12-2007

A MIDDLE-WARE LEVEL CLIENT CACHE FOR A HIGH PERFORMANCE COMPUTING I/O SIMULATOR

Michael Bassily
Clemson University, mbassil@clemson.edu

A MIDDLE-WARE LEVEL CLIENT CACHE FOR A HIGH PERFORMANCE COMPUTING I/O SIMULATOR

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Michael Bassily
December 2007

Accepted by:
Dr. Walter Ligon, Committee Chair
Dr. Tarek Taha
Dr. Samuel Sander

ABSTRACT

This thesis describes the design and run time analysis of the system level middle-ware cache for Hecios. Hecios is a high performance cluster I/O simulator. With Hecios, we provide a simulation environment that accurately captures the performance characteristics of all the components in a cluster-wide parallel file system. Hecios was specifically modeled after PVFS2. It was designed to be extensible and to easily allow for various component modules to be easily replaced by those that model other system types. Built around the OMNeT++ simulation package, Hecios' inner-cluster communication module, is easily adaptable to any TCP/IP based protocol and all standard network interface cards, switches, hubs, and routers. We will examine the system cache component and describe a methodology for implementing other coherence and replacement techniques within Hecios. Similar to other cache simulation tools, we allow the size of the system cache to be varied independently of the replacement policy and caching technique used.

ACKNOWLEDGEMENTS

I would like to thank Dr. Ligon for his support and guidance during my thesis development and my thesis work. I would also like to thank Dr. Taha and Dr. Sander for being on my committee. I also thank Brad for his guidance during my thesis writing and thesis work development, and Beshoy, George, Maggie and everyone else who provided me any feedback during my thesis proofreading and editing, and all my friends who have shown me nothing but support.

DEDICATION

I would like to dedicate this thesis to my parents, Fahim and Evon Bassily, for all the support, encouragement, and words of advice they've given me.

TABLE OF CONTENTS

Table of Contents (Continued)

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Cluster Computing

Due to the physical limitations on modern processor design techniques, and the low cost of COTS (commodity off the shelf) machines, parallel cluster computing is quickly becoming one of the most interesting areas of research around the globe. Parallel computing has become a prominent mechanism for research areas such as thermodynamics, heat transfer, weather predictions as seen in hurricane trajectory predictions, and even the area of computing for CPU trace layouts. This increasing popularity of cluster computing has lead to the development of many algorithms, protocols, and techniques that not only make it easier to program parallel code, but also speed up execution through methods such as using the physical network layout of the cluster as an outline for task distribution.

In cluster computing, there are two common and accepted memory architectures, message passing and shared memory. While on an architectural level the move towards multi-core and multi-processor systems might seem to indicate that the market could be leaning towards a shared memory approach, the leading parallel computation standard is in fact the Message Passing Interface (MPI). As popular as MPI is, its just a well defined standard by the parallel computation community[28]. The implementations of MPI, such as MPICH and LAMMPI, are the actual packages used by academic and industry programers. The reason MPI has become extensively used is its feature set. The extensive list of MPI library functions includes everything from data types to aid in the communication between processes to MPI I/O, MPI calls that take into account the specifics of disk access in a clustering environment.

Value added re-sellers and customized system vendors such as Atipa, Sun micro-systems and Cray, have taken the MPI I/O implementations and tuned

them specifically for optimum performance on their hardware. While maintaining identical or very similar function structures they allow generic MPI code to run in a manner that optimizes the resources of their highly customized systems. Similarly, Myrinet and many other network interface vendors have provided modules and drivers as well as MPI implementations that reduce overall latency when used in conjunction with their hardware. However, one area that has been untapped until recently, one of arguably the slowest bottlenecks of today's modern computational systems, is that of disk I/O bandwidth. Historically, and with the emergence of solid state hard drives, mass storage media has been magnitudes slower then even the slowest system memory or cache. Even with today's high performance disk drive arrays that can sustain transfer rates close to a couple of hundred megabytes a second, they can not compare to the multi-gigabit per second throughput of low latency DDR RAM. Parallel file system (PFS) development has attempted to solve this problem by grouping together the mostly unused compute node hard drives in an attempt to achieve greater performance.

