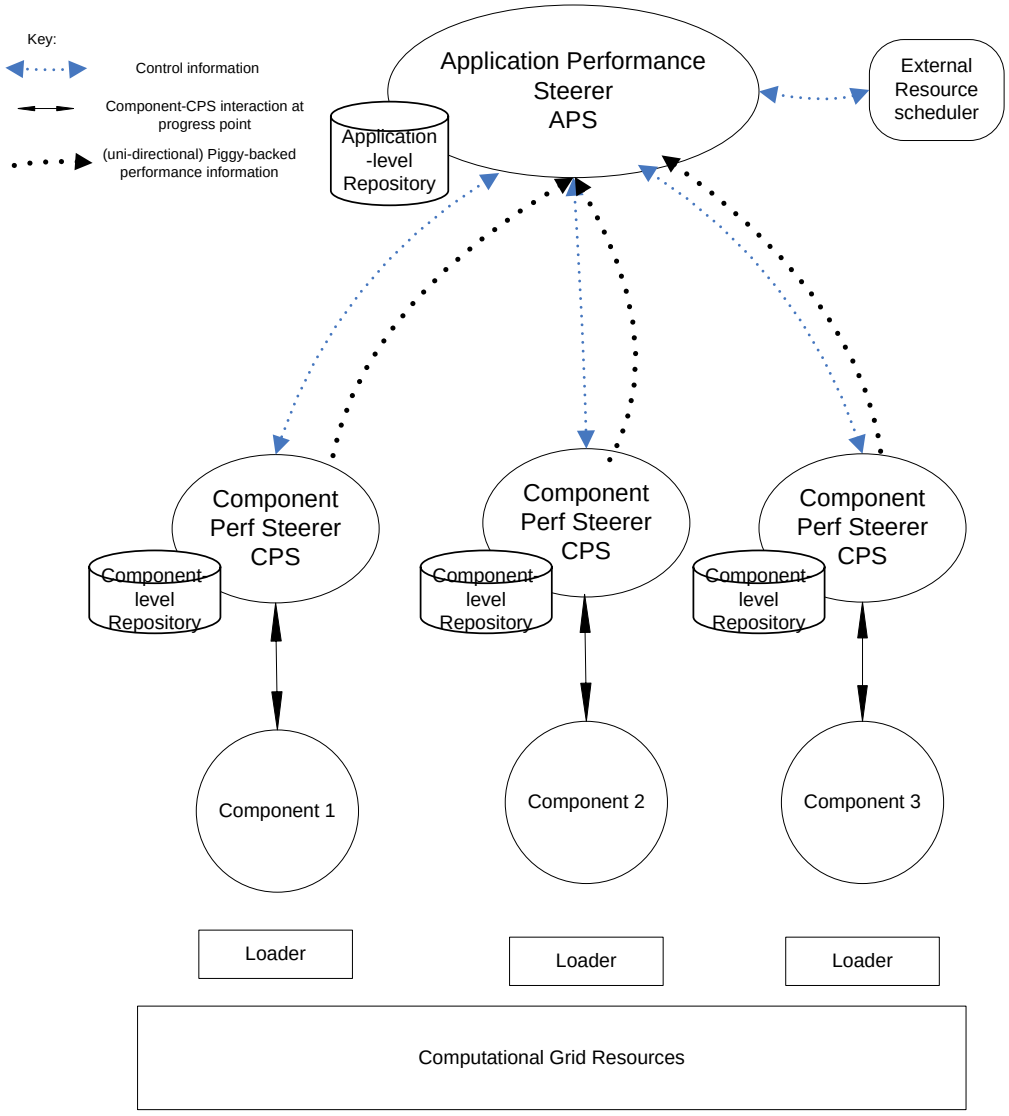

Figure 1 - PERCO Design. Resources are allocated to a component-based application by an external scheduler. The APS is responsible for application-level performance control. Each CPS is responsible for component-level performance control of its local component. PERCO loaders are used to launch and migrate the CPSs and their components.

3.2. The PERCO design

An application, consisting of loosely coupled and separately executable components, may be distributed over many grid computing resources for the execution to achieve concurrency and to gain access to large amount of computing resources (Mayes et al. 2003). There are at least two performance control issues relating to such a component based application executing on a grid.

Separation of concerns: There are two levels of abstraction of performance control: application and component. In an application view, the concern is to deploy components such that the overall application executes as fast as possible. From the component view, the performance concern is local to that component. That is, the component can be controlled locally by tuning its parameters.

Emphasising local control: Communication latency between distributed components is likely to be significant in performance control systems, which need to access monitoring information from the components. Maximising local control will reduce dependency on communication.

The PERCO design divides the functionality of the performance control system into two levels of abstraction (Mayes et al. 2003):

1. **The performance improvements at component-level**. This might include changing algorithms or tuning performance parameters.
2. **The performance improvements at application-level.** This might include monitoring the progress of the application as a whole. The performance control would then entail re-distributing resources to components that have become the performance bottleneck by re-deployment.

The PERCO design explicitly deals with these levels separately. The CPS is responsible for performance control at component-level. The APS is responsible for performance control at the application-level.

The roles of the APS and the CPS imply that they should have access to a performance prediction capability and to a performance information repository. The component loader is responsible for starting, restoring or migrating the component.

The PERCO design assumes that, ideally, the allocation and initial distribution of resources is carried out by the external resource scheduler. It is further assumed that run-time re-deployment of components to resources is the responsibility of the performance control system. In other words, a fixed set of resources would be allocated to a performance controlled application. The performance control system is responsible for re-distributing these resources to enable performance control to the application.

3.3. Basic roles of the APS and the CPS

As discussed in the previous section, the PERCO design gives distinct roles to the APS and the CPS. A performance prediction facility is required by both the APS and the CPS roles.

3.3.1. Role of the APS

The APS is provided with a set of resources by the external resource scheduler. The role of the APS is:

To distribute the available resources between the components in a manner that gives better overall application performance.

The goal is to achieve the minimum application execution time.

Finding this improved deployment involves:

1. finding the expected performance of each component for a given resource deployment; and

2. finding the expected performance of the application for each set of predicted component performance values.

The expected performance is found out by performance prediction, which is central to the APS role. Performance information is assumed to be available via some performance information repository.

3.3.2. Role of the CPS

The role of a CPS is:

To enable a component to tune its performance by adjusting control parameters of a component.

For example, a CPS may change an algorithm or data structure in its component to help the component perform better.

The APS distributes available resources amongst components of the application. Then, each CPS controls its component to achieve minimum execution time on the deployed resource. A CPS is also assumed to have access to performance information for that component through a performance information repository. However, the performance information required is for that component only.

The components may communicate with each other. This implies that, if the components have a dependency between them, then a slowly-executing component can become a bottleneck and make the whole application go slower.

A CPS attempts to optimise the performance of its component on its current resource, based on the collection of performance data for the component. A CPS "sees" only the local performance – that is, it has no overview of application performance (because the latter is the job

of the APS). The APS can detect the relative performance of components. A component may be progressing too slowly, compared with its peers, for a number of reasons. For example, this component might be computationally intensive and executing on a slow resource. In contrast, another component might not require much computing power but be deployed to a faster resource. Such deployment of the components might make the whole application slow. The APS detects such problems and redeploys the slow component to a faster grid resource. The detection of a problem depends on performance information. The re-deployment decision depends on performance prediction, which in turn also depends on the performance information. So, performance information underpins the whole process.

3.4. Performance steerer model of operation

An example application might be to observe how the behaviour of an ocean simulation and an atmosphere simulation interact to determine climate. The computation requires interaction between these two simulations through a series of simulated timesteps. In general, an application may move through different phases as shown in Figure 2.

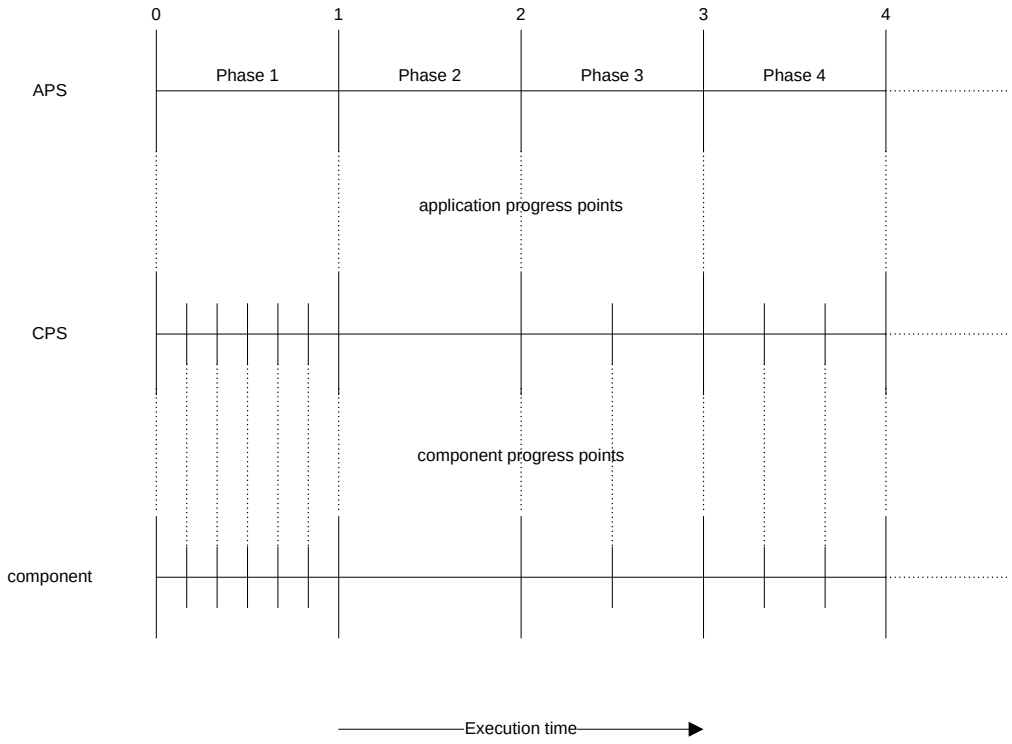

Figure 2 – A pictorial representation of the relationship between computation, CPS and APS. Dotted vertical lines indicate the exchange of status and control messages at progress points. There can be any number of component progress points in one application phase (Mayes et al. 2003).

3.4.1. APS model of operation

The PERCO design views the operation of the APS in terms of a cycle of activities. The execution of the component-based application is assumed to proceed in phases as shown in Figure 2. Notice that there are application progress points between the phases. These progress points include APS activity. The APS activity is essentially a decision-making process which may lead to a new deployment for the components of the application.

The role of the APS is to distribute the available resources between the components in a manner that gives the minimum execution time for the application. Therefore, the APS can perform two operations at each application progress point:

1. If the APS predicts that the current deployment of components on resources gives minimum execution time over other possible deployments, then instruct the CPSs to continue with the current deployment; however,
2. if the APS predicts that a faster deployment is available that might give lesser application execution time and would offset the time taken to deploy the improved deployment, then the components are re-deployed on resources according to that deployment.

3.4.2. CPS model of operation

Between application progress points, the PERCO design views the activity of a component in terms of component progress points. There can be many component progress points within one phase of the application, as depicted in Figure 2.

The CPS can autonomously tune the component at component progress points between the application phases or progress points. The number of component progress points in a phase marks the number of opportunities that are available for the CPS to monitor and tune the component. In Figure 2, the CPS can tune its component during phase 1, but to a lesser extent in phase 3 and phase 4. During phase 2, the CPS has no opportunity to tune the component. It is obvious that, the greater the number of component progress points in an application phase, the greater the opportunity for the CPS to tune its component during that phase. It is claimed that "greater autonomy for the CPS means less global activity, and opportunities for greater efficiency" (Mayes et al. 2003). This implies

that the most efficient setup possible would be to have one application phase, where there is minimal global activity and maximal CPS autonomy. The CPS will have a maximum number of opportunities to tune the component. However, this may not give the minimum execution time for the overall application since the APS will not have any opportunity to, say, re-distribute resources to a slow component, even though that component may be in its optimal component-level configuration. It is envisaged that a balance between numbers of application progress points and component phases in those phases would result in minimum execution time for the application.

3.5. The Perfrep design

3.5.1. Overview

As mentioned previously, Perfrep is an emulator which is an implementation of the PERCO design. Perfrep is implemented in different performance information repository configurations to investigate effective performance information management. The general design of Perfrep is depicted in Figure 3. The Perfrep emulator consists of emulated APS and CPSs, and is implemented in a geographically distributed context, as would be applicable to the grid environments. There are no loaders present; the APS and the CPSs are statically deployed onto the distributed resources. These resources are not grid resources, as they do not implement any grid middleware.

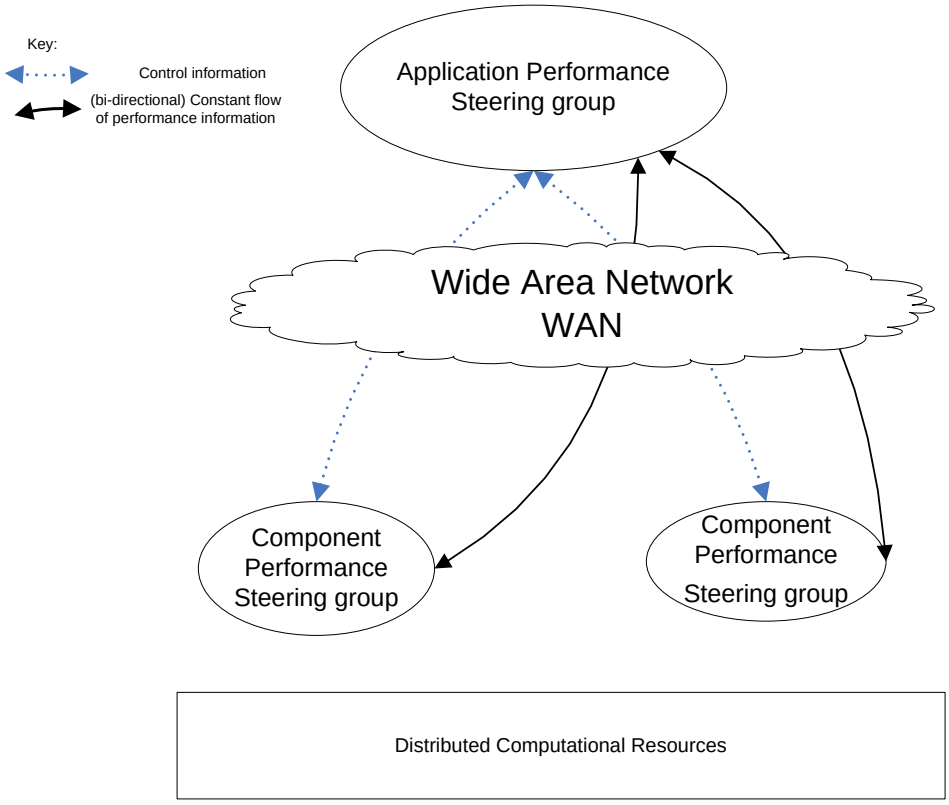

Figure 3 – General design of Perfrep. The Application Performance Steering group and Component Performance Steering group refer to a set of entities that reside with the APS and the CPSs. For example, a central repository may reside with the APS and would then form part of the APS group.

3.5.2. The Perfrep configurations

This section includes a brief description of each Perfrep configuration. A case study for each configuration of Perfrep has been used to investigate the communication overhead and the effects of component-level performance control. In all the cases, the CPSs can undertake component-level performance control.

Application-level performance control has not been empirically investigated as part of this work.

The implementation of the configurations is given in Section 3.6.

For all configurations:

- the CPSs send status messages to the APS, termed "checking-in", at every application progress point,
- the APS replies to status messages with control messages to the CPSs, and
- an application-level performance information repository (application-Rep) is present in the APS group.

Perfrep with centralised repository configuration (Perfrep-central)

This is the simplest configuration. The CPS updates the application-Rep with performance information at every component progress point.

The CPS needs to fetch information from the application-Rep for component-level performance control decisions.

Perfrep with piggyback configuration (Perfrep-piggy)

Performance information is piggybacked on the communication messages between the CPS and the APS. This happens whenever a CPS checks-in to the APS and the APS replies with a control message; this only occurs at application progress points. In this configuration, the performance information for component progress points is not stored into the application-Rep and is therefore lost.

Perfrep with cached piggyback configuration (Perfrep-cache)

In the Perfrep-piggy configuration, the performance information associated with component progress points is lost between application progress points. To avoid this loss, a component-level cache is introduced in the CPS group. This cache holds performance information for the component progress points between the application progress points. At an application progress point, all the performance information for the component progress points, since the last application progress point, is piggybacked into the application-Rep via the APS.

The CPS accesses this component-level cache for component-level performance control decisions.

Perfrep with distributed configuration (Perfrep-dist)

With the cached piggyback configuration, a subset of requests for performance information by the CPS performance control algorithm can be served by the component-level cache. Any requests that cannot be served by the component-level cache would need to be served by the application-Rep. Therefore, a component level performance information repository (component-Rep) is introduced, which allows the possibility of relieving the application-Rep from storing detailed component-level performance information. This possibility can increase the relevance of information for the APS in the application-Rep. That is, the application-Rep only needs to store performance information that is relevant to APS performance control.

3.6. Implementation

This section describes the Perfrep implementation of the basic PERCO design shown in Figure 1. This implementation is the foundation of all the Perfrep configurations and is called the base implementation.

Implementation descriptions of the Perfrep configurations are presented. Where the Perfrep configurations differ from the base implementation, the difference is highlighted in the description of their implementations.

3.6.1. Issues in implementation

Separation of concerns: A performance control system performs run-time adjustments based on the performance information. Such a system consists of two parts: (a) one part manages performance information and (b) the other part makes performance control decisions.

Clearly, these are two related but separate issues. Separating their implementations reduces coupling between them and so increases flexibility. For example, the frequency of generating performance information may be adjusted according to the accuracy of performance control decisions required, without modifying the performance control code.

This issue is discussed further in Section 3.6.6..

3.6.2. The Perfrep Implementation of the PERCO design --- base implementation

This section describes the mechanisms used to implement the design shown in Figure 1.

For simplicity, and to facilitate experimentation, the deployment of CPSs (with associated components) is simulated. That is, the APS and the CPSs (with associated components) are statically deployed to platforms. Therefore, there is no implementation of loaders. Simulation of migration is described below.

Application Performance Steerer (APS) --- The APS is responsible for receiving application progress point updates from CPSs. This involves the extraction of piggybacked performance information and updating of the application-Rep. The APS checks for performance imbalance, using performance information from the application-Rep. The APS send control commands (normally to continue) to the CPSs. The APS deals with communication, problem detection and rectification.

The APS checks for problems in the performance of the deployment of components. When a problem is detected, the APS may instruct some CPSs to migrate.

