Performance Models for Virtual Laboratory Modules

Sander Ketelaar
shketela@science.uva.nl
9921915
University of Amsterdam

May 20, 2005

Abstract

Large-scale distributed computing is becoming more and more popular after the Internet revolution has made it possible to access computers all over the world. Because network bandwidth and speed have rapidly increased in the last decade, applications that require much computing power or storage can gain a lot of performance by running distributed instead of on a single machine. By connecting such resources, a distributed heterogeneous environment is created. This environment is often referred to as "The Grid" [1, 7].

With the emergence of Grids better understanding on how applications perform in a distributed environment is needed. Especially this is the case, when the software tools developed for Grids mature and are being used in for example eScience projects [36], where people tend to have less knowledge of the underlying infrastructure. This thesis describes techniques and proposes an architecture for semi-automatically computing dynamic performance models for modules, the components to build applications in the Virtual Laboratory Amsterdam [8]. Our approach succeeds at modelling the most important factors (execution time and memory usage) that affect performance for the modules and enables to explore the interactions between a target architecture and application characteristics. The performance models built are parameterised by problem size.

For a Cooley-Tukey FFT, an example of the application class of divide and conquer algorithms, the performance modelling architecture builds an execution time model that predicts problem sizes with a mean error of 4,09% based on only three small problem sizes as input. The memory usage model based on 5 samples makes accurate prediction for lare problem sizes with a mean error of 1.38%. Both models are processor architecture independent.

Keywords

Grid computing, scheduling, resource management, performance modelling, performance prediction

Contents

1 Introduction **9**
1.1 Virtual Laboratory for e-Science 9
1.1.1 Case Study 10
1.2 Problem Statement 11
1.3 Related Research 12
1.3.1 GrADS 12
1.3.2 Prophesy 13
1.3.3 DynInst 14
1.3.4 PACE 14
1.3.5 Autopilot 14
1.4 Outline 15

2 Scheduling Grid Applications **17**
2.1 Grid scheduling problem 17
2.2 Scheduling in gVLAM 19
2.2.1 gVLAM Modules 19
2.2.2 Program model 20
2.2.3 Scheduling policy 21
2.2.4 gVLAM Runtime System 22
2.3 Rescheduling 23

3 Performance Modelling **25**
3.1 Introduction 25
3.2 Performance Prediction Techniques 26
3.2.1 Profiling 26
3.2.2 Simulation based method 28
3.2.3 Handmade analytical models 28
3.3 Performance model requirements 28
3.3.1 Execution Time 29
3.3.2 Memory Usage 30

4 Architecture Design **31**
4.1 Architecture 31
4.1.1 Instrumentation component 31
4.1.2 Analyser component 33
4.1.3 Storage component 34
4.1.4 Post processing component 34
4.2 Case Study 34
4.2.1 Fast Fourier Transform 35

5 Experiments **37**
5.1 Experimental Methodology 37
5.1.1 Testbed 38
5.2 Execution Time 38
5.2.1 Model Construction 38
5.2.2 Model Validation 38
5.3 Memory Usage 41
5.3.1 Model Construction 41
5.3.2 Model Validation 45
5.4 Different architectures 47
5.4.1 Execution time 47
5.4.2 Memory Usage 47

6 Conclusion **49**
6.1 Discussion 50
.1 Linear Least Squares Curve Fitting 54
.2 JWrapper 55

List of Figures

1.1 gVLAM user interface 11

2.1 Abstract representation for a module 20
2.2 Two topologies 21
2.3 gVLAM Runtime System 22

4.1 Architectural overview 32
4.2 FFT computation 36

5.1 “standalone FFT” 39
5.2 Curve fits and prediction errors for “standalone FFT” 40
5.3 Curve fits and prediction errors for *“vlmain”* 42
5.4 Difference between “standaloneFFT” and “module FFT” b. also including “vlmain” 43
5.5 Memory traces for “standalone” and“module” 44
5.6 Curve fits and prediction errors for “module” 46
5.7 Maximum and minimum memory usage for “standalone” and “module” 47
5.8 Difference in instruction counts for “vlmain” between two architectures 48
5.9 Model constructed on a Pentium 3 used for predicting “vlmain” on AMD 48
5.10 Model constructed on a Pentium 3 used for predicting ”vlmain” on AMD 48

1 JWrapper 55

List of Tables

5.1 Function coefficients for *"vlmain"* 41
5.2 Differences between three instrumentations 41
5.3 Function coefficients for "standalone" 45
5.4 Function coefficients for "module" 45
5.5 Coefficients for fitting "vlmain" on AMD 47

INTRODUCTION, n. A social ceremony invented by the devil for the gratification of his servants and the plaguing of his enemies.

Ambrose Bierce (1842 - 1914) US journalist, short-story writer

1 Introduction

The Grid [1, 6] will make it possible to share resources such as computing power, data, networks, expensive instruments between people so that they will be able to cooperate and collaborate. These resources are distributed according to geographical location and might be administrated by different organisations. Much research is aimed at the creation, maintenance and usage of Grids. The Globus Toolkit [5, 2] is an example of a middleware infrastructure that offers various tools to implement a Grid test-bed.

This chapter provides a general introduction into the research domain of this thesis, which is performance modelling in Grid-based environments. It discusses the related research performed within this domain. Furthermore, a motivation and problem description of the research performed is given.

1.1 Virtual Laboratory for e-Science

More and more research areas use computer technology to enhance and aid with their experiments. The emergence of Grid infrastructures allow for data sharing and global distributed collaboration between scientists. "e-Science" is the term used for this new way of doing research [36]. With the advent of Grid technology the e-Science vision can become reality. To realize the e-Science vision in the Netherlands a large project started called Virtual Laboratory for e-Science [37]. The mission of the VL-E project is:

> To boost e-Science by the creation of an e-Science environment and doing research on methodologies.

Within this VL-e project several Problem Solving Environments (PSE) are defined for applied scientific research areas. These PSE's use one shared Grid-based environment, namely the Virtual Laboratory Amsterdam (gVLAM) [8, 18]. The Virtual Laboratory Amsterdam provides scientists with a workflow management system in which they collaborate and perform their scientific research. In gVLAM, experiments are represented according to a generic experi-

ment model. This model consists of three components, a *process flow template* (PFT), a *study* and *a topology*.

A PFT provides a definition for a serie of formalised steps that together compose a scientific experiment. The PFT reveals the dependencies between the processes. The advantage of defining PFT's is that complex experiments become accessible to users with less domain knowledge. Because the PFT is defined by the scientist with the knowledge of a certain applied scientific field, his knowledge is captured in a formal and easy to use way.

The specific solution for the scientific problem defined by a PFT is described by a study. The study is an instantiation of the process flow template. In the study, the different parameters for the processes in the experiment can be filled in. The data gathered in an experiment need to be processed, analysed or visualised in some kind of way with the aid of computer technology. This is represented as a data-flow graph called topology, where software components are connected together. We define the topology as the gVLAM-application. In gVLAM, these different components are called modules [14]. A module could be any kind of software, ranging from a simple sequential program to parallel programs. Also third party software can be wrapped to be used as a gVLAM module. In section **2.2.1** a further explanation of a module will be given, as it forms the core element of this thesis.

In figure 1.1 the user interface of the gVLAM environment is displayed. Four windows can be distinguished. The upper-right window displays the study of the experiment. The yellow rectangles in the study represent the abstraction for a topology. It is possible to define multiple topologies within a PFT, as it is possible in an experiment to process certain data differently than other data. The details of one of the topologies (DCAnalysis) of the study is displayed in the right-bottom window. Four modules form the DCAnalysis topology. The upper-left window shows the repository of available modules that can be used for constructing a topology. Each module in the topology has parameters that can be adjusted by selecting the module in the right-bottom window (topology) and by adjusting the parameters in the left-bottom window. Examples of real life experiments that use the gVLAM environment are in the field of high energy physics, bio-informatics and medical diagnosis and imaging.

1.1.1 Case Study

To give a concrete example for the usage of the Virtual Laboratory, a scientist is interested in the analysis of the frequency spectrum of a signal coming from an instrument as well as applying certain filters to that signal. He models his experiment by defining it as a PFT. Next, he creates a study in which he sets the right experimental parameters, for example the location of his input data. Thereafter, he sets up a topology. He selects the modules needed for his experiment from the module repository into the topology. Finally, he has to create an environment to run his experiment. This includes choosing the appropriate resource requirements for his application, determining whether he has access to the resources, whether they are available, if the module's binaries are for the right machines and the whereabouts of the data to be processed. Unfortunately this involves much knowledge for the scientist about the underlying infrastructure. The gVLAM environment aims at transparency for the scientist where his experiment is run, as he is only concerned with the results of the experiment. The Resource Manager in the Virtual Laboratory aims at automating this process. It schedules the different modules and keeps track of these during the runtime.

Figure 1.1: *gVLAM user interface*

The experiment described here will serve as a case study throughout the thesis to verify the proposed architecture, motivate the decisions taken and show the relevance of this research.

1.2 Problem Statement

Resource management and scheduling in Grid environments still is a challenging task [3, 4, 33, 25, 13]. In order to find an efficient schedule for the mapping of the application on the available resources, information about the behaviour of the application should be available prior to actual execution of the application. This information can be described in the form of a program model and/or performance model. Therefore most research [20, 21, 19, 39, 28] aims at the development of program models and performance models, which can be done manually, but will be quite time consuming and difficult if one is dealing with large complex applications. Automatic model development is therefore preferred to speed up this task.

The main subject of this thesis is to develop an architecture for semi-automatically constructing performance models for gVLAM modules running in the Virtual Laboratory. Currently, the gVLAM scheduler only uses little static information specified by the user about the module for it's scheduling decisions. This information doesn't scale at all for different datasets that can be an input for the module. A method should be developed to construct a performance model that is parameterised by both static and dynamic information of the application and the system.

Another challenge is to use application metrics for constructing the models as independent as possible from architecture specific characteristics. The advantage of this approach is reuse of the performance model for the different architectures available in the heterogeneous environment. For instance, the

number of floating point operations is independent of the platform used for the processing, but the wall-clock time will heavily depend on the platform used. The number of instructions executed is semi-architecture dependent in the fact that it depends on the compiler and compiler options used. Similarly, memory access patterns will have a strong correlation with the application, while cache hit/miss ratios will depend significantly on the memory system.

The application chosen to test the method developed in this thesis comes from the case study described in section 1.1.1 a radix-2 Cooley-Tukey Fast Fourier Transform [32], which is a representative of the divide and conquer applications class. The Fast Fourier Transform is chosen as example application because it is widely used in many areas of applied scientific research. Because there exist much knowledge about FFT's, validating the performance model can be done easily.

1.3 Related Research

Quite a lot of research is done in the field of resource management and scheduling of applications within a Grid environment. Also performance modelling of applications to aid scheduling is used in several projects. This section will shortly discuss some of the projects that are trying to achieve things closely related to our work.

1.3.1 GrADS

The Grid Application Development Software (GrADS) project is one of the big Grid projects currently running in the United States. It is a collaboration between several universities and laboratories [26]. The GrADS Project's aim is to develop a comprehensive programming environment that explicitly incorporates application characteristics and requirements in application development decisions [25, 24, 23]. Activities such as compilation, scheduling, staging of binaries and data application launch and monitoring of application progress during execution are requirements necessary for successful fullfillment of the project's goal.

The core of the GrADS project is the program execution framework. A GrADS application is represented as Configurable Object Program (COP). A COP contains the actual program *source code*, a *mapper,* that determines how to map the application's tasks to a set of resources and a *performance model*, that estimates the application's performance on a set of resources. The construction of a COP is done in the Program Preparation System (PPS). The mapper and the performance model are closely tied together as the performance heavily depends on the set of resources on which the COP is run. A performance model is based on measurements of execution time for a trial run on a set of representative resources or left to the application developer to deliver. However, a lot of detail refrains to be given about the construction and structure of these performance models.

Most focus lies in the execution of a COP by the Program Execution System (PES). A modular scheduling approach is used that combines generic scheduling strategies with application specific components to determine the best candidate set of resources for a COP. Experiments have shown that this approach for building Grid applications and scheduling is the most promising one [25]. The class of applications for which the GrADS scheduler is optimised, are tightly-coupled parallel applications. Here lies a difference with gVLAM, as we deal

with dataflow applications. Only recently the GrADS scheduler was extended to be able to schedule this kind of applications [23]. Before executing the COP on the Grid, a Binder configures and builds the application executable for the selected resource set. The actual compilation for the application is therefore done on the target resource. This enables for architecture specific optimisation. The COP's source code is also instrumented by the Binder to support performance monitoring during runtime. When the application binary is ready, it is sent to the Grid Runtime System.

Every GrADS application has a specific performance contract. The performance contract is monitored during runtime using the Autopilot toolkit [33] (this will be considered future work for our performance modelling architecture. See also section 1.3.5). When the performance contract is violated, the re-scheduler is activated and it tries to find a new resource set for which the application's performance contract is within bounds. Then parts of the application are migrated to another resource.

The GrADS software evaluated on a macro grid, which consists of a set real resources distributed over the different universities collaborating in the project and in a controlled simulation grid called the micro-grid. In the micro-grid special conditions can be simulated, while others remain constant in order to check limitations of the software.