Another factor that has lead to the development of parallel file systems has been the increasing size of parallel task output. Scientific applications such as DNA mapping, mechanical system modeling, finite element analysis, and heat transfer simulations might sometimes require upwards of multiple gigabytes of data. Even a simple elastostatic model with only 10,000 vertices could require 3.6 GB [16] of storage space. It becomes evident that running multiple instances of a simulation could yield outputs that would stress most modern day hard disk storage drives. However, with the small sized hard drives that are found on today's compute nodes, a collection of 128 nodes each with small 80 gig hard drives, using only a portion of that available hard drive space would yield a high performance multi-terabyte storage solution.

Parallel File Systems

Purpose

The emergence of COTS machines as workhorses in the clustering community has lead to the development of some very interesting methods of reducing the overall impact of slow hard disks, and even slower external networks. Recognized as early as 1989 [9], it became obvious that parallel I/O techniques needed to be developed. While a number of parallel file systems have emerged with various performance characteristics, they have utilized the same underlying fundamental idea of file partitioning to achieve increased I/O throughput. The basic idea of parallel file systems is shown in Figure 1.1. In this illustration, a single file is physically partitioned across 5 I/O nodes, but still seen as one logical file located on one physical drive through the PFS software present on the I/O and client nodes. This partitioning is usually performed using 'striping'. In striping, the file is divided into a sequence of fixed-size blocks that are distributed to the disks round-robin. Striping is the same technique used in Redundant Array of Independent Disks (RAID). There are many network links to the I/O nodes however, unlike RAID which has a single network link connecting the typical network. As depicted in Figure 1.2, the increased bandwidth provides the system with a scalability property that allows many more client nodes to simultaneously read and write data at a much faster throughput then a standard RAID system. However, systems that yield the greatest performance often combine these two techniques and use RAID arrays at each I/O node.

PVFS2

A Clemson University research project, the Parallel Virtual File System (PVFS), is a parallel file system for cluster computers. PVFS1 was designed to foster research and experimentation, while PVFS2 was designed to be used primarily in production and easily integrate into a cluster environment. PVFS2's