Migration has two aspects: transportation of the component to a new platform, and the execution of the component on the new platform. Migration can be simulated as follows:

- Transportation time is simulated by taking the average *ftp*[1] time of transporting the component executable to a different platform. The ftp time depends on the geography of the resources involved in executing in a Perfrep configuration. It is determined manually for a geographical setup of resources and incorporated into the Perfrep configurations.

- Simulating execution on different platforms is performed by the notion of different configurations of the components of Perfrep. There are two types of platform configurations i.e. a slow configuration, called machA, and a fast configuration, called machB. In machA, the execution of the component is paused momentarily, for time **P**, at every component progress point in order to simulate a slow platform. In machB, there is no such

[1] ftp is a Linux utility to transfer files over a network.

pause (**P** = 0s). Consequently, the component is faster in its machB than machA.

An application-level performance control algorithm that can be used by an APS to rectify problems in performance is described in Section 4.1.3. (Recall that application-level performance control has not been studied empirically as part of this work).

Component (comp)--- A grid "application" is actually an assemblage of "components". An application component is transformed into a Perfrep component, simply called component, by wrapping in an interface code. The component source code is instrumented with calls to the CPS. The interface code is used to facilitate minimal changes required in the application component code to make it performance controllable. This interface wrapper passes outgoing instrumented calls from the component to the CPS, and interprets incoming commands from the CPS to the component. Such commands include:

- tuning of a parameter initiated by the CPS, for example change of a sorting algorithm; or
- performance control of the whole component initiated by the APS, for example request to migrate to a different resource by instructing the component to change its platform configuration.

Component Performance Steerer (CPS) --- A CPS is associated with a component and its component interface.

A CPS in Perfrep has three primary purposes:

1. The CPS checks-in to the APS.
2. The CPS performs component-level performance control.

3. The CPS updates/retrieves performance information to/from the application-Rep, via the APS, at application progress points.

The performance information is piggybacked on the communication messages between the CPSs and the APS.

The interactions between the APS and the CPSs are expressed as procedure calls which are implemented as message passing stub routines. Linux sockets have been used to implement the communication mechanism.

The component code, the interface code and the CPS code are linked together into a single binary image which executes in the context of a single process. Calls between the component, component interface and CPS are conventional procedure calls.

The algorithm used by the CPS to perform component-level performance control is described in Section 4.1.2.

Application-level performance information repository (application-Rep) --- The implementation of the application-Rep uses the BerkeleyDB library. This resides on the same resource as the APS. There is also an associated application-level cache for fast lookup of the most recent records. The application-Rep is updated via the APS.

Component-level performance information repository (component-Rep) --- There is no component-Rep in the base implementation.

Loaders --- As discussed in Section 3.1, the PERCO design relies on loaders to redeploy components. However, in Perfrep there is no actual implementation of loaders. The APS and the CPSs (with associated components) are statically deployed onto platforms. Migration is simulated, as described above.

Operation

The APS and the CPSs start executing on separate distributed resources. The APS waits for the CPSs to check-in.

Each component invokes its CPS at every component progress point in the component execution. The CPSs communicate with the APS at each application progress point. This happens after a fixed number of component progress points. While the CPS is reporting to the APS, it is in a blocked state until the APS notifies the CPS of what to do next – this is a choice taken for simplicity of synchronisation. The APS-CPS communication is primarily intended for control and status information. Such communication is inevitable due to the basic design of the system. In addition, the performance information is piggybacked on the status messages from the CPSs to the APS, and on the control messages from the APS to the CPS.

Other Perfrep configurations

The design depicted in Figure 3 and the base implementation in Section 3.6.2 underpin all the Perfrep configurations. Therefore, these configurations have similarities between them. The following sections highlight the implementation differences and/or additions to the base implementation to achieve each particular Perfrep configuration.

3.6.3. Perfrep with centralised repository configuration (Perfrep-central)

Figure 4 depicts the Perfrep-central configuration.

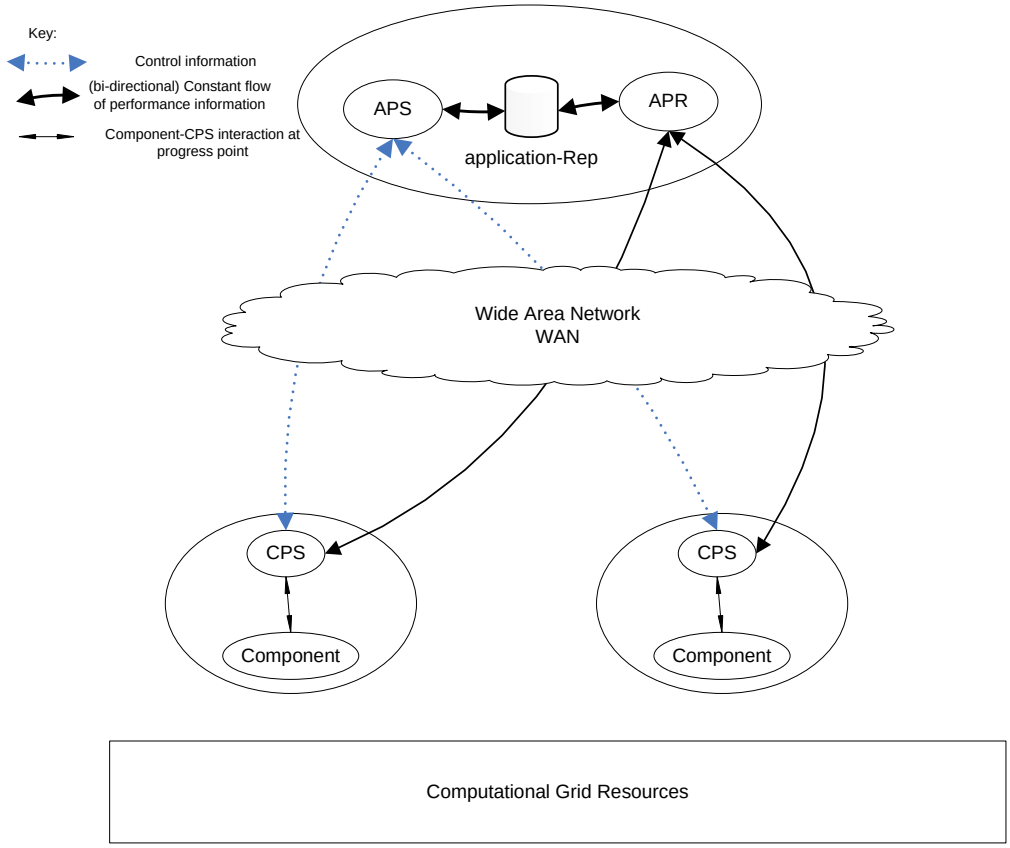

Figure 4 – Perfrep-central configuration. The APS group consists of an APS, an APR and an application-Rep (with its associated application-level cache). The CPS group consists of a CPS and a component. The APS-CPS channel is exclusively used for control and status information, and the APR-CPR channel is exclusively used for performance information.

Perfrep-central is the simplest configuration; it has an additional member in the APS group, namely i.e. Application Performance Repository (APR) process.

The APS and the APR are compiled into a single binary image. The APS forks into two processes. The APS is responsible for performance control only. The child process, APR, manages the application-Rep.

Application Performance Repository process (APR) --- The APR is responsible for managing performance information in the application-Rep. The Perfrep-central configuration implements an APR on the same computing resource as the APS. It has three roles:

1. The APR updates the application-Rep with the performance information received from the CPSs.

2. The APR serves requests for performance information from the CPSs.

3. The APR transforms dynamic performance information into static performance information while idle. This role is not yet implemented.

Component Performance Steerer --- As mentioned in the description of the base implementation, a CPS checks-in to the APS at every application progress point. While the CPS is checking-in to the APS, it is in a blocked state until the APS notifies the CPS of what to do next. In contrast to the base implementation, the CPS updates the application-Rep with performance information via the APR at every component progress point. In addition, if the CPS is activated to perform component-level performance control, then the APR is requested for performance information. This CPS-APR setup has three consequences:

1. The CPS has to wait for the delivery of performance information in a blocked state. This blocked time could have been used for execution of the component.

2. Performance information is requested and delivered over a network at every component progress point.

3. If component-level performance control is active, the CPS may request performance information from the APR over a network in order to control the component execution.

Application Performance Steerer --- The APS only uses APS-CPS communication for receiving status messages from the CPS and replying with control messages. In contrast to the base implementation, the APS does not expect performance information in the status messages from the CPS. Other than this, the APS in Perfrep-central behaves like the APS in the base implementation.

Discussion --- It is anticipated that a lot of communication takes place in this configuration over a network. Since a CPS blocks for the network communication, this might result in the CPS being blocked for a long period of time. This would waste execution time, which could have been spent on component execution. This blocked time might dominate any performance control gains, if component-level performance control is active. These two factors: the time spent in the blocked state for network communication, and the effect on component-level performance control are studied empirically in Chapter 4.

3.6.4. Perfrep with piggyback configuration (Perfrep-piggy)

It is worth emphasising that the Perfrep-central configuration dictates that a CPS stores performance information in the application-Rep at every component progress point. In addition, it is envisaged that a lot of potentially valuable execution time might be wasted for network communication while the CPS is in a blocked state. Assuming a synchronised CPS-APS communication, an obvious improvement might be to piggyback performance information on the status and control messages exchanged between CPS and APS. This would give the configuration shown in Figure 5.

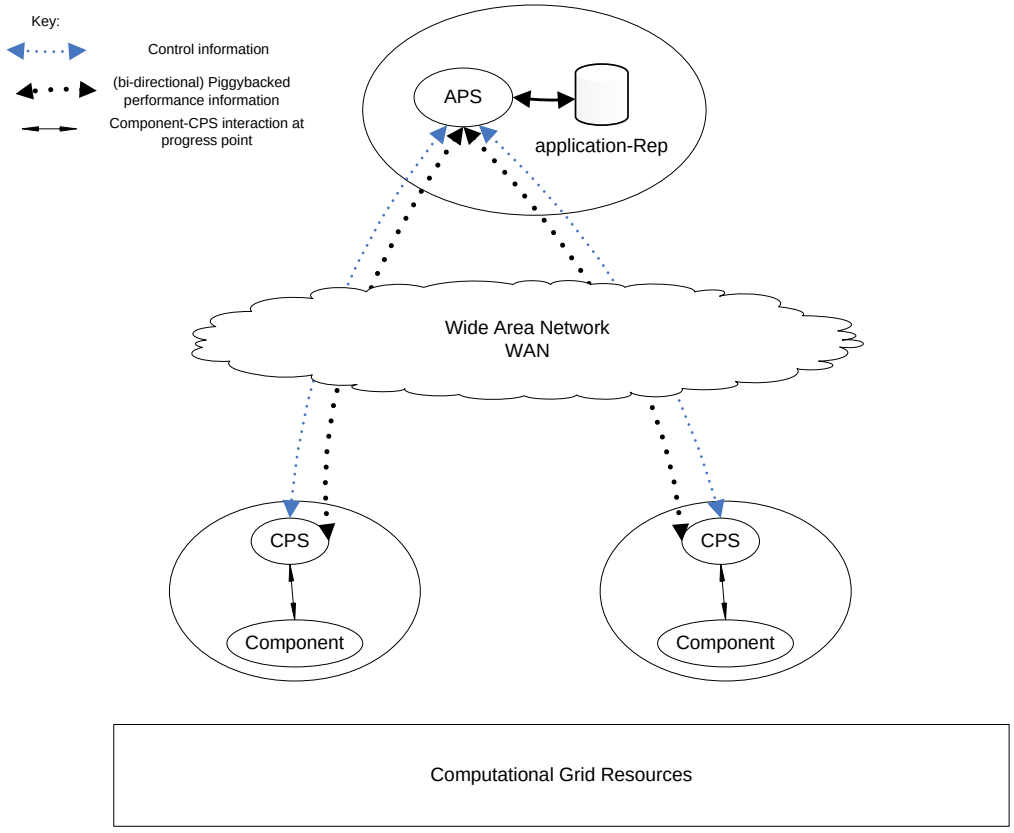

Figure 5 – Perfrep-piggy configuration. The APS group consists of an APS and an application-Rep (with its associated application-level cache). The CPS group consists of a CPS and a component. The APS-CPS communication channel is used for control and status information. In addition, performance information is piggybacked on the status and control messages between the CPSs and the APS.

This configuration is closest to the base implementation described in Section 3.6.2.

Notice that the status and control messages and the performance information are coupled. This configuration violates the principle of separation of concerns given in Section 3.6.1. It is inflexible to the extent that the exchange of performance information between the

CPS and the application-Rep can only happen at application progress points.

Another side-effect of the Perfrep-piggy configuration is that performance information for the component progress points is not being stored in the application-Rep. This implies that, if component-level performance control is active, then the CPS may be unable to perform efficient tuning of the component.

An obvious improvement would be to introduce a cache in the CPS, in order to store performance information for component progress points. This configuration is discussed next.

3.6.5. Perfrep with cached piggyback configuration (Perfrep-cache)

This configuration has a component-level cache in the CPS which stores performance information for the component progress points. Figure 6 depicts the architecture of the Perfrep-cache configuration.

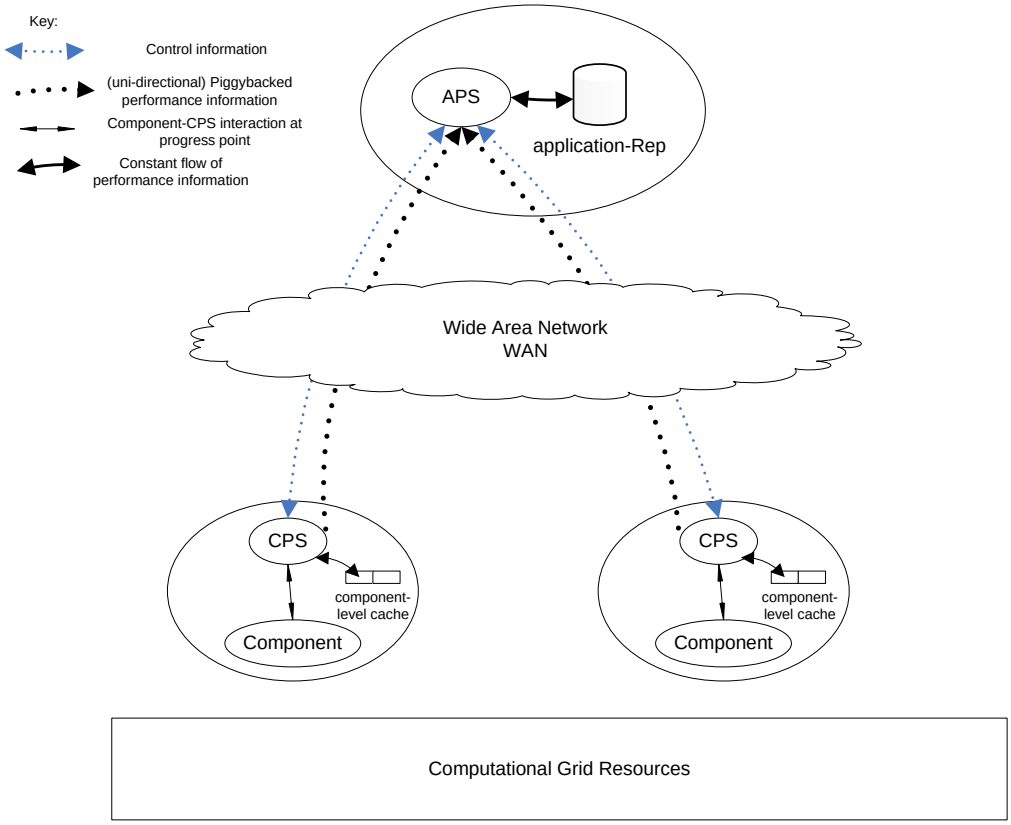

Figure 6 – Perfrep-cache configuration. The APS group consists of the APS and an application-Rep (with its associated application-level cache). The CPS group consists of a CPS, a component and a component-level cache. The APS-CPS communication is used for control and status information. Performance information is piggybacked on the status messages from the CPSs to the APS.

The CPS group has an additional member, a component-level cache, as compared to the base implementation.

Application Performance Steerer --- The APS in Perfrep-cache has responsibilities similar to those of the APS in the base implementation. The only difference is that the APS in Perfrep-cache does not yet implement mechanisms to serve any performance information request from the CPSs.

Component-level cache --- The component-level cache is responsible for storing performance information for the component progress points. The component-level cache should at least be of the same size as the number of component progress points between the application progress points. This setup has two implications:

1. The component-level cache can temporarily store performance information for all the component progress points between the application progress points. At an application progress point, the CPS can piggyback all the performance information for the component progress points cycles stored in the component-level cache and send it to the APS.

2. Requests for performance information by a CPS can be served by the component-level cache. Therefore, the ability to tune as efficiently as the Perfrep-central configuration may be regained.

Component Performance Steerer --- The CPS in this configuration differs from the CPS in the base implementation in three ways:

1. The CPS stores performance information for component progress points in the component-level cache.

2. The CPS is able to update the application-Rep with the performance information of the component progress points stored in the component-level cache. This is done by piggybacking performance information on the status messages from the CPS to the APS at an application progress point.

3. The CPS retrieves performance information from the component-level cache for component-level performance control.

Discussion --- This Perfrep-cache configuration has some potential shortcomings, too. If the CPS performance control algorithm requires

a detailed performance information history, then the component-level cache may not be able to serve all the requests, and requesting data from the application-Rep is inevitable. How this may affect network communication costs really depends on the frequency of the old performance information required by the CPS.

Therefore, a local component-level performance information repository (CPR) may be needed to rectify this.

The Perfrep-central and Perfrep-cache configurations dictate that the application-Rep is updated with the performance information for every component progress point. The APS may not require performance information at each component progress point, for application-level performance control.

Moreover, for all the above configurations, the CPS is responsible for dealing with performance information, for example by storing it in application-Rep. As discussed in Section 3.6.1, the management of performance information is clearly a separate concern from performance control. A separate entity (CPR) may be appropriate for dealing with the performance information. This leads to the configuration presented next.

3.6.6. Perfrep with distributed configuration (Perfrep-dist)

To address issues discussed in the previous section, the performance information is managed separately in the Perfrep-dist configuration, as depicted in Figure 7.