1.3.2 Prophesy

Prophesy [28] is a novel system that aims to automatically generate analytical performance models. Three ways are used for constructing the performance models. Two well established methods are incorporated such as curve fitting and the parameterisation method. But also a novel method called kernel coupling [29, 27], developed in the Prophesy project. Kernel coupling is based on correlation between kernels, where a kernel is defined as a small core function within a program. Given the performance models of different kernels the coupling parameter C_{ij} represents the way two kernels i and j influence each other in relation to running each kernel in isolation. A value of $C_{ij} > 1$ represents destructive interaction resulting in performance loss, whereas $C_{ij} < 1$ represents constructive coupling resulting in performance gain. In [29] it is shown that C_{ij} can be extended to a chain of kernels to construct total application performance models by means of a composition algebra. The way to calculate $C_{ij} = \frac{P_{ij}}{P_i + P_j}$ is by first executing kernel i and j isolated in a loop to extract the runtime P_i and P_j respectively and than by executing both kernels, where i immediately precedes j, together in a loop to extract P_{ij}.

The Prophesy framework consists of a data collection component and a data analysis component. The data collection component PAIDE (Prophesy Automatic Instrumentation and Data Entry) uses source code instrumentation for performance extraction. This information is then used for the data analysis to generate a performance model through one of the three methods. All the models from the data analysis component and the performance information from PAIDE are stored in databases. The databases store template information, runtime information, application information, performance information.

The Prophesy framework is the closest to our approach. The work done on coupling by Prophesy is very promising and might be useful in gVLAM environment when extending the methods of performance model building, but currently we focus on curve fitting for the automatic model development. Our architecture is set up in such a way for possibilities to incorporate kernel coupling in the future.

1.3.3 DynInst

DynInst [39] aims at changing the program while it is executing, and not have to re-compile, re-link, or even re-execute the program to change the binary. Runtime instrumentation of the object code has advantages over traditional source-based performance profiling systems. Most significant of which is the elimination of the interference of calls to the instrumentation with the compiler's optimisation passes. In section 3.2.1 instrumentation at different levels is further explained. DynInst provides an API to create a program that can attach to a running program, create a bit of code and insert it into the running application. The next execution of the program executes the modified code instead of the original code. There are two processes, the application and the mutator. The mutator contains calls into the DynInst API and to routines to manipulate the application process.

There are two abstractions defined for a program in execution. *Points,* the place where the instrumentation can be inserted, and *snippets,* the bit of code that is inserted to be executed at a point. In order to generate code, the snippet is translated to machine language code in the mutator process and then copied to the application's address space. The insertion is done via *trampolines*. A base trampoline replaces instructions at a specific instrumentation point. The base trampoline branches to a mini-trampoline, which contains the snippet. At the end of the base trampoline, the instructions that were replaced by the trampoline are executed before branching back to the main program.

Dynaprof [38] is a performance analysis tool based on DynInst and designed to insert performance measurement instrumentation directly into a running applications' address space at run time. DynInst is also used in Paradyn [40], which is a performance measurement tool like Dynaprof for parallel and distributed applications. The DynInst API is promising for the object instrumentation part of our architecture described in section 4.1.

1.3.4 PACE

At the University of Warwick a Performance Analysis and Characterisation Environment (PACE) [41] is used for performance evaluation and analysis for parallel applications in heterogeneous distributed environments. The methodology of PACE is describing the application as a layered framework. This way, the hardware layer is separated from the application layer. An application model is constructed through static source code analysis which shows control flow, operation counts and communication structure. The central part is the evaluation engine that combines the application model and the hardware characteristics to predict the application runtime. The system aims to simultaneously provide information regarding algorithmic choices, software implementation, hardware configuration, mapping, execution time, bottlenecks and scaling.

1.3.5 Autopilot

Autopilot [33] is an architecture for dynamic performance tuning based on closed loop adaptive control. This will help the application's changing requirements of resources and the resource availability. The application is instrumented by the user specifying which parameters he wants to investigate or tune. The application is connected with Autopilot sensors that extract qualitative and quantitative performance data. Autopilot actuators are also connected with the application and allow for modification of values of the applications variables. Both sensors and actuators are registered with the Autopilot manager, that keeps

track of all sensors and actuators in the system. The sensors and actuators have property lists to specify the metrics of the application that can be measured and changed dynamically. A client program can get information about the running application through the sensors and dynamically steer via the actuators. The client doesn't need to know where the application runs, as this information is stored at the Autopilot Manager. The decisions to dynamically steer the running application through the actuator(s) is based on a fuzzy logic rule base at the client. Autodriver is a visualisation tool that graphically shows the value of sensors connected to the client.

We consider performance monitoring and active steering in the gVLAM environment as future work.

1.4 Outline

This thesis is structured as follows. Chapter 2 will discuss problems and issues around scheduling in Grid environments. In chapter 3 we will focus on methods that can be useful for the construction of performance models. The description of the architecture can be found in chapter 4 and the results and validation of the performance model for the case study application in chapter 5. Finally, chapter 6 will conclude and discuss some future work.

2 Scheduling Grid Applications

In this chapter we will discuss the central issues facing scheduling applications in Grid environments. We will show that in Grids application-level scheduling is favoured over system-level scheduling. Scheduling dataflow applications is of most importance, since these are the type of applications in the gVLAM environment. Furthermore, the Runtime System of gVLAM is explained with the scheduler in particular.

2.1 Grid scheduling problem

Scheduling applications on heterogeneous networks has proved to be a challenging task. It is much more complicated than scheduling in a multi-threaded environment or even in a homogeneous distributed environment. Having a global generic scheduler for Grid applications appeared not possible due to a number of important reasons. First, because each resource will have it's own performance characteristics that might not always be known. Second, because the resource will have it's own local scheduler and/or access rights. Third, because the total set of applications running in the Grid environment is not known, resource concurrency between applications cannot be taken into account. Finally, because of the highly dynamic nature of the Grid environment, where network congestion and network failure are likely to occur. Therefore, another approach towards Grid scheduling has to be taken. All the problems encountered are referred to as the Grid scheduling problem [3, 13].

The choices made in scheduling are based on optimising some kind of performance metric represented as a cost function. On a single machine this metric is often system-specific (maximum throughput of applications or data, network latency and/or bandwidth, maximum memory usage, etc.) and will probably promote the performance of the system beyond the performance of individual applications. In a Grid environment this is practically impossible, since there

is less or sometimes no knowledge about the state of the environment, because of the Grid scheduling problem. Predicting future states of the environment is even harder. Therefore, it is recommended that the performance metric that is to be optimised is seen from an application centric point of view. It is called application-level scheduling [22], where everything about the system is evaluated in terms of the impact on the application. Often users consider application execution time as the cost function to be used in application-level scheduling. Within the AppLeS project [22] preliminary research on the application scheduling paradigm was done. They proved that this technique is successful and now it is a widely adopted paradigm in Grid scheduling and also used in this research. For a few important application classes, scheduling agents (called AppLeS) were implemented.

A problem with application-level scheduling is thrashing [22], because every application can select the same set resources as the most optimal. If they all seek to use these resources, they may find that the resources are not optimally available anymore, and all find another optimal set, etc. A solution to address this problem lies in the use of market economy or social models, such as bidding or auction structures or via resource reservations.

The basis for good scheduling is prediction. By predicting how an application behaves when executed on the system, more accurate and performance efficient schedules can be computed. In Grid-based environments very accurate predictions cannot be made, because the parameters needed are most of the time not available or too computationally intensive. Therefore evaluation of the cost function needed for the scheduling choices made are as good as the accuracy predictions that can be made.

As stated above, scheduling in Grids involves the need to have a Grid scheduler for every different application class. Every Grid scheduler has it's own scheduling model. A scheduling model consists of a *policy,* which is a set of rules or an algorithm for producing schedules, a *program model*, which abstracts the set of programs to be scheduled and a *performance model*, which abstracts the behaviour of the program on the underlying system for the purpose of evaluating the performance potential of candidate schedules. A scheduling model should have the following functionality [3]:

- produce performance predictions that are within a certain time-frame
- use dynamic information to represent variations in performance that can serve as feedback to the scheduler
- adapt to a wide spectrum of computational environments

In general, the necessary steps that should be performed to make a Grid application run according to the chosen schedule and based on the program and performance model are:

1. select a set of available resources
2. reduce this set for constraints like resource user authorisation and resource needs of the application
3. find a suitable mapping for the application components to the resources based on some kind of cost function
4. distribute the data to the resources
5. monitor the executing application

The resource discovery fase finds the set of resources where the user has access to and which comply to certain requirements (both static and dynamic) suitable for the application to run. After that information about the current state of availability of the resources is gathered and the scheduler tries to find the best set of resources based on the program model and the performance model of the application. This matching can be done in different ways [10, 11, 4]. After that the job is executed, monitored and if necessary re-scheduled.

2.2 Scheduling in gVLAM

Applications in the gVLAM environment are dataflow applications [8]. The main characteristics of a dataflow application are that they are composed of different components that can produce and consume data. A common way to represent such an application is via a directed acyclic graph (DAG). Dataflow-style graphs are becoming a common way to describe and model Grid programs. For example, Dome [20] and SPP(X) [19] provide a language abstraction as a program model, which is compiled into a low-level program dependency graph. MARS [21] assumes that programs are phased and builds a program dependency graph as part of the scheduling process. From now on we will refer to the total DAG based application in gVLAM as a topology.

The advantage of component-based application modelling is that it encourages for code reuse. It provides the ability to build dynamic applications in a natural way and it stimulates collaboration. The shift to dataflow applications has been a large impact for the performance community. Performance implications will be very difficult to evaluate, since applications become distributed and very decoupled.

The Resource Manager is the core component in the gVLAM Runtime System (RTS) [8]. It handles issues related to resource discovery, scheduling and spawning the topology to the Grid. In the next sections we discuss the scheduling model for the gVLAM scheduler, which consists as described in section 2 of a scheduling policy, a program model and a performance model (this is described in chapter 4). But first, we describe what gVLAM modules, the components that build the topology, are.

2.2.1 gVLAM Modules

Modules in the Virtual Laboratory may be any kind of software, written in either C++, Java or Python. They range from simple file processing programs, or mathematical functions to parallel MPI programs. It is also possible that they are a wrapping around third party software. A module is built by compiling and linking it against a special developed dynamic library for gVLAM called `VLport` [14, 15]. For developing Java and Python modules a wrapper is available to make use of the `VLport` library (see appendix B for a figure of the Java wrapper for `VLport`). The `VLport` library is responsible for handling the communication between modules, data-transfer and authentication to the Grid.

Figure 2.1 shows an abstract representation for a module. As can be seen, the part that is linked by the `VLport` library contains a connection handler and a GASS client. The connection handler is a CORBA 2 ORB compliant implementation called omniORB [16] and maintains the connection to it's neighbour module(s) in the topology and a reference so it can be located by the connection manager of the Resource Manager (shown in figure 2.3). All modules in a topology run in the same CORBA environment. The GASS and GridFTP compo-

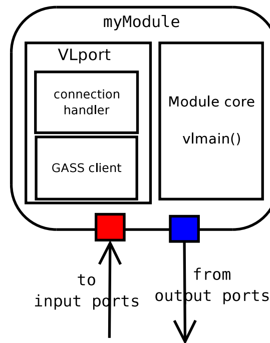


Figure 2.1: *Abstract representation for a module*

nents from the Globus Toolkit [5] that are incorporated in the `VLport` library are responsible for the data-transfer between modules. The data-transfer between two modules is done via ports. A module can have one or more input ports to receive all incoming data and it can sent data to other modules via one or more of it's output ports. The output port of a module can only connect to an input port of another module provided that both ports share the same data type.

The module core is the actual application part where a module developer implements the module's functionality. The module's core function is called `vlmain()` and a common implementation contains calls to read the input ports, process the received data and write it to the output ports.

Modules are continuously active. Once they are initiated, they either wait for the arrival of data on their input ports or enter a continuous processing loop. They finish when the connection to other modules is closed.

Together with a module, meta-information stored in a module description file (MDF) has to be specified. The MDF contains information such as developer information, input and output port information and extra parameters that can be set initially to the module. The module's resource requirements (cpu, memory and storage) can optionally be specified and serve as a very simple static performance model for the module, which is extended in this thesis, that is used by the scheduler. Also system-specific preferences can be specified in the MDF.

2.2.2 Program model

The topology is an abstract representation for a gVLAM application. The user constructs a topology in the bottom-right window of figure 1.1. The scheduler uses an analytical program model to determine how a topology performs on a candidate set of resources. We consider two possible types of topologies (see figure 2.2.2). The first type is a topology where the modules operate in a concurrent fashion and the latter is a topology where modules exchange data in a sequential manner. A hybrid form of the types is also possible. For each topology type we define a different cost function to model the whole application.

We define a set of modules that build the whole application as $C = \{c_1, c_2, ..., c_M\}$ and the set of available resources as $R = \{r_1, r_2, \dots, r_N\}$. Then $compT(c_i, r_k)$ is the computation time of module c_i $(i \leq M)$ when scheduled on resource r_k $(r \leq N)$. The communication link between two modules is described as $commT(c_i, c_j, r_k, r_l)$, which represents the total amount of data sent from the output port of the module to the input port of it's neighbour module. If this is combined with dynamic resource information from the Network Weather System [34] such as available bandwidth and network latency, a dynamic prediction

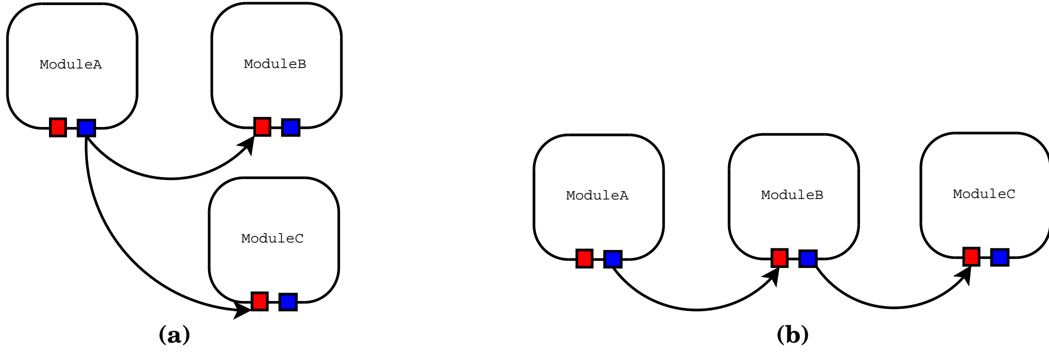


Figure 2.2: *Two topologies*

will be possible. The total amount of data sent to the output ports can be modelled as a function based on the input data size.