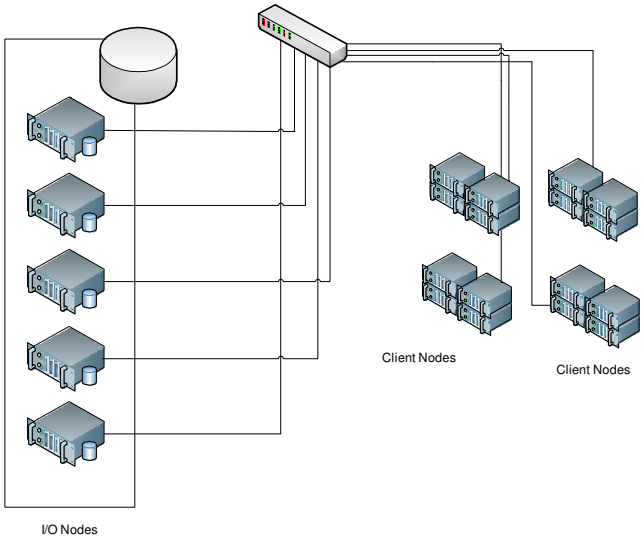

Figure 1.1 Common parallel file system structure

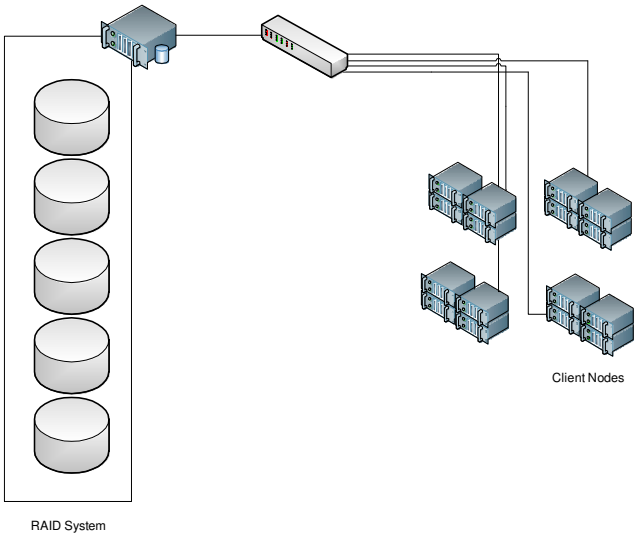

Figure 1.2 Common RAID system

layered software stack approach allows explicit separation of system components, enabling the same upper levels of the software stack to run on different lower level architecture specific kernel modules. It provides coherent, but not sequentially consistent, semantics in order to avoid locks or other system synchronizations [29]. Because of the nature and the design of PVFS2's stateless system, adding client-side caching would require significant internal modifications to ensure cache coherence [29].

PVFS2 achieves the performance goals most parallel file systems set to achieve with bandwidths of 700 Mbytes/sec with Myrinet and 225 Mbytes/sec with fast ethernet [5]. This has lead to the installation of PVFS2 at numerous national laboratories. It is also robust and scalable enough to handle cluster sizes up to hundreds of nodes. PVFS2 operates well with contiguous and non-contiguous accesses, providing fast operation completion for re-size operations that more common in clustering environments while also providing scalable meta-data access by allowing all servers to provide meta-data storage [29].

Motivation

In research environments, there exists three major research vehicles that facilitate theory exploration. Ranging from least to greatest in terms of complexity are analytical models, system simulations, and system prototypes. While analytical models are easy to work with, it becomes harder to accurately capture all system specific details as the project grows. Prototypes provide a real world implementation, but often the development can last at least weeks if not months or years depending on the complexity of the system. A compromise between these two research methodologies is a system simulation. Sometimes seen as an intermediate level between hypothesis formation and implementation, a flexible simulation can provide as little or as much detail about each system component as needed to test new PFS techniques.

Modern parallel file systems, as with any large scale project, have multiple contributors/users. As most of these file systems are developed in a dynamic

research environment, it has become a challenge to fully implement a new feature without first exhaustively considering the particular nuances of the system. Similar in difficulty, is starting a new parallel file system from scratch that examines and implements new research ideas and methodologies. In order to begin the development process of a new parallel system, numerous design decisions must be made, some of which will have most likely been explored by other projects. The development of Hecios, our High end computing I/O simulator, was not only fueled by this need, but also by other factors that must be considered when developing a parallel file system such as:

- Complexity - As parallel file systems have become increasingly complex, the implementation of a trivial traditional file system feature requires significantly more time for testing and implementation in a parallel file system.
- Security - Additionally, file systems must take into consideration system security to preserve the multi-user environment's data integrity.
- Scalability - File systems must also scale to large enough sizes to accommodate prevailing computational needs.
- Semantics - Data in system caches must be kept consistent through cache coherence.

Hecios and Modules

The modular design of Hecios allows for the different system simulation components to be designed as independent modules that attached together. Hecios' modules are conglomerated into a multilayer hierarchy shown in Figure 1.3. The server side resembles the standard PVFS2 server and most modern parallel file systems, taking into account the following components:

- Disk/Disk Cache - The main server side storage of a parallel file system, includes physical storage elements such as a hard drive or RAID array.