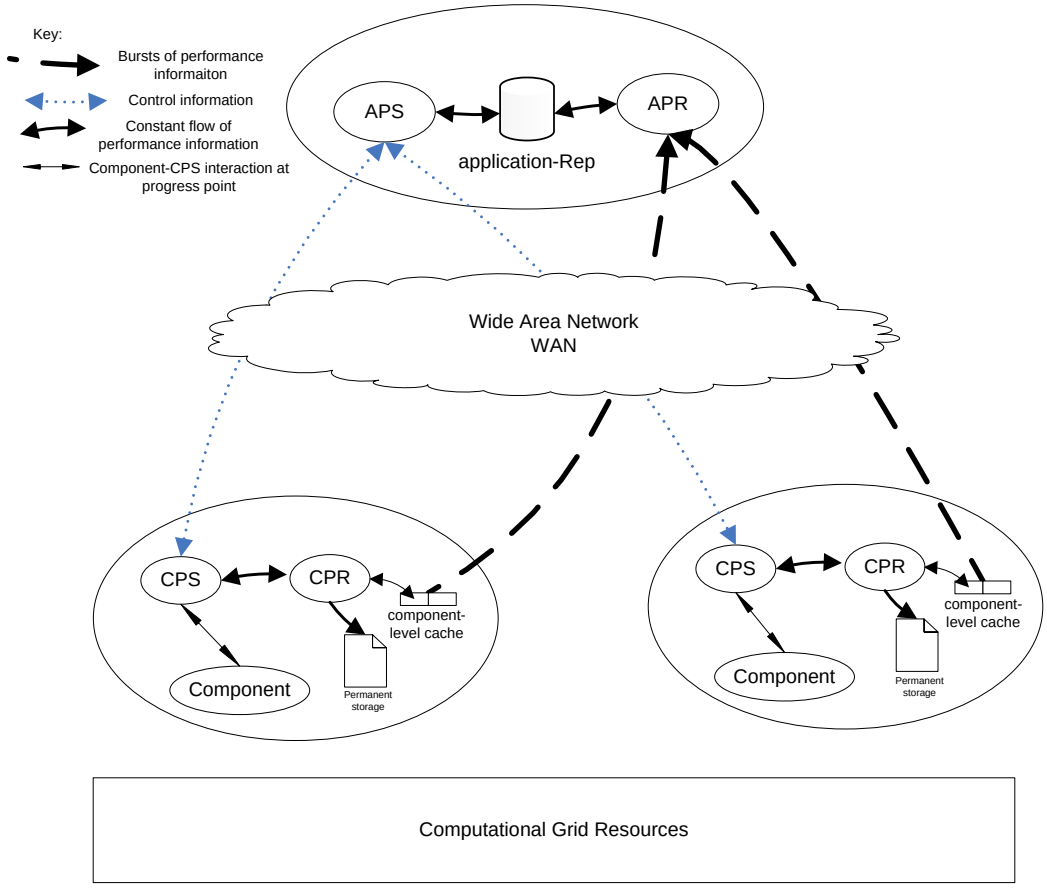

Figure 7 – Perfrep-dist configuration. The APS group consists of an APS, an APR and an application-Rep (with its associated application-level cache). The CPS group consists of a CPS, a component, a CPR, a component-level cache and a persistent storage. The APS-CPS communication is exclusively used for sending control and status information only. The APR-CPR communication channel is exclusively used for sending performance information by the CPR. The CPR manages a component-level cache and a persistent storage for performance information history.

The CPS forks into two processes. The CPS runs in the same process as the component. The child process, CPR, updates the application-Rep with performance information via the APR.

Component Performance Steerer --- The CPS in this configuration differs from the CPS in the base implementation in three ways:

1. The APS-CPS channel is used for status and control only.
2. The CPS stores the performance information at the component progress point locally to the CPR.
3. The CPS retrieves performance information from the CPR for performance control.

The CPR updating application-Rep via APR does not affect the execution time of the CPS. Instead, the CPR is responsible for management of the component-level cache, for updating application-Rep via APR, and for maintaining persistent performance information history using a data file.

Application Performance Steerer (APS) --- The APS has responsibilities similar to those of the APS in the Perfrep-central configuration. That is, the APS only uses APS-CPS communication for receiving status messages from the CPS and replying with control messages. Other than this, the APS in the Perfrep-dist configuration behaves like the APS in the base implementation.

Component Performance Repository process --- A CPR is responsible for dealing with performance information. The CPR resides on the same computing resource that the component resides on, along with the corresponding CPS. At each component progress point, the CPS can store performance information in the local CPR, rather than in application-Rep, and then continue execution. This performance information, or at least part of it, is required by the APS to make performance control decisions.

A CPR has three purposes:

1. It maintains a component-level cache to allow the CPS, using a generic interface, to have rapid access to recent performance information for the component progress points.

2. Long term performance information is maintained in some form of persistent storage. In the current implementation, a data file is used, though other storage mechanisms, such as a local database might be used.

3. The performance information in the component-level cache is delivered to application-Rep via network communication with the APR.

When the component-level cache is full, the cached performance information is sent to the APR and flushed to the local file.

Application Performance Repository process --- The APR is responsible for managing application-Rep. The Perfrep-dist configuration implements an APR on the same computing resource as the APS. The APR has two roles:

1. It updates the application-Rep with the performance information received from the CPRs.

2. It transforms dynamic performance information to form static performance information. This has not yet been implemented but could be done as future work.

Discussion --- This configuration has the following characteristics:

- The CPR stores performance information in a permanent storage format. This persistent storage is undertaken for two main reasons:

a. The CPS may require performance information history of a component which is not available in the short-term component-level cache.

b. Post-mortem analysis can be performed on the whole performance information history.

- This configuration frees the CPS from the responsibility of updating application-Rep. Consequently, the CPS does not need to block while the performance information is being communicated, as happens in the case of the Perfrep-central configuration. Instead, the CPS makes rapid updates to a component-level cache and carries on with execution.

- The CPR is responsible for updating the application-Rep via the APR, according to a modifiable policy. A simple, and currently implemented, policy is to update the application-Rep when the component-level cache is full. Performance information can take a significant amount of time to be delivered over a WAN. Since the CPR is a separate process, the performance information can be delivered while the component is executing. In addition, the performance information can be delivered at times other than progress points. This allows study of the effects of varying the update interval on the performance of the whole system.

- Perfrep-dist promotes separation of concerns and the possibility of updating the application-Rep with a subset of performance information relevant to the application-level performance control. However, in the current implementation, application-Rep is used to store performance information for every component progress point.

3.7. Summary

Perfrep is an implementation of the PERCO design. This chapter has described the PERCO design, building-blocks of PERCO design, and the PERCO model of operation.

The two level abstraction for performance control has been discussed. This includes application-level performance control and component-level performance control. Application-level performance control is concerned with the re-deployment of components on the given resources so as to achieve better performance of the overall application. Component-level performance control is concerned with the performance control of a component only by adjusting the control parameters of a component.

This chapter has presented implementations of all the Perfrep configurations. Each configuration is underpinned by a base implementation of the PERCO design. The configurations differ in the way that they manage performance information and its provisioning for component-level performance control. Possible implementation issues with these configurations have been highlighted; in particular, frequent network communication may result in wasting of component execution time.

The next chapter describes experiments that have been performed with the various Perfrep configurations and their results.

4

Evaluation of Perfrep

A performance information repository in a grid has many aspects that need to be studied empirically. The experiments presented here have been performed to emphasize the importance of localising information and performance control in a WAN environment, where a grid is likely to operate. The Perfrep configurations have been empirically studied in a WAN environment for effective and rapid performance information management for component-level performance control and also reducing intrusiveness.

This chapter begins by describing the experimental method in Section 4.1. This includes implementations for a component, component-level performance control and application-level performance control. Additionally, the experimental test bed is introduced and the experimental measurements are explained.

All the Perfrep configurations were described in Section 3.6. In Section 4.2, each Perfrep configuration is experimentally evaluated, and discussed in the light of two requirements, namely rapid access to information and non-intrusive, which naturally leads to the next configuration. The raw data of the experiments are given in Appendix 7.2.

In Section 4.3, the Perfrep configurations are evaluated against the requirements laid out in Chapter 1. In Section 4.4, the chapter concludes by presenting a table summarising the ability of each Perfrep configuration to meet these requirements.

Recall that application-level performance control has not been studied empirically as part of this work.

4.1. Experimental Method

In this section, simple implementations for a component, component-level performance control algorithm for the CPS and application-level

performance control algorithm for the APS are suggested. Only the suggested implementations of the component and the component-level performance control algorithm are later used to study the requirements under investigation. A comprehensive critique of the implementations of the component and the component-level performance control algorithm is presented in Section 5.3. In addition, a detailed critique of the implementation of component-level performance control algorithm is presented in Section 4.1.2.

The experiment is designed so that an application progress point happens after a fixed number, **App-Cpp**, of component progress points. The App-Cpp is set to be ten for the experiments documented in this thesis.

4.1.1. IterSort – A performance controllable component

A synthetic application has been used. A simple sort program, IterSort, has been developed as an example of a performance controllable component. The program uses a suite of sorting algorithms to iteratively sort a sequence of arrays. A tuneable parameter determines the specific sorting algorithm used at any iteration. This parameter determines which sorting algorithm is to be used to sort the array in each iteration. After sorting an array, IterSort reports to its CPS marking a component progress point. The CPS may perform performance control based on a component-level performance control algorithm (described in Section 4.1.2). Such a performance control decision has two outcomes:

1. Instruct IterSort to carry on executing with the same sorting algorithm as previously used, or
2. switch the current sorting algorithm to one with a better sorting performance.

To facilitate experiments, IterSort has been designed to exhibit three characteristics:

Iterative: IterSort uses a suite of sorting algorithms to iteratively sort a sequence of arrays. During each iteration, an array is sorted using a particular sorting algorithm. After each iteration, the CPS can be invoked to perform any component-level performance control.

Variation: The variation in sorting time of each iteration occurs due to two reasons: the particular sorting algorithm being used and due to the input data. For example, if the array contains data that is almost sorted then the bubblesort sorting algorithm would perform very well.

Repeatable: To generate repeatable behaviour of IterSort across multiple executions, the arrays are initialised pseudo-randomly. The repeatable executions makes it possible to calculate a meaningful average and a standard deviation.

4.1.2. A component-level performance control algorithm for the CPS

The component-level performance control algorithm used for IterSort consists of two parts: prediction and switching.

Prediction

Prediction is used to predict how the current sorting algorithm would perform up to the next progress point. The progress point can be a component progress point, if the Perfrep-central or Perfrep-dist configurations are used; or an application progress point, in the case of the Perfrep-piggy or Perfrep-cache configurations.

The prediction is made using a simple NWS formula (Wolski 1998). The formula bases the prediction on the average taken over a fixed-

length performance information history. For the experiments, the length of the history is equal to the sorting times of last **T** available progress points. The prediction requires data retrieval. Data may be retrieved remotely, from application-Rep, or locally, from the component-level cache or the CPR, depending on the Perfrep configuration. If there is not enough information available for the current sorting algorithm, i.e. the last **T** sorting times of the current algorithm are not available, for example when the Perfrep starts executing, no prediction is made. This is because the prediction based on less than **T** elements might be too inaccurate to base a control decision. The value of **T** is ten for the experiment in this thesis.

Switching

IterSort always starts executing with the slowest sorting algorithm. The switching algorithm uses history to record the best performance of each sorting algorithm, i.e. the least sorting times achieved so far by that sorting algorithm. The sorting algorithms in the history are listed from slowest to the fastest. At start-up, this history is initialised to zero for all algorithms, denoting that the fastest available sorting algorithm is as yet unknown.

Once the component-level performance control algorithm is invoked, its switching part uses the prediction to get the expected performance of the current sorting algorithm up to the next progress point. Assuming that a prediction is available, then the following switching condition is tested:

```
pred > currts * (1.0 + tolerance/100.0)
```

If the prediction (`pred`) is greater than the *tolerance* (`tolerance`) of the sorting time (`currts`)of the current sort, then the sorting algorithm is switched with another sorting algorithm with the least sorting time. The tolerance, specified as a percentage, indicates how

much concession the component-level performance control algorithm is willing to offer to a poorly performing sorting algorithm. During the whole process, the history of a sorting algorithm is updated as soon as a sorting time is available for that sorting algorithm, and further updated if there is a shorter sorting time for that sorting algorithm. For selection of new sorting algorithm, the history is consulted for algorithm with the least sorting time in the order with which sorting algorithms are listed in the history. If there are multiple algorithms with the same least sorting time, then the first algorithm from that list is selected. Since, the history was initialised to zero at start-up, every sorting algorithm gets executed at least once, assuming that the switching condition is true enough times.

Critique: It is worth emphasizing that only one component-level performance control algorithm is used. Further, this component-level performance control algorithm has been synthesized to highlight the need for a component-level cache and a component-Rep. In practise, a CPS might benefit from various component-level performance control algorithms with varying accuracies of predictions.

There are other limitations to this component-level performance control algorithm. For example, if the performance of the component slowly deteriorates, the sorting algorithm would not be switched. The reason for this is that the current sort time (`currts`) is updated at every progress point. Moreover, the switching condition specified above compares the predicted performance (`pred`) with the current sort time (`currts`), within tolerance (`tolerance`). Therefore, if performance slowly deteriorates, the predicted performance, though worse than the current sorting time, would still be inside tolerance and therefore, search for a new sorting algorithm would not be performed.

The existing component-level performance control algorithm is conservative; i.e. a new sorting algorithm would not be selected unless the current sorting algorithm performs really well i.e. the current sort time is fairly small compared with previous sorting times. In other words, search for a new sorting algorithm would not be performed unless the switching condition is true.

For the pseudo code of component-level performance control algorithm, see Appendix 7.1.

4.1.3. An application-level performance control algorithm for the APS

Although application-level performance control is not studied empirically in this work, an implementation for an application-level performance control algorithm is suggested here by describing the behaviour of an APS that uses it. Such an APS "notices" that one of the components is consecutively progressing slower than the other components. If a certain threshold is exceeded, the APS instructs a slow component to migrate to another platform. As discussed in Section 3.6.2, the migration has been simulated by two platform configurations (machA and machB).

4.1.4. Layout

Figure 8 depicts the generic experimental test bed.

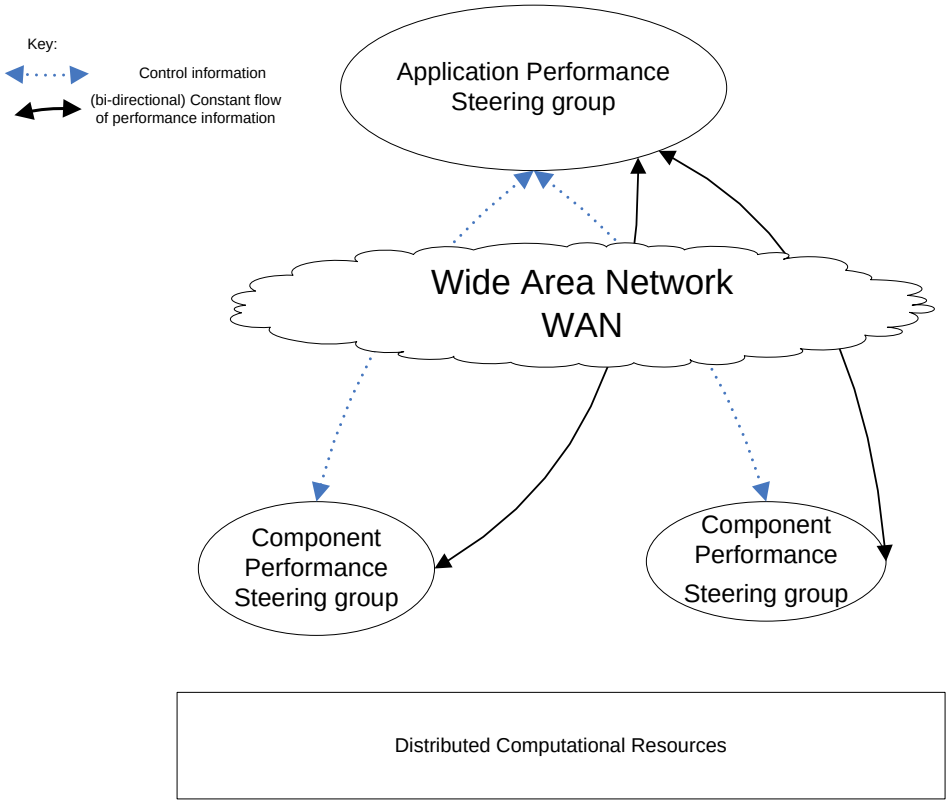

Figure 8 - Generic experimental test bed. A generic template for testing, which is used to perform all the experiments. Note that this figure is the same as Figure 3.

The experiments were carried out between two local area networks (LANs) connected via a WAN. Only one CPS group was used.

The Application Performance Steerer (APS) group may contain many entities. This group was hosted on a LAN at home. Similarly, a Component Performance Steerer (CPS) group may have many member entities. The CPS group was present on the LAN of The University of Manchester.

4.1.5. Measurements

The entities in the APS and CPS groups have been instrumented with calls to a timer, *gettimeofday()* --- a Linux library routine, to

measure the time taken for activities. These timings mainly split into two parts:

1. The time taken to execute between two component progress points --- in other words, the time taken by a component to sort an array. The total time taken for sorting all the arrays is represented as sort_time in the later bar charts.

2. The time taken not executing the component. This includes:

   a. The round-trip time taken by the CPS to check-in to the APS for status messages. The total round-trip time for check-in is represented as cps-aps checkin in the later bar charts.

   b. The round-trip time for updating the performance information storage, which is either application-Rep or the component-level cache, depending on the Perfrep configuration. The total round-trip time for this activity is expressed as update_Rep in the later bar charts.

   c. The round-trip for retrieving performance information from the performance information storage, which is either the application-Rep or a component-level cache, depending on the Perfrep configuration. This retrieval only happens if component-level performance control is active. The total round-trip time for retrieval is expressed as retrieve_Rep in the later bar charts.

   d. The following measurements are insignificant compared with the factors above and are not included in the later bar charts: the execution time of a particular sorting algorithm; the time for component-level performance control, if component-level performance control is active; writing performance information to a component-Rep, if there is one etc. These times are however given in Appendix 7.2.

The above timing results are written to disk in the form of comma separated values (csv) files. Each experiment for a particular Perfrep configuration has been repeated approximately 25 times. Any outliers from these results have been assessed by eye and removed from the results. The csv files have been imported into Microsoft Excel for further analysis. For every Perfrep configuration, a mathematical average and standard deviation have been calculated for the timings of the factors described above. Tables of all these experimental results together with averages and standard deviations, are given in Appendix 7.2. The average values have been used to draw the bar charts shown in the next few sections.

4.1.6. Other factors

The entities in the APS and CPS groups are executed at highest OS priority in order to minimise perturbations to the recorded values of the timers. The CPS group is executed in machB.

The average ping time between the two LANs over WAN is 25ms. The average ping time was assessed using 800 measurements of the *ping* utility in Linux. The minimum ping time was 21ms, while the maximum ping time was 75ms. The large variation in the ping time indicates the variability involved in communication between the two LANs.

The APS-CPS communication is used for control and status information. The design of PERCO dictates that a CPS needs to check-in to an APS at the application progress points. This communication is inevitable in every Perfrep configuration. The number of times such a communication happens is kept constant across the experiments. Only the parameters associated with the performance information are varied. Therefore, the round-trip-time for a CPS to check-in to an APS should stay constant throughout the experiments.