The function $commTT(c_i, r_k)$ defines the total communication of module c_i, summing over the different $commT(c_i, c_j, r_k, r_l)$ for all links of c_i to other modules.

$$commTT(c_i, r_k) = \sum_{c_j \epsilon NB(j)} commT(c_i, r_k, c_j, r_l) \tag{2.1}$$

Here $NB(j)$ is the set with modules c_j scheduled on r_l that are connected to the *output port* of c_i.

To describe how well a module runs on a certain resource, we define:

$$rank(c_i, r_j) = w_1 * compT(c_i, r_k) + w_2 * commTT(c_i, r_k) \tag{2.2}$$

Here $0 \leq w_1 \leq 1$ and $0 \leq w_2 \leq 1$ are scaling factors that can be optionally set, for example to emphasise computation over communication. The lower the rank, the better the mapping is. The resulting program model for a gVLAM application written in terms of both types of topologies T_c for the concurrent type and T_s for the sequential type is:

$$T_c = max_{(i,k)\epsilon P_q}\{rank(c_i, r_k)\} \tag{2.3}$$

$$T_s = sum_{(i,k)\epsilon P_q}\{rank(c_i, r_k)\} \tag{2.4}$$

The tuple $(i, k)\epsilon P_q$ describes the mapping for module c_i run on resource r_k for a certain candidate schedule P_q, where $P_q \epsilon Q$ and Q is the total set of candidate schedules calculated by the scheduler according to the scheduling policy described in section 2.2.3.

Currently, the module's parameters are static, but the framework proposed in this thesis improves this by calculating a more accurate model based on dynamic parameters such as problem size. The way the cost-function for $compT(c_i, r_k)$ looks like is further explained in section 3.3.1.

2.2.3 Scheduling policy

The scheduler present in gVLAM calculates an execution schedule for a given topology. Currently, the scheduler is able to choose between two greedy algorithms, one computation-network prioritised and one simulated annealing algorithm [17].

in the gVLAM environment and everybody using gVLAM can have it's own local repository. The internal representation of the topology is a XML file that describes the structure of the gVLAM application, thus the different modules that compose the application and their mutual connection relationships. The dotted arrows show the relation between the abstract description of a module in the topology and the actual module in the repository. The scheduler uses the performance models of the different modules and the retrieved dynamic resource information provided by the Monitoring and Discovery System (MDS-2) [35] and network information through the Network Weather System [34]. Both MDS-2 and NWS are services running on every resource in the Grid environment, which can be queried in order to retrieve dynamic resource information. NWS gives a forecast for a small time-span about the network conditions.

After the schedule for the topology has been calculated, the modules can be sent to the different resources, determined by the scheduler, in the Grid environment. The authentication and resource management service [9] from the Globus Toolkit activates and keeps track of the modules. The connection manager [16] sets up and manages the connections between the modules in the way described by the topology. Currently, there is no performance monitoring while running the modules. This is considered future work.

2.3 Rescheduling

When an application is running in a Grid production environment it is possible that the performance of a resource degrades. Therefore, it is necessary that monitoring tools are active to check whether the performance is decreasing. In the GrADS project [24, 23] performance contracts are used to see if a component needs rescheduling. A performance contract defines the boundaries in which a performance metric is said to be valid. If the performance contract is violated, a new schedule is computed and the component is migrated, if necessary. In Mars [21], several checkpoints are inserted in the source code of the application. When the checkpoint is reached, the schedule is re-computed and the application is optionally migrated. In gVLAM the re-scheduler is currently being developed but this is beyond the scope of this thesis.

All models are wrong, but some are useful

George E. P. Box

3 Performance Modelling

Accurate performance analysis and prediction is key to achieving optimum use of modern information systems. In this chapter we will describe general techniques and approaches that can be used for modelling application behaviour. The aim is to find parameters to describe this behaviour as much as possible in an architecture independent way. These parameters represent the performance model for the application for various performance metrics, where execution time and memory usage are considered the most important. However, predicting for example the execution time under varying circumstances is one of the most difficult things in performance analysis research.

3.1 Introduction

Performance is an important issue with any application. Efficient execution of applications requires insight into how system features impact the performance of applications. This insight results from experimental analysis and the generation of performance models. The term performance modelling has been used without a proper definition. We will now give a definition how performance modelling is seen in the light of this research. The term can be split into two words, which mean according to the dictionary (Compact Oxford English dictionary) giving only the :

performance per-form-ance (plural per-form-ances) , *noun*
manner of functioning: the manner in which something or somebody functions, operates, or behaves; working effectiveness: the effectiveness of the way somebody does his or her job

model *noun*
a simplified mathematical description of a system or process, used to assist calculations and predictions;

Based on the above definitions from the dictionary, this is how we define performance modelling:

Definition 1 *The way to devise a simplified mathematical description for the behaviour of an application rather architecture independent to assist calculations and predictions.*

In this chapter we will mainly focus on the possible ways to develop performance models for modules. The chosen method(s) is tested for a class of applications.

3.2 Performance Prediction Techniques

Performance estimation is a difficult task, because it involves reasoning about complex architectures using incomplete information about the program and its data-structures. In general, there are three well established techniques that can be used for performance prediction: profiling, through simulation and by hand constructed analytical models. It depends on the kind of performance thats needs to be predicted, which one should be applied. In distributed computing, profiling is the most used technique.

3.2.1 Profiling

Profiling is one of the most used techniques for performance prediction. In general profiling consists of three steps that need to be done: *code instrumentation*, *performance data extraction* and *analysis*. Profiling is not only used for performance prediction, but also for performance analysis. It reveals the parts where the program spends most time. With this information a program developer can see which part of his application contains bottlenecks and is worth optimising in order to achieve a performance increase. Most available tools only aim at performance visualisation and were not useful for our approach, because we aim at developing an architecture independent model for a module. When someone tries to determine a bottleneck in an application, the resource on which the application is run will not be of much relevance.

Code instrumentation The application code is instrumented, which can be at source code level or at object level. At source code level, instrumentation is done mostly manually by inserting calls in the source code to instrumentation library routines. At the object level a classification can be made between static and dynamic binary instrumentation. The advantage of binary instrumentation is the removal of the instrumentation's dependency on the compilation process. The type and format of the instrumentation can be changed without recompiling the application. PACE [41] is an example for static binary instrumentation, whereas DynInst [39] is an example for dynamic binary instrumentation. Both projects are further explained in section 1.3.4 and 1.3.3 respectively.

Performance data extraction The instrumented application is executed on resources on which the application is likely to run and the instrumentation library records the performance data.

Visualisation and analysis The performance data is visualised to determine bottlenecks. A common way to visualise the performance data is via a call graph, that shows the hierarchy of calls between the functions, or via flat profile, which is a table containing the function calls.

gprof

A profiling tool available on all unix-like operating systems is `gprof` [30]. With `gprof` the instrumentation is done at compiler level. The source code should be compiled by the GNU C compiler with profiling enabled (option -pg). During the compilation inline statements are added to the code to increase the counters. After execution a file containing the performance data is available and can be analysed with `gprof`. `gprof` is able to generate two tables, the flat profile and the call graph. The flat profile shows how much time is spent in the functions and the call graph shows the function hierarchy. The standard sampling interval is 0.01 seconds. The overhead of `gprof` is low. However, the major disadvantage of `gprof` is that is doesn't interact well with threads and dynamic libraries. Therefore, `gprof` was not suitable for using it to extract a module's behaviour, because the modules are multithreaded.

Counter-based profiles

There are two broad classes of profiles. Timing profiles, where the amount of time spent in routines and statements is shown, for which`gprof` is an example. Another class is counter-based profiles, that show the number of times a specific event occurred. Such a counter can be implemented at software level or at hardware level.

Processors nowadays are equipped with hardware performance counters. These counters are special registers dedicated to events that occur at microprocessor level or at the memory subsystem when the application is executed. Events like instruction execution, branch prediction, cache hits/misses can all be measured using these hardware counters. The Performance Application Programming Interface (PAPI) [31] is a library that can access the performance counters built in the processor for a lot of different processor architectures. Although the number of available counters and the existence of some events are architecture specific, it is still a useful tool to use in heterogeneous systems.

Statistical profiling

An alternative for gathering performance data through code instrumentation is statistical profiling. Statistical profiling works by sampling the program counter at some interval during execution. From the statistical distribution of the samples an execution time profile of program can be constructed. The sampling can be done relatively independent of the program clock. This makes the samples to appear “random” , as than they can be compiled into statistics to show the programs execution patterns. This extra overhead will be around 3%-5% of the execution time [30].

Statistical profiling does not evaluate the absolute coverage of a program and therefore it can only predict performance with a certain accuracy. The accuracy of a statistical profiler's measurement is determined by the number of samples the profiler collects. Given the desired accuracy we can calculate what the number of samples (and thus the sampling interval) should be for the profiler:

$$N = \frac{pqZ^2((1+g)/2)}{e^2}$$

This function states that for a confidence coefficient g of a confidence interval e you need at least N samples of a binary process. p is the probability of the binary event and $q = 1 - p$. Z is the inverse of the Normal distribution integral function. A binary event for instance is that a sample contains a routine in

question. In other words, to be g sure within an error of e, at least N samples need to be collected.

It is also possible to do statistical profiling based on event counters (explained in section 3.2.1). A sampling event is then triggered when the counter reaches a certain threshold. Subsequently, a histogram is constructed.

3.2.2 Simulation based method

Performance prediction on simulation is a very accurate way to get information about execution times. First, the application instrumented to find out what events occur and then it is executed to generate a trace file. The trace file is fed to a simulator that simulates the event on the target architecture. The disadvantage is that the simulation time increases with the granularity of the accuracy of the prediction.

A simulation based engine for Linux is Valgrind [44] which simulates memory accesses and can be used for detecting memory leaks. Unfortunately Valgrind has a large overhead. Programs profiled with Valgrind run 20-30 times slower.

3.2.3 Handmade analytical models

Program and algorithm developers generally have a lot of knowledge about the performance and complexity. Therefore, they can construct accurate analytical performance models themselves and this is also the traditional way of building these. Because such a deep understanding is needed of the application and while most applications are developed by teams of developers this technique is not widely used in the industry. However, this doesn't mean that they aren't used at all anymore. The performance models constructed by humans are still the most accurate ones. In our architecture it is possible to use handmade analytical models as a performance model.

3.3 Performance model requirements

For performance models to become useful, they have to adhere to certain requirements. In general, a performance model should be able to provide a calculable explanation of why a program performs as it does, what factors affect the performance and a prediction according to a quantifiable objective. It must be able, given the accuracy of the model, to guide the application to the best machine and adapt to different execution environments including performance of future systems. The generated prediction made by the performance model is only allowed to have an error below a certain threshold.

When we want to develop a dynamic performance model, we need to use dynamic resource information as well. Because the dynamic resource information used is only valid for a certain timeframe, the prediction made by the model is that as well. The performance model currently available for the gVLAM scheduler is a static, user provided one. This doesn't fullfill the stated requirements. The performance models that were generated semi-automatically from the method developed in this thesis do fullfill these requirements.

The four parameters we consider most important to reflect the performance of a module are:

- Execution time

- Memory usage
- Communication
- I/O

In this research we will only develop models for a module based on execution time and memory usage, as they are the most important ones. If a module is a parallel application, also the internal communication needs to be modelled. For now we will only concentrate on modules that are sequential applications and hence communication can be discarded. The I/O is defined as the amount of storage needed for the files the module stores on the filesystem and will be also considered beyond the scope of the thesis.

The exact character of the performance model will be equational based. It describes the execution of module's core: the `vlmain()` function. The input parameter will be problem size for both execution time and memory usage and the output is execution time and memory consumption respectively. In the next two sub sections modelling execution time and memory usage for a module will be further explained.

3.3.1 Execution Time

For the construction of a semi-architecture independent performance model that predicts the execution time it not sufficient to calculate the wall-clock times or CPU time, because these are affected by the hardware on which the module runs. The most application specific parameters are based on the source code at basic block level and use other parameters such as control flow and loop and operation counts. Because these parameters mostly account for user based instrumentation, we will choose an intermediate solution, namely the number of instructions issued, which is compiler dependent.

A way to measure the number of instructions issued is by counter-based profiling (see section 3.2.1). The PAPI library [31] will be the most suitable to use, as it is able to count number of instructions executed. Another possibility that is more architecture independent, is counting the number of floating point operations executed. Unfortunately, some architectures do not support a hardware counter for this event.