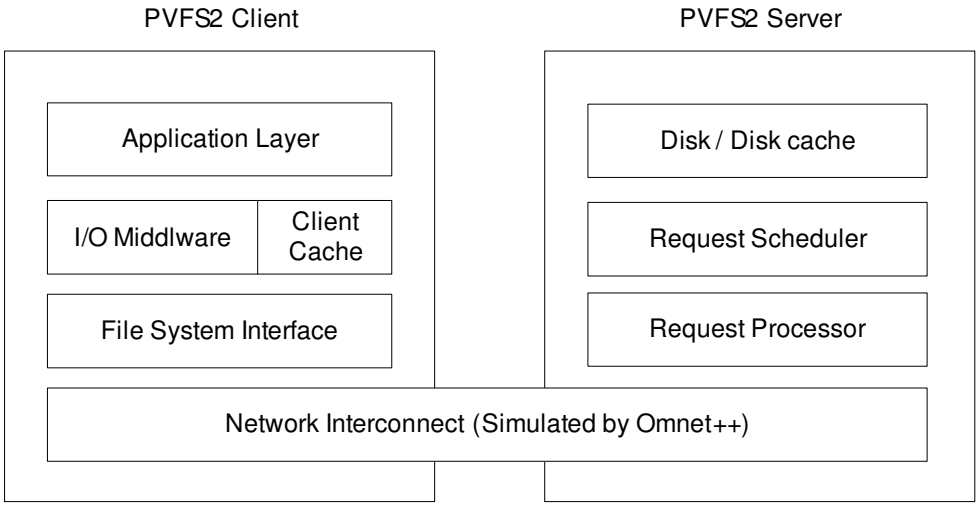

Figure 1.3 Hecios' architectural layers

- Request Scheduler - This component orders incoming requests when consistency semantics dictate serialization.
- Request Processor - Includes functionalities of the flow component which facilitates pipelining of data between network and disk, and the progress engine that manages the state to state transition of the components internal state machines.
- Network Interconnect - The network component includes the network protocol, connection medium and switching device.

On the client side, a similar layering scheme to PVFS2's also exists, however, the client cache component at the I/O middle-ware level is currently not implemented in PVFS2:

- Application Layer - Includes the cluster configuration files and application parser for scheduling I/O events.
- I/O Middle-ware - Includes a message parser to cache appropriate messages and possible coherence mechanisms and replacement techniques.

The Current PVFS2 middle-ware does not implement a client cache, hence the need for our simulation.

- File System Interface - Provides functionality seen in the client-side Flow, and state machine progress engine similar to operations of the request processor and request scheduler on the server side.

- Network Interconnect - This component operates the same as the network interconnect on the server side, linking the clients and servers.

Client Cache Module

In today's commodity computers, caches are found in almost every layer of the system memory hierarchy, ranging from L1 cache to system level memory paging. They have become critical in sustaining overall system throughput. When caches are referred to during a discussion of parallel file systems, they are often recognized for their complexity due to the coherence techniques required for their implementation in a system where concurrent reads and or writes might be occurring . Although PVFS2 does not provide any cache coherence mechanisms, one of the main goals of this simulation project is to examine the effects of adding an I/O middle-ware level cache to a PVFS2 I/O node. As we've previously mentioned, in most scientific applications the relaxed semantics provide accurate program execution. However, the performance gains associated with caching warrants that we take a closer look at its effects within the constraints of our PFS in a controlled simulation environment.

A client cache component could easily be acknowledged by many as one of the essential building blocks in a modern highly efficient parallel file system. By keeping a local copy of the most commonly used files or file portions, a large number of network transactions are eliminated, which, depending on the file access to computation ratio of the application and the coherence mechanism chosen, can lead to significant amounts of speedup. The client data cache component structure discussed in this thesis follows an easily replicable

form to allow for new cache technique implementations. The included cache structure could also be used as a building block for a more in depth replacement policies, other then the provided LRU and FIFO implementations. A middle-ware component data cache is often used in projects where the effects of multiple techniques are studied in order to select a highly effective cluster utilization technique. This may be in the environment of a parallel file system implementation change, a general parallel file system simulation, or a cluster specific analysis. Due to overwhelming complexities in modern production file systems, thorough system simulation is needed in, most cases, in order to determine the best implementation caching techniques. While the goal of keeping or adding to the system's scalability takes high precedence, consideration is given to cache coherence as it greatly increases the implementation complexity.

We determined that a highly scalable and configurable file system simulator would satisfy the community's needs. By accurately simulating all node and interconnect components; hard drives, client level cache, network link: we have created a tool that can be adapted to any of the ever growing assortment of parallel file systems. Careful detail was given in the design and implementation phases of each component to allow for a great level of extensibility and ease of use. *In this thesis we pursue the design and implementation of a cache simulation module that will serve as the cornerstone for studies of client cache organization in parallel file systems.* These studies are to include details of consistency, scalability, and security as the cache is implemented at different levels from the operating system to the middle-ware.

Simulation Package

Many parallel system simulation tools are built from the ground up with an in-house simulation kernel. We have chosen to instead build our tool using the popular OMNeT++ simulation package. Known in the network simulation circles as a viable alternative to commercial simulation packages, OMNeT++

provides the necessary simulation kernel as well as basic networking components and protocol implementations. This flexibility allowed Hecios development to progress very quickly, without sacrificing our desire to have a parallel file system simulation model that is easily extensible to other file systems and network structures.

