XCPU3: Workload Distribution and Aggregation

Pravin Shinde

Student Number: 1824368

Department of Computer Science

Vrije Universiteit, Amsterdam

A thesis submitted for the degree of

Master in parallel and distributed systems

July 2010

Supervisors:

Eric Van Hensbergen

Prof. dr. Herbert Bos

Abstract

With growing acceptance of clusters and grids, the parallel programming paradigm is coming out from the research community to the business domain. Unfortunately the applications in the business domain are typically dataflow workloads which differ significantly from the traditional high performance computing(HPC) workloads. Most of the existing workload distribution solutions concentrate more on HPC applications and are not efficient in quickly deploying the dynamic workload involving large number of small jobs which is the typical case in dataflow applications. Also developing and deploying applications on existing infrastructures still is a non-trivial task requiring special runtimes, middleware support and/or language dependence.

We have explored an alternate approach for simplifying workload distribution and aggregation based on synthetic filesystems and the private namespace concepts of Plan 9. Instead of sending workloads to a remote cluster, XCPU3 works by bringing the remote cluster to the user's desktop, leading to full control over the compute environment. XCPU3 provides a filesystem interface, making it runtime and language agnostic. It also allows executing multiple jobs simultaneously in isolation, leading to better resource utilization. XCPU3 makes all nodes independent and equivalent in functionality leading to the ability to localize the decision making and ability to handle dynamic workloads. XCPU3 provides easy to use, flexible and scalable infrastructure for workload distribution and aggregation that can be used by dynamic workloads and dataflow applications.

Contents

List of Figures

Chapter 1

Introduction

The aim of this project is to develop an infrastructure for scalable deployment of dataflow applications which is simple to use. We have evaluated the infrastructure on the Blue Gene supercomputer[Tea08]. We have developed the XCPU3 filesystem in the Inferno[Vit] kernel which can be deployed in a hosted environment on most operating systems. XCPU3 provides a filesystem interface to the applications and workload distribution/aggregation in a language and runtime agnostic way. This filesystem interface also provides much needed flexibility by allowing control over each and every job, enabling the ability to deploy dataflow applications. We show that the XCPU3 infrastructure is simple to use, and at the same time flexible and scalable.

1.1 Problem statement

As silicon technology used in chip design of computers is reaching its peak, it is also approaching the limitations imposed by physics. The size of transistors is reaching the minimal limit which can reliably work. The heat generated by these chips is also increasing rapidly with the increase of clockspeed. This problem is commonly referred as the heat wall and is a major obstacle in increasing the cpu clockspeed.

These limitations has pushed the computer industry in the direction of multicore architectures. Having a large number of less powerful computers is assumed to be economically more viable than having a single more powerful computer. Blue Gene

is an example of how one can achieve higher performance by using more processors in parallel.

This new trend in hardware is also affecting the way applications are written. The traditional approach of writing applications with a single sequential flow of control is no longer efficient. The key lies in composing and developing the application to exploit underlying multiple cores and nodes.

The scientific community has been developing applications using parallel computing to solve large problems. These are compute intensive applications primarily developed from the beginning for deployment on clusters. Typically these applications work by dividing the big problem into smaller problems and allocating these small problems to separate nodes. Each small problem can be solved either independently or with limited communication with other nodes. These parallel applications are typically written in special languages like Fortran or depend on some runtime environment or library like MPI to work.

But this compute model is not universal. Specially the applications in business domains fall into category of data intensive instead of being compute intensive. Most of business applications fall into the category of **dataflow applications**. Dataflow applications can be visualized as a **directed acyclic graph(DAG)** where each vertex is individual computation and every edge is communication between these computations. Figure 1.1 shows typical example of these dataflow applications.

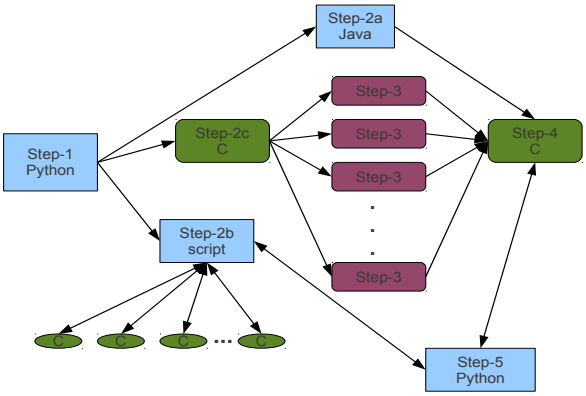

Figure 1.1: Typical dataflow application

The increasing popularity of clusters, grids and recent availability of cloud computing[ec2] [azu] [app] has made the availability of these resources commercially feasible for most businesses. But the major obstacle in widespread adoption is the expertise needed for writing parallel applications. Also, dependence on a specific language or run-time environment for parallelism dictates the need for major re-writing all existing applications for exploiting the parallelism.

Deployment of scientific applications do not focus much on job startup time as it typically includes deploying a single big application running for a long period. Most of the performance improvement is measured after the job is started, ignoring the job startup time. But dataflow applications typically include running large number of small applications for short periods of time. This requirement gives significance to job startup time and demands more efficient job deployment solutions.

Another crucial property exhibited by dataflow applications is that the amount of resources needed for total computation is unpredictable in the beginning as it may depend on results of intermediate computations in the DAG. This leads to the requirement of dynamically allocating new resources whenever needed. Most of the current job schedulers do not support such flexibility.

1.1.1 Blue Gene

Blue Gene [Tea08] is a supercomputer aimed at reaching speeds in the petaFLOPS. This architecture evolved from Blue Gene/L, Blue Gene/C, Blue Gene/P and Blue Gene/Q. These are massively parallel machines with 65,536 nodes in the biggest installation.

Blue Gene is a hierarchical architecture with a controller node at the root managing IO nodes. Each IO node manages 64 compute nodes, providing distribution and aggregation points for them. The compute nodes are the leaf nodes responsible for computation.

Compute nodes run a light weight operating system called *Compute Node Kernel(CNK)*. CNK is a minimal operating system supporting a subset of the POSIX calls and it allows only one process to run at a time. Typically, applications are developed using C, C++ or Fortran with use of MPI for communication. All compute

nodes under an IO node are connected with each other using the 3D torus network enabling direct point-to-point communication between these compute nodes.

IO nodes run the Linux operating system and are primarily responsible for data distribution, aggregation and filesystem operations on behalf of compute nodes. It enables compute nodes to communicate with the outside world. Blue Gene nodes are also connected to each other by the hierarchical tree network where IO nodes constitutes a root of a tree of compute nodes.

All IO nodes are connected to a service node which works as a gateway for users by providing a command interface. The service node runs a Linux operating system and is mainly responsible for resource reservation using special partitioning. It allows multiple users to use Blue Gene by partitioning the nodes into electronically isolated sets. This gives isolation and protection to the user applications which are running simultaneously. These partitioned nodes are rebooted with a clean environment which provides a pristine environment to every user in every reservation. Once the reservation is over, these partitioned nodes are released.

1.1.2 Drawbacks

Blue Gene was primarily aimed at compute intensive and large scientific applications like simulation of protein folding and quantum chemistry. It is not an ideal environment for dataflow applications which mostly consists large number of small individual applications. Traditional Blue Gene applications are tightly bounded to this environment and do not run anywhere else. As these applications don't run on typical workstations, the development and testing process is difficult and needs access to the Blue Gene setup. As all applications needs to be re-developed specifically for this platform, it leads to huge initial investments and commitment to this platform which is not desirable for commercial dataflow applications.

In the current setup, every node runs one process, and all nodes in one partition runs the application, which makes sense for compute intensive scientific applications. But typical dataflow applications may not utilize the entire CPU, leading to inefficient use of resources.

The partitioning of nodes is done electronically at the granularity of IO nodes, each partition may contain multiple of 8, 16, 32, 64 or 128 nodes. This base value is

configurable at the hardware level and not at the software level. So, once any value (suppose 64) is chosen, then all partitions on that hardware will contain multiples of 64 nodes. As most of the scientific applications are designed to scale up or down to any degree of parallelism, absorbing additional nodes available because of such partitioning is easy. But typical dataflow applications are scheduled by treating them as DAG and mapping each computational vertex to a node. Hence they cannot easily absorb the nodes beyond the number of computational vertices.

Every new reservation leads to the creation of a new partition, and all nodes in that partition are rebooted. This leads to a longer job-startup time. Again, it does not affect scientific applications much as these applications run for long duration. In contrast, dataflow applications involve starting large number of small jobs. Many times, these jobs are run multiple times. Hence these applications are adversely affected by longer job-startup time.

1.1.3 Alternate possibilities

Recently the Plan 9 [P+95] operating system has been ported to the compute and IO nodes of Blue Gene.[VHE07] Figure 1.2 presents a new Blue Gene setup with Plan 9.

This setup creates new possibilities for deploying dataflow applications and other execution models. With the availability of Plan 9 on compute nodes we can overcome many limitations posed by the CNK kernel and availability of Plan 9 on IO node can be used to give much needed flexibility to dataflow applications.

1.2 Goal

Our goal is to create a scalable infrastructure with the flexibility needed for running dataflow applications. We primarily target the Blue Gene supercomputer as our testbed as it allows us to test our infrastructure on up to 64K nodes. Typically, the Blue Gene installations are administered by single administrative domain. Also the Blue Gene hardware is designed to provide high reliability. By relying on these features of Blue Gene, we are able to ignore many complications faced by typical distributed systems which are aimed to run on unreliable hardware under different

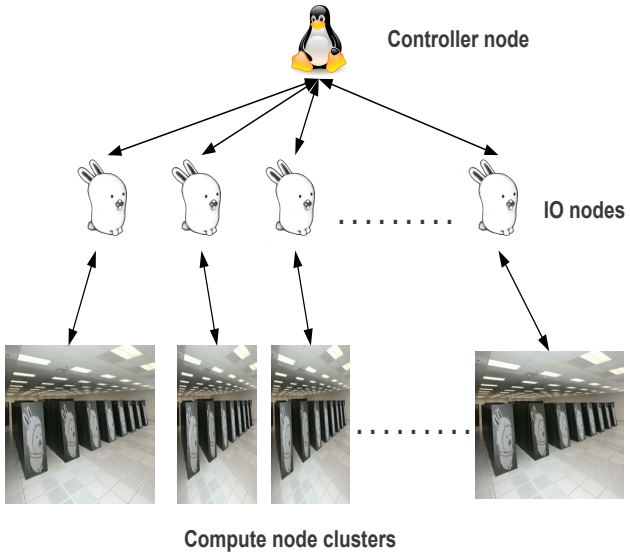

Figure 1.2: Bluegene setup

administrative domains. Fault tolerance is a desirable feature of this infrastructure but it is out of scope for this project.

The challenge is to provide the flexibility needed by dataflow applications using a very simple interface. We need to provide an infrastructure with support for dynamically adjusting the resource reservation which can help in coping with unpredictable resource requirements. This infrastructure needs to provide quick job-startup time even for large numbers of jobs, allowing overall good performance for dataflow applications. It also needs to provide a simple interface for communication between nodes even when those nodes cannot communicate directly with each other. We do not want the application development for this infrastructure to be constrained to a particular language, runtime, middle-ware or platform. And we need to run most of the existing applications without modification.

This project is not aimed at performance improvement, but to present a new and elegant way for workload distribution and aggregation. It is aimed at demonstrating that simple filesystem interfaces and a simple design based on the basic principles of the Plan 9 distributed operating system can be easy, intuitive and scalable to a large

number of nodes.

This project does not aim to entirely solve the problems of dataflow applications, but to provide a scalable and flexible infrastructure which can be effectively used to implement them.

This thesis aims to explore alternate possibilities which will simplify the deployment of dataflow applications on infrastructure like Blue Gene. Even though we are taking Blue Gene as a test setup, we aim to develop a generic solution which can be used in any cluster setup. The next chapter presents the background work which has inspired this project. Chapter 3 will explain the design of our solution. Chapter 4 will discuss how it is implemented. Chapter 5 explains the filesystem interface used and shows a few examples to show how easily this interface can be used. Chapter 6 presents some evaluations showing that this approach is feasible. Chapter 7 presents related work and positions this thesis in the context of them. Chapter 8 will conclude this thesis.

Chapter 2

Background

There have been work going on to open Blue Gene for more generic applications, and Plan 9 has been ported to Blue Gene[VHE07] as part of that effort. This has lead to exploration of alternatives for workload distribution on this platform.

Following are the concepts/work that inspired our exploration. We have borrowed many concepts from these works and used them to meet the requirements of dataflow applications.

2.1 Plan 9

Plan 9 is a distributed operating system which is able to simplify development of distributed applications. The Plan 9 design is based on three primary design principles.

1. **Everything is file** : Plan 9 presents everything as file. This includes devices and the processor itself. All these resources are accessible in a hierarchical namespace.

2. **9P: Standard communication protocol** All files are accessed using the 9P protocol. This allows accessing both local and remote resources in same way.

3. **Namespaces** Each process has it's own namespace view. This namespace view can be separately built for every process, allowing every process to have a different view of the world.[PPT+93]

As everything in Plan 9 is a file, it is easy to share everything as file over network. It also allows transparently sharing its resources as files over network. The 9P protocol removes the differentiation between local and remote resources by providing the same interface for accessing both of them. Private namespaces allow a process to build its own namespace using any resources anywhere over the network. These features allow arranging any remote resources in any view and using them transparently. In the following sections, we will see how these concepts can be used to simplify the deployment of applications on remote nodes.