We prefer to base the performance model on the problem size of the module, then for problem sizes not measured a prediction can be made. An initial amount of executions will be required to determine an initial model. For small problem sizes N, the instructions counts are determined empirically. Then a regression technique is used to fit a curve through the measured data points. Once the coefficients for the function are determined we are able to predict instruction counts for data sizes we haven't encountered before. If it is not possible to fit a curve through the data points, we will use historical information to find whether the module has run before with data size N.

Recall the formula $compT(c_i, r_k)$ for calculating execution time from section 2.2.2 for a module c_i on a specific resource r_k. It combines dynamic resource information with dynamic application information to predict the runtime, resulting in the following equation:

$$compT(c_i, r_k) = \frac{Instr(c_i, N) * \frac{Cycles}{Instr}}{CycAvail(r_k, t)} \tag{3.1}$$

The dynamic resource information can be the amount of available CPU cycles: $CycAvail(r_k, t) = CpuAvail(r_k, t) * Clockrate(r_k)$. MDS-2 [35] is able to supply the relative amount of available CPU time for a certain time-frame $CpuAvail$ and the $Clockrate$ for resource r_k. The dynamic application information is determined by our proposed architecture and returns the number of instructions that will be executed based on the input data size N of the module c_i. The value $\frac{Cycles}{Instr}$ is a both application and resource dependent measure and hard to determine, because it depends on the instruction mix of the application and the amount of cycles needed for a certain instruction. An average may be taken, but we will assume this value is known and we will only concentrate on predicting instruction counts.

3.3.2 Memory Usage

Modelling memory usage in a semi-architecture way is more difficult than modelling execution time. As with the execution time model we prefer to base the performance model on the problem size of the module, which for example can be:

$$memReq(c_i, N) = memUnit * N \tag{3.2}$$

$memUnit$ depends on the size of one unit of the data structure that is stored and N is the problem size. This model will only be accurate if the datastructure based on the problem size is large and accounts for most of the required total memory. If $memReq$ is modelled according to 3.2, it can only define a minimum bound on memory usage. We are also interested in the maximum bound on memory usage. Therefore a different approach should be taken.

Counterbased profiling using the PAPI hardware counter library [31] as in section 3.3.1 is not possible, because it can only measure cache hit/misses which cannot be transformed to maximum amount of memory usage. We decided on a modelling method based on statistical profiling of the program's memory during executing. Unfortunately this results in a more architecture and operating system specific implementation, but the advantage is that minimum and maximum bounds now can be measured. We chose for an empirical approach and sample the memory usage of the module every 10 microseconds. Hence the way to determine $memReq$ will become:

$$memReq(c_i, N) = memUsage(N) \tag{3.3}$$

Where $memUsage$ is the model constructed by the model builder (see section 4.1.2) and for the case study in chapter 5. $memReq(c_i, N)$ is used by the scheduler to determine if a module maps to the resource for the requested amount of memory that it needs.

4 Architecture Design

This chapter describes the proposed architecture for generation of performance models for gVLAM modules. The architecture is evaluated through the case study from section 1.1.1. The algorithm for computing a Cooley-Tukey radix-2 FFT is explained which is implemented as a module for the case study.

4.1 Architecture

The aim of this research is to develop a framework that can assist the gVLAM scheduler [17] to improve the accuracy of it's scheduling models, by providing dynamic module information in the form of a performance model. The current performance models used by the scheduler are very basic and should be provided by the module developer. A balance needs to be found between the accuracy and the time it takes to create such a model during run-time.

An overview of the performance modelling architecture is presented in figure 4.1. It consists of four main components visualised as dotted rectangles. The following components can be distinguished: *the instrumentation component, the analyser component, the storage component* and the *post processing component*. Because the architecture is set up in a modular way, profiling tools developed in other related research projects can be used. This is done because performance modelling is a research area that has already quite some history and a most tools have already proven their value. Therefore, a large part of this research was spent on finding the most suitable tool(s) to be used in the gVLAM environment. Currently, PAPI [31] is the most promising and it also used in a number of other projects.

4.1.1 Instrumentation component

The first component is the instrumentation component. It is required to mark important events that could trigger a profiler to count the number of times those events occur. At several places during the building phase of the module, instru-

mentation can be inserted as described in section 3.2.1. In the architecture of figure 4.1 instrumentation is either done before or after the compilation phase. For object instrumentation PACE [41] and DynInst [40, 39, 38] provide suitable libraries. Either source code or object instrumentation should be added when a new module is added to the repository. The instrumented version of the module is added to the module repository.

As we make use the PAPI library [31] to count the amount of instructions executed for our execution time model, the current implementation of the architecture uses the source code instrumentation. The advantage of using PAPI is that it is able to count this event on a wide variety of architectures with hardly any overhead. Currently, the instrumentation should be done by hand. A call to start the counters is done when the module enters the main function (`vlmain()`) and a call to stop the counters is done when the module leaves the main function. Automation is possible when the calls to the library are integrated in the `VLport` library. Prior to the call to start the `vlmain()` the event counters should be initialised and started. When the module leaves the `vlmain()` the counters should be stopped and the results will be collected.

For the calculation of the memory usage model, there is no instrumentation needed in the source code or object, as it uses statistical profiling on the memory usage. If the model is enhanced to become more accurate and application specific, instrumentation might be necessary and it should be added to this component of the architecture.

4.1.2 Analyser component

The Analyser component consists of two parts. The first part is the execute/profile environment. This can be the Runtime System of the Virtual Laboratory (figure 2.3) when the module is used in an experiment or a profile environment that is optimised for executing trial runs. It is important, that before a new module is added to the module repository initial runs with small datasets in order to gather initial meta-information are executed. The profile environment doesn't have the overhead of the total gVLAM Runtime System. The profile environment consists of a few scripts and an executable that has the functionality to connect and activate modules. The extracted information from either the Runtime System or profile environment will be in terms of computation, memory and communication. When conducting our experiments of chapter 5, we only used the profile environment to gather performance data.

The performance model builder is fed with this information and builds the performance model, as architecture independent as possible. The performance model builder tries to fit an analytical function through the information supplied after the execution. In the current implementation a linear regression technique for only one function class is used. The model builder can be extended with other function classes and for modules with more difficult behaviour nonlinear regression techniques can be used. For more detailed information about the least squares linear curve fitting technique we refer to appendix A.

Each time when the module is executed in the Runtime System, the performance information is saved in the storage component. If the performance model is later built again, after for example a real execution of the module, the model builder first queries the historic information and the function template of the performance model from the storage component. It fits a function combining the old and new information and the performance function template's parameters are updated. Therefore, each time when the module is executed the performance function can become more accurate.

Of course a different analysis method may also be used. The analyser can use his profiling tool to construct a call graph (see section 3.2.1), which visualises the dependencies between the different functions in the module.

4.1.3 Storage component

The storage component acts as a repository for all the meta-information gathered from all the modules available in the gVLAM environment. We define different levels of meta-information:

- Expert knowledge about the requirements and architectural restrictions for a module
- Trial executions to determine run times, memory usage, etc of an individual module
- Run-time information during real execution of the application for the modules it uses
- Integrated trial executions and historical run-time information into an analytical performance model with problem size as input parameter

First, expert knowledge delivered by the user that developed the module defined as quantitative meta-information from the Module Description File (MDF) about the module's requirements and restrictions. This can be seen as the static part of the performance model, as it doesn't scale when the module is used for different datasets. Another form of expert knowledge is described in section 3.2.3. This is a qualitative form of meta-information. Most of the time this qualitative expert knowledge will not be available.

The trial executions and the historical information can be combined into an analytical performance model generated by the Analyser component. The performance model will be represented as a function template with the problem size for the module as input parameter. The coefficients calculated by the model builder are also stored in the function template. The historical information can also be used for table lookup, if the module was executed for the same problem size and the analytical performance model is not available or inaccurate.

The current implementation of the database consist of text files for the different levels of meta-information per module.

4.1.4 Post processing component

In this component the the performance model of the module, retrieved from the database component, together with dynamic information about the candidate resource, from NWS [34] and MDS-2 [35], are used to compute equations 3.1 and 3.3. These equations are able to make dynamic performance predictions for the determination of a candidate schedule.

Although the post processing component is not a stand alone component in the architecture, because it is actually available in the scheduler, it is placed in figure 4.1 for completeness.

4.2 Case Study

The architecture described in figure 4.1 will be used to determine a performance model for the case study described in section 1.1.1. Recall the case study, where

the scientist wants to analyse the frequency spectrum that comes from an instrument. He sets up a topology containing a Fast Fourier Transform module. The reason why FFT was chosen as example module is because it can be used in a lot of scientific fields. The most common is signal analysis, where time domain signals are transformed to frequency domain signals (or vice versa). Other application fields are data compression, partial differential equation solving and the multiplication of large integers.

The way the performance model will be constructed is via the architecture described in section 4.1. The module is profiled by calls in the source code to the PAPI library to count the number of instructions executed in the main part of the module. For a few known small data sizes the average was taken from **t**rial runs. This was stored in the database in the storage component of the architecture. With the information available from the trial runs a function based on the input data size was fit trough the datapoints. The module was run again, but now for more realistic larger data sizes and compared with the prediction made by the determined function. In the next chapter the results and the performance model are presented. In the following subsection a brief explanation will be given on the FFT algorithm.

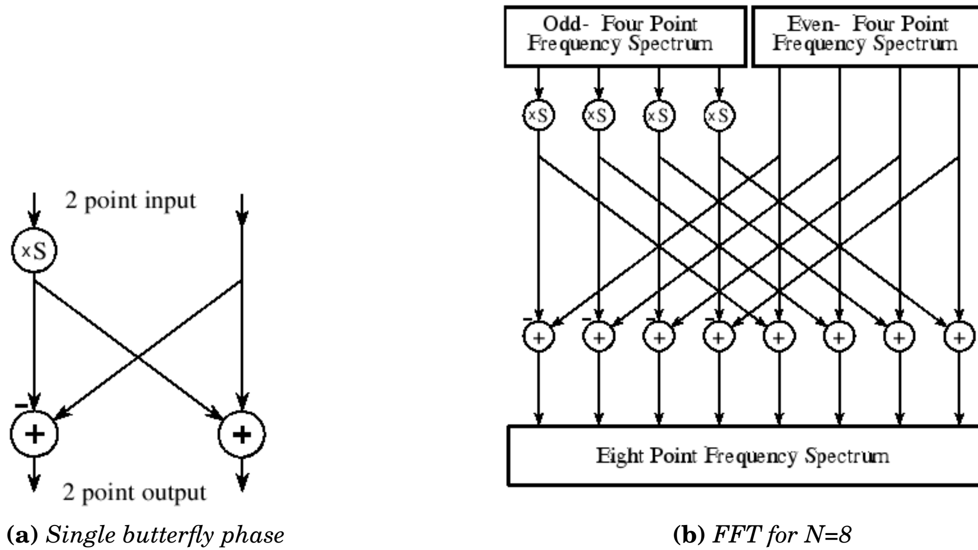
4.2.1 Fast Fourier Transform

The FFT is the name for an algorithm for a very fast computation of a complex Discrete Fourier Transform (DFT). A complex DFT transforms a complex N point time signal into a complex frequency domain signal. The DFT has a synthesis part 4.1, where the frequency domain signal is computed from the time signal and an analysis part 4.2 that calculates the time domain signal from the frequency domain signal.

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn} \tag{4.1}$$

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{\frac{j2\pi}{N}kn} \tag{4.2}$$

Computing a DFT as in formula 4.1 is slow and has a complexity of $O(N^2)$, but by using a FFT for computing a DFT, the complexity reduces to $O(Nlog_2(N)$. Another advantage is precision. Due to less calculations, there are less rounding off errors. There is no such thing as *the* FFT, because numerous implementations exist for any kind of FFT. The most popular algorithm for the FFT is the Cooley-Tukey algorithm [32]. The general idea behind the Cooley-Tukey FFT is a divide and conquer approach. Cooley and Tukey proved that because of the symmetry of the signal, it could be shown that the DFT of the signal is equal to the sum of two $N/2$ point signal, or in general $N = N_1N_2$. Calculating the DFT over a small signal is less complex. The radix for an FFT is defined by either N_1 or N_2. If N_1 is the radix, it is called a decimation in time (DIT) algorithm, whereas if N_2 is the radix, it is decimation in frequency (DIF) algorithm. The algorithm we use is a radix-2 DIT Cooley-Tukey FFT, thus $N_1 = 2$ and $N_2 = N/2$ and the signal length N has to be some power of 2, or padded with zero's to make it a power of 2. Other radix FFT's use prime factorisation to determine the $N_1$and N_2. Cooley and Tukey also defined $W_N \equiv e^{-j\frac{2\pi}{N}}$ (called twiddle factor) the DFT can be written as:



(a) *Single butterfly phase* **(b)** *FFT for N=8*

Figure 4.2: *FFT computation*

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \tag{4.3}$$

If the signal is split into an *even* and an *odd* signal of equal length and we define $k = k_1 + N_1 k_2 = k_1 + 2k_2$ then:

$$X[k_1 + 2k_2] = \sum_{n=0}^{N/2-1} x[2n] W_{N/2}^{k_1 n} + (-1)^{k_2} W_N^{k_1} \sum_{n=0}^{N/2-1} x[2n+1] W_{N/2}^{k_1 n} \tag{4.4}$$

This process can be iterated until there are two signals of length 1. A clever way to separate the even and the odd signals is via bit-reversal. The addition of the two signals multiplied with a twiddle factor is called the butterfly phase (figure 4.2.1). It is called butterfly, because the operation in figure 4.2.1a looks like a butterfly. This phase is critical to the speed of computation, since the twiddle factors can be pre-computed and stored to be used again for many different sizes of N.