The time taken between two component progress points has been designed to be the order of the round-trip-time of a WAN communication, say between CPS-APS, for slower sorting algorithms. The reason for this is to avoid domination of round-trip time over time taken between two component progress points, or vice versa.

The specification of the resource hosting the APS group was: Pentium II i686 233 MHz, with 512 KB cache size and 128MB physical memory. The Linux operating system with kernel release 2.4.19-4GB was used. The compiler was gcc 3.2.

The specification of the resource hosting the CPS was: AuthenticAMD i686 2091 MHz, with 512 KB cache size and 512MB physical memory. The Linux operating system with kernel release 2.6.10-1.12_FC2 was used. The compiler was gcc 3.3.3.

The BerkeleyDB version used for both resources was 4.2.52.

4.2. Experimental results

The experimental results for different Perfrep configurations are described here.

4.2.1. Perfrep with centralised repository configuration (Perfrep-central)

Figure 9 illustrates the experimental results for the Perfrep-central configuration with no component-level performance control.

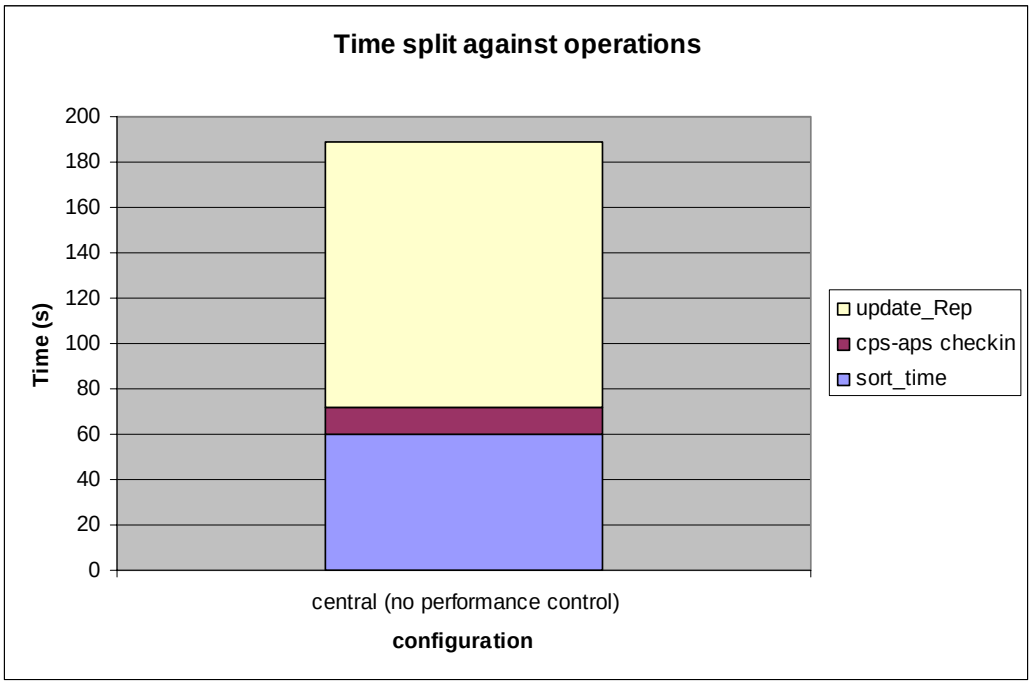

Figure 9 - Results for the Perfrep-central configuration for one cps with no component-level performance control. 67% of total execution time is spent in communication (mostly in updating application-Rep via the APR).

Perfrep-central is the simplest configuration, as depicted in Figure 4. The CPS communicates with the APR at every component progress point.

Figure 9 shows that 67% of the total execution time is spent not executing the component (i.e. not sorting). Breaking this overhead down further, 61% of the time is spent updating application-Rep via the APR.

If component-level performance control is active, then a reduced sorting time results. However, there is a further increase in communication overhead. Figure 10 compares the Perfrep-central configuration with no component-level performance control against the same configuration with component-level performance control.

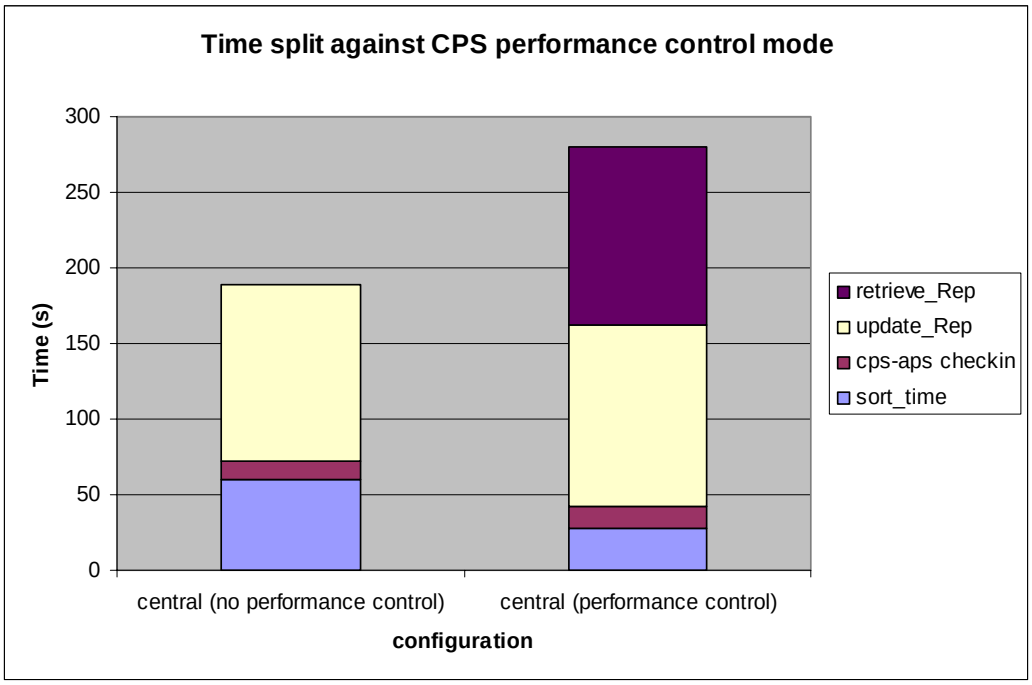

Figure 10 - Results for the Perfrep-central configuration (no component-level performance control against component-level performance control). Reduced sorting time ensures, but an increase in communication (due to retrieval of performance information from the application-Rep via the APR) is evident in the Perfrep-central configuration when component-level performance control is active.

Sorting time has been considerably reduced, from 32% to 10%, when component-level performance control is active. This is because the CPS successfully tunes the component by choosing progressively faster sorting algorithms. However, the overall execution time has increased by 48%. This increase is due to the fact that the performance information is retrieved from the application-Rep via the APR for component-level performance control.

4.2.2. *Perfrep with piggyback configuration (Perfrep-piggy)*

Figure 11 compares the Perfrep-central and Perfrep-piggy configurations when component-level performance control is inactive. The Perfrep-piggy configuration is depicted in Figure 5.

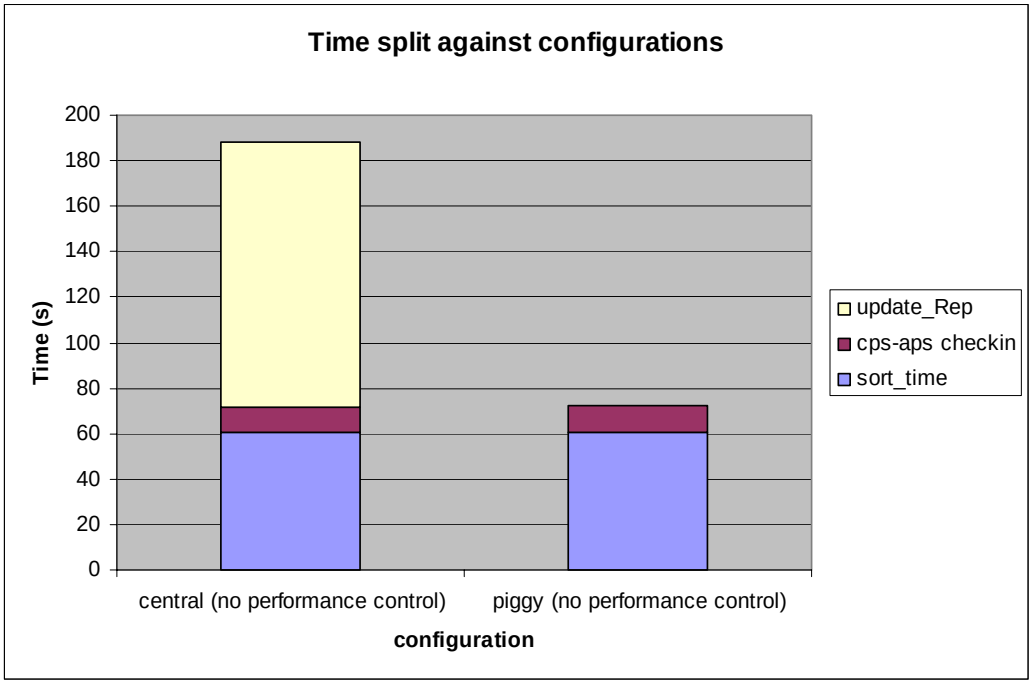

Figure 11 - Comparison of the Perfrep-central configuration against the Perfrep-piggy configuration with no component-level performance control. The Perfrep-central configuration spends 61% of total execution time in updating the application-Rep, whereas the Perfrep-piggy configuration avoids that by piggybacking the performance information.

It is worth emphasising that the Perfrep-central configuration dictates that the CPS stores performance information into the application-Rep via the APR at every component progress point over the WAN. Figure 11 shows that the update_Rep time is practically eliminated by the Perfrep-piggy configuration. The CPS now updates application-Rep with the performance information via the APS at the application progress points only. The round-trip-time for the CPS to

check-in to the APS does not increase, even though larger messages are exchanged in the Perfrep-piggy configuration. This indicates that latency is a critical factor and inevitable. Also, note that the time spent in sorting is the same (60s) in both cases.

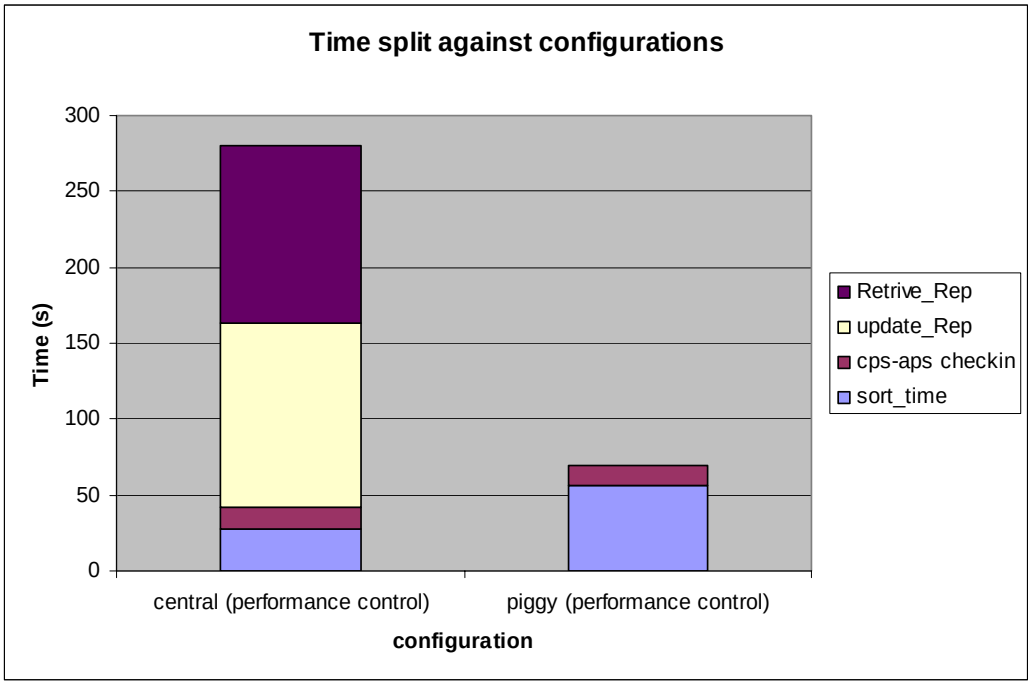

Figure 12 - Comparison of Perfrep-central against Perfrep-piggy when component-level performance control is active. The sorting time in the both configurations reduces because of the component-level performance control.

Figure 12 compares the Perfrep-central configuration against the Perfrep-piggy configuration when component-level performance control is active. Although the sorting times in both configurations reduces, it does so more in Perfrep-central (28s). This is because performance information for component progress points is not being stored by the CPS for the Perfrep-piggy configuration, and is lost between application progress points. This has the implication that the CPS needs to wait until enough performance information is available to make an accurate prediction, followed by any component-

level performance control decisions. Therefore, if component-level performance control is active, then the CPS takes a longer time to tune to faster sorting algorithms for the Perfrep-piggy configuration, giving a sort time of 56s.

In contrast, the Perfrep-central configuration tunes faster than the Perfrep-piggy configuration since the former has access to extra performance information, including performance information for the component progress points.

4.2.3. Perfrep with cached piggyback configuration (Perfrep-cache)

Figure 6 depicts the architecture of the Perfrep-cache configuration.

The component-level cache of the Perfrep-cache configuration is of the same size as the number of component progress points in an application phase.

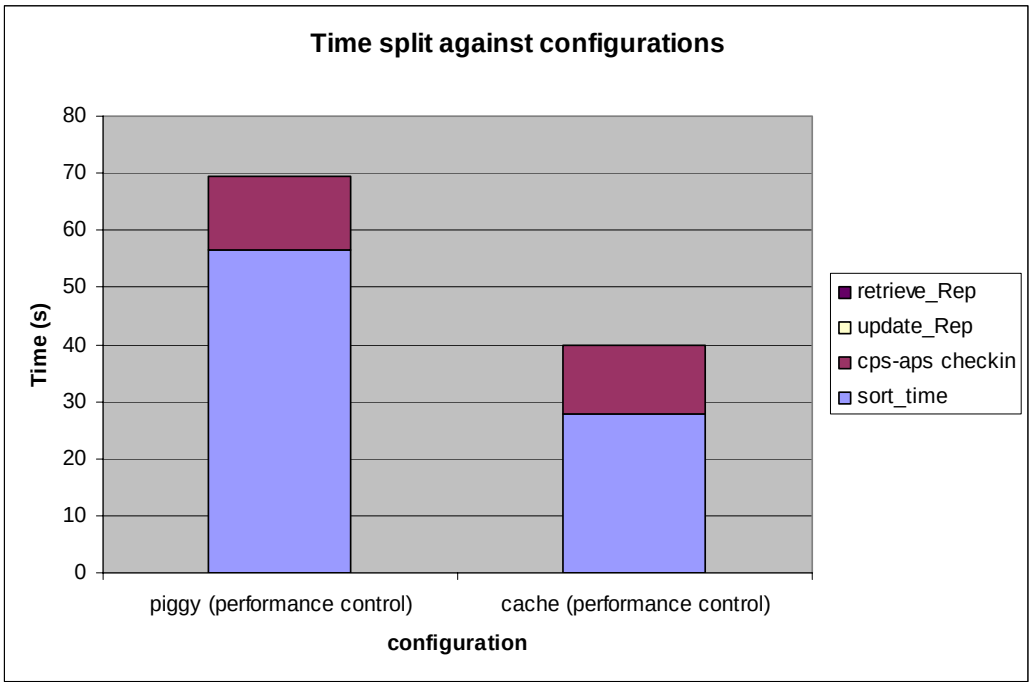

Figure 13 - Comparison of the Perfrep-piggy configuration against the Perfrep-cache configuration when component-level performance control is active. The sorting time in the Perfrep-cache configuration is shorter than the Perfrep-piggy configuration. There is (virtually) no cost associated with piggybacking performance information on the CPS-APS control and status messages.

Figure 13 compares the Perfrep-piggy configuration against the Perfrep-cache configuration with component-level performance control switched on. The sorting time in Perfrep-cache is 28s. This is because the cache of Perfrep-cache temporarily stores performance information of all the component progress points between the application progress points. Therefore, the ability to tune as fast as in the Perfrep-central configuration is regained.

However, this Perfrep-cache configuration also has some shortcomings. If the component-level performance control algorithm requires detailed history, then the cache may not be able to serve all

the requests, and requesting data from the application-Rep is inevitable. How this would affect the communication cost really depends on the frequency at which detailed performance information is required by the CPS.

4.2.4. Perfrep with distributed configuration (Perfrep-dist)

The Perfrep-dist configuration has been introduced in Figure 7.

The following diagram compares the performance of the Perfrep-cache configuration against the Perfrep-dist configuration.

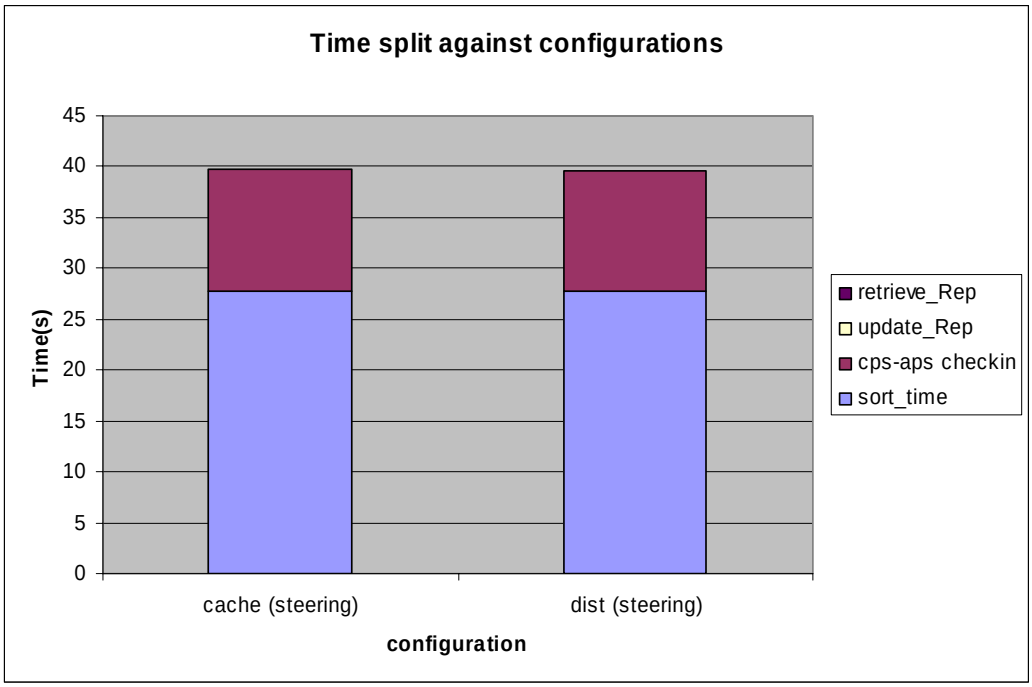

Figure 14 - Comparison of the Perfrep-cache configuration against the Perfrep-dist configuration when component-level performance control is active. All the times, including the sort time, are similar. Note that the CPR updates application-Rep via the APR separately in the Perfrep-dist configuration. This update does not require the CPS to block.