2.1.1 CPU

CPU[plaa] is Plan 9's concept for using remote compute resources. It is different from traditional rlogin, telnet or ssh. Most of these protocols work by logging in on a remote node and providing the environment of the remote node to user. Only the local terminal is used while all other resources like the processor, disk and filesystem are from the remote node.

In contrast, CPU works by bringing remote resources to the user. Figure 2.1 presents CPU.

It works by creating new private namespace[PPT+93] with a local filesystem view, disk, network, console, etc on the client side. This namespace is mounted by the remote processor and used for all the processes started there as part of the CPU session. These processes have the view of the client system while the execution is still performed on a remote node. One of the key advantages of this model is that now users don't have to worry about the environment of the remote node when running any job on it. Any execution requested on a remote node will see the same local view of the user's environment, simplifying the deployment of applications.

Even though CPU does simplify the life of the developer, it does not solve all of our problems. Firstly, it is Plan 9 specific and does not work with any other operating system. This keeps it out of realm of most of the users. Another limitation is that CPU works in one-to-one fashion. This limits the usefulness of it in cluster like environments. Controlling each and every node in a cluster individually does not scale well for larger numbers of nodes.

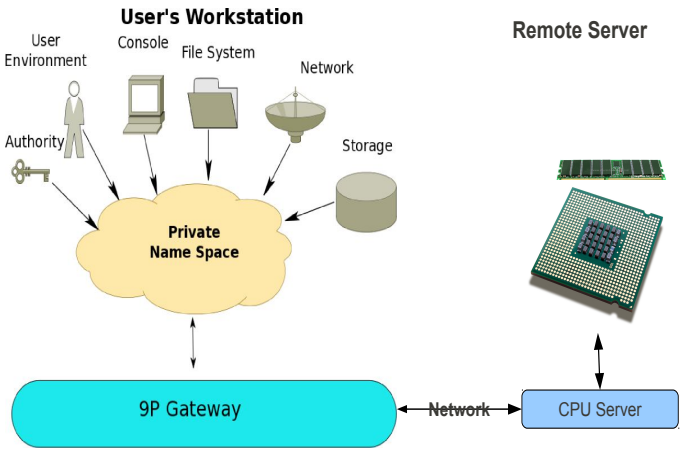

Figure 2.1: CPU

2.2 XCPU

XCPU[MM06] [IMM06] was an attempt to bring the functionality of CPU into mainstream operating systems like Linux. It aims at providing the remote resources using a **filesystem interface** and improve the scalability. XCPU can be seen as the remote process execution system presenting execution and control services in the form of files in a hierarchical filesystem. This filesystem can be exported over the network so that other nodes can use it. As XCPU presents remote compute resources as files, deployment of applications does not need any special privileges as long as user can read/write these files.

XCPU has two types of nodes, the controller node which is responsible for starting and controlling the job and compute nodes which actually do the computations.

In addition to execution of the program, XCPU is also responsible for distributing them to remote nodes. XCPU differs from CPU in that it does not use the private namespaces to create local environments on remote nodes. It uses the **push model** and automatically pushes the executable and all the dependencies of it to the remote

node. The dependencies are analyzed using static analysis of the executable requested by the user.

To achieve scalable pushing of programs and their dependencies, XCPU uses a few compute nodes for distributing the programs and dependencies during the program startup time. This is known as the **tree spawn mechanism** and it improves the scalability for XCPU to larger numbers of nodes. Figure 2.2 presents a typical tree hierarchy used by XCPU for better scalability.

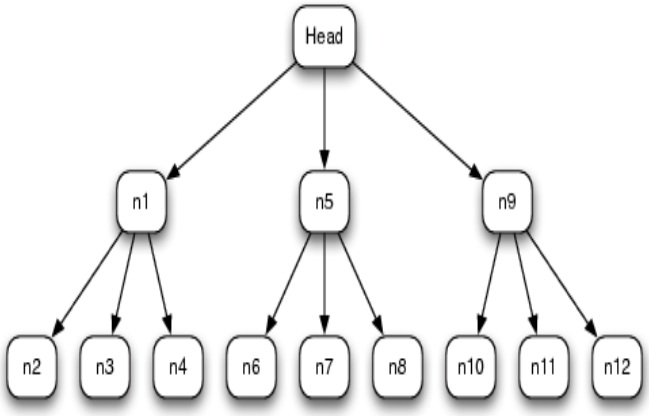

Figure 2.2: XCPU tree spawn mechanism

With automatic pushing and the tree spawn mechanism, XCPU not only simplifies the deployment of applications but it also scales them better with quicker job startup time.

2.2.1 Drawbacks

One major drawback of XCPU is the result of the push model. All dependencies are needed to be pushed to compute nodes. The difficulty lies in detecting the dependencies of the requested executable. Static analysis is not capable of detecting dependencies like configuration files which are known only at runtime. The user needs to explicitly provide information about these dependencies so that XCPU can push them.

XCPU is designed for executing one application on the cluster at one time. This is not very useful when multiple users want to use different parts of the cluster at the same time. XCPU does not aim for such situations, leading to limited flexibility.

XCPU has properties like quick job startup time and a simple filesystem interface which makes it a good first step towards scalable dataflow application deployment. But there is still a lot of gap to cover for meeting all the requirements of dataflow applications.

2.3 XCPU2

XCPU2 [IH09] is evolved from XCPU and is aimed to provide more flexibility. XCPU2 was inspired by the use of **private namespace** of CPU. Recently the Linux kernel has added support for private namespace[Bie06], and this facilitated the development of XCPU2. The much needed flexibility is provided by the ability of building a separate view of the filesystem for each process using these private namespaces.

XCPU2 works by dividing the nodes into three categories. Each type is responsible for different functionality.

1. **A Control Node** is responsible for handling **reservation** requests for compute nodes. There will be a single control node per cluster for this role. XCPU2 does not take responsibility of reservation. This part is typically delegated to some other software which accepts the reservation requests and returns the list of nodes assigned for reservation request.

2. **A Job Control Node** is responsible for **managing a single job**. The nodes provided by the control node for reservation requests are then managed by a job control node which is responsible for starting, stopping the job and manipulating the filesystem view of compute nodes. Users will typically interact with these job control nodes for running their applications.

3. **A Compute Node** is responsible for **actual computation**. These nodes communicate with the job control nodes and accepts the application names that are to be executed. These nodes also allow job control node to modify the filesystem view of process executing the application by using private namespaces.

A user typically starts by requesting the reservation with the control node. Once the reservation is done, the user will get an access to the job control node which can be used to manage the job. The user can export his local filesystem view to the job control node with the name of the executable to execute. All compute nodes mount the user's local filesystem via the job control node. All compute nodes then create processes with the namespace modified in such a way that it will inherit the user's filesystem view instead of the compute node's filesystem view. This way, processes running on a compute node will see the same uniform view as processes in the user environment. In contrast to XCPU which pushes the executables and dependencies, XCPU2 uses the **pull model** where any dependency is automatically pulled because of the inherited user filesystem view using private namespace.

This propagation of the user's environment on the compute nodes simplifies the application deployment to a great extent. If an application is properly running on the user environment, then it will also run on the compute nodes as they have the same filesystem view. This is a big improvement over other environments where developers are forced to develop the applications for an environment of the compute nodes which may differ significantly from the user environment. Also, developers can use any libraries or customized tools within a distributed application without worrying much about availability of those in the compute environment. Users neither need special privileges nor do they need to make special requests for installing special libraries and modified tools on every nodes of the cluster. This model also simplifies the maintenance of the cluster as there is no need to maintain every tool and every version of the libraries on each compute node.

The diagram 2.3 shows how multiple users can use the cluster at the same time for executing different applications with contradicting requirements on the filesystem view. The control node provides the partitioned set of compute nodes with one job control node to each user requested reservation. Now each user can execute its own application with it's own view of filesystem without any interference from other users who are running their applications at the same time on the same cluster. This provides added flexibility and better utilization of resources.

XCPU2 also inherits scalability from XCPU by using the tree spawning mechanism in compute nodes within a single reservation. This hierarchical structure reduces the load on the job control node and provides better job-startup time.

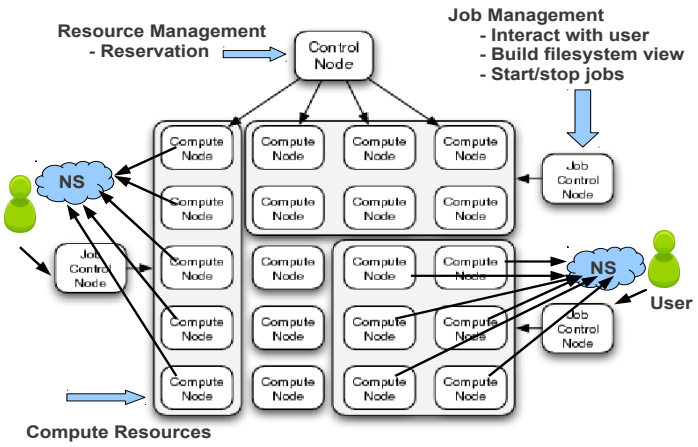

Figure 2.3: XCPU2 in action

XCPU2 is definitely a forward step toward providing the flexibility needed to deploy dataflow applications. It provides flexibility in controlling the namespace view and it allows multiple users to run their applications simultaneously with good job startup time.

2.3.1 drawbacks

The fundamental drawback with XCPU2 is the granularity of control. The control provided by the XCPU2 on the filesystem view of compute nodes is not fine grained. Users can control the filesystem view of all nodes together, but they can not control each and every compute node individually.

This lack of control on every node is a major deterrent for dataflow applications. As these applications can be best scheduled by treating them as DAG and mapping each computational vertex to one compute node. As each computational vertex can have different requirements about environment and filesystem view, control of each compute node is important for successfully mapping dataflow application on compute resources. *XCPU2 fails to provide this much needed fine grained control on all nodes.*

This limitation was major motivation for us to develop XCPU3 which can handle all needs of dataflow applications.

Chapter 3

Design

The key challenge in this design was to provide the flexibility needed by dataflow applications while achieving scalability to the large number of nodes available with Blue Gene.

Even though a lot of work was present in the form of XCPU and XCPU2, we decided to only borrow the concepts and lessons from them, but design and implement a new system from scratch instead of just extending XCPU2.

3.1 Key design decisions

We decided to adopt a few guiding principles which will help us to reach our goal of flexibility with scalability. We also decided to keep the design simple, and we believe that the simple design should lead to the simple and flexible interface. We also hope that the lack of complexity may help in improving scalability. In this section we present those decisions which influenced most aspects of system design and implementation.

3.1.1 Localization of decision making

The key requirement for us was scalability to a large number of nodes. We planned to design the system without any central component which should have knowledge of the entire system. This design decision implies that there is no central entity making decisions. We plan to distribute and localize the computations as well as decisions.

These decisions include scheduling, job management and workload distribution and aggregation.

The downside of this design decision is the lack of a global view at one location. We avoid decision making based on global knowledge and promote use of local information. We hope that if each node tries to attempt a local optimum, we will reach the global optimum. This may not be true in all cases, We hope that in those cases, even if we do not reach global optimum, we will perform acceptably well.

We also assume that there will be applications which will need a detailed global view of the system. We support generating such a global view by aggregating the information from each node. These operations are comparatively expensive because of communication involved. We leave the decision of using such a detailed global view to application developers who will be in best position to make such trade-off.

3.1.2 Independence of every node

In some sense, this design decision is a side-effect of the above decision. As we plan to distribute and localize all functionalities, it was essential to replicate these functionalities at multiple levels making localization possible. The granularity of functionality replication decides the granularity of control, and hence influences the flexibility provided by the system. As we aim for maximum flexibility, we have decided for replicating the following three functionalities at each node.

1. **Resource reservation**: Each node should be able to reserve more resources on it's own without involvement of any central entity.

2. **Job management**: Every node should be capable of starting new jobs using his reservations. The node should also be able to manage and interact with these jobs. And every node should also be able to terminate the jobs it started.

3. **Computation**: Every node should be able to perform the computation by running the requested application in isolation and returning the results.

We intend to provide each node with the capability to perform all of these roles simultaneously instead of binding them in one role at one time.

This design makes the interface to every node a building block identical to each other, and provides the flexibility to build any structure with these building blocks.

There are a few downsides in making every node independent. Now the traditional master-slave model is no longer valid where a master can assume the control over all resources of slave nodes. Each node has to politely request other nodes for help in performing the tasks and they should be ready to handle their requests for help being rejected.

With independence of every node, each node can be a source of failures and faults. One will need to come up with better ways to deal with faults and failures when so many sources of them are present. This takes the XCPU3 in realms of **Distributed Systems** opening many more possibilities and questions.

We avoid these complexities by assuming a very simple model for handling failures. Any failure anywhere in the system will result in the failure of the entire operation. We assume that failures will be in-frequent, so aborting and restarting operation should be acceptable for such infrequent failures.

We believe that this approach will work on setups like Blue Gene where failures are infrequent, but it will not be acceptable on other grid like system which are more prone to failures because of network, hardware and administrative issues. We limit ourself to reliable systems for the purpose of this project and keep other systems for our future work.

3.1.3 Filesystem Interface

We want to keep XCPU3 interface agnostic from language, runtime and middleware. Plan 9 has proven that the filesystem interface is very flexible and yet powerful in the world of distributed applications. We aim to follow the same principle of **Everything is a file** from Plan 9.