The FFT itself represents a class of algorithms. The complexity of the algorithm depends on the implementation used. The Cooley-Tukey algorithm makes our module implementation an example of the divide and conquer class of applications and therefore the performance model developed in the next section also represents this class of algorithms.

An implementation of a large number of different FFT's is the FFTW [42] library and it is widely used in the scientific community. Because our main concern is not to optimise, but to analyse the FFT we use a simple implementation of a recursive radix-2 Cooley-Tukey FFT.

5 Experiments

This chapter reviews the results we obtained from testing the small implementation of our framework for the described case study. Most performance models we prefer to build are parameterised by the problem size of the module. The performance model for our case study will therefore also be parameterised by it's problem size. As described in section 4.1 the framework is able to extract execution data from application runs. The exact execution data will be instruction counts and memory consumption. To test the scalability of the model, the FFT module was executed with small data sizes as input. With the collected data of multiple runs for different data sizes we were able to compute parameterised curves for both execution time and memory usage.

5.1 Experimental Methodology

For our experiments we used two implementations for the FFT algorithm. The first is where the FFT algorithm is implemented in a standalone C program, totally independent of the `vlport` library functionality. The instrumentation needed for the execution time model is placed around the function that computes the FFT. We will refer to this implementation as "standalone FFT". For the memory usage model the total program is traced and we will refer to the measurements as "standalone". This implementation is used in order to validate the behaviour characteristics of the FFT for our case-study module.

The second implementation uses the same code for the FFT algorithm as the first implementation, but now embedded in a *module*. The instrumentation for the execution time model is placed at two positions in the module. Around the FFT function, verifying any occurrence of overhead in application instruction counts between a module and it's standalone version. To these measurements we will refer as "module FFT". The other position with calls to the PAPI library is around the `vlmain()` function of the module (see section 2.2.1 for the module's structure). To these measurements we will refer as "vlmain" and the resulting execution time model will be the one used in the scheduler. For the memory

usage model we did sampling over the complete execution of the module and hence refer to this as “module”.

5.1.1 Testbed

The testbed used during the experiments consists of two machines with different architectures and operating systems.

The first machine has a Intel Pentium III processor with a clock-rate of 700 MHz. Both the D-cache and I-cache are 16Kb 4-way associative and the L2-cache is 256 Kb 8-way associative. The cache line sizes are 32 bytes. The total amount of RAM is 64 Mb. The operating system is a Debian distribution of Linux. All testdata for the construction of the performance models were run on this machine.

The second processor is an AMD mobile AthlonXP 1600+ processor with a clock-rate of 1393 MHz. There is a 64Kb 2-way associative D-cache and I-cache. The L2 cache is also 64 Kb, but 4-way associative. The cache line sizes are 64 bytes. The total amount of RAM is 352 Mb. On this architecture run two operating systems, a Debian distribution of Linux and Microsoft Windows XP SP2. This machine is used in section 5.4 and **??**, to verify architecture and operating system independence.

5.2 Execution Time

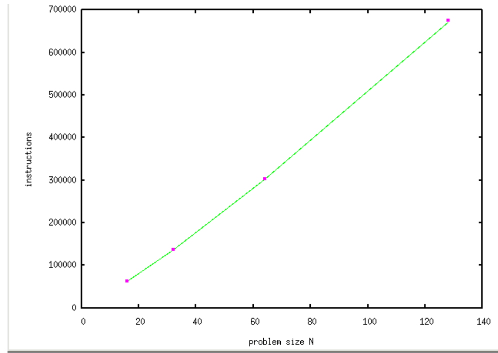
5.2.1 Model Construction

First, we construct a model based on the collected data from the number of instructions executed for *“standalone FFT”*. For every small problem size, the program was executed 21 times in a loop. Because of instruction and data caching, the instruction counts from the second run on differed from the first run. The test set consists of 4 different small sized problem sizes shown in figure 5.1a, with corresponding averages and standard deviations over runs 2-21. Because in general the module will only run one time, we will only use value of the first run. This value was constant for every first run of the application, while the later runs in the loop fluctuated due to earlier mentioned caching.

The sample runs from from figure 5.1a didn’t show any polynomial behaviour, when the linear regression technique was applied. However, they showed the characteristic of a function in a power law class. For a power law the following family of functions hold: $y = \alpha_2 x^{\alpha_1}$. The regression technique closely fit the number of instructions executed, when we consider x as the problem size. Figure 5.1b shows the fit for the small test set. The purple points in the figure are the measured values and the green line is the prediction function. The coefficients for the function are: $\alpha_1 = 1.1456$ and $a_2 = 2589.707$. The residual of the fit is $R = 0.01017$, which represents a close fit through the data. In the next section we will validate the model by predicting large problem sizes for a module using a function derived from a few samples with a small problem size.

5.2.2 Model Validation

The power law function class appears to be a good model for the FFT. This can be validated if we look at the FFT algorithm. For the Cooley-Tukey FFT it is known that is has a complexity of $O(N\, log_2 N)$. This matches the power law of $y = \alpha_2 x^{\alpha_1}$. If on both sides the logarithm is taken, we get: $log(y) = log(\alpha_2) + \alpha_1 log(x)$. For this function the linear regression technique is easily applied.



(a) *Fit for four samples*

sample size N	run 1	run 2-21	
		μ	σ
16	62382.00	50817.75	0.536
32	136284.00	121179.00	0.000
64	302854.00	281193.15	0.357
128	674468.00	677016.05	0.218

(b) *Instruction counts*

Figure 5.1: *"standalone FFT"*

To validate the function's scalability for predicting instruction counts for large problem sizes based on a model estimated for small problem sizes, we executed "standalone FFT" also for larger, more realistic problem sizes N up to $2^{17} = 131072$. Figure 5.2a shows that the constructed model (the green line), based on four test samples from figure 5.1 (the purple points), has a reasonable prediction for the larger problem sizes (the red points). The accuracy of the prediction results are evaluated as follows :

$$Error = \frac{|measurement - prediction|}{measurement} * 100 \tag{5.1}$$

Also the mean error was calculated. Figure 5.2b show this relative error for all the predictions. Figure 5.2c and 5.2d show the model and error, when the model was based on five other samples and figure 5.2e and 5.2f show the results when all the problem sizes were used for fitting the model.

The model build based on the first four samples, has a large relative error for $N > 10000$, resulting in 25.62% for $N = 131072$. When all samples are taken into account the prediction error decreased to only 7.12% for $N = 131072$. The mean relative error is 3.54% . A reason for non-scalability for larger N could be that $N = 128$ still fits in cache while larger N do not and therefore need more instructions.

Now, we will verify whether the same function class applies for "vlmain" . Figure 5.3a-f shows the results for "vlmain". Figure 5.3a-b show that the first four samples are not able to the result in an accurate model. The maximum relative error is 68.60%, with a mean error of 33.91%. It appears that there is too much extra overhead because the application is a module and that the small samples are not able to overcome this overhead and scale to large N.

However, sample N taken not much larger (128-512) were able to compensate for the introduced module overhead and result in an accurate prediction. The error is larger than 2.20% only for the three smallest N and and the mean error is 4.08%. Figure 5.3c-d show this result, which is even better than when

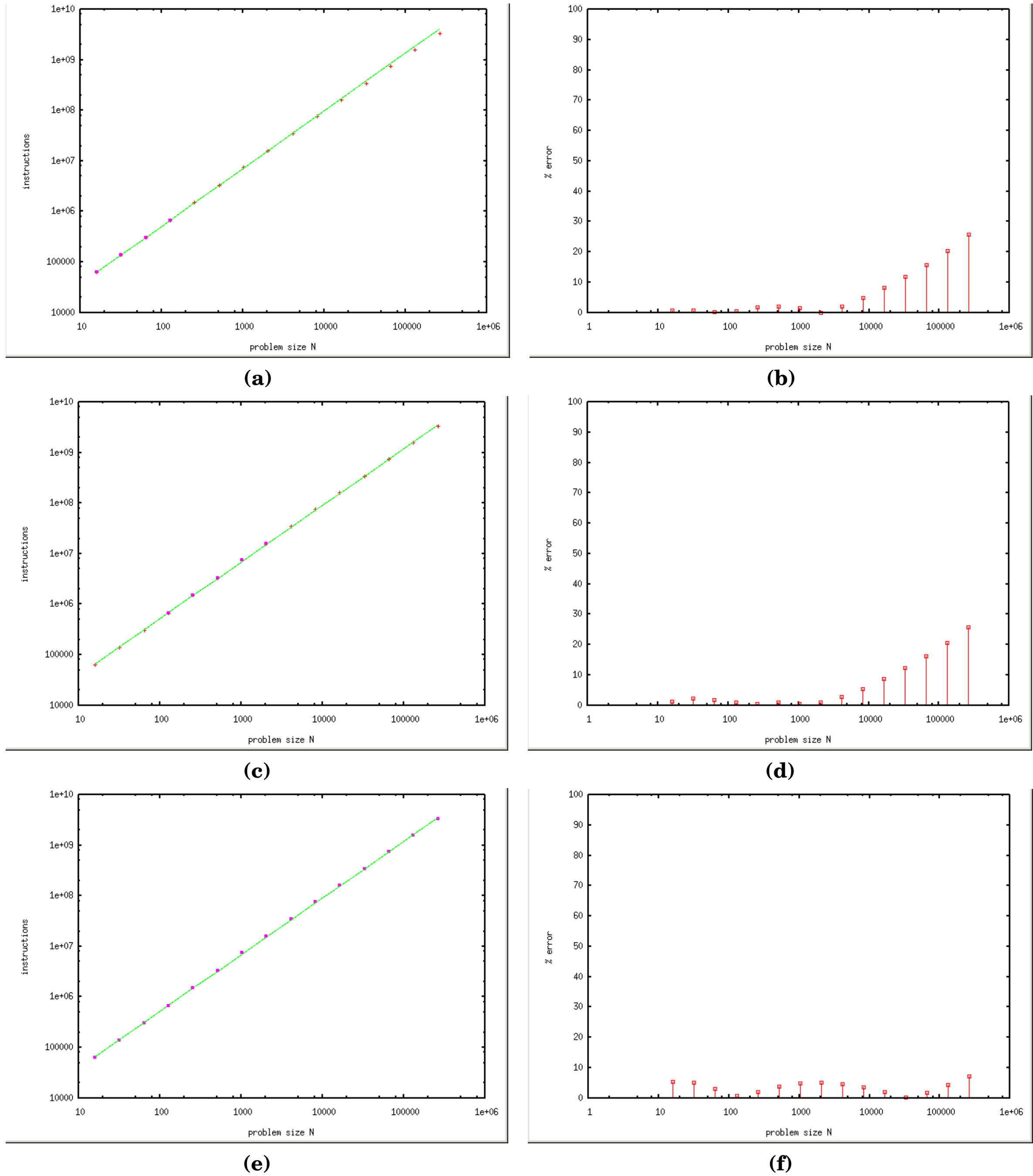


Figure 5.2: *Curve fits and prediction errors for "standalone FFT"*

sample N	α_1	α_2	$log(\alpha_2)$	μError (%)	Residual
16-128	0.92371	20244.33	9.91563	33.91	0.08325
128-512	1.0858	9570.11	9.1664	4.08	0.004300
all	1.0671	11463.23	9.3469	5.12	0.2737

Table 5.1: *Function coefficients for* "vlmain"

sample N		α_1	α_2	$log(\alpha_2)$	μ Error (%)	Residual
128-512	standalone FFT	1.1570	2463.40	7.8093	10.30	0.003911
	module FFT	1.1485	4990.54	8.5153	7.42	0.0025602
	vlmain	1.0858	9570.11	9.1664	4.08	0.004300
all	standalone FFT	1.1232	2917.84	7.9786	3.54	0.15384
	module FFT	1.1196	5820.26	8.6691	2.51	0.10674
	vlmain	1.0671	11463.23	9.3469	5.12	0.27366

Table 5.2: *Differences between three instrumentations*

all samples were used for fitting the model, figure 5.3e-f. Compared to "standalone FFT", "vlmain" results in more accurate predictions.

Table 5.1 shows the different values for α_1 and α_2. The residual value shows how well the function fits through the test samples and not how accurate the total prediction is. The performance model based on all samples is better than when based on the four smallest samples , although it's residual is higher.

The best model for predicting the instruction counts, only based on three samples, for "vlmain" is:

$$y = 9570.11 * x^{1.0858} \tag{5.2}$$

Executing an application as a module introduces overhead. But, we are still able to use the same function model, although with different coefficients. To verify the quantity of the actual overhead introduced by the module, we will compare "standalone FFT" and "module FFT". Figure 5.2.2a shows the measurements, and the overhead is relatively the same for every problem size. The mean relative overhead is 93,37% with a standard deviation of 4.72%.

Table 5.2 shows the comparation in coefficients for the three different instrumentations and figure 5.2.2b includes "vlmain" in the graph of Figure 5.2.2a. The extra instructions needed for reading and writing the signal in "vlmain" have a larger account in the total for small problem sizes. For large problem sizes, the computation of FFT takes the upper hand.

5.3 Memory Usage

5.3.1 Model Construction

The performance model for memory usage is build the same way as for execution time. For a few small data sizes the peak amount of memory consumption is measured. This is done by sampling the program's memory consumption during small time intervals. In a Linux operating system, this information is available in the `/proc` filesystem. Every program has a process ID (PID), which is a subdirectory in the `/proc` filesystem. In the PID directory is a *statm* file, which contains information regarding the memory behaviour of the program. This file is sampled during execution of the program at an interval around 1/100 second.