Alternative simulation packages such as OPNET and NS2 were suggested and considered. As OPNET is one of the more popular network simulation tools, it was considered an obvious choice for a simulator of this ambition. However, the non-open source license and very low level network component layout lead us to consider other alternatives. While NS2 seemed to fit the open source community bill, additions were ultimately not as easy to implement as those we designed for OMNeT++. In addition, the OMNeT++ community had already provided an open source detailed disk simulation module that could easily be integrated into OMNeT++ simulations. Using OMNeT++ proved to be a challenge, and an intricate compilation system was developed to deal with our desired module structure. However, this compilation system has greatly simplified module implementations and made the complicated linking process almost transparent.

Another feature that allows OMNeT++ to be so flexible is the ned file structure. OMNeT++ .ned files specify input and output interconnections between modules through the gates mechanism. Each Hecios module is associated with a .ned file that specifies how it connects to other system components and the type of connection. Connections can be physical connections such as between the network transport and disk layers, or logical connections that make it easier for component separation, such as with the connection between the application and I/O middle-ware components. Communications passed along the connections are in the form of custom OMNeT++ messages that can be similar in form to MPI I/O calls or of the lower level PVFS, OS, or BMI calls. Since we will be discussing the client cache module which handles only MPI messages received from the application, only MPI I/O calls will be

examined.

A large portion of the configuration layer is implemented through the parameter settings found in the omnetpp.ini file. This file is included with all OMNeT++ projects and contains network topology and configuration information. Additional fields were added to the omnetpp.ini file to constrain most configuration parameters to a single location. The eviction policy used for the data cache is also selected through this file.

Thesis Overview

In Chapter 2, we start by building a solid background of MPI I/O and PVFS and continue with an evaluation of middle-ware and cache implementations in parallel file systems. We then extensively describe the message structure for our client cache and how it handles MPI I/O messages. We will then note overall performance. While we don't directly compare performance of our system, we do evaluate the run time overhead associated with the particular cache mechanism implemented in our module. We conclude by offering suggestions to improving the cache module either by whole or partial replacement of the insertion and replacement policies along with possible coherence implementations.

CHAPTER 2
BACKGROUND AND RELATED WORK

In this chapter we go more in depth about MPI and its use in parallel computing, as well as present some background knowledge about common caching techniques. We then explore the similar works that we feel provides the background for the creation of our PFS simulator. First, we present an overview of the related project, provide it's negative and positive aspects, we then provide a comparative analysis of the project to ours.

MPI-IO

MPI is the message passing system installed on most clusters. This add-on to C and C++ allows for data communication and access in a highly structured manner. Through the use of MPI library calls, programmers can easily transfer data between tasks without having to resort to low level TCP/IP calls. MPI also provides the tasks with a structured naming system that affords easy task recognition. To facilitate optimal process distribution, MPI includes a mechanism for describing the network layout through the use of communicators and topologies. MPI communicators define a group or subgroup of allocated nodes. MPI_COMM_WORLD is a communicator provided by the MPI implementation that includes all the nodes allocated to the running program; for creation of new topologies and associated communicators a call to the MPI_Cart_Create function is made.

While the preceding MPI features make MPI a very powerful library, we are more interested in the later part of the MPI specification, MPI2. MPI2 or MPI-IO provides function calls that abstract the underlying file system structure. For example, the MPI_File_Open operation provides a file handler given a specific file name. Through a series of internal system calls, MPI is able to convert the system specific name to a universal handler [14]. Similarly, other

MPI_File_ operations allow for a mostly architecture free file access mechanism allowing for easily portable user level code [13]. Since HECIOS is a trace driven simulation, the MPI_File_ function calls traverse the simulation stack in a similar fashion to traversing a parallel file system such as PVFS2. Each MPI_File function call is associated with an Omnet++ message type that is specific to the layer it is currently on, the next layer's message is constructed once execution has completed on the current layer.