Figure 14 shows that all the times are similar.

The communication overhead is similar because, in the Perfrep-dist configuration, the CPS writes performance information locally to the CPR and carries on with the execution. Updating the application-Rep via the APR does not affect the execution time of the CPS. Instead, the CPR is responsible for management of the component-level cache, updating the APR, and a component-Rep using a file.

The sorting and tuning times are similar because the Perfrep-dist configuration has access to performance information at the component progress points, managed by the CPR in the component-level cache.

Recall that the Perfrep-dist configuration promotes separation of concerns and the possibility of updating application-Rep via the APR with a subset of performance information relevant to the APS.

4.2.5. Comparison of all the Perfrep configurations

It is interesting to compare the results of all the Perfrep configurations in a single bar chart, as depicted in Figure 15.

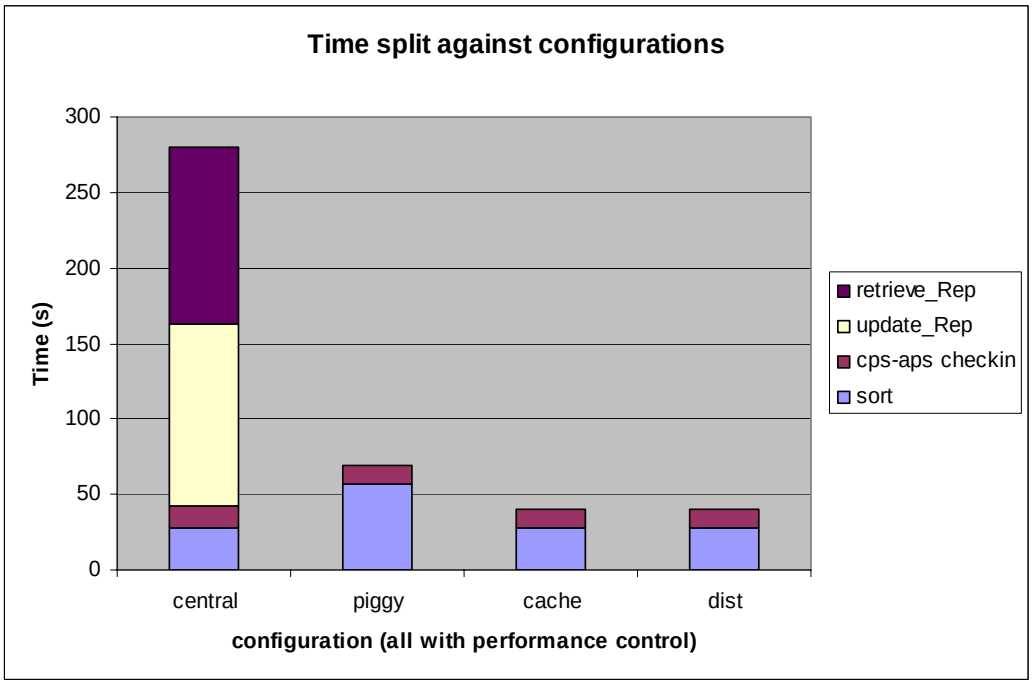

Figure 15 - Comparison of all the Perfrep configurations when component-level performance control is active.

It is evident that the communication costs play a major role for executing Perfrep over a WAN. Therefore, network communication becomes a performance bottleneck for this test case (and any other with similar ratio of execution time to CPS call-out). This is because most of the execution time is spent updating and retrieving performance information from the application-Rep, as in the case of the Perfrep-central configuration. In order to alleviate this, performance information may be piggybacked on to the communication messages. Firstly, this violates separation of concerns. Secondly, piggybacking affects the granularity of performance information that is stored, and performance information may be lost (as in the case of the Perfrep-piggy configuration). As a consequence, the CPS has access to less performance information. Therefore, tuning to the faster algorithms is slow and the sorting time is high. To rectify this, a component-level cache may be

introduced, as in the case of the Perfrep-cache configuration. This setup has low communication overhead and fast tuning. However, separation of concerns is still violated, as performance information is still piggybacked on control and status information. In addition, if the CPS requires a detailed history, then the component-level cache may not be able to serve all the requests, and requesting data from the application-Rep is inevitable. Therefore, the equally fast Perfrep-dist configuration ought to be used.

4.3. Evaluation of the Perfrep configurations against requirements set forth in Chapter 1

In this section, the Perfrep configurations are evaluated against the requirements set forth in Chapter 1.

Rapid availability of monitoring information: It is evident from the empirical results from different configurations that increasing locality of performance information improves the performance. This is due to two reasons. Firstly, the network communication overhead is reduced. Secondly and more importantly, a CPS can perform efficient component-level performance control in this test case. The Perfrep-cache and Perfrep-dist configurations promote component-level cache or component-Rep (collectively called component-level storage) at the CPS. Consequently, they can serve performance information requests from the CPS, more quickly. Therefore, these two configurations satisfy this requirement.

The Perfrep-central and Perfrep-piggy configurations do not have component-level storage at the CPS. Any data requests by the CPS needs to be served by application-Rep. Therefore, these configurations violate this requirement.

Non-intrusive: Any time spent on monitoring and controlling the application can be regarded as intrusive. In the above experiments, it is assumed that intrusiveness accounts for the time spent not executing the component. The CPS intrusiveness pays off if the component-level performance control algorithm is effective. However, it is evident from the experimental results that network communication cost lies at the heart of intrusiveness. In fact, it is so high in this test case that it dominates any other intrusiveness (such as managing the component-level storage). This is evident from the results of the Perfrep-central configuration, where more time is spent communicating than the actual execution of the component. Therefore, the Perfrep-central configuration violates this requirement. In other configurations, the intrusiveness is considerably decreased by reducing communication cost. This is achieved in two ways: piggybacking performance information and/or having component-level storage.

The Pefrep-piggy, Perfrep-cache and Perfrep-dist configurations are equally intrusive, even though they have different completion times for application execution. The completion time varies due to different sorting times. The sorting time differs due to varying effectiveness of component-level performance control, and the effectiveness varies due to granularity of performance information available. The Pefrep-piggy, Perfrep-cache and Perfrep-dist configurations meet this requirement, at least for this test case.

Minimal software demands on distributed resources: None of the configurations put more software demands on the distributed resources than necessary. The minimal demands include mechanisms to communicate with remote hosts, which are inevitable.

A CPS is implemented as C code. The CPS and the component are compiled into the same binary image. Any resource that executes the component can execute the combined CPS-component binary.

For all configurations, the application-Rep at an APS is borrowed from the PERCO prototype (Mayes et al. 2004), and is implemented as a BerkeleyDB database. BerkeleyDB can be statically compiled into the binary of the application that uses it. This allows the APS to execute on resources where BerkeleyDB may not be present.

The Perfrep-central and Perfrep-piggy configurations do not implement any component-level storage.

The Perfrep-cache configuration implements the component-level cache as an in-memory cache. Again, this implies the CPS can execute on any resource.

The component-level storage in a CPR in the Perfrep-dist configuration has two parts. One is an in-memory cache, which is similar to the component-level cache of the Perfrep-cache configuration. The second part contains more performance information history and is implemented as a file. The second part implies that the resource must provide permanent storage, which is not a great software demand compared with a installed database.

This requirement is met by all the configurations.

Optimised for performance information: Only the dynamic part of the performance information repository is implemented in the Perfrep configurations. The application-Rep is implemented using BerkeleyDB for all the Perfrep configurations. The current implementation of application-Rep uses a B+tree configuration in BerkeleyDB. A tree data structure is not optimal for insertion, since insertion is an O(logN) operation. The out-of-date data is not removed at present.

Since, the Perfrep-central and Perfrep-piggy configurations only implement application-Rep, they violate this requirement.

In addition to implementing application-Rep, the Perfrep-cache configuration also implements a component-level cache as a cyclic in-memory cache at a CPS. Such a cache is optimal for updates. Once the cache gets filled, it is overwritten with new entries. This implies that performance information older than the size of the cache is trimmed whether still relevant or not. Therefore, the Perfrep-cache configuration only partially meets this requirement.

Similarly, the Perfrep-dist configuration implements application-Rep. As mentioned previously, the Perfrep-dist configuration implements the component-level storage in two parts. The component-level cache is similar to the component-level cache of the Perfrep-cache configuration. The component-Rep implemented as a file is used to contain a longer history. Searching the file for particular performance information is complex but inserting performance information is a constant-time operation. In the current implementation of the Perfrep-dist configuration, there is no trimming of the out-dated performance information from the file. Therefore, the Perfrep-dist configuration only partially meets this requirement. This file is not trimmed, so that post-mortem analysis may be performed on the performance information of the component. This has the side-effect that the file could grow embarrassingly large and be spatially intrusive. The management of this file may be further investigated as a future study. This might include keeping only up-to-date

information in the component-Rep (file in this case) and performing post-mortem analysis from the application-Rep.[1]

Adaptive: Perfrep is an emulator of the PERCO design. Although the experiments are performed in a WAN environment, the environment is controlled. For example, the dynamics of a distributed system are not present in this system – neither the APS nor the CPS are assumed to crash. Perfrep has undefined behaviour for such a crash, if it happens. Therefore, this requirement is not met by any Perfrep configuration.

Post-mortem analysis: Potential benefits from post-mortem analysis have not been studied empirically as part of this work. However, it is anticipated that port-mortem analysis could be easily incorporated.

Performance information is stored at various granularities in different Perfrep configurations. Post-mortem analysis for this information could result in performance gains. However, these performance gains would be bounded by the granularity of performance information stored.

In the case of the Perfrep-piggy configuration, only performance information for application progress points is stored in application-Rep. It is hoped that post-processing this information into static performance information could result in performance gains at application-level only. The static performance information needs to be stored.

In the cases of the Perfrep-central and Perfrep-cache configuration, all performance information generated is stored in application-Rep.

---

[1] The current implementation of application-Rep is kept up to date with the performance information of component progress points.

Sophisticated post-processing is required to distinguish between performance information for application progress points and component progress points. If achieved, this could result in the performance gains at both application-level and component-level.

In the case of the Perfrep-dist configuration, all performance information generated is stored in the application-Rep. In addition, the performance information for components is stored in their component-Reps. Again, sophisticated post-processing is required at the application-Rep, as in the case of the Perfrep-central configuration, to distinguish between performance information for application progress points and component progress points. However, post-mortem analysis could be simplified if only performance information for application progress points is stored in the application-Rep. Then post-mortem analysis could be divided into two levels:

1. the post-mortem analysis of performance information for application progress points in the application-Rep, and
2. the post-mortem analysis of performance information for component progress points in the component-Reps.

This simplified post-mortem analysis setup could result in performance gains at both application-level and component-level.

Since the necessary infrastructure is present to incorporate post-mortem analysis but not an implementation, this requirement is partially met by all Perfrep configurations, but to differing degrees. In other words, this requirement is partially met by the Perfrep-piggy and Perfrep-cache configurations, but to a greater extent by the Perfrep-central configuration, and even more by the Perfrep-dist configuration.

Post-mortem analysis has been used in practise elsewhere and found to be useful (Chung and Hollingsworth 2004).

Relevant information in repository: The application-Rep is intended for application-level performance control. Similarly, the component-Rep is intended for component-level performance control. Since the Perfrep-piggy configurations only store performance information for application progress points in the application-Rep, it meets this requirement.

In contrast, the Perfrep-central and Perfrep-cache configurations store all the generated performance information in the application-Rep and violate this requirement.

Similarly, the Perfrep-dist configuration stores all the generated performance information in the application-Rep. However, performance information for component progress points is stored in the component-Rep of a component. Therefore, the Perfrep-dist configuration only partially meets this requirement.

For all the Perfrep configurations, static performance information needs to be stored in the static part of the performance information repository. Perhaps a database can be used as the static part of the performance information repository.

4.4. Discussion

Table 2 summarises the Perfrep configurations reviewed in Section 4.3 in terms of their meeting the requirements from Chapter 1.

Systems	Rapid availability	Non-intrusive	Minimal demands	Optimised	Adaptive	Post-mortem analysis	Relevant information
Perfrep-Central	×	×	√	×	×	–	×
Perfrep-piggy	×	√	√	×	×	–	√
Perfrep-cache	√	√	√	–	×	–	×
Perfrep-dist	√	√	√	–	×	–	–

Table 2 – The Perfrep configurations compared against performance information repository requirements from Chapter 1. Key: √ means requirement met, × means requirement not met, – means requirement partially met.

From Table 2, the Perfrep-dist configuration is the best configuration implemented. In addition, the following three points are worth highlighting:

1. The Perfrep-dist configuration may be extended to satisfy the requirement of relevant information by only storing performance information for application progress points in the application-Rep.

2. The Perfrep-dist configuration is as efficient as the Perfrep-cache configuration, with a clear separation of concerns, the possibility of serving detailed requests from the component-level cache and isolated execution of a CPS.

3. However, the Perfrep-dist configuration implies more resource consumption on the computing resources by using the component-Rep. It is envisaged that such implications might be a small price to pay as compared to the gains. These consequences have not been studied in detail.

4.5. Summary

The experimental method for evaluation has been described. This entailed the use of a synthetic test case component which allowed a degree of component-level performance control (but no application-level performance control). The experimental test bed has been described and the experimental measurements defined.

Each Perfrep configuration has been empirically studied, followed by a brief discussion leading (mostly) to the next configuration.

As can be seen from the results of the Perfrep-central configuration, a considerable amount of time is spent in transferring of information over the WAN. Network communication costs may be reduced by piggybacking performance information on existing control messages, but this results in the loss of efficient tuning for the sorting example used as an illustration. In contrast, localising performance information at component-level reduces communication time while maintaining efficient tuning.

This chapter concludes by comparing the Perfrep configurations against the requirements for performance repository requirements laid in Chapter 1.

5

Conclusions

This chapter starts by summarizing the investigation undertaken in this study in Section 5.1. The contributions are stated in Section 5.2. The context in which where the work presented in this thesis might be useful is discussed in Section 5.2. Benefits and limitations of using the techniques employed by Perfrep are outlined in Section 5.3. The chapter concludes by suggesting future work in Section 5.4.

5.1. Summary of investigation undertaken

It has been highlighted that monitoring of a geographically distributed component-based application is critical to effective performance control. Requirements have been identified that are needed in a performance information repository that can support performance control in a grid. This list has been extracted from the published literature.

A literature review has been undertaken. It is worth noting that monitored information needs to be managed in many systems across many domains from performance control to monitoring. However, the focus may be different. For example, rapid availability of performance information is critical in performance control systems but not so in a performance modelling system. The literature review consists of two parts:

1. The first part discusses existing systems: (a) that may have a performance information repository component as part of their architecture, or (b) that present useful ideas for a performance information repository.

2. The second part discusses the suitability of using a conventional database as a performance information repository.

For each system, a brief description of its functionality has been presented, followed by any performance information management issues addressed by the system.

Three points from the literature review that are worth re-iterating:

1. Performance information management is a "building block" in the overall system. Therefore, it is not necessarily discussed in detail in the cited literature.

2. Generally, monitored information generated by producers is transferred to consumers located remotely. There are at least two problems associated with such a setup:

    a. Firstly, the transfer of information may have large communication overheads.

    b. Secondly, the consumer is remote in all of the systems reviewed here. There might be benefits in localising consumers at the producer-site. For example, part of a performance control system might be localised at the site that is executing the application component.

Following this, the focus has been to study the performance information repository from the view of a performance control system for a grid application. In particular, the focus has been to study:

1. the reduced communication overhead of managing local performance information repository i.e. component-level storage at the distributed sites, and

2. gains in the performance of a distributed application by performing component-level performance control.

The design of an existing performance control system, PERCO, has been used. The roles of the various building-blocks in the PERCO

design have been specified. The model of operation for PERCO design has been presented. The PERCO design explicitly makes a two-level distinction for the performance control of an application consisting of components. Component-level performance control is concerned with the performance control of a component only by adjusting the control parameters of a component. Application-level performance control is concerned with the re-distribution of resources to the components to achieve better performance of the overall application.

Thereafter, the implementation of a performance information repository, Perfrep, based on the PERCO design has been described. Perfrep allows different configurations of performance information repository to be investigated. These configurations differ in the way that they manage performance information and its provisioning for component-level performance control. Possible implementation issues with these configurations have been highlighted; in particular, frequent network communication may result in wasting of component execution time.

A performance information repository in a grid has many aspects that need to be studied empirically. The experiments presented here have emphasized the importance of localising performance information in a WAN environment, in which a grid is likely to operate and the impact of component-level performance control.

Simple implementations for a component, component-level performance control algorithm for the CPS and application-level performance control algorithm for the APS have been presented.

It has been established that network communication costs indeed play a major role when executing Perfrep over a WAN. This communication is definitely a performance bottleneck. This is because most of the execution time is spent updating and retrieving

performance information from the application-Rep, as in the case of the Perfrep-central configuration. In order to alleviate this, performance information may be piggybacked on to the communication messages. However, firstly, this violates separation of concerns and, secondly, piggybacking affects the granularity of performance information stored and performance information is lost as in the case of the Perfrep-piggy configuration. As a consequence, the CPS has access to less performance information. Therefore, tuning to the faster sorting algorithms is slow and the sorting time is correspondingly high. To rectify this, a component-level cache has been introduced, as in the case of the Perfrep-cache configuration. This setup has low communication overhead and fast tuning. However, separation of concerns is still violated, as performance information is still piggybacked on control and status information. In addition, if the CPS requires a detailed history, then the component-level cache might not be able to serve all the requests, and requesting data from the application-Rep becomes inevitable. Therefore, the equally fast Perfrep-dist configuration might be used.

Finally, all the of the Perfrep configurations have been evaluated against the requirements laid out in Chapter 1 for a performance information repository to be used in a performance control system.

5.2. Contributions

This is the first study that presents an implementation for managing performance information across distributed resources on which components of a component-based application are being executed. In addition, this investigation empirically demonstrates some effects of local performance control at the distributed resource.

This work investigates different architectures of managing the performance information. Advantages and disadvantages of these

architectures have been studied. All the experiments were performed over the wide area network.

This work is particularly suitable for those systems that generate plentiful performance information at the distributed sites, and require that performance information both locally and on a remote site. In addition, this work is particularly valuable to those systems that require minimal overhead in managing such information.

The work presented is directly applicable to the PERCO prototype.