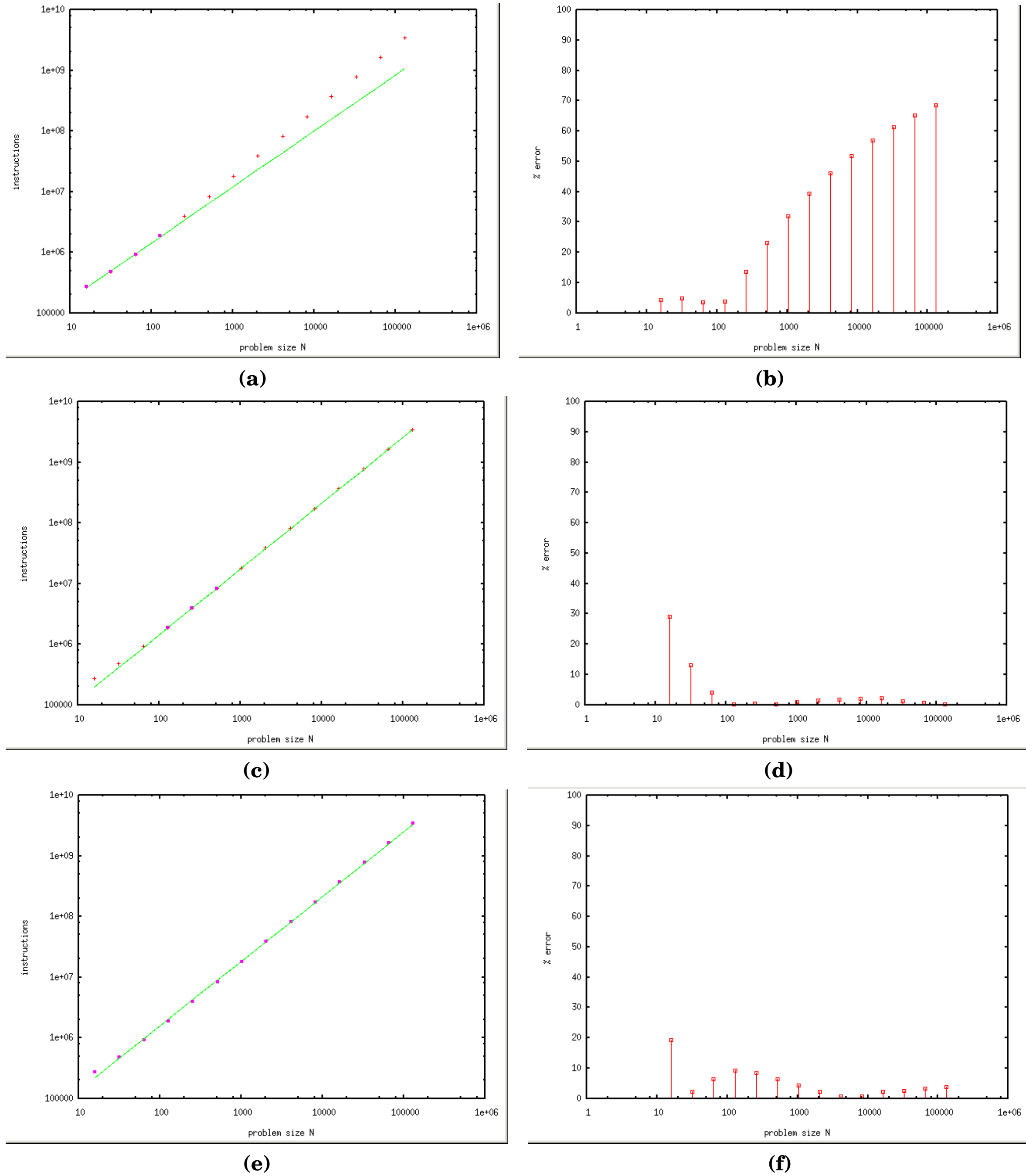
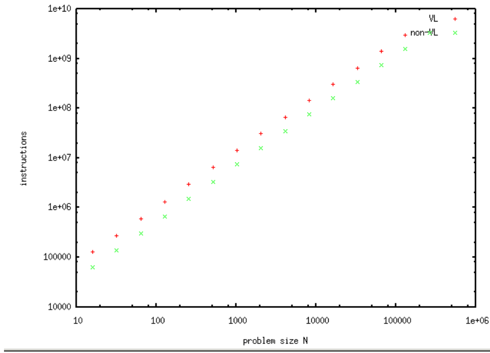
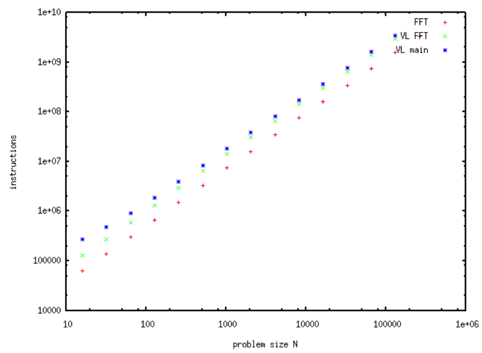


Figure 5.3: *Curve fits and prediction errors for* “vlmain”



(a)



(b)

Figure 5.4: *Difference between "standaloneFFT" and "module FFT" b. also including "vlmain"*

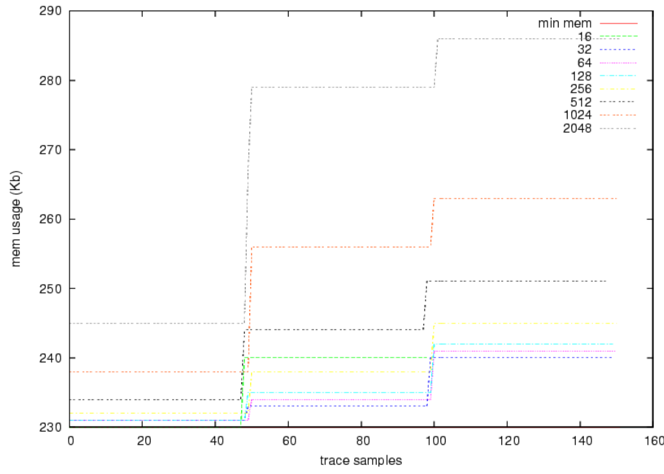
The maximum amount of memory used by the program serves as input for the regression technique of the model builder component (section 4.1.2). The trace shows the memory consumption behaviour. For resolving more accurate initial and ending memory consumption a `sleep(1)` was introduced at the beginning and before termination of the program. This was only done for modelling purposes and is not needed in the real execution.

Profiling "module" was more difficult than "standalone", because of it's multi-threaded behaviour. However, all the threads for the `vlmain()` function shared the same memory space and this was profiled.

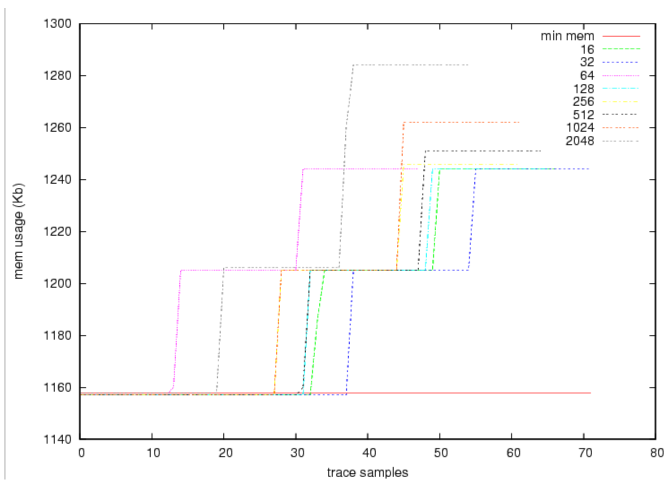
Figure 5.5a-d shows the traces for all the problem sizes for "standalone" and "module". Figure 5.5a and 5.5b show traces for the small problem sizes up to $N = 2048$ for both "standalone" and "module". In the traces an exponential behaviour is seen. Furthermore, the initial amount of memory claimed for "module" is much higher than for "standalone". Executing a program as a module introduces overhead. Figure 5.5c and 5.5d show the traces for the largest problem sizes. Remarkable is that for $N = 16384$ the decrease of memory consumption as a result of cleaning the claimed memory in both "standalone" as "module" is much higher than for the other problem sizes.

For the maximum memory consumption, a polynomial behaviour in the logarithm of the problem size was found. For the linear regression we found a good approximation for a polynomial of degree 5, where y is memory peak and x is the $log_2$of the problem size N.

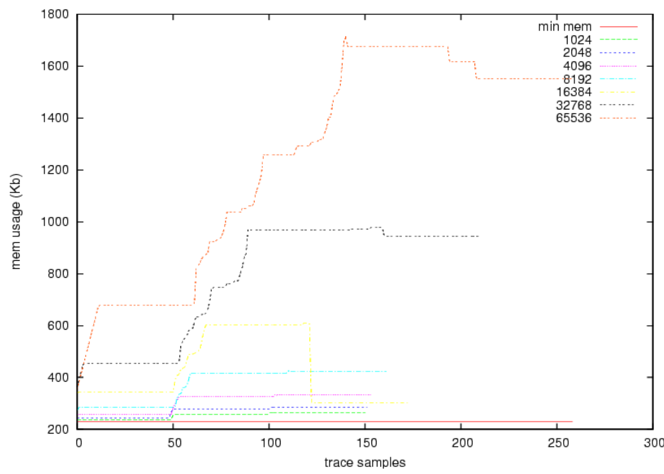
$$y = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \alpha_3 x^3 + \alpha_4 x^4 + \alpha_5 x^5 \tag{5.3}$$



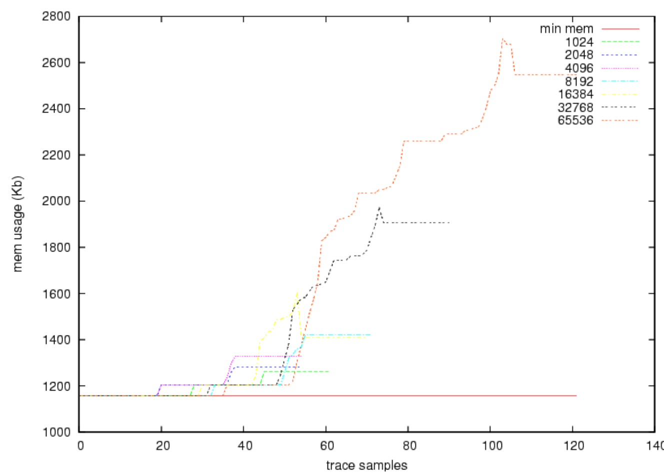
(a)



(b)



(c)



(d)

sample $log_2 N$	α_0	α_1	α_2	α_3	α_4	α_5	μError (%)	re
4-10	332.25	-70.202	20.438	-2.8314	0.18371	-0.0041667	18.83	0.
8-12	120.68	219.64	-81.556	12.584	-0.89369	0.024540	6.41	3.092
all	-3309.8	2378.9	-604.38	73.199	-4.2607	0.096438	4.13	6

Table 5.3: *Function coefficients for "standalone"*

sample $log_2 N$	α_0	α_1	α_2	α_3	α_4	α_5	μError (%)	re
4-10	1253.4	-9.3258	3.2917	-0.4924	0.02651	$2.496 * 10^{-14}$	5.86	0
8-12	311.59	581.16	-142.86	17.41	-1.0602	0.026204	1.86	1.69
all	-524.64	1227.3	-324.72	41.18	-2.5233	0.006048	0.38	2

Table 5.4: *Function coefficients for "module"*

5.3.2 Model Validation

The same method was used for validating the models as for the execution time model (section 5.2.2). Figure 5.6a-f shows the curve fits and prediction errors for "module" when different samples where taken for fitting the model. The maximum problem size was $N = 2^{16}$. Figure 5.6a shows that the constructed model (the green line), based on the five smallest data samples (the purple points), is not able to result in a reasonable prediction for the larger problem sizes (the red points). This is due to the lack of increase in memory consumption for the small data sizes. Hence, the maximum prediction error was 38.90% for $N = 2^{16}$. Taking samples for $N \geq 2^8$ resulted in a more accurate model and using all samples (5.6e-f) results in the most accurate model with a mean prediction error of 0.386%. The orange line in the figure is the initial amount of memory consumption,which remains constant for all problem sizes.

Table 5.3 and table 5.4 show the function's coefficients with error and residual of the fit. There is much fluctuation in the coefficients for both "standalone" and "module" as the number of samples and the problem size of the samples increase. This is due to high order polynomial that is chosen. The scalability of the memory usage model based sampling is worse than for the execution time model. The best model for "module" is when all the samples are used:

$$y = -524.64 + 1227.3x + -324.72x^2 + 41.18x^3 + -2.5233x^4 + 0.006048x^5 \quad (5.4)$$

The extra overhead introduced by the vlport accounts for a more scalable model, because the fits for "standalone" had a mean predicition error of 18.83% (maximum error of 82.32% for $N = 2^{16}$) for the fit based on problem sizes $2^4 - 2^{10}$ and a mean prediction error of 4.13% (maximum error of 9.85% for $N = 2^{16}$) for the fit of the total test set.

To determine the overhead of the module in memory usage, the maximum and minimum values of the memory traces for all profiled problem sizes for "standalone" and "module" were taken and drawn in one graph. Figure 5.7 shows the results. The amount of overhead is almost constant and scales when problem sizes becom larger. There is only initial overhead in memory consumption for the module.