PVFS2

Parallel file systems have been developed and will continue to be developed at research laboratories and universities in order to gain greater understanding of parallel I/O and to develop parallel systems that suite computational needs. Here at Clemson, the PVFS project was started as a research endeavor to understand the intricacies of parallel file system development. The next iteration of the file system, PVFS2, emerged soon after as a production file system meant specifically for parallel computational science I/O [29]. The main goals of PVFS2 were to be efficient and scalable to a large number of clients and I/O nodes, yet provide this functionality in a modular design. The software stack of Hecios seen in the previous chapter mimics the standard parallel file system stack seen in PVFS2 (Figure 2.1). It also has built in hooks to allow file system access through MPI I/O.

As PVFS2 was developed as a high performance file system, it provides coherency on a byte level, due to the fact that sequential consistency is expensive to implement because of the required atomic locks. The file system uses a mechanism for caching meta-data to reduce traffic to the meta-data servers and increase response time. As PVFS2 serializes all writes occurring to the same area of the file at the I/O node level, write-through cache implementation at the client node level would be the easiest technique for adding a coherent cache.

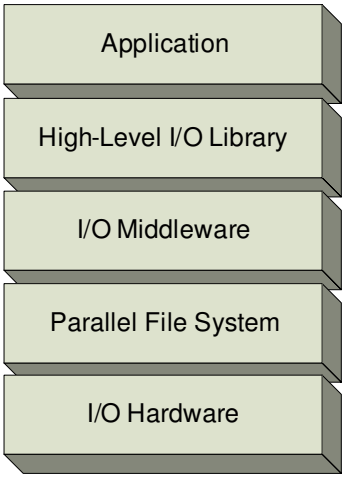

Figure 2.1 Layers of typical parallel file system

Cache Techniques

Caches have always been, and will continue to be, an important part of parallel file systems and computing systems. This is particularly true in environments where accesses to the working data set results in completion delays. Intermediate level caches feed data to the requesting source at a much faster rate then their larger, but slower, hierarchial counterparts. Often times moving from one level of memory to the next results in at least a magnitude of access time and throughput lose. We implement a highly extensible and scalable client level cache module in an attempt to accurately capture the performance characteristics of caching at this level of the PFS stack.

Often, just the mere fact that a cache is present is not conclusive that it provides a performance advantage. Careful consideration must be taken when choosing a parallel caching algorithm. The cache insertion, replacement policies, and size of the cache must all be taken into consideration when evaluating its effectiveness. Pre-fetching and caching have previously been examined in a research environment and found to reduce the overall system wide I/O requests [19]. Also, as we are dealing with a parallel environment, cache coherence must be considered for instances of multiple write accesses. PVFS2 handles this by only allowing one outstanding write transaction to the same area of the file, we replicate this feature in our simulator. However, since PVFS2 only caches meta-data, our implementation could lead to a future PFVS2 feature where all access types are cached.

While parallel simulators exist in the community, we are attempting to provide a definitive simulator that can be as characteristically correct as the user chooses. Many projects have chosen to neglect this level of simulation depth and system extensibility. We examine these simulators and note on the contributions of each work and how we have chosen to expand upon it in our implementation. However, we first look at file system implementations where caching has been put into practice, examining the positives and negatives of each approach.

Caching in NFS

The idea of using caches to speedup file system performance has for a some time been implemented in a system most of us use everyday, NFS. Most notably, the effects of caching in NFS are often seen when modifying a file located on an NFS partition at one location and viewing those same modification at another. These changes usually take a couple of seconds at least to show up at the other location, and this is most likely a best case scenario for NFS. However, the file at the moments of editing experiences no lag, and it almost seems as though the file is being edited locally. This exact same scenario can appear to happen in a similar manner, although without direct user interaction, to a parallel file system. We now look at NFS's caching mechanism in order to understand possible performance gains and how a cache can successfully be implemented in a heavy usage environment.

The caching technique described above in NFS complicates the problem of cache consistency. In earlier versions of NFS, close-to-open cache consistency was implemented in order to reduce the amount of network transactions needed during file access [12]. However, this method was dropped in favor of weak cache consistency in version 3. Weak cache consistency also became too bothersome and it was decided that in most cases, data locks would be the answer. There is still as small amount of caching going on in the most current version of NFS. For example, when doing an LS, the directory and file meta-data could be cached to avoid the network delay involved with doing an operation that should be instantaneous to the user. In version 4 a callback mechanism was introduced to allow clients to modify their own cache and write back to the server only when the server needs to know the caches status.