Every node will provide access to its services via filesystem interfaces. This interface will be exported as the filesystem over the network so that other nodes can use it. Other nodes can mount this filesystem and use it as interface for interacting with that node. Multiple remote nodes can be aggregated into the filesystem hierarchy providing a clean and easy way to access them.

Multiple overlay views can be created by *binding* the same filesystem at multiple locations with different names. This ability of creating ad-hoc overlays allow users to arrange remote resources as per his needs without worrying about their actual locations.

Other advantages of using filesystem interfaces are that

1. Existing tools/commands used with traditional filesystem can be directly used with XCPU3.

2. filesystems come with their own mechanism for access control list for providing the security. We can leave the security to these already proven mechanisms instead of implementing our own.

3. We inherit the ability to export, mount and bind the filesystem without writing any explicit code for it.

4. Filesystem interfaces are simpler to program than socket interfaces. This can lead to simpler code and hence lesser bugs.

5. Users don't need special privileges or administrative access to interact with the filesystem. This simplifies the user experience in running XCPU3 based applications.

With all the good features offered by the filesystem interface, it does impose some requirements/limitations. Following are a few limitations which affect our design decisions.

1. The requirement imposed by using this interface is that all operations on these synthetic filesystems should go to actual filesystems. This puts limitations on the application or middleware level caching. Such caching may end up providing stale values leading to errors.

2. Another more critical limitation concerns the POSIX standard for failure reporting in filesystem. POSIX standard dictates that file operations should report their success or failure in the form of a number. But a single number can not report enough information about the reason behind failures. This makes the

filesystem interface less desirable where failures are common and need more information for debugging. Plan 9 overcomes this limitation by returning a string instead of number. This string can provide much more valuable and verbose information if anything goes wrong. But as we are also aiming to deploy XCPU3 filesystem on POSIX compatible operating systems, we need alternative way to report failures.

3. Another drawback is the way a failure of remote services is detected. The filesystem interface relies on the underlying networking protocol for detecting failures by waiting for timeouts, and then reporting them back to users as an error. This makes the filesystem interface less desirable where quick failure detection and recovery is needed. This limitation can be overcome by an alternative design of using callback functions. If applications can register callback functions with operating systems, then these callback functions can be used to quickly report status/error and take recovery actions.

Our decision to choose the filesystem interface despite of it's drawbacks is the trade-off we are willing to do for flexibility and simplicity. We do understand the limitations imposed by this choice and we try to overcome them and inform the users of this system about these limitations.

1. We strongly encourage developers to directly interact with filesystems by using system calls and avoid using any middleware that might do caching. We also advise users to use the latest content of the file instead of previously read content.

2. We plan to use separate files in the filesystem interface to report errors in detail instead of entirely relying on returned error codes. That way, users can get more details about the error by reading this file.

3. We limit ourself in this project with the assumption that failures will be infrequent. With this assumption, we are willing to accept the delays in reporting failures at the remote end.

We want to provide flexibility without loosing the simplicity of the interface. With the above assumptions and restrictions, we believe that our decision of choosing the filesystem interface should provide the best flexibility.

3.2 Summary

We have built our system with the above three design decisions as our guiding principles. These decisions are taken by understanding the requirements of the dataflow applications based on a few assumptions about the reliability of the underlying system and a few restrictions on the developers. This solution may need modifications if any of the above parameters change. This solution is not designed to solve all the problems in all possible setups, it is targeted to the requirements we have discussed.

Chapter 4

Implementation

This chapter discusses the implementation issues in more detail. It also explains the filesystem interface and how it can be used. We will show how the filesystem interface and independence of nodes gives much desired flexibility. We will see some examples showing how easy it is to run applications on this platform. This chapter concentrates on how XCPU3 was implemented and describes implementation decisions which have proven to be important.

4.1 Inferno

Before we start discussing our decision to use Inferno, let us explain what Inferno is. Inferno[Vit] is an open source distributed operating system developed and maintained by *Vita Nuova*. It is a direct descendant of the **Plan 9** operating system. It runs natively on multiple hardware platforms and can also run as a user-space application on top of other operating systems. For the purpose of this project *"hosted Inferno OS"* or *"userspace Inferno"* refers to Inferno running as user-space application on top of other operating systems. So we will use these terms interchangeably.

As we are aiming for a flexible heterogeneous environment with different operating systems and different architectures, we found the Inferno operating system to be an attractive platform. It allowed us to develop XCPU3 once and easily deploy it on different platforms while enjoying the Plan 9 features on all those platforms.

This decision does come with the cost of some performance loss. Each node needs to run hosted Inferno in user-space which takes up some resources of the node.

Also every communication with the filesystem involves invocation of the host kernel which then passes the call to the guest inferno. The responses follow the same path. This does add latency in communication leading to poorer performance compared to filesystems implemented natively on host operating systems. But we are willing to accept this performance loss for achieving the portability to a large number of platforms.

We also had a choice of developing XCPU3 as part of the Inferno kernel or as an Inferno user-space application. User-space development is comparatively easy and user-space tools can be used for debugging any problems. On the other hand, kernel-space development comes with more constraints and practically no support for debugging. Any bug in kernel-space code typically leads to a kernel panic giving very little information about the cause of the bug. The advantage of kernel-space development is that serving user requests coming to XCPU3 will not involve context switches between Inferno user-space and kernel-space. As the choice between Inferno kernel-space and user-space does not affect the portability, we decided to implement XCPU3 in Inferno kernel-space for better performance. We are also hoping that this decision will help in easily developing a high performance native Plan 9 port for XCPU3 as the code-base of Plan 9 and Inferno kernel are similar.

Our choice of Inferno gave us the flexibility and features of the Plan 9 operating system on other operating systems. It allowed us to quickly implement and deploy our filesystem on multiple platforms easily at the cost of some performance.

4.2 Implementation

XCPU3 is implemented as the filesystem in the Inferno kernel. Figure 4.1 shows the placement of XCPU3 in the context of applications, host operating system and the Inferno.

The XCPU3 filesystem is exported by Inferno using the `9P` protocol. This protocol is used by Plan 9 and Inferno extensively to access any file. Recently the Linux kernel has added support for the `9P` protocol[HM05]. This allows Linux to mount filesystems exported over the 9P protocol. Other Unix based operating systems can use FUSE[FUS] for accessing the 9P based filesystems. As FUSE based filesystems run in user-space, they perform slower compared to native filesystems implemented in

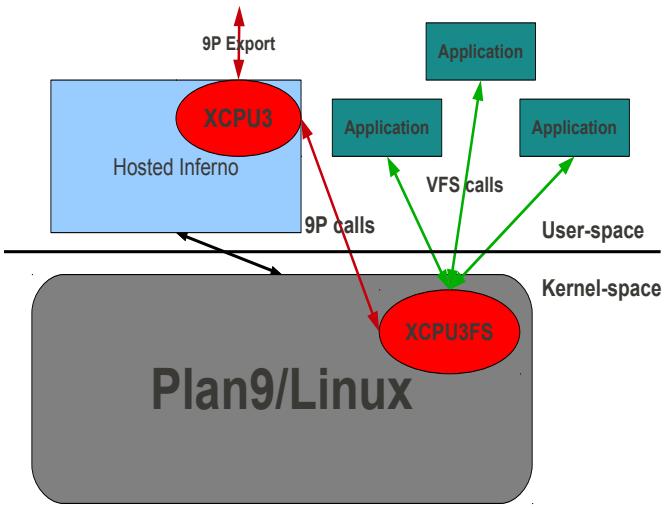

Figure 4.1: XCPU3 Structure

the kernel. But implementing FUSE based filesystems is much easier and faster. Applications can directly use the FUSE based filesystem just like a kernel implemented filesystem with the same API. Applications can also directly talk with filesystems exported over `9P` by implementing the client side of the `9P` protocol. This way they can get better performance by bypassing VFS and/or FUSE interfaces.

In a typical setup, Inferno runs in the hosted environment on an operating system like Linux or Plan 9. On its startup this hosted inferno will export the XCPU3 filesystem. This exported filesystem is mounted by the local host operating system. The applications interact with this mounted XCPU3 filesystem using the native interface. Any interaction with this XCPU3 filesystem is communicated to Inferno using 9P. The XCPU3 kernel module inside the inferno kernel then receives and interprets the user actions. If needed, it uses services from the host operating system or from other XCPU3 filesystems deployed on remote locations. The XCPU3 filesystem then sends the prepared response over 9P. The host operating system will relay this response back to the application via the local filesystem interface.

As we discussed before in the Design chapter this XCPU3 is an independent unit,

providing the functionality of reservation, job management and computation. In the case of an isolated instantiation, any reservation request is satisfied using local resources. This way, the application can use the same semantics for both local and remote executions.

4.3 The big picture (connecting multiple XCPU3 nodes)

The real power of the XCPU3 is the ability to connect with each other and form bigger instance of XCPU3. XCPU3 filesystem uses the `9P` protocol to connect with each other. This protocol provides access to both local and remote filesystems in the same way. This allows every node to access the XCPU3 filesystem hosted on any node without worrying about complexities of accessing remote filesystems. Those complexities are handled by the `9P` protocol.

4.3.1 Central Services

The ability to configure many XCPU3 nodes into hierarchy is provided by the *central services*. In contrary to the name, central services is highly distributed and every XCPU3 runs an independent instance of the central services. The administrator or the user of each XCPU3 instance need to provide only the information about its parent and children in the hierarchy and the all XCPU3 nodes initiate these connections leading to the distributed creation of the XCPU3 node hierarchy.

Central services is aimed to make all the resources addressable in the network-oblivious fashion. It also aims to have external entities (such as end-user workstations) participate in the hierarchy. It is also aimed to be able to work through multiple networks, NAT gateways and firewalls. Overall, central services is aimed to grants us a flexible facility for building hierarchies across different types of network and network boundaries in a distributed and secure fashion.

4.3.1.1 Implementation

The central services synthetic file server itself is quite simple. It provides a simple hierarchy of directory mount points representing remote nodes. Mounts of the remote nodes or binds of previously mounted remote nodes are accomplished within this file system such that anyone who mounts our name space can also see (and access) anyone we have mounted transitively, in such a way a child node can access a parent nodes, other children, or the parents nodes parent and so forth.

The original implementation of the central services mechanism used an auto-mounter-like interface. When you referenced a name within the synthetic file server for the first time, it would establish a connection to that systems central-services server and establish a duct. A duct is essentially a two way pipeline that allows the client to mount the sever and vice-versa – allowing each side to access the others resources. Each would create an entry in their respective mount table based on the node-name. This allowed clients sitting behind network-address-translation gateways to contact a parent and the parent to access that child without having to re-traverse the gateway. A command-line option could be used when starting central services allowing it to connect to a single remote resource over an ssh-connection establishing a tunnel capable of crossing firewalls.

It proved difficult to debug and did not allow us to leverage previously mounted filesystem (such as the parent's root filesystem from which we get our root filesystem). A second lighter weight version of the filesystem was created which allowed manipulation of the name space via a single control file at the top level with a simple name space oriented syntax:

- Establish a network connection and mount table.

  ```
  mount <remote-address> <name>
  ```

- Bind previously mounted resources.

  ```
  bind <path-to-remote-namespace> <name>
  ```

- Remove a resource.

```
remove <name>
```

Using this simple mechanism nodes could establish themselves within a hierarchy by binding a parent's central service directory to the name /csrv/parent and then tell the parent to back-mount their name space (allowing two way traversal). In this way children register with parents triggering the cross-mounts and establishing a graph which spans the entire system.

On the Blue Gene, the machine already has a natural aggregation topology of compute nodes organized under IO-nodes which accessed from front-end servers. During boot, our compute nodes mount IO-nodes which mount name spaces from the front-end servers for the purposes of establishing a distributed filesystem. We leverage this same aggregation topology for the purposes of central-services and the execution model.

4.3.2 Example topology

The figure 4.2 gives an example of how the XCPU3 filesystems can be be connected to form a larger system.

This example shows a hierarchical topology with three levels and with a couple of nodes behind a firewall. As ducts between XCPU3 filesystem are two-way, it allows node behind the firewall to participate in the larger topology as long as they at least have one link with any other node. In our example, node F can be seen and accessed by any other node via node B. The interface for such access is explained in *Filesystem interface* section. This way, central services play's key role in overcoming the partitioned networks and providing the view of all resources in network oblivious manner.

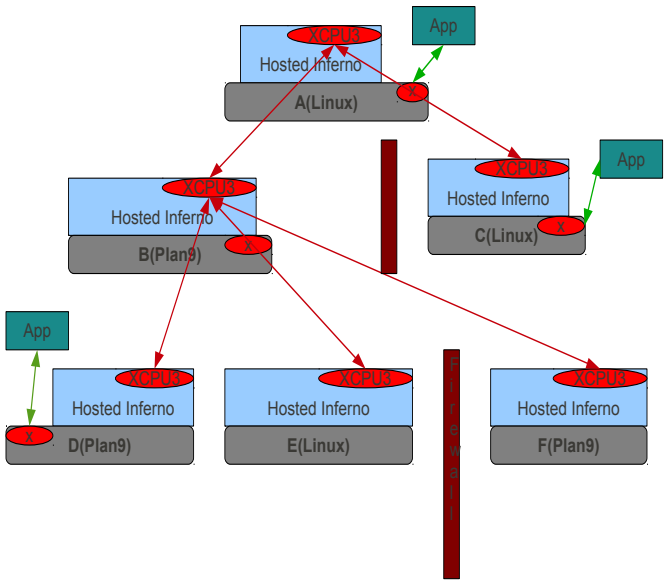

Figure 4.2: Sample topology of XCPU3

Chapter 5

Filesystem Interface

This chapter discusses the filesystem interface and how it can be used to access any node. We will mainly concentrate on the interfaces for accessing the nodes, local resources, sessions and sub-sessions.