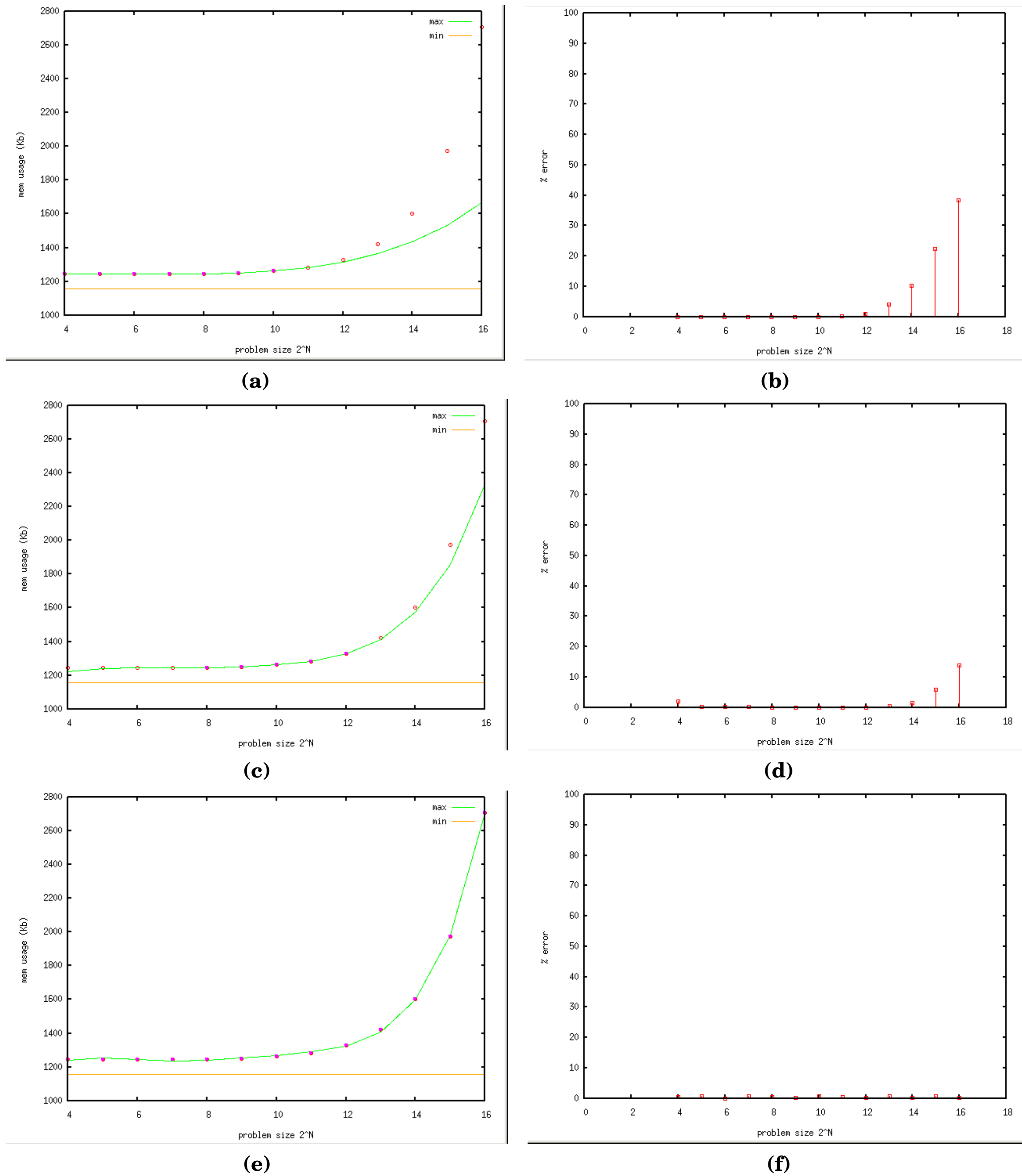


Figure 5.6: *Curve fits and prediction errors for “module”*

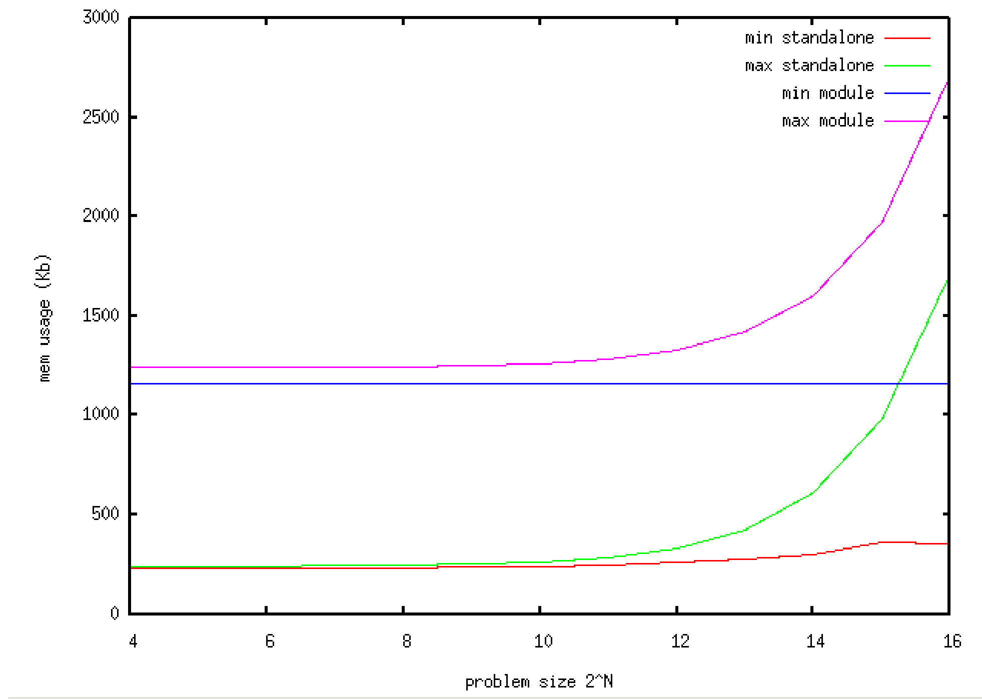


Figure 5.7: *Maximum and minimum memory usage for "standalone" and "module"*

sample N	α_1	α_2	$log(\alpha_2)$	μError (%)	residual
16-128	0.92347	20257.29	9.91627	33.96	0.083982
128-512	1.0858	9596.15	9.1663	4.09	0.0043761
all	1.0671	11463.23	9.3469	5.12	0.27366

Table 5.5: *Coefficients for fitting "vlmain" on AMD*

5.4 Different architectures

5.4.1 Execution time

In order to test the architecture independence of the execution time model, the same source implementation of the module was compiled and executed on a resource with an AMD processor (see section 5.1.1). Figure 5.8 shows the performance data for both architectures and they don't differ much, therefore it should be possible to use the performance model for "vlmain" generated on the Pentium 3, to predict the instruction counts on the AMD processor. Figure 5.9a-b shows the model and the prediction errors. The mean prediction error is 4.09%, which is the same as for the model for the Pentium 3 and mostly due to large errors of the three smallest samples. Table 5.5 shows the coefficients for the model when it was constructed in the same way on AMD as in section 5.2.2. The differences in coefficients compared to table 5.1 are very small.

5.4.2 Memory Usage

In order to verify architecture independence of the memory model, we compiled and executed the module on the resource with the AMD processor and run all experiments again. The model of equation (XXX) was used to predict the memory usage for "module" on AMD. Figure 5.10a-b shows the results. The P3 model can make accurate predictions for AMD samples. The mean error is 0.55% with a maximum error of 1.10%.

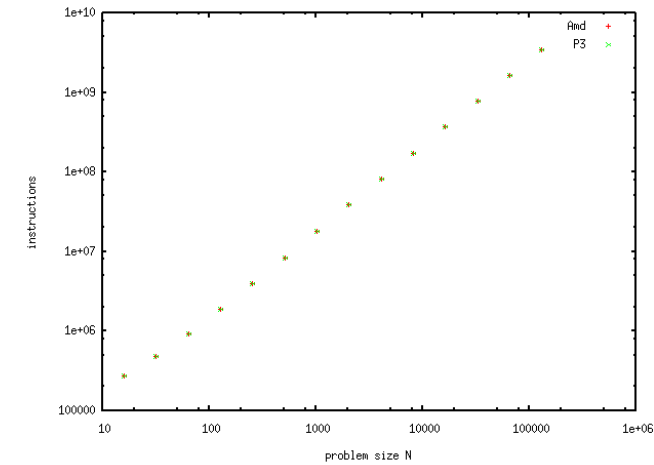
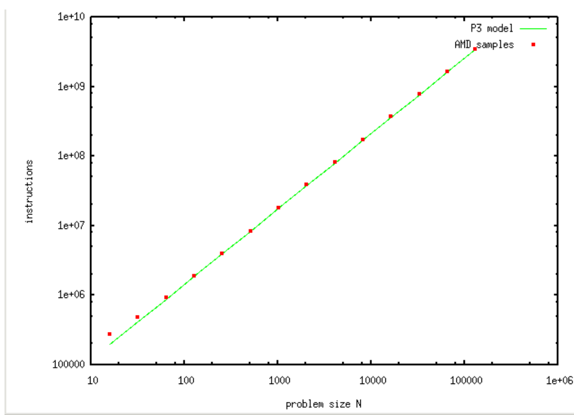
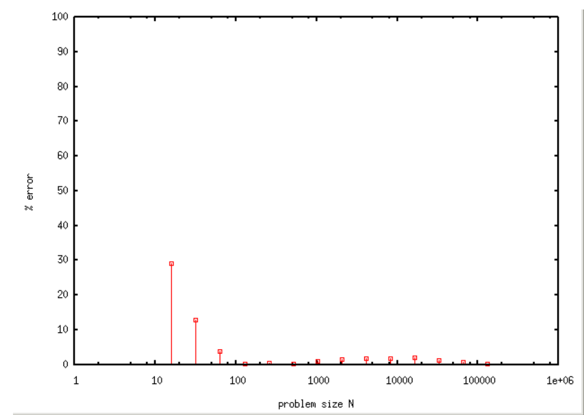


Figure 5.8: *Difference in instruction counts for “vlmain” between two architectures*

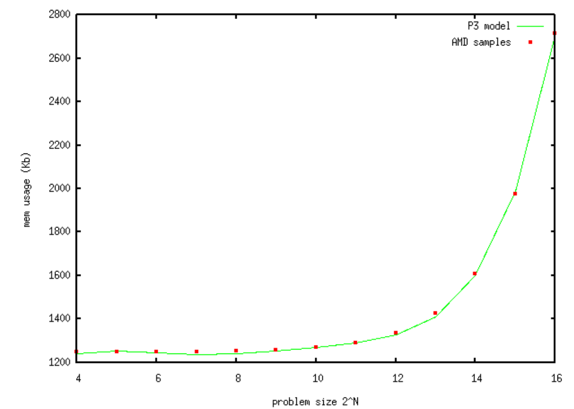


(a)

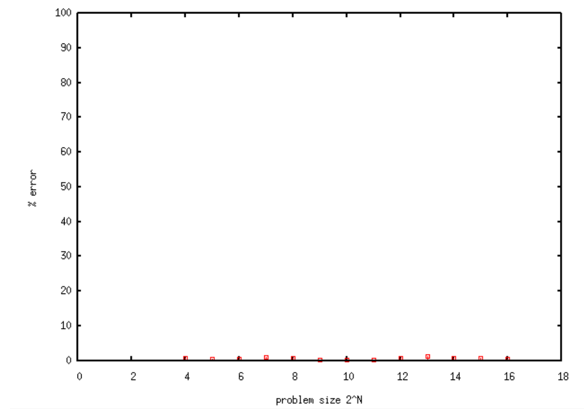


(b)

Figure 5.9: *Model constructed on a Pentium 3 used for predicting “vlmain” on AMD*



(a)



(b)

Figure 5.10: *Model constructed on a Pentium 3 used for predicting ”vlmain” on AMD*

6 Conclusion

In the past the gVLAM scheduler relied on the accuracy of the data provided by the module developer in the module description file. The architecture proposed in this thesis provides a more elaborate way to increase accuracy and scalability of this data by the development of semi-architecture independent dynamic performance models for Virtual Laboratory modules. A prototype of the architecture was able to construct an execution time model and a memory model for a case-study containing a Fast Fourier Transform module. As the implementation for the FFT used belongs to the divide and conquer class of algorithms, the model's skeleton extends to other applications in this class of algorithms. For these applications the model builder only has to find coeffients for the model's skeleton. The accuracy of the model is automatically increased, because the module's execution information is stored in a database and the model is updated each time new performance information becomes available.

For the module of the case study an execution time model and a memory model were built, based on the module's problem size as input parameter. Given a small number of initial problem sizes a reasonable accurate model was constructed. For the prediction of instruction counts, the resulting model was based on only three small problem sizes:

$$y = 9570.11 * x^{1.0858}$$

The mean prediction error over fifteen different problem sizes is 4.08% and is mostly due to the smallest problem sizes, not realistic in real life research, used. For the memory usage model more samples were needed to build the model with reasonable accuracy. The resulting model based on a fit for all problem sizes is:

$$y = -524.64 + 1227.3x + -324.72x^2 + 41.18x^3 + -2.5233x^4 + 0.006048x^5$$

The mean prediction error over fourteen different problem sizes is 0.38% with a maximum error of 1.10%

The profiling techniques used for building the models proved to be architecture independent, because the model constructed on a Pentium 3 processor architecture was able to make accurate predictions on a AMD processor architecture. For the execution time model the mean error was 4.09%, which is almost the same as the mean error for the predictions made on the Pentium 3. For the memory usage model used to predict executions on AMD, the mean error was 0.55%, compared to 0.38% for predictions made on the Pentium 3.