5.3. Benefits and Limitations

Perfrep is an emulator and implementation of the PERCO design. This work empirically highlights effective management of performance information (**rapid availability),** and intrusiveness **(non-intrusive)** of such a process. Not all the requirements laid out in Chapter 1 have been investigated in detail experimentally.

Perfrep was not incorporated into a performance control system and tested with real applications. In contrast, the Perfrep configurations have been empirically studied with a synthetic component that simulate an application. As discussed in Section 4.1.1, the components were simple sorting segments of code. Sorting is a well studied area. The sorting algorithms used are also well understood. However, a combination of a sorting segment with different available sorting algorithms were intentionally used to highlight the possible benefits from the component-level performance control. Real applications may not have natural progress points and well understood algorithms. There is only one component-level performance control algorithm used with simple prediction. For a real component-based application, a range of sophisticated component-level performance control algorithms may be required to truly exploit component-level performance control.

Moreover, the Perfrep was implemented with the assumption that there is no interaction between the components. In practise, such an assumption may not be true. The application components may and do interact with each other. Such interactions may have impact on the overall application behaviour. For example, a component may require information from another component as an input.

Minimal experiments were performed to verify that the Perfrep configurations can perform application-level performance control, although the experiments are not documented in this thesis due to limited time. Ideally, application-level performance control needs to be studied along with component-level performance control.

The experiments have been performed in a controlled WAN environment. For example, the computing resources were statically assigned for the APS and the CPS. Typically, a grid is dynamic, and expects a grid application to be adaptive. Achieving adaptivity is suggested as a possible future work.

The distributed resources were exclusively assigned to the APS and the CPS. In practice, such distributed resources may be part of the grid, servicing other requests simultaneously. This aspect has not been investigated as part of this work.

5.4. Future Work

The future work for Perfrep can be categorized in three categories: (a) detailed and experimental study of the outstanding requirements, (b) evaluating the Perfrep configurations in a realistic environment and (c) other interesting leads.

5.4.1. Outstanding requirements

In Section 4.3, Perfrep is evaluated against the requirements outlined in Chapter 1. Section 1.3 makes it clear that the implementations of the Perfrep configurations were constructed in order to empirically study: (a) the rapid availability of performance information to the CPSs and (b) the intrusiveness of managing the performance information.

The intrusiveness of managing the performance information is dominated by the network communication cost, as can be seen in the experimental results. Although intrusiveness has been discussed as part of this thesis, a much wider and detailed study of intrusiveness needs to be performed. This might include studying the spatial intrusiveness of the local storage at CPR.

Storage of the performance information is handled in three ways: a component-level cache implemented as an in-memory cache, a component-Rep implemented as a file, and an application-Rep implemented as BerkeleyDB. The suitability of these storage systems is evaluated in Section 4.3. A detailed investigation into a storage system whose data structures are optimised for the unique characteristics of dynamic performance information needs to be done. Furthermore, there is no way of recognising out-of-date dynamic performance information and then its removal. This is another possible future avenue.

Also, there is no post-mortem analysis of the performance information collected. Experience across multiple executions of the application is useful, as discussed in Section 4.3. Therefore, a study of effective post-mortem techniques for producing static performance information from the dynamic performance information might be carried out.

Application-level performance control might need only a subset of the performance information being generated by the components. Currently, there is no logic to decide what information is relevant for application-level performance control and then to instruct the CPSs or the CPRs to deliver the relevant information only. Therefore, selective update of performance information, and having relevant information at different levels (i.e. application-level and component-level) of Perfrep, may make an interesting future study.

Security issues in managing performance information have not been pursued in this work, and are left for future investigation.

As mentioned previously, the empirical evaluation of Perfrep has been performed in a controlled environment. For example, the dynamics of a distributed system were not present in this system – the APS or the CPS did not crash. Studying infrastructural adaptivity and repository robustness are interesting issues left for further study.

The Perfrep configurations have been implemented and evaluated in a distributed environments using Linux sockets, and not using a particular grid implementation such as Globus. The lessons learnt are still valid for a grid. However, re-implementing the communication routines of the Perfrep configurations to use a particular grid middleware has been left for further study.

5.4.2. Realism

The component used in the experiments is a simple sorting program. As mentioned in Section 5.3, sorting is a well studied area and the sorting algorithms used are also well understood. Moreover, a simple component-level performance control algorithm has been used. The experiments have been performed with the synthetic application and the simple component-level performance control algorithm to

demonstrate that localised information can make this component faster by tuning it locally. An interesting study might be to evaluate Perfrep with a real application with not-so-well understood tuning parameters, with different component-level performance control algorithms of varying prediction accuracy.

In addition, application-level performance control has not been empirically studied as part of this work. Hence, this is another possible future work. With application-level performance control implemented in the future, a question arises whether the simultaneous decision of a APS and the CPS can make a component perform worse. An investigation could be undertaken to promote situations in which both the APS and the CPS can mutually work to make the components go faster.

One of the requirements that a grid places on the application is the ability to cope with dynamic behaviour. Consequently, the performance control systems and Perfrep must be adaptive to the dynamic behaviour and heterogeneity of grid. In other words, Perfrep should be adaptive to any fluctuations, such as varying load on the resources. Similarly, Perfrep should be fault tolerant. Things can and do go wrong in distributed systems. Therefore, it is essential that Perfrep should be able to identify failures and recover from them. In addition, Perfrep is statically deployed on the particular resources. In a true performance control system, the CPSs can be deployed to any resources for execution. Perfrep needs to be extended to cope with this.

The CPSs block when they check-in to an APS. The APS responds with a continue or migration command. The CPSs can do useful execution instead of blocking. If the APS is instructing the CPSs to continue most of the time, then the ability to recognise this pattern

and implement non-blocking CPSs may yield an improve in performance.

If non-blocking CPSs are implemented, they can be extended to execute in isolation. In other words, if the APS is not responding to the requests of CPSs then the CPSs may keep executing in isolation with some customisable policy.

Scalability of Perfrep has not been studied here. It may be investigated by experimenting on Perfrep with a different number of CPSs and seeing their effects on the application-Rep. This implies that the application-Rep might become a bottleneck and, hence, suggests an investigation into a distributed APS with a number of application-Reps.

So far, the Perfrep prototype has been tested with only a small number of performance parameters. The effect of changing the number of performance parameters on the performance control using Perfrep is as yet unstudied. An interesting study can look into this.

5.4.3. Other interesting leads

While implementing Perfrep, it was observed that there seems to be a trade-off between ease-of-use and efficiency. For example, BerkleyDB is an embedded database with less richer query semantics than a relational database. This implies that the custom code is required to achieve sophisticated queries and, consequently, any changes to queries result in this custom code having to be modified. This is at best inconvenient. There were two reasons for using the BerkeleyDB for the implementation of application-Rep. Firstly, BerkeleyDB can be statically compiled into the application binary, not requiring any BerkeleyDB installation on the computing resources that execute components. Secondly, BerkeleyDB was expected to be faster than a relational database (an untested opinion).

Assuming, the BerkeleyDB is the optimal choice for storage, there is a need to define the balance between ease-of-use and efficiency. For example, if efficiency is not a major concern, then a more generic data format may be used.

At the moment, a CPS, a component and a CPR (if present) are compiled into a single binary image. If these entities were to be dynamically bound, that would allow more flexibility.

There might be external parties that might be interested in the performance information produced by a CPS. The external parties may be able to register with a CPS. This scheme might require some kind of registry for enquiry of a CPS with certain information and dynamic binding of parties to CPSs.

6

References

Adve, V. S., R. Bagrodia, et al. (2000). "POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems." IEEE Transactions Software Engineering **26**(11): 1027-1048.

Berman, F., A. Chien, et al. (2001). "The GrADS project: Software support for high-level grid application development." International Journal of High Performance Computing Applications **15**(4): 327-344.

Berman, F., R. Wolski, et al. (2003). "Adaptive computing on the grid using AppLeS." IEEE Transactions on Parallel and Distributed Systems **14**(4): 369-382.

Boghosian, B. M., P. V. Coveney, et al. (2000). "A three-dimensional lattice-gas model for amphiphilic fluid dynamics." Proceedings of the Royal Society of London, Series A (Mathematical, Physical and Engineering Sciences) **456**(1998): 1431-54.

Caltech (2005). "Synthetic aperture radar atlas project http://www.cacr.caltech.edu/SDA/digital_puglia.html."

Cao, J., D. J. Kerbyson, et al. (2000). "Performance modeling of parallel and distributed computing using PACE." IEEE International Performance, Computing & Communications Conference, Proceedings: 485-492.

Chin, J. and P. V. Coveney (2004). "Towards tractable toolkits for the Grid: a plea for lightweight, usable middleware." London, University College London.

Chung, I.-H. and J. K. Hollingsworth (2004). "Automated cluster-based Web service performance tuning." Proceedings. 13th IEEE International Symposium on High Performance Distributed Computing, 4-6 June 2004, Honolulu, HI, USA, IEEE.

Chung, I.-H. and J. K. Hollingsworth (2004). "Using Information from Prior Runs to Improve Automated Tuning Systems." Proceedings of the 2004 ACM/IEEE conference on Supercomputing, IEEE Computer Society.

Cooke, A., A. J. G. Gray, et al. (2003). "R-GMA: an information integration system for grid monitoring." On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE. OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2003. Proceedings, 3-7 Nov. 2003, Catania, Sicily, Italy, Springer - Verlag.

Czajkowski, K., S. Fitzgerald, et al. (2001). "Grid Information Services for Distributed Resource Sharing." In Proceedings of the Tenth IEEE 13 International Symposium on HighPerformance Distributed Computing (HPDC10). IEEE Press.

Foster, I. and C. Kesselman (2004). "The grid: blueprint for a new computing infrastructure." Amsterdam; Oxford, Morgan Kaufmann : Elsevier Science.

Foster, I., C. Kesselman, et al. (1996). "The Nexus Approach to Integrating Multithreading and Communication." Journal of Parallel and Distributed Computing **37**(1): 70.

Kaufman, J. H., G. Deen, et al. (2003). "Grid Computing Made Simple." Industrial Physicist **9**(4): 31-33.

Levy, R., J. Nagarajarao, et al. (2003). "Performance management for cluster based Web services." 8th International Symposium on Integrated Network Management, 24-28 March 2003, Colorado Springs, CO, USA, Kluwer Academic Publishers.

Mayes, K. (2005). "Personal communications."

Mayes, K., G. D. Riley, et al. (2003). "The Design of a Performance Steering System for Component-Based Grid Applications." Performance Analysis and Grid Computing. V. Getov, M. Gerndt, A. Hoisie, A. Malony and B. E. Miller, Kluwer Academic Publishers**:** 111-127.

Mayes, K. R., M. Lujan, et al. (2004). "Towards Performance Control on the Grid." Submitted to Philosophical Transactions of the Royal Society of London Series A.

Nanda, M. G., S. Chandra, et al. (2004). "Decentralizing execution of composite web services." 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'04, Oct 24-28 2004, Vancouver, BC, Canada, Association for Computing Machinery, New York, NY 10036-5701, United States.

Olson, M. A., K. Bostic, et al. (1999). "Berkeley DB." USENIX Annual Technical Conference, Monterey, California, USA.

Ozsu, M. T. and P. Valduriez (1999). "Principles of distributed database systems." Upper Saddle River, NJ ; London, Prentice Hall : Prentice-Hall International (UK).

Prodan, R. and T. Fahringer (2003). "A Web service-based experiment management system for the Grid." International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, IEEE Comput. Soc.

RealityGrid (2005). " The RealityGrid project. http://www.realitygrid.org/."

Ribler, R., J. Vetter, et al. (1998). "Autopilot: Adaptive Control of Distributed Applications." Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago.

Sahai, A., V. Machiraju, et al. (2002). "Automated SLA monitoring for Web services." Management Technologies for E-Commerce and E-Business Applications. 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2002. Proceedings, 21-23 Oct. 2002, Montreal, Que., Canada, Springer-Verlag.

Su, A., F. Berman, et al. (1999). "Using apples to schedule simple SARA on the Computational Grid." International Journal of High Performance Computing Applications **13**(3): 253-262.

Tierney, B., R. Aydt, et al. (2002). "A Grid Monitoring Architecture." Global Grid Forum Performance Working Group. http://www.ggf.org/documents/GFD.7.pdf.

Vetter, J. and K. Schwan (1999). "Techniques for High-Performance Computational Steering." IEEE Concurrency **7**(4): 63-74.

Vetter, J. S. and D. A. Reed (2000). "Real-time Performance Monitoring, Adaptive Control, and Interactive Steering of Computational Grids." CALTECH ASCI TECHNICAL REPORT 053.

Wolski, R. (1998). "Dynamically forecasting network performance using the Network Weather Service." Cluster Computing **1**(1): 119-32.

Wolski, R., N. T. Spring, et al. (1999). "The Network Weather Service: a distributed resource performance forecasting service for metacomputing." Future Generation Computer Systems **15**(5-6): 757-68.

7

Appendices

7.1. A component-level performance control algorithm for the CPS

This appendix presents code for a component-level performance control algorithm for the CPSs for the Perfrep configurations. This algorithm is same for different Perfrep configurations except the way it retrieves performance information for prediction.

In the component-level performance control algorithm, a major timestep is synonymous to the phase between two application progress points and a minor timestep is synonymous to the execution between two component progress points.

The following is the component-level performance control algorithm for the Perfrep-dist configuration.

```
/* associate the name of sorts to numbers */
#define TOTAL_ALOG 7
#define BUBBLE_SORT 0
#define INSERTION_SORT 1
#define SELECTION_SORT 2
#define SHELL_SORT 3
#define HEAP_SORT 4
#define MERGE_SORT 5
#define QUICK_SORT 6
#define NWS_ELEMENTS 10

/* mapping of alog to their performance
Index is used to for algorithm type
Associated value is used to record best performace
for that index */
double alogs[TOTAL_ALOG];

...
/* as part of start up routine for CPS when the 1st
sort time is encountered, initialize alogs[].
BUBBLE_SORT is the default sort algorithm. tsarg
is a struct used to report timing info */
    int i;
    for(i=0; i<TOTAL_ALOG; i++)
    {
```

```
      //i is equal to the current sort algorithm
      if(i == tsarg->alog)
      {
           // set current sort time to be
           // the history of current

           // algorithm
          algos[i] = tsarg->timestep;

      }
      else
      {
      // initialize performance to 0, so every
      // algorithm has a chance
          algos[i] = 0;

      }
    }
...

/*
  WARNING: inappropriate for slowly deteriorating
performance, since current steering only makes a
comparison with current timestep, not the history.
*/
// allow a new algorithm to at least execute
// NWS_elements times for a smoother

// prediction
static u_int counter = 0;


int cpsSteering(u_int progPt, double currts,
sortype alog)
{
  counter++;
  double pred = prediction(counter, currts);

  /*
    history:
      - performance of an alog
  */

  if((algos[alog] == 0) || (currts < algos[alog]))
  {
      // update best sort time
    algos[alog] = currts;
  }
```

```
// if pred is a lot worse than curr_ts then
// switch

if(pred > currts*(1.0+(double)tolerance/100.0))
 {
     /* search for alogrithm that has shown
     better performance than the current one */
   sortype new_alog = alog;
   double least_ts = currts;
   int index_alog;
   for(index_alog=0; index_alog<TOTAL_ALOG; inde
   x_alog++)
   {
     if (alogs[index_alog] < least_ts)
     {
          /* continue searching as may find even
          a better alogirthm with a lowerer sort
          time */
       new_alog = index_alog;
       least_ts = alogs[index_alog];

     }
   }

   /* see if we have got a new alog */
   if(new_alog != alog)
   {
     /* change alogrithm */
     // ChangeConf is a C struct
     ChangeConf _alog;
     _alog.alog = new_alog;
     compControl(CONT_CHANGE, &_alog,
     sizeof(ChangeConf));
     // reset counter so that new algorithm
     // perform NWS_ELEMENTS for a smoother
     // prediction
     counter = 0;
   }
   else
   {
     // continue with current configuration
     compControl(CONT_RESUME, 0, 0);
   }
 }
 else
 {
   // continue with current configuration
   compControl(CONT_RESUME, 0, 0);
```

```
  }

  return 0;
}

/*
  cpsSteering
    - prediction based on simple NWS formula,
average of last 10 sort timings
*/

double prediction(u_int counter, double currts)
{
  // can only perform sensible steering when can
  // accurate prediction
  // i.e. counter => NWS_ELEMENTS
  // therefore,
  if(counter >= NWS_ELEMENTS)
  {
      // TimestepArg is a C struct,
      // representing a record in repository


    TimestepArg *datarec = NULL;
    TimestepArg * rec = NULL;
    u_int ndata = NWS_ELEMENTS;

 // get NWS_ELEMENTS number of performance data,
 // which is retrieved differently for different
 // Perfrep-configurations
 // For example, Perfrep-dist retrieves data by
 // requesting CPS to retrieve it from CPR, as
 // follows:

    getDynamicData_cps(&datarec, &ndata);

    //NWS: calculate average for last 10 runs
    int ele = -1;
    double pred = 0.0;
    for(ele = 0; ele < ndata; ele++)
    {
      rec = &(datarec[ele]);
      // timestep here refers to the sort time
      pred += rec->timestep;
    }
    free(datarec);
    //including curr_ts
    return pred /= (double)(ndata);
   }
```

```
    // return we cant predict accurately
    return -1.0;
}
```

7.2. Experimental Results

This appendix contains the raw experimental results for the evaluations of the Perfrep configurations presented in Chapter 4.

A common description of the experimental results is presented here: format of the experimental results table, terms, outliers and analysis.

Format of the experimental result table: Each row in the table indicates an instance of the CPS execution. An average and the standard deviation of normal (non-outlier) results are at the end of the table. Each experiment for a particular Perfrep configuration has been repeated 25 times or more. Any outliers from these results are assessed by eye and removed from the results.

Terms: The terms used in the experimental results are defined in Table 3.

timesteps	Total number of component progress points, equivalent of the number of arrays sorted.
array_ele	Number of elements in the array. Each array has same number of elements that are pseudo-randomly initialized.
tolerance	Tolerance in percentage to the CPS performance control algorithm.
ttl_time	Total time taken by the CPS to execute.
sort_time	Total time taken to sort all the arrays.
time_bw_ts	Total time spent not sorting.
cps_aps	Cumulative round-trip-time for CPS to check-in to APS.
steering	time used by the component-level performance control decisions including data retrieval.
upAPR	Cumulative round-trip-time for updating the application-Rep via the APR with performance information (only applies to the Perfrep-central and Perfrep-dist configurations).
downAPR	Cumulative round-trip-time for retrieving performance information from APR (only applies to the Perfrep-central configuration).
upCPS/R	Cumulative round-trip-time for updating the component-level cache or the CPR with performance information (only applies to the Perfrep-cache and Perfrep-dist configurations, respectively).
downCPS/R	Cumulative round-trip-time for retrieving performance information from the CPS cache or the CPR with the performance information (only applies to the Perfrep-cache and the Perfrep-dist configurations respectively).
alog	Algorithm type: BUBBLE_SORT=0, INSERTION_SORT=1, SELECTION_SORT=2, SHELL_SORT=3, HEAP_SORT=4, MERGE_SORT=5, QUICK_SORT=6
Timestep	Number of times an algorithm was executed.
exe_time	Total sorting time of a sorting algorithm.
diff_mach	A Boolean to indicate if the CPS migrated while executing this algorithm. This field is unused.