5.1 Interface for remote nodes

XCPU3 follows certain conventions which allows every node to easily interpret the filesystem view of the other nodes. This section describes the XCPU3 interface and the conventions for accessing the remote nodes.

Figure 5.1 tries to give a simple overview of how this synthetic filesystem view is populated based on the underlying topology of the nodes. The nodes in the figure are arranged in the simple hierarchical topology. The XCPU3 filesystem starts with the `[/csrv]` directory. This directory contains the synthetic contents dynamically generated by the XCPU3. The location `[/csrv/local/]` presents the local resources whereas `[/csrv/parent/]` presents the XCPU3 (ie `[/csrv/]`) filesystem of the node which is the parent of this node in the physical topology. All other directories in the `[/csrv/*]` represents the XCPU3 filesystem of the remote nodes which are the children in the physical topology. The *App-1 view* is the XCPU3 filesystem view on the root node `[t]`. You can see that it has two directories in the `[/csrv/]` representing the local resources and the remote child L.

The good part of this setup is that every node has to worry about only its children and the parent, other topology falls in the place automatically. In our sample case,

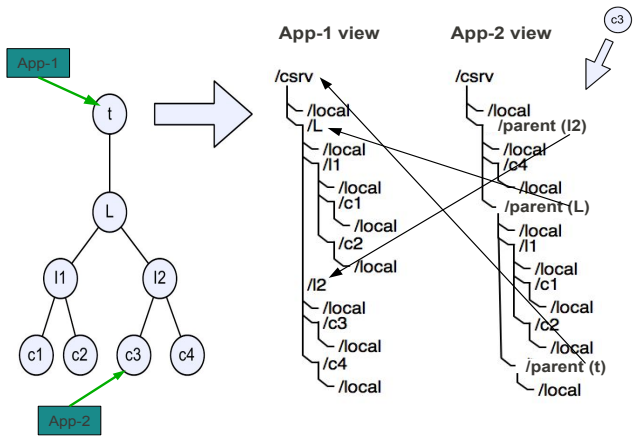

Figure 5.1: Sample filesystem interface for sample the topology in XCPU3

the node `[t]` only needs to connect to node `[L]`. It can access other nodes from the filesystem view of the node `[L]`. In tern, the node `[L]` is responsible for connecting to the node `[t]` as parent and node `[l1]`, `[l2]` as children. These connections allow him to access the entire cluster. This way, responsibility of making the connections with other nodes is also **localized** leading to better scalability. The *App-2 view* provides the `[/csrv/]` filesystem view at the leaf node `[c3]`. This node `[c3]` only connects to the node `[l2]` as it's parent and this single connection allows it to view the entire cluster.

This particular view of resources is highly dependent on the underlying cluster. Nodes are directly connected to the neighbors and indirectly connected to other nodes, hence maintaining the relationship with underlying cluster. Most applications may not want a topology based view of underlying resources, but the uniform view. For that reason we create an uniform overlay view separately for every application when they request the reservation. This overlay view binds the resources from this underlying hierarchy and hence successfully hides the complexities created by underlying topology. As every application receives its own session directory and all the requested resources are provided within the same session directory, the application can use all the resources within its session without worrying about interfering with others or the physical location of resources. As each application lives in it's own session, this design also enables the capability of running multiple applications simultaneously and

independently.

The XCPU3 infrastructure is designed to work by providing resources in a uniform way irrespective of the underlying cluster topology. But there is a class of applications and users that want to have the view of resources based on the cluster topology. The view provided by the `[/csrv/]` caters to the need of these applications as this view maps the resources to the underlying cluster topology.

5.1.1 Creating a global view

Even though the filesystem view at different nodes differ from each other, all nodes have access to the full topology. The `[/csrv/]` filesystem view encodes enough information within itself that any nodes can construct the global view and figure out their own position within the global view.

Just because every node can construct the global view, does not mean that it must use this global view for making any decision or performing a typical operation. The nodes mostly use only the local view for decision making and operations. This local view includes the parent node and the children nodes.

The ability to construct a global view is available to the applications running on any node which need this information. This operation is comparatively expensive as it involves traversing the `[/csrv/]` filesystem view. As each directory in `[/csrv/]` location except `[/csrv/local/]` is the XCPU3 filesystem of a remote node, any node can access the XCPU3 filesystem of other nodes using this traversal and use this information to build the global view. The XCPU3 infrastructure provides the interface to generate such global views, and leaves the decision to use this information at the expense of performance to the application developers and users.

5.2 Interface for local resources

This section describes the interface for accessing and managing the local resources at any node. Again this interface is uniform across all the nodes enabling applications and other nodes to access it in the same way. All local resources can be accessed in the location `[/csrv/local/]`. Figure 5.3 shows the typical content of this location.

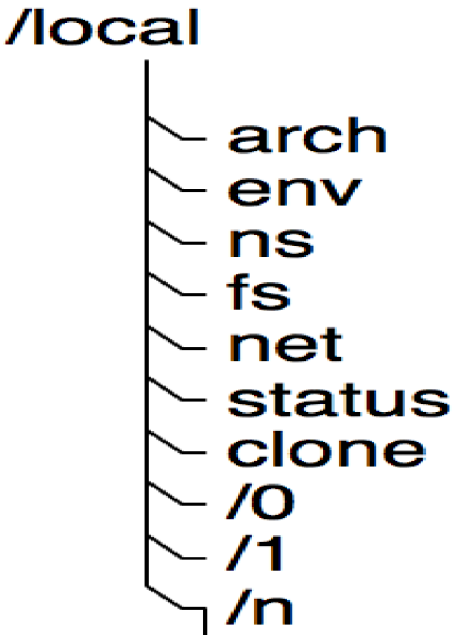

Figure 5.2: Sample filesystem interface for local resources in XCPU3

Each file and directory in the `[/csrv/local/]` has a specific purpose in accessing and managing the local resources.

5.2.1 arch

`[/csrv/local/arch]` is a *read-only* file. A read operation on this file returns the name of the host operating system and the hardware architecture. This information is non-destructive and can be read any number of times. The typical use-case of this file is

```
$ cat /csrv/local/arch
Linux 386
$
```

Typically, this information will be used by the scheduler to satisfy the constraints imposed by the applications for the underlying operating system and architecture.

5.2.2 env

`[/csrv/local/env]` provides the interface for specifying the environmental variables. This file can be opened for writing and multiple tuples of `[Name Value]` pairs can be written into it. Following is the example of how this file can be used

```
$ echo "PATH /home/example/" > /csrv/local/env
$
```

The environment variables added this way are the default variables and they will be available to all the sessions started on this node. We do provide another interface for controlling the environmental variables within a session. We will discuss that interface in more details in the next section.

5.2.3 ns

`[/csrv/local/ns]` file provides the interface to assemble the namespace that will be inherited by any application process created on this node.

```
$ echo "mount /exp/root/ /" > /csrv/local/ns
$ echo "bind /some/path /other/path" > /csrv/local/ns
$
```

This is the default namespace and can be overwritten for a particular session using another interface, allowing more fine-grained control.

5.2.4 fs

`[/csrv/local/fs]` location gives the interface to the host filesystem. This interface allows the applications to access the local filesystem if needed.

5.2.5 net

`[/csrv/local/net]` location gives the interface to the host network. This interface is in the Plan 9 style filesystem which can be used create new network connections[net]. The network connections created using this interface will use the local networking infrastructure. This is quite useful when the application wants to rely on the local networking interfaces instead of the abstractions provided by XCPU3.

5.2.6 status

`[/csrv/local/status]` is the read-only file providing information about the available resources at local node and remote nodes. The information includes the number of nodes for each combination of an operating system and an architecture. These statistics include all the remote nodes which are the descendant of this node. These descendants are traversed by avoiding the *parent* links.

Every node reads the `[/csrv/<child-node-name>/local/status]` file of all his children nodes and then aggregate this information to produce its own status file. As all the children are XCPU3 nodes themselves, when the parent reads their status file, they repeat the same action of reading and aggregating the status files of their children. This way the process is recursively repeated till it hits the leaf nodes. These nodes terminate the recursion by returning the information about themselves. Following is an example of the content of status file at a compute node of the Blue Gene which is a leaf node.

```
$ cat /csrv/local/status
1 Plan9 power
$
```

This information is aggregated back till the root node which then returns to the user. Following is the example of an aggregated information on the Blue Gene computer.

```
$ cat /csrv/local/status
1 Linux power
65 Plan9 power
$
```

The above description shows how nodes help each other to perform the operation that needs the global information. Unfortunately, these global operations do involve lot of communication making them comparatively expensive. So, hereby we inform the users about being careful in using these global operations.

We have plans to reduce the cost of such global operations by caching the information of previous global operations. We expect that the global view of the system will not change very frequently, allowing us to cache this information. We have not implemented this yet but we plan to implement such caching in near future.

5.2.7 clone

`[/csrv/local/clone]` is a read-write file. This file is responsible for creating the sessions which can be used to run the applications. Whenever this file is opened, it allocates the session to it. This session can be a new session with the new id or a previously used but free session. The session is identified by the session-id which is a non-negative number and it is represented as a directory with the session-id value as the name. The directories `[/csrv/local/0/]`, `[/csrv/local/1/]` and `[/csrv/local/n/]` are the session directories. Whenever the `[/csrv/local/clone]` file is opened, a session is allocated to it. And all subsequent reads on this file will return the session-id of the session allocated. Any writes on this opened file will be considered as writes on the `[/csrv/local/<session-ID>/ctl]` file. If this file is closed and if there is no other open file within this session directory then this session is released for future use.

The typical way for using the clone semantic is

1. Open clone file in read-write mode.
2. Read the session-id.
3. Use the files within that session directory for starting and managing the computation.
4. Once your computation is over, close all those files within session.
5. Close the clone file to release the session and the resources allocated.

5.2.8 Sessions

`[/csrv/local/0/]`, `[/csrv/local/1/]` and `[/csrv/local/n/]` are the session directories in figure 5.3. New sessions are created as and when needed and the session-id value increases monotonically. Every session directory has its own files with its own structure. Next section will discuss the session interface in more details.

5.3 Session interface

This section explains the filesystem interface of the sessions in the XCPU3. These sessions are important because they are the single unit of execution which can be entirely controlled by the users. The figure 5.3 shows the directory structure of a single session.

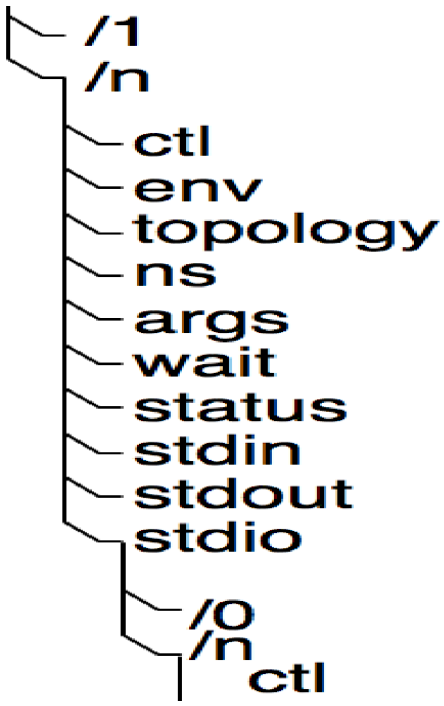

Figure 5.3: Sample filesystem interface for sessions in XCPU3

5.3.1 ctl

`[/csrv/local/<session-ID>/ctl]` is the read-write file which provides the control over the session. Just like the clone file, a read operation on this file will return the session-id. Write operations on this file are treated as the management commands for this session. These commands can be used to do the reservations of the remote resources, requesting the execution or the termination of an ongoing execution.

5.3.1.1 Reservation request

The reservation request is one of the most complicated and essential operations in XCPU3, so we are giving detailed information bellow about how it works.

```
$ echo "res 4" > /csrv/local/<session-id>/ctl
$
```

In above request, *res* is the keyword which is followed by the number of resources needed. Users can also append the type of an OS and an architecture as the constraints if needed, but we are dropping these advanced options to simplify the explanation.

Satisfying the above request involves the following steps.

1. Checking the availability of the remote nodes which can be used for the reservations. As we have described in the section of the remote node interface, these remote nodes are mounted at the `[/csrv/]` location. The current implementation selects the children nodes in the round robin fashion. But we do have future plans for making more informed scheduling by accounting for the load on the remote nodes.

   Depending on the availability of the children nodes and the number of resources requested, the reservation request is divided among the selected children, where every selected node is allotted part of the request.

2. New sessions are created on the selected children nodes by reading their `[/csrv/<remote-node-name>/local/clone]` file. In this situation, these newly created sessions are dispersed among the children nodes and are accessible from the filesystem interface for remote nodes using the path `[/csrv/<remote-node-name>/<remote-session-id>/]`. These resources are not easy to use in the current form as one must have the knowledge of all the remote nodes and corresponding session-id for accessing these resources. We simplify this access by binding these remote sessions as sub-sessions in the local session directory. In other words we bind the `[/csrv/<remote-node-name>/<remote-session-id>/]` to the `[/csrv/local/<session-id>/<sub-session-id>/]`. The sub-session-id's used are monotonically increasing integer numbers starting from the zero. This

way, all remote resources allocated as part of the reservation request for this session will end up as the directories within the current session directory. This arrangement simplifies the interface for accessing all the resources associated with one reservation request.