The major disadvantage of the execution time model is compiler dependence and the memory usage model is restricted to the Linux operating system. More application specific approaches are future work and discussed in the next section.

6.1 Discussion

The current implementation of the proposed performance modelling architecture is only a prototype. Future plans are to move the instrumentation part of the architecture to become part of the `VLport` library. The database currently consists of textfiles, this should become a more advanced structure. Historic performance data will become searchable, easier retrievable and it can be used if the analytical model's prediction is to inaccurate or given the problem size, performance data is available. Furthermore, an appropiate format should be developed to store the analytical performance model, developed by the model builder, in the database. For one application class the architecture has shown to be useful. Future plans are to add support to other application classes.

The execution time model and memory model should be enhanced to become more architecture independent. A few possibilities were already discussed in this thesis. For example, DynInst [39], a library for dynamic binary instrumentation adds possibilites to overcome the compiler dependence now present at both models. This also has an advantage for the memory model. If load and store calls in the programs binary are profiled, memory reuse distance [43] can be measured. Memory reuse distance is a measure for the number of distinct memory locations that are referenced between two references of the same location. Given memory reuse distance, the data locality can be predicted. However, the overhead is rather large, because each memory reference has to be profiled.

The gVLAM execution framework currently does not have a performance monitoring component. Autopilot [33] provides the appropiate infrastructure for performance monitoring and active steering. By making use of performance monitoring, feedback to the scheduler will serve as input for rescheduling decisions.

Bibliography

[1] L. Smarr, *Grids in Contexts*, Chapter 1, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers Inc., 1999, pp. 1-13.

[2] I. Foster, C. Kesselman, *The Globus Toolkit*, Chapter 11, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers Inc., 1999, pp. 259, 278.

[3] F. Berman, *High-Performance Schedulers,* Chapter 12, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers Inc., 1999, pp. 279-309.

[4] M. Livny and R. Raman, *High-Throughput Resource Management*, Chapter 13, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers Inc., 1999, pp. 311-337.

[5] The Globus Project, http://www.globus.org

[6] I. Foster, C. Kesselman, S. Tuecke. *The Anatomy of the Grid: Enabling Scalable Virtual Organizations.* International J. Supercomputer Applications, 15(3), 2001.

[7] I. Foster, C. Kesselman, J. Nick, S. Tuecke, *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[8] A. Belloum, D. Groep, Z. Hendrikse, B. Hertzberger, V. Korkhov, C. de Laat, D. Vasunin, *VLAM-G: a grid-based virtual laboratory*, Future Generation Computer Systems 19, 2003, p 209-217.

[9] K. Czajkowski et al., *A Resource Management Architecture for Metacomputing Systems*, Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, pg. 62-82, 1998.

[10] C. Liu, L.Yang, I. Foster, D. Angulo, *Design and Evaluation of a Resource Selection Framework for Grid Applications.*

[11] J. Schopf, L. Yang, *Using Predicted Variance for Conservative Scheduling on Shared Resources,* Chapter 15, Grid Resource Management: State of the Art and Future Trends, Kluwer Academic Pub, 2003, pp. 215-237.

[12] H. Dail et al. *Scheduling in the Grid Application Development Software Project,* Chapter 6, Grid Resource Management: State of the Art and Future Trends, Kluwer Academic Pub, 2003, pp. 73-95.

[13] J. Schopf, *Ten Actions when Grid Scheduling,* Chapter 2, Grid Resource Management: State of the Art and Future Trends, Kluwer Academic Pub, 2003, pp.15-23.

[14] D. Vasunin, V. Korkhov, A. Belloum, Z.G. Hendirkse, R.G. Belleman, VLAM-G Modules Developer's Guide, revision 1.17, februari 2004.

[15] D. Vasunin, Jwrapper developers guide.

[16] http://omniorb.sourceforge.net/

[17] V. Korkhov, A.S.Z. Belloum, L.O. Hertzberger, Evaluating Meta-scheduling Algorithms in VLAM-G Environment, 2003.

[18] Usersguide for GVLAM, 2005.

[19] P. Au, J. Darlington, M. M. Ghanem, and Y. Guo. *Co-ordinating Heterogeneous Parallel Computation*. In L. Boug, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, Euro-Par '96, pages 601-614, August 1996

[20] .N.C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, P. Stephan, *Dome: parallel programming in a distributed computing environment,* in: Proceedings of the 10th International Parallel Processing Symposium (IPPS 96), IEEE Computer Soc. Press, Silver Spring, MD, April 1996

[21] J. Gehring and A. Reinefeld. *MARS - A Framework for Minimizing the Job Execution Time in a Metacomputing Environment*. Future Generation Computer Systems, 12(1):87–99, 1996.

[22] F. Berman and R. Wolski, *The AppLeS Project: A status report*, Proceedings of the 8th NEC Research Symposium, Berlin, Germany, May 1997.

[23] F. Berman et al., *New Grid Scheduling and Rescheduling Methods in the GrADS Project*, Parallel and Distributed Processing Symposium 2004 Proceedings, pp. 199-206.

[24] K. Kennedy et al. *Towards a framework for preparing and executing adaptive grid programs*, Proceedings of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium, April 2002).

[25] H. Dail et al, *A modular scheduling approach for Grid Application Development Environments,* 2002.

[26] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselamn, D. Reed, L. Torezon and R. Wolski, The GrADS project: Software support for high-level Grid application development. International Journal of High-Performance Computing Applications, 15(4):327-344, 2001.

[27] V. Taylor et al., *Prophesy: Automating the Modeling Process,* Third Annual International Workshop on Active Middleware Services, 2001.

[28] V. Taylor, X. Wu, R. Stevens, *Prophesy: An Infrastructure for Performance Analysis and Modeling of Parallel and Grid Applications,* ACM SIGMETRICS Performance Evaluation Review, Volume 30, Issue 4, March 2003

[29] V. Taylor, X. Wu, J. Geisler, *Using Kernel Couplings to Predict Parallel Application Performance,* in Proc. of the 3rd international Symposium on High Performance Distributed Computing, Edinburgh Scotland, July 24-26, 2002.

[30] S.L. Graham, P.B. Kessler, M.K. McKusick, *gprof: a Call Graph Execution Profiler,*

[31] Dongarra, J., London, K., Moore, S., Mucci, P., Terpstra, D. *Using PAPI for Hardware Performance Monitoring on Linux Systems* , Conference on Linux Clusters: The HPC Revolution, Urbana, Illinois, June 25-27, 2001.

[32] J.W. Cooley and J.W. Tukey, An algorithm for the machine calculation of complex Fourier series,. Mathematics of Computation, 19(90):297-301, 1965.

[33] Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci, and Daniel A. Reed, Autopilot: Adaptive Control of Distributed Applications, Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, IL, July 1998.

[34] R. Wolski, N.T. Spring and J. Hayes, *The network weather service: a distributed resource performance forecasting service for metacomputing,* Future Generation Computer Systems, 15(5-6):757-768, 1999.

[35] K. Czajkowski, S. Fitzgerald, I. Foster and C. Kesselman, Grid information services for distributed resource sharing. In Proceedings of the Tehnth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), August 2001.

[36] T. Hey and A.E. Trefethen, The UK e-science core programme and the grid, Future Generation Computing Systems,18(8):1017-1031, 2002.

[37] Vl-e, Virtual Laboratory for e-Science, http://www.vl-e.nl

[38] DynaProf, http://icl.cs.utk.edu/˜mucci/dynaprof/

[39] Bryan R. Buck, Jeffrey K. Hollingsworth , *An API for Runtime Code Patching* , Journal of High Performance Computing Applications 14 (4), Winter 2000.

[40] B.P. Miller, M.D. Callaghan, J.M. Cargille, et al./ The Paradyn Parallel Performance Measurement Tool, IEEE Computer, 28(11):37-46, November 1995.

[41] J. Cao, D. J. Kerbyson, E. Papaefstathiou and G. R. Nudd. Performance Modeling of Parallel and Distributed Computing Using PACE. Proc. 19th IEEE Int. Performance, Computing and Communications Conf., Phoenix, AZ, USA, 485-492, 2000.

[42] Fast Fourier Transform of the West, http://www.fftw.org

[43] Chen Ding , Yutao Zhong, *Predicting whole-program locality through reuse distance analysis*, ACM SIGPLAN Notices, v.38 n.5, May 2003.

[44] Valgrind, http://valgrind.org

[45] [http://www.techonline.com/community/ed_resource/feature_article/6397]

.1 Linear Least Squares Curve Fitting

Curve fitting is concerned with the fitting of an analytical function to a set of data points. Given m pairs of data (x_i, y_i), where $i = 1, \ldots, m$, try to find the parameters a and b such that $y = a * x + b$ is a good fit for the data. This is an example of linear curve fitting. Linear curve fitting deals with functions that are linear in the parameters, but the variables them self don't have to be necessarily linear. $y = a * x * x + b * x + c$ can be solved with linear curve fitting, but the variable x is quadratic. The parameters are linear, because they appear as multipliers for the variable x. There also exist non-linear problems for curve fitting. Exponential functions, where the parameters are part of the exponential have this property. For example, $y = a * e^{(-b*x)}$. This one can be linearised by taking the logarithmic of both sides. If a constant is added to the function, taking the logarithmic does not work anymore. The system is not able to be solved with linear methods anymore. A way for solving linear parameterised models is by Gaussian elimination. Given a linear system that is over determined, we can solve any linear combination of functions, as long as the following property is fulfilled:

$$y = a_1 f_1(x) + a_2 f_2(x) + \ldots + a_n f_n(x) \tag{1}$$

It is important that $f_i(x)$ contain no parameters in the function, like for example $sin(c_1 x)$. For any given points (x_i, y_i) fitting the polynomial coefficients for the vector matrix $a = \{a_1, a_2, ..., a_n\}$yields for the matrix vector multiply:

$$\begin{bmatrix} 1 & x_1 & \ldots & x_1^k \\ 1 & x_2 & \ldots & x_2^k \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \ldots & x_n^k \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_k \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{2}$$

$$y = X * a \tag{3}$$

$$X^T y = X^T * X * a \tag{4}$$

$$a = (X^T * X)^{-1} * X^T * y \tag{5}$$

A measure to determine the goodness of the fit for the function's coefficients a, is called residual. This represents the mean squared error between the measured data and the predicted data.

$$r^2 = \sum [y_i - f(x_i, a_1, a_2, ..., a_n)]^2 \tag{6}$$

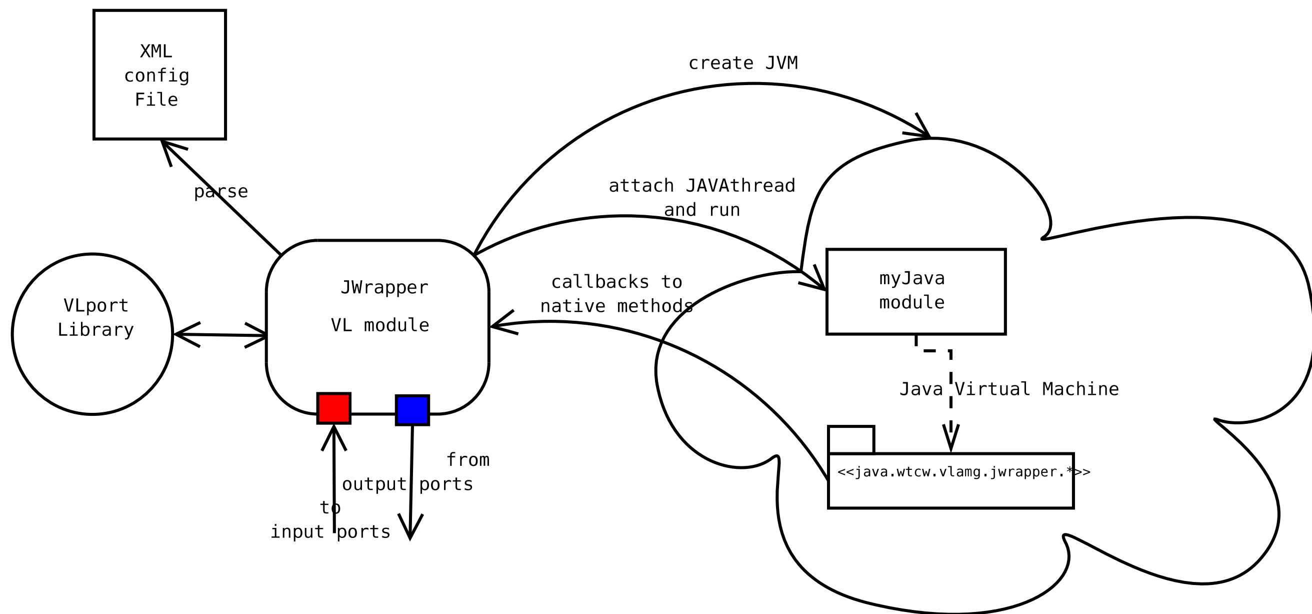


Figure 1: *JWrapper*

.2 JWrapper

To be able to construct modules as Java applications, a module called JWrapper is available [15]. Figure 1 shows a picture of the Jwrapper functionality.