Table 3 – Terms used in the experimental results.

7.2.1. Perfrep with centralised repository configuration (Perfrep-central)

Experimental results for the Perfrep-central configuration with no component-level performance control are presented here. These results are used in Figure 9, Figure 10 and Figure 11.

timesteps	array_ele	tolerance	ttl_time	sort_time	time_bw_ts	cps_aps	steering	upAPR	downAPR	upCPS/R	downCPS/R	alog	timestep	exe_time	diff_mach
1000	3000	1	188.9741	60.30642	128.6677	11.67177	-1	116.6314	-1	-1	-1	0	1000	60.30642	-1
1000	3000	1	189.6136	60.31534	129.2982	11.61851	-1	116.3189	-1	-1	-1	0	1000	60.31534	-1
1000	3000	1	192.6464	60.30003	132.3463	11.6128	-1	119.3732	-1	-1	-1	0	1000	60.30003	-1
1000	3000	1	189.9889	60.30238	129.6866	11.61771	-1	116.7098	-1	-1	-1	0	1000	60.30238	-1
1000	3000	1	190.0055	60.29595	129.7096	11.72858	-1	116.6214	-1	-1	-1	0	1000	60.29595	-1
1000	3000	1	189.9277	60.31358	129.6141	11.66969	-1	116.577	-1	-1	-1	0	1000	60.31358	-1
1000	3000	1	189.645	60.30438	129.3406	11.61433	-1	116.3682	-1	-1	-1	0	1000	60.30438	-1
1000	3000	1	189.6207	60.29462	129.3261	11.66452	-1	116.3031	-1	-1	-1	0	1000	60.29462	-1
1000	3000	1	189.9852	60.31232	129.6729	11.6197	-1	116.6866	-1	-1	-1	0	1000	60.31232	-1
1000	3000	1	189.5554	60.30135	129.254	11.62002	-1	116.2734	-1	-1	-1	0	1000	60.30135	-1
1000	3000	1	189.8216	60.30231	129.5193	11.65814	-1	116.499	-1	-1	-1	0	1000	60.30231	-1
1000	3000	1	188.7424	60.29636	128.4461	11.60921	-1	116.5051	-1	-1	-1	0	1000	60.29636	-1
1000	3000	1	189.6834	60.29626	129.3871	11.61399	-1	116.3636	-1	-1	-1	0	1000	60.29626	-1
1000	3000	1	190.1178	60.29428	129.8236	11.65684	-1	116.8123	-1	-1	-1	0	1000	60.29428	-1
1000	3000	1	189.7603	60.3079	129.4524	11.61763	-1	116.4623	-1	-1	-1	0	1000	60.3079	-1
1000	3000	1	189.6785	60.29691	129.3816	11.61564	-1	116.4102	-1	-1	-1	0	1000	60.29691	-1
1000	3000	1	189.5819	60.29507	129.2868	11.60875	-1	116.3204	-1	-1	-1	0	1000	60.29507	-1
1000	3000	1	189.6765	60.29472	129.3818	11.61365	-1	116.4043	-1	-1	-1	0	1000	60.29472	-1
1000	3000	1	189.7054	60.31253	129.3929	11.61801	-1	116.3997	-1	-1	-1	0	1000	60.31253	-1
1000	3000	1	189.6974	60.30158	129.3959	11.65455	-1	116.3823	-1	-1	-1	0	1000	60.30158	-1
1000	3000	1	189.6757	60.30644	129.3693	11.61817	-1	116.393	-1	-1	-1	0	1000	60.30644	-1
1000	3000	1	189.6585	60.29723	129.3612	11.66077	-1	116.341	-1	-1	-1	0	1000	60.29723	-1
1000	3000	1	190.0399	60.29483	129.7451	11.61309	-1	116.7593	-1	-1	-1	0	1000	60.29483	-1
1000	3000	1	189.5887	60.30524	129.2835	11.62112	-1	116.31	-1	-1	-1	0	1000	60.30524	-1
1000	3000	1	189.6833	60.30333	129.38	11.60429	-1	116.4178	-1	-1	-1	0	1000	60.30333	-1
1000	3000	1	190.2318	60.29384	129.938	11.64677	-1	116.9212	-1	-1	-1	0	1000	60.29384	-1
1000	3000	1	189.7864	60.30435	129.482	11.65798	-1	116.4635	-1	-1	-1	0	1000	60.30435	-1
1000	3000	1	189.6869	60.30708	129.3798	11.61999	-1	116.4006	-1	-1	-1	0	1000	60.30708	-1
avg			189.8135	60.30202	129.5115	11.63379		116.5867					1000	60.30202	
stddev			0.63072	0.006499	0.631281	0.028648		0.571813					0	0.006499	

Experimental results for the Perfrep-central configuration when component-level performance control is active are presented here. These results are used in Figure 10, Figure 12 and Figure 15.

timesteps	array_ele	tolerance	ttl_time	sort_time	time_bw_ts	cps_aps	steering	upAPR	downAPR	upCPS/R	downCPS/R	alog	timestep	exe_time	diff_mach
1000	3000	1	278.4421	27.73277	250.7094	14.23084	116.1662	119.9228	116.1523	-1	-1	0	35	2.111305	-1
1000	3000	1	280.2333	27.72832	252.505	14.16783	116.2268	120.7265	116.2117	-1	-1	0	35	2.11061	-1
1000	3000	1	280.0256	27.73035	252.2952	14.15254	116.1136	120.6022	116.0991	-1	-1	0	35	2.111244	-1
1000	3000	1	280.1976	27.73117	252.4664	14.09622	116.4938	120.4692	116.4793	-1	-1	0	35	2.11113	-1
1000	3000	1	280.57	27.73053	252.8395	14.1642	116.8261	120.4352	116.812	-1	-1	0	35	2.111182	-1
1000	3000	1	282.9611	27.72703	255.2341	14.75152	117.792	121.3012	117.777	-1	-1	0	35	2.110635	-1
1000	3000	1	285.4908	27.73019	257.7606	14.47355	118.8339	123.0662	118.8191	-1	-1	0	35	2.111097	-1
1000	3000	1	286.5047	27.73224	258.7725	14.27855	118.5857	122.9186	118.5712	-1	-1	0	35	2.111094	-1
1000	3000	1	279.9698	27.73281	252.237	14.08436	117.2006	119.5697	117.1863	-1	-1	0	35	2.110622	-1
1000	3000	1	278.654	27.73016	250.9239	14.17964	115.8537	119.5044	115.8386	-1	-1	0	35	2.110921	-1
1000	3000	1	280.1902	27.7261	252.4641	14.14418	116.4855	120.4492	116.4716	-1	-1	0	35	2.111717	-1
1000	3000	1	279.9825	27.73132	252.2512	14.1628	116.7609	119.9402	116.7462	-1	-1	0	35	2.110832	-1
1000	3000	1	280.2572	27.7266	252.5306	14.35067	116.7748	120.0147	116.7603	-1	-1	0	35	2.110489	-1
1000	3000	1	280.8117	27.73047	253.0812	14.61756	116.8644	120.2091	116.8497	-1	-1	0	35	2.111029	-1
1000	3000	1	280.4229	27.72992	252.693	14.61333	116.5653	120.1284	116.5505	-1	-1	0	35	2.110813	-1
1000	3000	1	281.9916	27.72893	254.2627	14.38921	117.0861	121.3898	117.0709	-1	-1	0	35	2.110432	-1
1000	3000	1	280.2812	27.73147	252.5497	14.39954	116.469	120.2924	116.4549	-1	-1	0	35	2.110846	-1
1000	3000	1	282.7137	27.72657	254.9871	14.27777	117.5906	121.7027	117.5757	-1	-1	0	35	2.111198	-1
1000	3000	1	284.0497	27.728	256.3217	14.19846	119.5716	121.1623	119.5569	-1	-1	0	35	2.110887	-1
1000	3000	1	279.6351	27.73207	251.903	14.23713	116.308	119.9696	116.2935	-1	-1	0	35	2.11096	-1
1000	3000	1	280.0339	27.72812	252.3058	14.35478	116.4229	120.0996	116.4088	-1	-1	0	35	2.11056	-1
1000	3000	1	279.6389	27.72942	251.9095	14.53168	116.3159	119.6508	116.3012	-1	-1	0	35	2.111848	-1
1000	3000	1	282.7155	27.72938	254.9861	14.288	117.6375	121.6744	117.6233	-1	-1	0	35	2.110334	-1
1000	3000	1	282.6446	27.73035	254.9143	14.36911	117.3307	121.7975	117.3162	-1	-1	0	35	2.110745	-1
1000	3000	1	282.7051	27.72784	254.9773	14.2716	118.3551	120.9634	118.3405	-1	-1	0	35	2.110693	-1
Average			281.2449	27.72968	253.5152	14.3114	117.0652	120.7184	117.0507				35	2.110929	
Std. Dev			2.019103	0.001966	2.019424	0.175104	0.948267	0.959072	0.948207				0	0.00037	

alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach
1	17	0.429891	0	2	948	25.19157	0	3	0	0	0	4	0	0	0
1	17	0.429856	0	2	948	25.18785	0	3	0	0	0	4	0	0	0
1	17	0.429853	0	2	948	25.18925	0	3	0	0	0	4	0	0	0
1	17	0.430007	0	2	948	25.19004	0	3	0	0	0	4	0	0	0
1	17	0.429879	0	2	948	25.18947	0	3	0	0	0	4	0	0	0
1	17	0.429723	0	2	948	25.18667	0	3	0	0	0	4	0	0	0
1	18	0.454796	0	2	947	25.16429	0	3	0	0	0	4	0	0	0
1	17	0.429925	0	2	948	25.19122	0	3	0	0	0	4	0	0	0
1	18	0.455077	0	2	947	25.16711	0	3	0	0	0	4	0	0	0
1	17	0.429902	0	2	948	25.18934	0	3	0	0	0	4	0	0	0
1	20	0.505939	0	2	945	25.10844	0	3	0	0	0	4	0	0	0
1	17	0.429694	0	2	948	25.19079	0	3	0	0	0	4	0	0	0
1	17	0.429841	0	2	948	25.18627	0	3	0	0	0	4	0	0	0
1	17	0.429958	0	2	948	25.18948	0	3	0	0	0	4	0	0	0
1	17	0.4298	0	2	948	25.18931	0	3	0	0	0	4	0	0	0
1	17	0.429716	0	2	948	25.18878	0	3	0	0	0	4	0	0	0
1	17	0.429809	0	2	948	25.19082	0	3	0	0	0	4	0	0	0
1	18	0.454708	0	2	947	25.16066	0	3	0	0	0	4	0	0	0
1	18	0.455029	0	2	947	25.16208	0	3	0	0	0	4	0	0	0
1	17	0.429835	0	2	948	25.19128	0	3	0	0	0	4	0	0	0
1	17	0.429801	0	2	948	25.18776	0	3	0	0	0	4	0	0	0
1	17	0.429868	0	2	948	25.1877	0	3	0	0	0	4	0	0	0
1	17	0.429723	0	2	948	25.18933	0	3	0	0	0	4	0	0	0
1	17	0.430343	0	2	948	25.18926	0	3	0	0	0	4	0	0	0
1	17	0.429807	0	2	948	25.18734	0	3	0	0	0	4	0	0	0
		0.436911			947.72	25.18184									
		0.017144			0.678233	0.018111									

alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0

7.2.2. Perfrep with piggyback configuration (Perfrep-piggy)

Experimental results for the Perfrep-piggy configuration with no component-level performance control are presented here. These results are used in Figure 11.

timesteps	array_ele	tolerance	ttl_time	sort_time	time_bw_ts	cps_aps	steering	upAPR	downAPR	upCPS/R	downCPS/R	alog	timestep	exe_time	diff_mach
1000	3000	1	72.4165	60.28554	12.13096	11.96285	-1	-1	-1	-1	-1	0	1000	60.28554	0
1000	3000	1	73.40631	60.28922	13.11709	11.88932	-1	-1	-1	-1	-1	0	1000	60.28922	0
1000	3000	1	72.34494	60.28794	12.05701	11.89046	-1	-1	-1	-1	-1	0	1000	60.28794	0
1000	3000	1	72.29826	60.28507	12.01319	11.84667	-1	-1	-1	-1	-1	0	1000	60.28507	0
1000	3000	1	73.38894	60.28225	13.10669	11.88746	-1	-1	-1	-1	-1	0	1000	60.28225	0
1000	3000	1	72.3137	60.28225	12.03144	11.8683	-1	-1	-1	-1	-1	0	1000	60.28225	0
1000	3000	1	72.33853	60.28421	12.05432	11.88886	-1	-1	-1	-1	-1	0	1000	60.28421	0
1000	3000	1	72.33557	60.29193	12.04364	11.86747	-1	-1	-1	-1	-1	0	1000	60.29193	0
1000	3000	1	72.3319	60.29605	12.03585	11.86949	-1	-1	-1	-1	-1	0	1000	60.29605	0
1000	3000	1	73.37049	60.28511	13.08538	11.89269	-1	-1	-1	-1	-1	0	1000	60.28511	0
1000	3000	1	73.74287	60.29805	13.44482	12.22302	-1	-1	-1	-1	-1	0	1000	60.29805	0
1000	3000	1	72.35483	60.28923	12.0656	11.89779	-1	-1	-1	-1	-1	0	1000	60.28923	0
1000	3000	1	72.34735	60.28651	12.06084	11.84771	-1	-1	-1	-1	-1	0	1000	60.28651	0
1000	3000	1	72.30185	60.28418	12.01767	11.85146	-1	-1	-1	-1	-1	0	1000	60.28418	0
1000	3000	1	72.32594	60.29451	12.03143	11.86397	-1	-1	-1	-1	-1	0	1000	60.29451	0
1000	3000	1	72.35449	60.33082	12.02367	11.82846	-1	-1	-1	-1	-1	0	1000	60.33082	0
1000	3000	1	73.35778	60.28313	13.07464	11.8835	-1	-1	-1	-1	-1	0	1000	60.28313	0
1000	3000	1	72.32137	60.29134	12.03003	11.86217	-1	-1	-1	-1	-1	0	1000	60.29134	0
1000	3000	1	72.32536	60.28382	12.04154	11.84408	-1	-1	-1	-1	-1	0	1000	60.28382	0
1000	3000	1	72.30664	60.28669	12.01995	11.85537	-1	-1	-1	-1	-1	0	1000	60.28669	0
1000	3000	1	72.28426	60.28645	11.99781	11.83083	-1	-1	-1	-1	-1	0	1000	60.28645	0
1000	3000	1	72.61818	60.30046	12.31772	12.15175	-1	-1	-1	-1	-1	0	1000	60.30046	0
1000	3000	1	73.41001	60.28522	13.12479	11.88799	-1	-1	-1	-1	-1	0	1000	60.28522	0
1000	3000	1	72.33336	60.28411	12.04925	11.88046	-1	-1	-1	-1	-1	0	1000	60.28411	0
1000	3000	1	72.33214	60.29676	12.03539	11.87233	-1	-1	-1	-1	-1	0	1000	60.29676	0
1000	3000	1	72.36515	60.30186	12.0633	11.86257	-1	-1	-1	-1	-1	0	1000	60.30186	0
1000	3000	1	72.34567	60.31781	12.02786	11.86073	-1	-1	-1	-1	-1	0	1000	60.31781	0
1000	3000	1	72.31944	60.2965	12.02295	11.85735	-1	-1	-1	-1	-1	0	1000	60.2965	0
1000	3000	1	72.34831	60.3037	12.0446	11.86626	-1	-1	-1	-1	-1	0	1000	60.3037	0
1000	3000	1	72.36154	60.30347	12.05807	11.89017	-1	-1	-1	-1	-1	0	1000	60.30347	0
avg			72.56672	60.29247	12.27425	11.89272							1000	60.29247	
stddev			0.454873	0.011033	0.457042	0.084458							0	0.011033	

Experimental results for the Perfrep-piggy configuration when component-level performance control is active are presented here. These results are used in Figure 12, Figure 13 and Figure 15.