3. The next step is to inform these newly created sub-sessions about their allotted part of the reservation. This is done by writing the reservation request into their `[/csrv/local/<session-id>/<sub-session-id>/ctl]` file. As the sub-session is an independent session entity, it recursively handles the reservation request by spreading it among it's children. This recursion terminates when a node receives the request for reserving zero nodes. A request for reserving zero nodes is interpreted as the request for local execution. Once the reservation request hits this terminating condition, the node reports success to the parent node which in turn reports back success if all its children nodes also report success. This reporting leads back to the node which started the reservation request. Once this report reaches back this node, the write operation requesting the reservation is completed with success.

   The current design treats all the errors as fetal. When any read/write operation fails at any node, it will stop processing, release any resources that are already alloted and report the error back to the parent. Once such error reaches the parent, it will follow the similar suit and release all resources it has allocated for this request. This also includes releasing the sub-sessions created on other children nodes. The error is reported to its parent when all resources are released. This way error propagates back till the user/application as error in the write request.

You can observe that above small method creates a tree of sub-sessions where each sub-session maps to the session on the child node and not necessarily a session where execution will happen. The leaf sessions are responsible for performing the actual execution. The number of children that any non-leaf node can have is limited by number of direct neighbors it is having. This is because every node can create only one sub-session on remote nodes, hence limiting the number of sub-sessions to number of neighboring nodes. The side-effect of this design is that the structure of

sub-session's tree is highly dependent on how nodes are connected with each other at underlying cluster topology.

Applications which need to control all the computations individually need to know the topology for locating the leaf sessions. This is undesirable as we do not want to expose the applications to underlying network topology.

We could have easily created a flat hierarchy of sub-sessions where each sub-session represents a leaf session where the actual execution will happen. This design ensures that sessions responsible for the actual computation are at the same level, hence can be easily accessed. But we believe that this representation of sessions will not scale well for very large numbers of nodes. The problem in this representation is that it will lead to a session directory with a large number of sub-session directories making it slow for traversing and other directory level operations. The work done in the tree spawn algorithm of XCPU shows that hierarchical trees scale better than a flat hierarchy. Also, the flat hierarchy with a single root design puts all the responsibility of workload distribution on the single root node. Such uneven distribution of the load makes this approach non-scalable.

On the other hand, the hierarchical tree of the sub-sessions adds more layers in-between leading to the better distribution of the leaf sessions. Each non-leaf session is responsible for a relatively small number of children nodes. Another advantage with the hierarchical tree design is that each non-leaf session helps in the workload distribution. Even though the hierarchical tree design complicates the process of locating the leaf nodes, the benefits provided on the scalability front out-weigh this limitation. We also try to overcome this limitation by providing the `[topology]` file to simplify the problem of locating the leaf nodes of the session.

Each node has to initiate this recursive reservation process on each selected child node by performing a write operation. If these nodes do these reservations sequentially on each child, the total time taken for the entire reservation to complete will be proportional to the total number of nodes involved, and hence not scalable for large reservation requests.

Our implementation does these recursive reservations in parallel for each child node. Separate Inferno kernel threads are used to perform the writes on the children nodes. This parallel reservation is proportional to the height of the hierarchical sub-session tree instead of the number of nodes in the tree. This way, even though we

use multiple recursions for making the reservation, we do them in parallel and save significant amount of time in the process.

A reservation request is the first thing that a user needs to do with the session. Once the reservation is done, the session is ready for execution. But in case the execution request is submitted without the reservation request, the XCPU3 will still honor that request by treating it as a local execution request.

5.3.1.2 Execution request

Execution requests are made by writing them in the `[ctl]` file. Following is a sample execution request.

```
$ echo "exec date" > /csrv/local/<session-id>/ctl
$
```

The above example is the request for the execution of the `date` command. Here *exec* is the keyword which marks this write operation as an execution request. If this session is the leaf session (without any sub-sessions) then this request is executed locally by a specially created process with the proper namespace and environment variables.

If this session has sub-sessions, then this execution request is distributed to all the sub-sessions by writing it to their `[ctl]` file. This way, the execution request is propagated till it reaches the leaf sessions where it is actually executed. All non-leaf sessions act as the workload distribution points leading to a scalable distribution. Figure 5.4 shows how the write operations on the `[ctl]` are distributed in the session tree.

Just like reservation requests, execution requests are also blocking. So we use the Inferno kernel threads to handle them in parallel for performance reasons.

5.3.1.3 Execution termination

Any ongoing executions can be terminated by writing the *kill* keyword into the `[ctl]` file. Following is a sample of its usage.

```
$ echo "kill" > /csrv/local/<session-id>/ctl
$
```

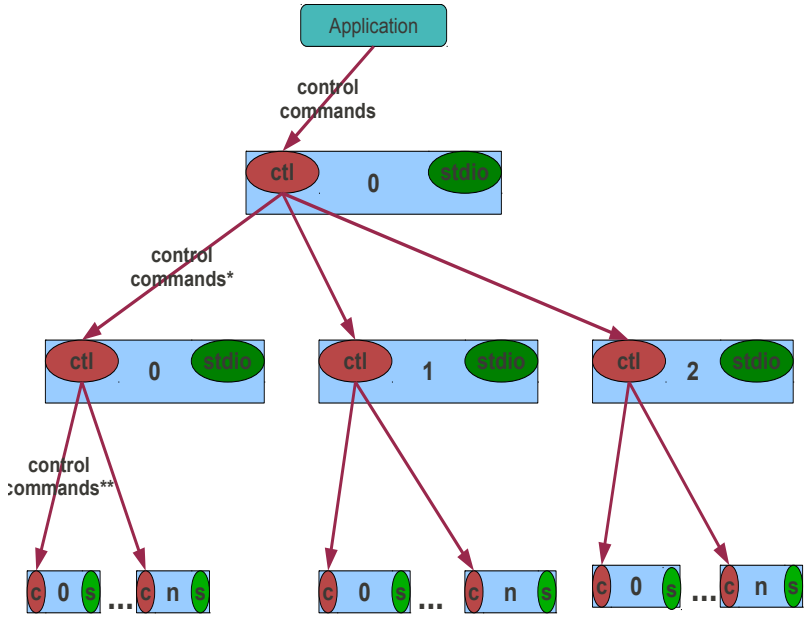

Figure 5.4: Distribution of ctl commands

Just like the *exec* request, this request is recursively propagated till the leaf sessions where it is actually performed by killing the process with the help of the host operating system.

5.3.2 stdio

`[/csrv/local/<session-ID>/stdio]` is the read-write file which works as both standard input and standard output for the application depending on in which mode the file is opened. The historical reason behind merging the standard input and the standard output into one file is to maintain the compatibility with XCPU and XCPU2. This file is responsible for providing the input to the application and returning the output that application has generated.

5.3.2.1 Distributing input

The input can be provided to the application by writing it into the `[stdio]` file. If this session is responsible for the execution, and does not have any sub-sessions then the input written into this file is directly provided to the standard input descriptor of the process executing the requested application.

When the session is the aggregation point of the sub-sessions then writing data into the `[stdio]` leads to the distribution of that data to all the sub-sessions. This distribution is done by writing the received data into the `[stdio]` file of all the sub-sessions. This process for distributing the input data is recursively repeated till the leaf sessions where this data is actually consumed. The diagram 5.5 presents a simple visual example of how this happens.

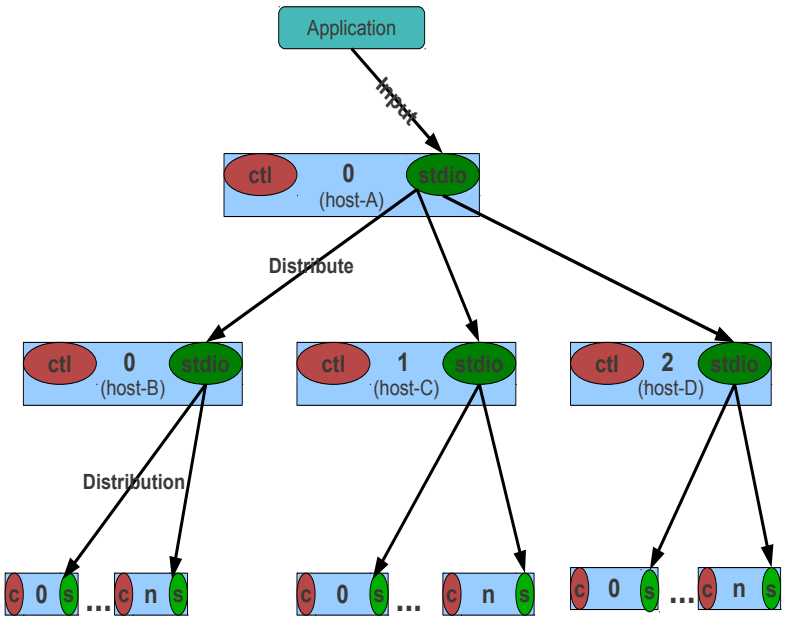

Figure 5.5: Distribution of input data

We use separate Inferno kernel threads for distributing the data to each sub-session. This way, we avoid the sequential handling of each sub-session leading to better performance and scalability. This approach also ensures that if the write

operation on one sub-session is blocked for a long time, then it will not delay the delivery of the data to other sub-sessions.

5.3.2.2 Aggregating output

The output of the application can be accessed by reading it from the `[stdio]` file. If this session is the leaf session without any sub-sessions, then it signifies that this session is not the aggregation session but the execution session and the application is running in this session. In such cases, the output is read out from the standard output descriptor of the process executing the application. This operation is performed with the help of the host operating system.

If it is the aggregation session (i.e. sub-sessions are present), then the read operation is forwarded to the sub-session's `[stdio]` files. This way the read operation progresses till it reaches the leaf sessions. As data is only produced at the leaf sessions, all the read operations are satisfied by them only. Figure 5.6 presents the visual example of how this aggregation works.

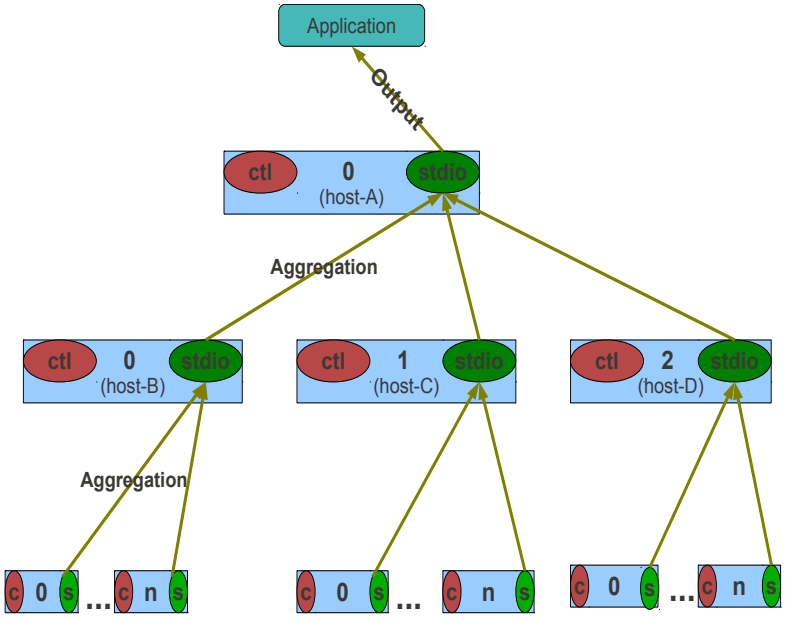

Figure 5.6: Aggregation of input data

If each node may execute the application with different speed and produce the output at a different time, then aggregating these outputs becomes complicated. We use threads to read the data from each child sub-session so that slower executions will not block the faster executions. We use intermediate buffering at each aggregation sessions. Data from sub-sessions is read and stored into these buffers and then any read requests are answered from this buffer.

Output aggregation generates many questions like how does one separate the output of the two leaf sessions? how does one find out which leaf session produced which part of the output? We solve this problem in a related project (PUSH[EVH09a]) which is beyond the scope of this document.

5.3.3 wait

`[/csrv/local/<session-ID>/wait]` is read only file which provides a simple way to detect if a requested execution has terminated or not. This is done by opening the `[wait]` file and reading from it. This read operation blocks till the process performing the requested execution terminates. Whenever the read on this file returns, the user can safely assume that the requested execution is complete.

Just like the write operations on the session files, this read operation on the `[wait]` file is an aggregated read over all the children sub-sessions. All the leaf sessions block the read operations on this file within their session till the process created for the execution is terminated. Once that process dies, the read is returned. This way, the `[wait]` file facilitates the easy detection of the termination of the computation.

5.3.4 env

`[/csrv/local/<session-ID>/env]` provides the interface for specifying the environmental variables. This provides the same interface as the `[/csrv/local/env]` but it affects only the current session instead of the entire node. This allows the user to customize each and every session individually. If there are sub-sessions, then any write operation on this file leads to separate and identical write operations on the `[/csrv/local/<session-ID>/<sub-session-ID>/env]` files for all the sub-sessions present in the session. This way, setting up the environment for the session leads to setting up the environment for all the reserved nodes.

5.3.5 ns

`[/csrv/local/<session-ID>/ns]` provides the interface for building the namespace for the session. This provides the same interface as the `[/csrv/local/ns]` but again it affects only the current session instead of the entire node allowing the user to customize the namespace of each and every session individually. Similar to env file, the write operations on the `[/csrv/local/<session-ID>/ns]` file trigger separate write operations on the corresponding `[ns]` files in every sub-session, leading to the replication of the namespace on every reserved node.

5.3.6 status

`[/csrv/local/<session-ID>/status]` is a read only file which reports the status of the each execution. This information is provided for monitoring purposes to the users. Again, this information is generated at the leaf sessions with the help of the host operating system and is reported back to the requesting parents. These parents aggregate all the information and propagate it back to their parent, till it reaches the starting node.

5.3.7 topology

As we have discussed in the ctl file section, the `[/csrv/local/<session-ID>/topology]` file is supposed to provide the information about the exact location of the sessions which are doing the executions. The `[topology]` file does this by providing the relative path from the current session to every leaf sub-session which is responsible for the execution. Any application which needs this information can read and parse the `[topology]` file to get this information.

As no single node has global knowledge, the content of the topology file is generated by recursively merging knowledge of all the nodes participating in reservation.

5.3.8 sub-sessions

Figure 5.3 shows an example of sub-sessions `[0]` and `[n]`. These sub-sessions automatically appear in the filesystem view once the reservation request is successfully submitted to the `[ctl]` file. As we have discussed in the reservation section, these

sub-sessions are nothing but the bindings to the actual sessions running somewhere on the remote child node. These sub-session directory bindings are temporary and will dis-appear as soon as all the files in this session directory, including the `[clone]` file is closed. Any operation on the files in session's directory affects the same file in all the sub-sessions.

The main objective of these aggregation sub-sessions is to achieve better scalability. This is done by breaking the large number of requests into the manageable chunks in hierarchical way.

5.4 Examples

We have been claiming the simplicity of the usability of this infrastructure throughout the document. Now is the time to show how easily this system can be used. The following examples are designed to show the simplicity of the system-interface and not to present all available features. We are giving these examples of using the bash shell in Linux. Most parts of these examples are self explanatory. We will be giving additional explanations wherever needed.

5.4.1 Example of traditional application deployment

In this example, we show how to run the same application on a large number of nodes and to collect back their output. The infrastructure does the workload distribution and output aggregation automatically on behalf of the application.

We use two terminals for deploying this simple job. We will use the first terminal for creating the session, and the other terminal to manage the session and execution.

We assume that the XCPU3 filesystem is mounted on `./mpoint/` directory. This can be done by using FUSE or 9VFS[HI].

```
$ less ./mpoint/csrv/local/clone
0
```

The above shows one of the ways to create a new session. We use the `less` command instead of `cat` so that the clone file will not be closed once the End-Of-File (EOF) is detected. If the clone file is closed before initiating other actions on this newly

created session, then this session will be terminated. We avoid this issue by using the `less` command which does not close the file after detecting the EOF. The contents read from the clone file represent the session-ID. In our example, the session-ID is "0". We will not terminate the `less` command till we are done with execution of the application.

Now we use session 0 for performing actual execution.

```
$ echo "res 4" > ./mpoint/csrv/local/0/ctl
$ echo "exec date" > ./mpoint/csrv/local/0/ctl
$ cat > ./mpoint/csrv/local/0/stdio
Fri May 7 13:53:58 CDT 2010
Fri May 7 13:53:58 CDT 2010
Fri May 7 13:53:58 CDT 2010
Fri May 7 13:53:58 CDT 2010
$
```

The first echo command sends the request for reserving 4 remote resources. The next echo command submits the request for executing the `date` command. And the `cat` command after that returns the aggregated output to the user.

This example shows all the complexities about finding, connecting and using the remote resources is hidden behind the filesystem interface. The reservation request is responsible for locating the resources, and the execution request is responsible for initiating the execution of the `date` command. As the namespace of the user is shared with all the remote sessions by default, the same `date` command from the client namespace is used for all the executions. Reading the stdio file will return the output aggregated from all the sub-sessions involved in the execution.

Once the execution is over, the `less` program in the first terminal can be terminated. This will close the clone file leading to reclamation of the used resources and closing of the session.

This approach can be used in the *trivially parallelizable applications* where the same application is deployed on all the nodes but with different inputs. The output from all the nodes involved is merged back to generate the final result.

This approach can be also used in *high performance computing(HPC) application* deployments where the underlying infrastructure is only responsible for deploying the

same application on all the nodes, and then the application itself co-ordinates the workload distribution by communicating with each other.

5.4.2 Example of dataflow application deployment

As we have discussed in the *introduction* chapter, dataflow applications need flexibility. We will show here, how this flexibility is provided by XCPU3. In this mode of operation, the role of XCPU3 is limited to the reservation. An application is given the capability of doing the workload distribution and aggregation with the help of the filesystem interfaces provided by the XCPU3. Instead of interacting with the aggregation points, these dataflow applications can interact directly with the sessions responsible for the actual execution using the filesystem hierarchy of that session.

In this example, we will try to create a small pipeline of two commands `date | wc`. But we will create this pipeline across multiple nodes. The objective of this example is to show how the underlying complexities are hidden from the application.

```
$ less ./mpoint/csrv/local/clone
0
```

The above command creates the session in the same way we described in the previous example. The following commands will create the desired pipeline.

```
$ echo "res 2" > ./mpoint/csrv/local/0/ctl
$ echo "exec date" > ./mpoint/csrv/local/0/0/ctl
$ echo "exec wc" > ./mpoint/csrv/local/0/1/ctl
$ echo "xsplice 0 1" > ./mpoint/csrv/local/0/ctl
$ cat ./mpoint/csrv/local/0/1/stdio
1 6 29
$
```

Now, we want to create a pipeline of two commands, so we have reserved two remote resources using the first command. Now, instead of the aggregation, the following commands directly traverse the session directory structure one level deeper and request the individual executions on each of the nodes. The first command `[exec date]` is sent to 0'th sub-session and the second command `[exec wc]` is sent to the

1st sub-session. The `[xsplice 0 1]` request tells the parent session to redirect the output of the 0'th session to the input of the 1st session. The **xsplice** command can be seen as a pipe operator of the shell script for redirecting the output of one command to the input of other command. The only difference is that the xsplice operator connects the two sessions instead of connecting two commands.

The above example is equivalent of executing `date | wc` on the shell, but with the difference that both commands are executed on a different remote machines while sharing the same namespace.

We do need few more mechanisms for easily mapping the DAG of dataflow applications onto XCPU3 infrastructure. Most notable mechanisms are the **inflow** and the **outflow** operators. The inflow operator can be seen as the mechanism for distributing the output generated by one compute vertex to the input of multiple compute vertices. The outflow operator aggregates the output of multiple compute nodes and redirects it as input to one compute node.

We do not implement these inflow and outflow operators in the XCPU3 infrastructure directly but we have left this responsibility to userspace applications like the **PUSH shell**[EVH09b]. This shell is designed to take a dataflow application, convert it into the DAG, map it onto the remote resources reserved by the XCPU3 and orchestrate the flow of the data between all the compute vertices (i.e. compute sessions). The XCPU3 is the infrastructure where operators needed for dataflow applications like the inflow and the outflow can be easily implemented.

Chapter 6

Evaluation

This chapter presents the evaluation of the XCPU3 infrastructure from the perspective of job deployment. XCPU3 is an implementation of a new methodology for the workload deployment for a new class of problems. Also, XCPU3 concentrates on quicker deployment of a large number of small jobs while giving clean interfaces and abstractions. Existing infrastructures do not concentrate directly on this issue of deploying a large number of small jobs. As XCPU3 is venturing into this unexplored domain, we do not have any fair ground for comparing the performance of XCPU3.

Another issue stopping us from performing extensive evaluations with other infrastructures is that the Blue Gene infrastructure is not very hospitable to other infrastructures which are primarily aimed at traditional HPC operating systems like Linux. There are efforts going on in the form of the Kittyhawk[AUW08] project for using Linux for providing the traditional environment on the Blue Gene. Once the Kittyhawk infrastructure is up, comparisons with other solutions are possible.

XCPU3 is the part of the project HARE[VHFMM08]. Figure 6.1 shows the global picture and how these components work with each other. This is a layered design where XCPU3 uses the services provided by **CSRV**. The services provided by XCPU3 can either be directly used by an application or can be used by another infrastructure like PUSH for providing targeted interfaces for dataflow applications.

As we are still working on this integration with PUSH, we will limit our performance evaluations to simpler applications. We plan to do more in-depth performance evaluation after the integration process is completed.

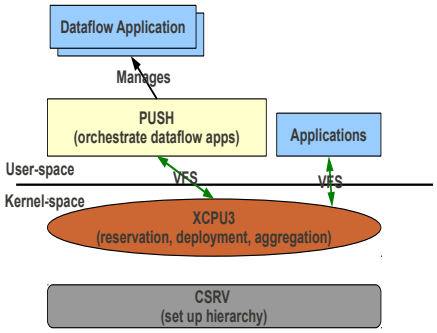

Figure 6.1: The big picture

6.1 Experimental setup

We have performed our evaluations on a Blue Gene setup with 512 nodes. This setup is visually presented in a figure 6.2. We run hosted Inferno on all the compute nodes, IO nodes and the controller node. The user interacts with the XCPU3 instance on the controller node for job submission.

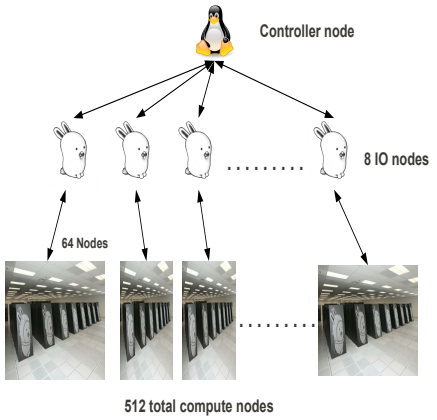

Figure 6.2: Setup for evaluation

6.1.1 Scalability of XCPU3

Our first objective is to show how quickly we can do deployment and execution of a large number of small applications. We have avoided using larger applications as the bigger runtimes of larger applications tend to amortize the overhead in the deployment of the application. We have used the `date` command as the application for deployment. This is a small application and does not need any external inputs and produces small output. Each deployment involves session creation, reservation, execution, output aggregation and termination of the session. We deployed varying numbers of execution of this application on the cluster of 512 nodes. The number of requested executions increased exponentially from 1 to 2048 executions.

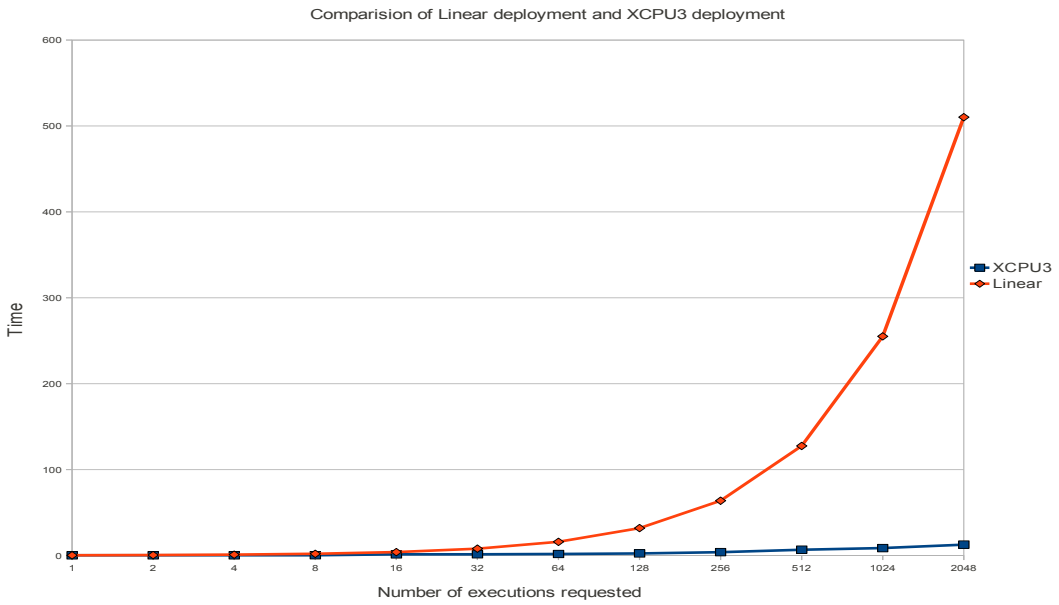

Figure 6.3: Comparison if XCPU3 with sequential deployment

Figure 6.3 gives an initial perspective of how XCPU3 performs relative to sequential performance. This graph plots the total time taken by XCPU3 and the hypothetical time it may take for performing the same amount of work on one machine. This graph shows that the XCPU3 is successfully able to exploit the parallelism for deploying the jobs quickly. The XCPU3 deploys 2048 jobs in 12.66 seconds whereas

sequential execution would take upto 510 seconds. In the graph, the line showing sequential scaling looks exponential, but that is because number of requested executions increase exponentially.

6.1.2 Job deployment without input

This section aims to further analyze the performance of XCPU3. Again we are concentrating on similar deployment. We have recorded the time taken by each of the following stages in the deployment on the XCPU3 infrastructure.