timesteps	array_ele	tolerance	ttl_time	Sort_time	time_bw_ts	cps_aps	steering	upAPR	downAPR	upCPS/R	downCPS/R	alog	timestep	exe_time	diff_mach
1000	3000	1	94.90543	56.42496	38.48047	31.00209	0.001007	-1	-1	-1	-1	0	890	53.6525	0
1000	3000	1	68.55848	56.42775	12.13073	11.96036	0.000984	-1	-1	-1	-1	0	890	53.65447	0
1000	3000	1	68.62048	56.42722	12.19326	12.02226	0.000991	-1	-1	-1	-1	0	890	53.65499	0
1000	3000	1	68.58334	56.42989	12.15345	11.98166	0.000966	-1	-1	-1	-1	0	890	53.65758	0
1000	3000	1	69.64113	56.45744	13.18369	11.93673	0.000984	-1	-1	-1	-1	0	890	53.68514	0
1000	3000	1	68.56119	56.42894	12.13225	11.96222	0.000992	-1	-1	-1	-1	0	890	53.65396	0
1000	3000	1	68.60921	56.46387	12.14534	11.97338	0.000968	-1	-1	-1	-1	0	890	53.69145	0
1000	3000	1	68.89299	56.43675	12.45624	12.28596	0.000993	-1	-1	-1	-1	0	890	53.66231	0
1000	3000	1	68.55871	56.42794	12.13077	11.95636	0.000997	-1	-1	-1	-1	0	890	53.65561	0
1000	3000	1	69.59679	56.43461	13.16218	11.96137	0.000991	-1	-1	-1	-1	0	890	53.66174	0
1000	3000	1	73.45217	60.29104	13.16112	11.9636	0.001065	-1	-1	-1	-1	0	1000	60.29104	0
1000	3000	1	68.60016	56.43735	12.16281	11.99235	0.001005	-1	-1	-1	-1	0	890	53.66407	0
1000	3000	1	69.61874	56.42922	13.18952	11.99391	0.000995	-1	-1	-1	-1	0	890	53.65586	0
1000	3000	1	68.57653	56.42943	12.1471	11.97629	0.000963	-1	-1	-1	-1	0	890	53.65684	0
1000	3000	1	68.57772	56.43846	12.13926	11.96786	0.000964	-1	-1	-1	-1	0	890	53.6661	0
1000	3000	1	68.55288	56.43334	12.11954	11.94937	0.000988	-1	-1	-1	-1	0	890	53.66083	0
1000	3000	1	68.61155	56.43864	12.1729	12.00318	0.000968	-1	-1	-1	-1	0	890	53.66572	0
1000	3000	1	69.60581	56.42543	13.18038	11.98406	0.000985	-1	-1	-1	-1	0	890	53.65288	0
1000	3000	1	68.6221	56.4343	12.1878	12.01731	0.000962	-1	-1	-1	-1	0	890	53.66036	0
1000	3000	1	68.55779	56.43548	12.12231	11.95014	0.000983	-1	-1	-1	-1	0	890	53.66298	0
1000	3000	1	68.57645	56.42602	12.15043	11.97676	0.000996	-1	-1	-1	-1	0	890	53.65346	0
1000	3000	1	74.94913	56.4336	18.51554	17.28409	0.000978	-1	-1	-1	-1	0	890	53.66119	0
1000	3000	1	68.56518	56.42745	12.13773	11.96658	0.000984	-1	-1	-1	-1	0	890	53.65527	0
1000	3000	1	62.45387	50.29244	12.16143	11.99321	0.000906	-1	-1	-1	-1	0	710	42.83043	0
1000	3000	1	72.45266	60.29936	12.15331	11.98437	0.001056	-1	-1	-1	-1	0	1000	60.29936	0
1000	3000	1	68.62376	56.4373	12.18646	12.01596	0.000988	-1	-1	-1	-1	0	890	53.66474	0
1000	3000	1	68.56988	56.42566	12.14421	11.97222	0.000976	-1	-1	-1	-1	0	890	53.65326	0
1000	3000	1	73.47175	60.29677	13.17498	11.9782	0.001046	-1	-1	-1	-1	0	1000	60.29677	0
Average			70.17735	56.62824	13.54911	12.85757	0.000989						895.3571	53.98503	
Stddev			5.328973	1.736356	5.038422	3.694415	3.04E-05						50.14661	3.02105	

alog	timestep	exe_time	diff_mach	Alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach
1	110	2.772455	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.773286	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772228	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772308	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772307	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.774981	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772428	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.774436	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772331	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772866	0	2	0	0	0	3	0	0	0	4	0	0	0
1	0	0	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.773284	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.773356	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772596	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772352	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772513	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.77293	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772544	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.773943	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772501	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.77256	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772405	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772179	0	2	0	0	0	3	0	0	0	4	0	0	0
1	180	4.538299	0	2	110	2.923707	0	3	0	0	0	4	0	0	0
1	0	0	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772555	0	2	0	0	0	3	0	0	0	4	0	0	0
1	110	2.772403	0	2	0	0	0	3	0	0	0	4	0	0	0
1	0	0	0	2	0	0	0	3	0	0	0	4	0	0	0
	100.7143	2.538787			3.928571	0.104418									
	37.90129	0.95547			20.78805	0.552529									

alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0

7.2.3. Perfrep with cached piggyback configuration (Perfrep-cache)

Experimental results for the Perfrep-cache configuration when component-level performance control is active are presented here. These results are used in Figure 13, Figure 14 and Figure 15.

timesteps	array_ele	tolerance	ttl_time	Sort_time	time_bw_ts	cps_aps	steering	upAPR	downAPR	upCPS/R	downCPS/R	alog	timestep	exe_time	diff_mach
1000	3000	1	39.91307	27.72748	12.18559	11.99255	0.011456	-1	-1	0.003516	0.003783	0	35	2.110478	0
1000	3000	1	39.83787	27.72342	12.11445	11.92024	0.011405	-1	-1	0.003539	0.003799	0	35	2.110551	0
1000	3000	1	39.95742	27.72613	12.23129	12.04023	0.011467	-1	-1	0.003557	0.00371	0	35	2.111147	0
1000	3000	1	39.91344	27.72606	12.18739	11.99485	0.011463	-1	-1	0.00353	0.003822	0	35	2.110143	0
1000	3000	1	39.95305	27.72495	12.22809	12.03624	0.011535	-1	-1	0.003572	0.003793	0	35	2.110484	0
1000	3000	1	40.04486	27.72356	12.32131	12.09882	0.011522	-1	-1	0.003538	0.00376	0	35	2.110993	0
1000	3000	1	39.91503	27.72529	12.18974	11.99696	0.011492	-1	-1	0.003492	0.003778	0	35	2.110189	0
1000	3000	1	39.96125	27.72439	12.23686	12.04463	0.01154	-1	-1	0.003552	0.003773	0	35	2.110356	0
1000	3000	1	39.91309	27.72123	12.19186	11.99978	0.011434	-1	-1	0.003587	0.0038	0	35	2.110221	0
1000	3000	1	41.0339	27.72666	13.30723	12.08923	0.011466	-1	-1	0.003551	0.003753	0	35	2.110095	0
1000	3000	1	40.63	27.72476	12.90525	12.68032	0.011582	-1	-1	0.003511	0.003797	0	35	2.11066	0
1000	3000	1	40.90826	27.72393	13.18434	11.9645	0.011376	-1	-1	0.003562	0.003761	0	35	2.110283	0
1000	3000	1	39.92799	27.72511	12.20287	12.00973	0.011429	-1	-1	0.003535	0.003716	0	35	2.110789	0
1000	3000	1	39.95791	27.73417	12.22374	12.00901	0.011282	-1	-1	0.003543	0.003787	0	35	2.11034	0
1000	3000	1	40.98736	27.72416	13.2632	12.04565	0.011493	-1	-1	0.003527	0.003769	0	35	2.110325	0
1000	3000	1	39.95525	27.73351	12.22173	12.02764	0.011329	-1	-1	0.003556	0.003764	0	35	2.111148	0
1000	3000	1	39.93585	27.7254	12.21044	12.01811	0.011385	-1	-1	0.003518	0.003757	0	35	2.11059	0
1000	3000	1	39.93194	27.72353	12.2084	12.01589	0.011482	-1	-1	0.00354	0.003727	0	35	2.110459	0
1000	3000	1	39.9394	27.72274	12.21666	11.99424	0.011492	-1	-1	0.003509	0.003798	0	35	2.110915	0
1000	3000	1	40.92786	27.72359	13.20427	11.98145	0.011544	-1	-1	0.003561	0.003778	0	35	2.110669	0
1000	3000	1	39.91477	27.7256	12.18917	11.99558	0.011387	-1	-1	0.003615	0.003712	0	35	2.110388	0
1000	3000	1	41.0446	27.72367	13.32092	12.10166	0.011497	-1	-1	0.00351	0.003797	0	35	2.110276	0
1000	3000	1	40.03624	27.72535	12.31089	12.11725	0.011506	-1	-1	0.003551	0.003685	0	35	2.110164	0
1000	3000	1	39.95002	27.72467	12.22535	12.03325	0.011501	-1	-1	0.003536	0.003742	0	35	2.110384	0
1000	3000	1	40.03218	27.7239	12.30828	12.10112	0.011397	-1	-1	0.003485	0.003746	0	35	2.110467	0
1000	3000	1	40.43901	27.72564	12.71338	12.52058	0.011534	-1	-1	0.003534	0.003868	0	35	2.11078	0
1000	3000	1	40.06765	27.72624	12.34141	12.102	0.011503	-1	-1	0.003552	0.00374	0	35	2.110824	0
1000	3000	1	40.90894	27.72407	13.18487	11.9682	0.011543	-1	-1	0.00353	0.003715	0	35	2.110507	0
1000	3000	1	40.18537	27.72277	12.4626	12.27111	0.011493	-1	-1	0.003564	0.003794	0	35	2.110955	0
1000	3000	1	40.61046	27.7268	12.88365	12.02422	0.011431	-1	-1	0.003553	0.003716	0	35	2.110637	0
Average			40.22447	27.72529	12.49917	12.07317	0.011466			0.003541	0.003765		35	2.110541	
Stddev			0.424826	0.002694	0.425253	0.158496	6.88E-05			2.7E-05	3.94E-05		0	0.000296	

alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach
1	17	0.429708	0	2	948	25.18729	0	3	0	0	0	4	0	0	0
1	17	0.429809	0	2	948	25.18306	0	3	0	0	0	4	0	0	0
1	17	0.430069	0	2	948	25.18491	0	3	0	0	0	4	0	0	0
1	17	0.429749	0	2	948	25.18616	0	3	0	0	0	4	0	0	0
1	17	0.429709	0	2	948	25.18476	0	3	0	0	0	4	0	0	0
1	17	0.429908	0	2	948	25.18266	0	3	0	0	0	4	0	0	0
1	17	0.429653	0	2	948	25.18545	0	3	0	0	0	4	0	0	0
1	17	0.429753	0	2	948	25.18429	0	3	0	0	0	4	0	0	0
1	18	0.454637	0	2	947	25.15637	0	3	0	0	0	4	0	0	0
1	17	0.429678	0	2	948	25.18689	0	3	0	0	0	4	0	0	0
1	18	0.45466	0	2	947	25.15944	0	3	0	0	0	4	0	0	0
1	17	0.429891	0	2	948	25.18375	0	3	0	0	0	4	0	0	0
1	17	0.429905	0	2	948	25.18442	0	3	0	0	0	4	0	0	0
1	17	0.429688	0	2	948	25.19414	0	3	0	0	0	4	0	0	0
1	17	0.429808	0	2	948	25.18403	0	3	0	0	0	4	0	0	0
1	17	0.429858	0	2	948	25.19251	0	3	0	0	0	4	0	0	0
1	18	0.454665	0	2	947	25.16015	0	3	0	0	0	4	0	0	0
1	18	0.454563	0	2	947	25.15851	0	3	0	0	0	4	0	0	0
1	18	0.454616	0	2	947	25.15721	0	3	0	0	0	4	0	0	0
1	18	0.454595	0	2	947	25.15832	0	3	0	0	0	4	0	0	0
1	17	0.429687	0	2	948	25.18552	0	3	0	0	0	4	0	0	0
1	17	0.429737	0	2	948	25.18366	0	3	0	0	0	4	0	0	0
1	17	0.429695	0	2	948	25.18549	0	3	0	0	0	4	0	0	0
1	17	0.430019	0	2	948	25.18427	0	3	0	0	0	4	0	0	0
1	17	0.429746	0	2	948	25.18369	0	3	0	0	0	4	0	0	0
1	17	0.429864	0	2	948	25.18499	0	3	0	0	0	4	0	0	0
1	17	0.429835	0	2	948	25.18559	0	3	0	0	0	4	0	0	0
1	17	0.430392	0	2	948	25.18317	0	3	0	0	0	4	0	0	0
1	18	0.454862	0	2	947	25.15695	0	3	0	0	0	4	0	0	0
1	17	0.429749	0	2	948	25.18642	0	3	0	0	0	4	0	0	0
	17.23333	0.435617			947.7667	25.17914									
	0.430183	0.010685			0.430183	0.012041									

alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0

7.2.4. Perfrep with distributed configuration (Perfrep-dist)

Experimental results for the Perfrep-dist configuration when component-level performance control is active are presented here. These results are used in Figure 14 and Figure 15.

timesteps	array_ele	tolerance	ttl_time	sort_time	time_bw_ts	cps_aps	steering	upAPR	downAPR	upCPS/R	downCPS/R	alog	timestep	exe_time	diff_mach
1000	3000	1	39.83359	27.78907	12.04452	11.7572	0.027939	0.001601	-1	0.005352	0.020084	0	35	2.157881	0
1000	3000	1	39.84379	27.7928	12.05099	11.76518	0.027014	0.001549	-1	0.005251	0.019096	0	35	2.156223	0
1000	3000	1	40.92403	27.78844	13.13559	11.82332	0.027557	0.0016	-1	0.005398	0.019542	0	35	2.155864	0
1000	3000	1	40.91567	27.78544	13.13022	11.81702	0.027309	0.001543	-1	0.005326	0.019277	0	35	2.156629	0
1000	3000	1	40.91941	27.78963	13.12978	11.81918	0.027496	0.001544	-1	0.005381	0.019514	0	35	2.155937	0
1000	3000	1	40.91742	27.78906	13.12835	11.81793	0.026998	0.00153	-1	0.005315	0.019055	0	35	2.156374	0
1000	3000	1	40.91253	27.78738	13.12515	11.81326	0.027408	0.001576	-1	0.005343	0.019336	0	35	2.156272	0
1000	3000	1	40.91785	27.7904	13.12746	11.81946	0.027065	0.00152	-1	0.005235	0.018987	0	35	2.156267	0
1000	3000	1	40.91438	27.78936	13.12502	11.81094	0.027666	0.001566	-1	0.005328	0.019588	0	35	2.156211	0
1000	3000	1	41.23011	27.78829	13.44182	12.12706	0.027103	0.001615	-1	0.005364	0.019241	0	35	2.156063	0
1000	3000	1	41.30306	27.79052	13.51254	12.19879	0.027187	0.00157	-1	0.005378	0.019349	0	35	2.157108	0
1000	3000	1	40.9256	27.79154	13.13406	11.81864	0.027139	0.001566	-1	0.005307	0.01909	0	35	2.156846	0
1000	3000	1	40.56891	27.78756	12.78136	12.53818	0.026828	0.001546	-1	0.005232	0.018827	0	35	2.15646	0
1000	3000	1	39.85565	27.79037	12.06528	11.77973	0.027414	0.001584	-1	0.005358	0.019368	0	35	2.156689	0
1000	3000	1	40.92535	27.79626	13.12908	11.82059	0.027084	0.001551	-1	0.005379	0.019229	0	35	2.159319	0
1000	3000	1	40.99323	27.79051	13.20272	11.89547	0.027127	0.001567	-1	0.005331	0.019163	0	35	2.156782	0
1000	3000	1	40.8381	27.78972	13.04838	11.74045	0.027476	0.00155	-1	0.005275	0.019405	0	35	2.156044	0
1000	3000	1	40.83802	27.79063	13.04738	11.73664	0.027195	0.001561	-1	0.005189	0.019185	0	35	2.156303	0
1000	3000	1	40.83384	27.78622	13.04762	11.73742	0.027437	0.001572	-1	0.005349	0.01939	0	35	2.155962	0
1000	3000	1	40.83088	27.78908	13.0418	11.732	0.027173	0.001631	-1	0.00529	0.019181	0	35	2.155957	0
1000	3000	1	40.8395	27.78947	13.05003	11.74245	0.027078	0.001601	-1	0.005365	0.019058	0	35	2.156173	0
1000	3000	1	40.84299	27.78665	13.05634	11.74946	0.026927	0.001565	-1	0.005257	0.01885	0	35	2.156265	0
1000	3000	1	40.93122	27.78993	13.14129	11.83117	0.027164	0.001594	-1	0.005281	0.019124	0	35	2.156894	0
1000	3000	1	40.92144	27.78505	13.13639	11.82852	0.026803	0.001561	-1	0.005324	0.018786	0	35	2.156044	0
1000	3000	1	40.9173	27.78646	13.13084	11.82303	0.026746	0.001556	-1	0.005244	0.018658	0	35	2.156976	0
1000	3000	1	40.99723	27.78765	13.20958	11.90324	0.027196	0.001581	-1	0.005301	0.019156	0	35	2.156269	0
1000	3000	1	40.84441	27.78696	13.05745	11.74876	0.027015	0.001575	-1	0.005324	0.018955	0	35	2.155814	0
1000	3000	1	40.93224	27.78608	13.14616	11.83417	0.026965	0.001559	-1	0.005228	0.018968	0	35	2.156627	0
1000	3000	1	40.91904	27.78919	13.12985	11.82223	0.026753	0.001528	-1	0.005282	0.01865	0	35	2.155914	0
Average			40.80644	27.78896	13.01748	11.85005	0.027181	0.001568		0.00531	0.019176		35	2.156489	
Std. Dev			0.354677	0.002352	0.355139	0.168293	0.000281	2.6E-05		5.45E-05	0.000302		0	0.00071	

alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach
1	18	0.455153	0	2	947	25.17603	0	3	0	0	0	4	0	0	0
1	17	0.430016	0	2	948	25.20656	0	3	0	0	0	4	0	0	0
1	17	0.430002	0	2	948	25.20257	0	3	0	0	0	4	0	0	0
1	17	0.429956	0	2	948	25.19886	0	3	0	0	0	4	0	0	0
1	17	0.429842	0	2	948	25.20385	0	3	0	0	0	4	0	0	0
1	17	0.429844	0	2	948	25.20285	0	3	0	0	0	4	0	0	0
1	17	0.429857	0	2	948	25.20125	0	3	0	0	0	4	0	0	0
1	17	0.430023	0	2	948	25.20411	0	3	0	0	0	4	0	0	0
1	17	0.429986	0	2	948	25.20316	0	3	0	0	0	4	0	0	0
1	17	0.430029	0	2	948	25.2022	0	3	0	0	0	4	0	0	0
1	17	0.430142	0	2	948	25.20327	0	3	0	0	0	4	0	0	0
1	17	0.429922	0	2	948	25.20478	0	3	0	0	0	4	0	0	0
1	17	0.429837	0	2	948	25.20126	0	3	0	0	0	4	0	0	0
1	17	0.430005	0	2	948	25.20368	0	3	0	0	0	4	0	0	0
1	17	0.430284	0	2	948	25.20666	0	3	0	0	0	4	0	0	0
1	17	0.430009	0	2	948	25.20372	0	3	0	0	0	4	0	0	0
1	17	0.430076	0	2	948	25.2036	0	3	0	0	0	4	0	0	0
1	17	0.430468	0	2	948	25.20386	0	3	0	0	0	4	0	0	0
1	17	0.429881	0	2	948	25.20038	0	3	0	0	0	4	0	0	0
1	17	0.429958	0	2	948	25.20317	0	3	0	0	0	4	0	0	0
1	17	0.429906	0	2	948	25.20339	0	3	0	0	0	4	0	0	0
1	17	0.429887	0	2	948	25.20049	0	3	0	0	0	4	0	0	0
1	17	0.430085	0	2	948	25.20295	0	3	0	0	0	4	0	0	0
1	17	0.42984	0	2	948	25.19916	0	3	0	0	0	4	0	0	0
1	17	0.43011	0	2	948	25.19938	0	3	0	0	0	4	0	0	0
1	17	0.429839	0	2	948	25.20154	0	3	0	0	0	4	0	0	0
1	17	0.429964	0	2	948	25.20118	0	3	0	0	0	4	0	0	0
1	17	0.430053	0	2	948	25.1994	0	3	0	0	0	4	0	0	0
1	17	0.429889	0	2	948	25.20339	0	3	0	0	0	4	0	0	0
	17.03448	0.430857			947.9655	25.20161									
	0.185695	0.004675			0.185695	0.005306									

alog	timestep	exe_time	diff_mach	alog	timestep	exe_time	diff_mach
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0
5	0	0	0	6	0	0	0