1. Reservation: Create a new session, and request the reservation by writing `res n` into the session `[ctl]` file. Here **n** varies from 1 to 2048 representing the number of executions requested.

2. Execution: Request the execution by writing `exec date` into the session `[ctl]` file.

3. Aggregation: Collect the output generated by all the executions by reading the session `[stdio]` file.

4. Termination: Closing all the files and terminating the session.

5. Housekeeping: Additional time taken before, between and after above steps.

Every deployment starts with the creation of the session followed by the reservation, execution, aggregation and then ending with termination of the session. We have run every deployment three times, and measured the time taken by each step. We have taken the average value over these multiple runs for our analysis. As the Blue gene setup is in controlled environment, we do not expect big variations in multiple runs. Also, we have repeated the entire experiment multiple times and always got similar results. We are discussing the measurements from one such experiment.

Figure 6.4 presents the results of deployment of the date command in the form of graph. This graph presents the breakup time for various stages of the deployment using the XCPU3 infrastructure.

From this graph we can observe that the session termination and the housekeeping overheads are negligible compared to the time taken by reservation, execution and

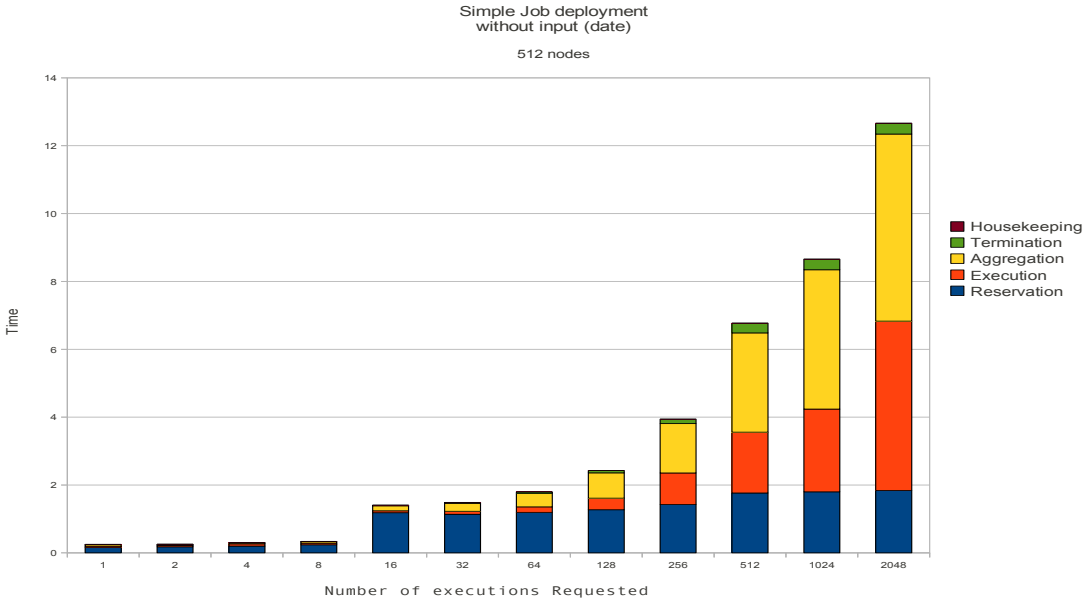

Figure 6.4: Deployment without input

aggregation. So, we can ignore these two overheads in our future evaluations. For jobs of up to 128 deployments, the reservation time dominates everything else. But for larger numbers of deployment execution and aggregation time increases rapidly while reservation time remains relatively constant. This shows that reservation time is not directly dependent on the number of deployments, whereas execution and aggregation time are directly proportional to the number of deployments.

Now, let us try to analyze why reservation time is independent of the number of deployments. The reservation process involves traversing the underlying topology tree of nodes till the reservation requirements are satisfied. All the children on the same level are traversed parallelly at the same time. This way, each level is traversed in the constant time, independent of number of nodes in that level. Another aspect of the reservation mechanism which helps here is that the amount of data written and read from the `[ctl]` file and the amount of data exchanged between nodes for communicating reservation request is fixed in size and independent of the number of deployments requested. With these two properties, the reservation time becomes directly proportional to the depth of the tree and not with the number of nodes.

We can observe the above relation in figure 6.4. The reservation time remains relatively constant for deployment requests from 1 to 8. Then it sharply increases between 8 to 16 and remains almost constant for all the requests between 16 to 2048. This can be attributed to underlying cluster topology. Figure 6.2 shows the presence of the 8 IO nodes in the first level. This enables satisfying the requests which are smaller than 8 executions. For larger requests, one more level needs to be traversed in the topology, introducing delays. The time taken for reservation remains almost constant between 16 and 2048 executions as all these reservation requests essentially traverses the same depth. We can conclude from these observations that *the time taken for the reservation is directly proportional to the depth of the tree.*

Now let us discuss, why the same property is not exhibited by execution time or aggregation time. We have discussed in the implementation chapter that all read and write requests are performed in parallel between all the nodes in the same level. But the amount of data exchanged for aggregation and execution is not constant. This data is directly proportional to the number of nodes involved. With the increase in the number of requested deployments, the amount of data to be exchanged also increases, leading to larger aggregation time. The execution time is also similarly affected as all compute nodes will try to fetch the binary of the executable from the initiating node leading to the copy of the data. These observations lead us to to conclusion that *the time taken for the execution and aggregation is directly proportional to the number of deployments requested.*

6.1.3 Job deployment with input

Our next evaluation involve the deployment of an executable `wc` which needs input. This command counts the number of lines, words and characters in the input file. This is an interesting case for our infrastructure as this deployment involves the distribution of inputs to all the sessions. This introduces a new stage in the deployment process in addition to the 5 stages we described in the above section. This stage will be the **input** stage and involves distributing the input data to all the sessions which are responsible for execution. By default XCPU3 will broadcast the input to all the compute nodes, but we have plans to introduce filters in the PUSH shell which can partition the input given to the compute nodes.

Figure 6.5 presents the results of evaluations involving the distribution of the input.

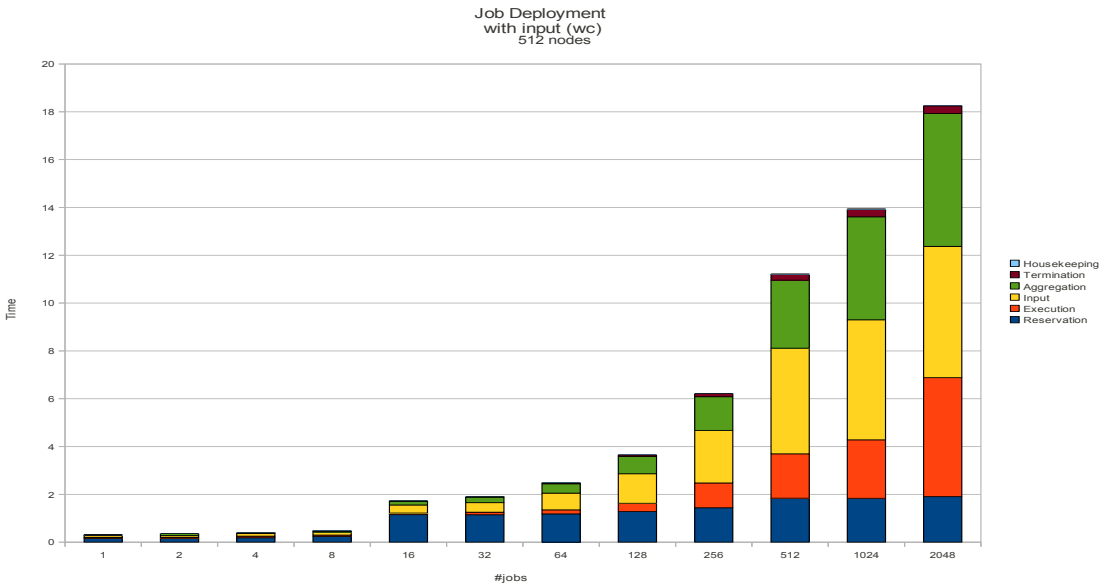

Figure 6.5: Deployment with input

These results enforce our observations that reservation time is directly proportional to the the depth of the tree whereas aggregation and execution time are directly proportional to the number of deployments requested. In addition to these observations, we can also observe that the input time exhibits behavior similar to the execution and aggregation time. This observation can be attributed to the fact that input distribution implementation is similar to the output aggregation implementation. And also the amount of data to be exchanged for input distribution depend on the number of deployments requested as this data needs to reach all sessions responsible for execution. This increases the amount of data to be exchanged with any increase in the number of the deployments requested. We can conclude with this observation that *the input time is directly proportional to the number of deployments requested*

6.2 Dataflow workloads

The evaluations presented in this chapter helps us in understanding how fast XCPU3 can deploy small jobs and aggregate the results produced by them. Now let us relate how these observations can justify our claims about dataflow applications. Typical deployments of the dataflow applications are similar to the above experiments as it involves starting up a large number of small jobs. But the similarity is over at this point. Dataflow deployment does not always involve the input distribution and output aggregation stages. These deployments work by feeding the output of one computation as input for other computation. At the end of the computation, a user needs to read the output of only selected compute sessions which does not need any aggregation. What we can conclude from the above description is that XCPU3 performs really well in the stages like reservation which are important for dataflow deployments. The stages like input distributions are output aggregation where XCPU3 is relatively slow are not needed by dataflow deployments. This makes XCPU3 ideal for rapid deployment of dataflow workloads.

Unfortunately we do not have concrete measurements and evaluations to back our predictions. XCPU3 is one of the piece in the envisioned solution for problem of efficient and easy deployment of large-scale dataflow applications. We are still working on the integration of userspace applications like PUSH with XCPU3 which will further simplify the dataflow deployment. As the entire envisioned solution is out of the scope of this document, we are leaving out the details about it. We are also working on reaching out the high performance computing community for porting real life large scale dataflow applications to this platform for evaluation purpose. We are hopeful that we will have have extensive performance evaluations for real life dataflow applications on this platform in near future. In this document, we have only attempted to provide a strong case for the abilities of XCPU3.

XCPU3 is an infrastructure which provides the needed flexibility, speed and ease of use for dataflow workloads. This chapter demonstrates the speed that can be achieved. It is difficult to measure the properties like *ease of use* and *flexibility* but the examples presented in the filesystem interface chapter should give some insights of all the possibilities opened up by the XCPU3.

Chapter 7

Related work

This chapter tries to put XCPU3 in the context of other work which has been done in this area. We will briefly discuss the various related projects and how they differ from XCPU3.

7.1 Historical solutions

Using multiple computers for getting the job done is not new. With the availability of cheap networking, there were attempts in the research community to create a *distributed operating system* which can be used for workload distribution. This section discusses few of those research projects.

7.1.1 Amoeba

Amoeba [TVR85] was developed as a general purpose distributed operating system which can solve the problem of scarcity of compute resources. It works by creating a *processor pool* using idle workstations and use this pool for performing long running computations. It introduced and used concepts like process migration. It provided tightly integrated mechanisms for remote process execution and control allowing the resources of the entire network to be utilized as a single system.

Amoeba mainly concentrated on turning the collection of workstations into a transparent distributed system. It concentrated more on executing the applications reliably and not on deploying the applications quickly.

7.1.2 Cambridge Distributed Computing System

The Cambridge distributed computing system[NH82] system was aimed to tackle the problem faced during 1980 when the machines were less powerful and more expensive. The direct assignment of the machine to a user was inflexible as the computational requirements of the users change depending on the work he is doing. This system works by having central *processor bank* which does not belong to any user. Any user can remotely use the processors from the processor bank and release them when computation is over. This system also implemented the network filesystem which can be accessed by any processor in the processor bank or workstation.

This system provided the mechanism for organizing the multiple resources, but it was aimed at sharing the resources between users.

7.2 Traditional job deployment solutions

Increase in the capability of the personal computers (pc) and decrease of their cost made the above research systems outdated. As we are reaching the limits of silicon technology, it is becoming increasingly difficult to increase the speeds of these processors. This has lead to the design of multicore and manycore chips. This trend has also promoted research in clusters and grids. In this new incarnation, most of these clusters run the Unix based operating system and the responsibility of extracting parallelism from these clusters is left to the applications. These new incarnations of clusters also used different mechanisms for job deployment. As clusters run the Unix based loosely coupled operating system on each node, the job deployment was based on the services like remote shell. This section discusses a few of these job deployment solutions used in recent clusters.

7.2.1 Secure SHell(SSH)

SSH[ssh] is the security aware extension of remote shell. SSH uses public key cryptography for key exchange and provides a secure channel by encrypting all the traffic. The job deployment is done by the client node logging on the compute node using SSH and using this remote shell to execute the desired application. The applications

and other dependency files are transferred to the remote node either by copying them using *Secure CoPy (SCP)* or fetching them over a network/distributed filesystem.

Most of the existing middleware and workload distribution systems use the above approach. Even though they provide wrappers around it to simplify the user interface, the above are the fundamental working services.

The limitation of using SSH is that the application has to run in the environment of the compute node and not in the environment of the client node. This puts additional burden on the application developer to ensure that the application will work properly in the compute node environment.

Another limitation is that the SSH connections are one-to-one, and this one-to-one model does not scale up for controlling large numbers of nodes.

7.2.1.1 PSSH

Parallel SSH (PSSH)[pss] is an attempt to overcome the limitation of scalability. This tool allows connecting to multiple remote nodes simultaneously. It allows client to control all the remote nodes simultaneously by replicating the same commands to all the remote shells.

This tool definitely improves the scalability but at the loss of flexibility as client node loses the fine grained control on each remote node. Also, as PSSH does not use any kind of hierarchy for scaling, client node is responsible for maintaining all parallel connections.

7.2.2 BProc: Beowulf Distributed Process Space

BProc[CHM+02] is developed as an alternative to the loosely coupled SSH based job deployment. BProc works by tightly coupling the kernels running on cluster nodes and gives better control on remote processes than SSH. BProc is set of kernel modifications, utilities and libraries. These modifications provide mechanisms for starting and managing processes on remote node and process migration by using the OS API. BProc creates a distributed process ID space which allows a front node to locally control all remotely started processes.

BProc is a good step towards getting more control over the processes cluster environment, but it also imposes few restrictions. It needs a modified kernel on all

the nodes. Another restriction is on the namespace. The remotely started process will work in the namespace of the compute node and not in the namespace of the client node. Also, the process migration is restrictive as all the open files except standard input and standard output are lost when a process is migrated. These restrictions lead to the loses of the flexibility which is needed for dynamic workloads.

7.2.3 Multicast Reduction Network (MRNet)

The MRNet[MRN] is designed for efficient multicast and data aggregation. It uses the tree of internal processes between the front-end and the back-end of the tool. The components of this tool communicate using logical channels called streams. The internal processes attach the filters with these streams to efficiently computing the averages, sums and complex operations. These processes internally transfer the data with high bandwidth by using packed binary representation and zero copy data paths. It uses multicast to reduce the cost of delivering the messages.

This project concentrates on optimizing group communication but it is language dependent. The MRNet API is in C++ and one needs to develop the applications using this API. It needs a lot more efforts from the developer side to use this infrastructure in comparison to XCPU3.

7.2.4 Streamline

The idea of using filesystem interface for data-streams is also used by the *pipefs*[dBB08] which is part of the streamline[str] project. The pipefs concentrates on extending the Unix pipeline model for fast I/O in the kernel. The pipefs presents the kernel operations as directories and live data as pipes using a synthetic filesystem interface. The pipefs also implements a splicing mechanism which allows copy free movement of the data. This approach of using filesystem interface and splice semantics is quite similar to XCPU3, but the objectives of both these projects are different. Pipefs concentrates on faster I/O with filesystem interface for the Linux kernel while XCPU3 concentrate on the flexible workload management. We hope that we can use the concepts from pipefs to improve the performance of XCPU3.

7.2.5 Dryad

Dryad[Y+08] is a distributed engine for data-parallel applications which is designed with a primary focus on simplicity of the programming model, reliability, efficiency and scalability. Dryad application combines computational vertices with communication channels to form a dataflow graph. This system explicitly forces the developer to consider the data parallelism of the computation. On the other hand the system deals with hard problems like resource allocation, scheduling and transient or permanent failures. Dryad is flexible in giving the developers fine control over the communication graph as well as subroutines that live on the computational vertices. It also allows the application developers to specify an arbitrary directed acyclic graph to describe the application's communication patterns. These graph vertices can use arbitrary number of inputs and outputs.

In many senses, Dryad addresses the issues faced by dataflow deployments. But there are certain issues with dynamic workload which are not tackled. Dryad assumes that the vertices in the DAG are deterministic and will not change dynamically which is not true for all dataflow deployments. Sometimes the size of computation is affected by the results of a few intermediate computations. The dryad API is C++ specific which limits the usability. It needs additional wrappers to use Dryad from other languages. Dryad also tries to support legacy executables by putting them in a *process wrapper* but this support is quite restrictive about what these legacy executables can do.

Dryad addresses the issues like fault tolerance which are currently ignored by XCPU3, but XCPU3 is more flexible as it can support dynamic workload and changing DAG of dataflow deployments. Also the filesystem interface of the XCPU3 is much simpler and cleaner compared to the class based C++ interface of Dryad.

7.2.6 Condor

Condor[TTL05] is *high-throughput distributed batch computing* system which exploits *opportunistic computing* for better performance. It uses the idle CPU cycles of voluntary workstations solve the large computational problem. Condor works by breaking the large problem into smaller tasks and submitting these smaller tasks to voluntary

workstations. Condor also provides the fault-tolerance by frequent check-pointing and restarting the tasks which failed.

Condor also provide the way to submit scientific workflow application by using the meta-scheduler DAGMan (Directed Acyclic Graph Manager). It allows user to specify the various dependencies within tasks in form of directed acyclic graph and then DAGMan takes the responsibility of scheduling these tasks by maintaining the proper order between them and feeding the results from predecessor to successor.

Condor differs from XCPU3 as it is aimed for voluntary computing and hence concentrate more on fault tolerance. It assumes that voluntary workstations are unreliable and can leave the system anytime. This design principle is quite opposite of XCPU3 which assumes that resources are mostly reliable. This difference in the assumptions about underlying hardware has lead to entirely different systems. Another difference which has cropped in due to differences in underlying asumptions is that the tasks given to the Condor voluntary workstations do not communicate with other workstations directly. They typically communicate with the agent responsible for submitting the job. This lack of the ability to communicate directly with other computational nodes limit the usefulness of Condor for dataflow applications.

7.2.7 Other commercial solutions

Apart from the research and development going on in the academic and open source industry, there has been lot of interest in the workload deployment solution in the commercial domain. The *Oracle Grid Engine*[oge] is a batch-queuing system which can accept, schedule, dispatch and manage the remote executions on clusters. Other solutions involve *PBS Works*[pbs] which is workload and resource management solution and *platform LSF (Load Sharing Facility)*[plab] batch job scheduler which is aimed to help in scheduling jobs on private clouds. Most of these solutions target to simplify the resource management and job submission/scheduling. But these commercial solutions mainly differ from XCPU3 as they do not try to simplify the communication issues within applications. This issue of effective communication between nodes is left to the application developers.

7.3 Conclusion

The above discussions shows that there is still lot of scope available for working on workload management systems for dataflow workloads. There are many existing solutions which tackle either only part of the problem or tackled this problem with different constraints in mind. We have worked on XCPU3 with scalability, flexibility and ease of use as primary objectives and these objectives have lead us to the current design of XCPU3. We hope that XCPU3 infrastructure will open up the doors for more solutions targeting specific use-cases.

Chapter 8

Conclusion

This chapter concludes the XCPU3 work that we have presented in this document. We will discuss our accomplishments and the limitations in this chapter.

8.1 Accomplishments

We have created the XCPU3 infrastructure as the workload deployment solution to tackle problems faced by dataflow deployments which were mostly ignored in previous solutions. We will discuss a few of our accomplishments in this section.

1. **Generic**: Even though we used Blue Gene as our primary target We do not use any feature specific to the Blue Gene. Also, by using Inferno, we have ensured that the XCPU3 will work on most of the traditional operating systems. These features make XCPU3 infrastructure generic so that it can be used by other clusters and grids also.

2. **Agility**: This infrastructure is quick in deploying large numbers of small jobs. The overhead involved in starting and closing XCPU3 sessions is small. The evaluations chapter provides us with some data about how this infrastructure performs and how well it scales for large numbers of deployments.

3. **Ease of use**: We have used the filesystem interface for providing ease of use without losing the flexibility. This interface allows the user to use any language

and runtime for interacting with the system. All existing tools which work on the files can be easily used with this interface.

4. **Flexible**: The XCPU3 infrastructure achieves flexibility by making each node an independent entity. As each node is independent and the reservation is decentralized, this infrastructure can provide the flexibility of dynamically adjusting the reservations based on the changing requirements and computations. Any node can start a new reservation whenever the computation increases in the size. These nodes can be organized in any arrangement as per user needs. XCPU3 also provides the means for creating an overlay of nodes and sessions by binding them in any organization.

With these accomplishments we believe that the XCPU3 facilitates the easy and quick dataflow deployments without losing the flexibility.

8.2 Limitations

We understand that the XCPU3 is not a silver bullet, but a step towards improving the infrastructure. This section discusses the limitations of the XCPU3 which need more work.

1. **Virtualization layer**: By using Inferno for the implementation, we have added an additional layer which reduces the performance. This decision was made to quickly get the portability across multiple operating systems but at the cost of the performance.

   This virtualization layer can be removed by natively implementing the XCPU3 filesystem on the different operating systems. We have plans for this in the near future and these plans should improve the performance further.

2. **Infrastructure**: XCPU3 is the infrastructure and not the entire solution. Even though it can be used in stand alone mode, it needs support from other services like CSRV and PUSH to facilitate the dataflow deployments in an easy way. We do not consider this a serious limitation as the XCPU3 is designed to solve

only part of the problem. Other services can be easily used in combination with XCPU3 to get the complete solution.

3. **Fault tolerance**: The current XCPU3 implementation assumes that faults will be infrequent and it expects the user to restart the job in case of a fault or problem. This approach is not scalable for grid-like installations. Failure of any node of link will lead to discarding the work done by all other nodes and restarting everything again.

8.3 Future work

We have the following future plans about overcoming the existing limitations.

We plan to implement the XCPU3 filesystem natively on traditional operating systems. This will improve the performance and allow different nodes with different operating systems to directly use each other's resources using the XCPU3 interfaces.

We also have plans for providing better fault tolerance by remembering the state of each compute session and selectively restarting the sessions which had failed. This mechanism can save a lot of computations where failures are frequent.

We also envision the use of XCPU3 in voluntary and collaborative computing where each participating user installs the XCPU3 filesystem and then it can either provide its resources to the community or use the community's resources using these interfaces.

8.4 Conclusion

We conclude this discussion with stating that the XCPU3 opens up a new way for dataflow workload deployment. This infrastructure simplifies quick deployment of large numbers of small applications while maintaining flexibility. It hides all the networking complexities behind the simple filesystem interface, providing ease of use. This is one more step towards enabling commercial dataflow applications on cluster-like setups. We hope that this work will speed up the acceptance of HPC in business domains.

References

[app] Google app engine. 3

[AUW08] Jonathan Appavoo, Volkmar Uhlig, and Amos Waterland. Project kittyhawk: building a global-scale computer: Blue gene/p as a generic computing platform. *SIGOPS Oper. Syst. Rev.*, 42(1):77–84, January 2008. 49

[azu] Microsoft azure services platform. 3

[Bie06] Eric Biederman. Multiple instances of the global linux namespaces. pages 101–112, 2006. 12

[CHM+02] Sung-Eun Choi, E.A. Hendriks, R.G. Minnich, M.J. Sottile, and A.J. Marks. Life with ed: a case study of a linux bios/bproc cluster. pages 33–39, 2002. 59

[dBB08] Willem de Bruijn and Herbert Bos. Pipesfs: fast linux i/o in the unix tradition. *SIGOPS Oper. Syst. Rev.*, 42(5):55–63, 2008. 60

[ec2] Amazon elastic compute cloud. 3

[EVH09a] N. Evans and E. Van Hensbergen. Push: a DISC shell. 2009. 43

[EVH09b] Noah Evans and Eric Van Hensbergen. Push, a disc shell. In *In the Proceedings of the 2009 Principles of Distributed Computing Conference*, 2009. 48

[FUS] Fuse: Filesystem in userspace. 22

[HI] E. Van Hensbergen and L. Ionkov. The v9fs project. http://v9fs.sourceforge.net. 45

[HM05] E. Van Hensbergen and R. Minnich. Grave robbers from outer space using 9p2000 under linux. In *In Freenix Annual Conference*, pages 83–94, 2005. 22

[IH09] L. Ionkov and E. Van Hensbergen. Xcpu2: Distributed seemless desktop extension. 2009. 12

[IMM06] Latchesar Ionkov, Ron Minnich, and Andrey Mirtchovski. The xcpu cluster management framework. In *First International Workshop on Plan9*, 2006. 10

[MM06] R. Minnich and A. Mirtchovski. Xcpu: a new, 9p-based, process management system for clusters and grids. *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10, 2006. 10

[MRN] Multicast reduction network (mrnet). 60

[net] Ip(3) man page in plan 9. 32

[NH82] R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System*. Addison-Wesley Publishers Limited, London, 1982. 58

[oge] Oracle grid engine. 62

[P+95] R. Pike et al. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995. 5

[pbs] Pbs works. 62

[plaa] Cpu man page. *Plan 9 Programmer's Manual*. 9

[plab] Platform lsf(load sharing facility). 62

[PPT+93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. *SIGOPS Oper. Syst. Rev.*, 27(2):72–76, 1993. 8, 9

[pss] Pssh: Parallel ssh. 59

[ssh] Rfc 4251: Ssh. 58

[str] The streamline. 60

[Tea08] IBM Blue Gene Team. Overview of the IBM BlueGene/P project. 2008. 1, 3

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005. 61

[TVR85] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed operating systems. *ACM Comput. Surv.*, 17(4):419–470, 1985. 57

[VHE07] Mckie Jim Minnich Ron Van Hensbergen Eric, Forsyth Charles. Petascale plan 9 on blue gene. *USENIX*, 2007. 5, 8

[VHFMM08] E. Van Hensbergen, C. Forsyth, J. McKie, and R. Minnich. Holistic aggregate resource environment. *SIGOPS Oper. Syst. Rev.*, 42(1):85–91, 2008. 49

[Vit] Vitanuova. Inferno operating system. 1, 21

[Y+08] Y. Yu et al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating System Design and Implementation (OSDI), San Diego, CA, December*, pages 8–10, 2008. 61