Applying the NFS strategy to parallel file systems would prove too complex and might not reduce network transactions due to the per file nature of the algorithm. Applying the same policies to subsets of files found in PFS would require a callback transaction to all file portions in order to accurately account for data continuity. In our client level cache simulation model, we write back

to the I/O node as soon as a write occurs, while caching the written data locally. Our cache also allows us to store only those file portions that were accessed and to combine cache entries when a request is made for overlapping memory addresses. This simple hybrid temporal locality algorithm should provide sufficient for most uses. However, our system is also able to simulate more complex systems.

OceanStore System

The OceanStore system provides caching based on the idea that systems will fail, and a system-wide backup of the system is required at all times. In an attempt to do so, it caches files many times at many different locations decoupling the information from the physical hard drive where it was originally stored [20]. Through it's internal caching techniques, OceanStore provides users with a built in automatic backup mechanism. This caching procedure provides up to date copies of data at locations where it is accessed most frequently, significantly reducing transfers across what could possibly be a slow network connection. One of the downfalls of this technique is the update mechanism. It becomes much harder to do data invalidation and updating when a large number of cached copies are present. To resolve this, a master replica is assigned and that copy is considered the most up to date and is distributed to all cached clients. The OceanStore system maps a tree like structure unto the system nodes, similar to a collective communication operation segmentation, updating the root of each sub-tree and then forcing those roots to broadcast the update to their nodes.

We provide a similar mechanism for cache updates in Hecios since it seems as though the caching technique of Hecios closely resembles that of the OceanStore system. The same caching rule is implemented where the cache is filled with the those files that have recently been accessed, either in a FIFO or an LRU fashion. We consider our master replica to always be the original I/O node, since writes are propagated to the I/O node as soon as the request is received

at the client node level. Contention is handled by the request scheduler that allows only one outstanding write to any one file. Although this request scheduler is not yet implemented, for now, a simple global broadcast when that write request is received invalidates all cached copies of the newly written file. We believe that although this adds network traffic to the system, invalidations have been known to be less taxing on a system then having a system without a cache. However, this factor is truly dependent on the application and the size of the cache and will vary when those factors are changes. The Frangipani file system implements a similar locking mechanism and delivers a highly scalable and performance oriented pfs.

Frangipani File System

Frangipani leverages the Petal file system's performance and capacity [21] while providing users with a simple upgrade mechanism for adding storage capacity [30]. Petal provides fault tolerance consistent backup through its virtual disk snapshot mechanism. This mechanism requires pausing the application running on the system while the snapshot is being taken. Frangipani virtually combines Petal systems into one contiguous file system to create a highly available large storage system with a process of addressing up to 2^{64}TB. It caches the most recently used files to the kernel's buffer pool. This cache is kept coherent through the use of write locks that are divided in a manner that divides the disk structures into segments with each segment containing its own lock. A segment is locked only if one of the clients is in the process of a write. After the write is completed, data is written to underlying Petal system and the lock is either released or downgraded depending upon outstanding system requests.

We could argue that if our caching module was to be implemented in a live system, the most obvious solution would be to follow Frangipani's implementation and use the kernel's buffer pool. This keeps the data highly accessible to other processes running on the system, given the assumption that the sys-

tem is trusted. However, our locking mechanism differs in that we do not have explicit locks, but rather a queueing mechanism. This implementation reduces the complexity of the system, but still delivers similar performance. Even though this system was implemented successfully, sometimes it is much harder to achieve a full implementation because of system complexity, in such cases, a parallel simulator is used.

Parallel Simulators

Although parallel file systems have made it easier to deploy enormous multi peta-byte storage wonders, most systems today, along with various file system additions, would not be installed without the approximate performance predictions given by parallel simulations. The usefulness of having a simulator that can accurately predict system performance is seen in the cluster development process as more system designers become reliant upon these tools. They depend on these simulators to provide them with feedback in determining optimal network hardware, interconnect and by predicting the systems expected performance. We examine the design of parallel simulators to differentiate popular caching mechanisms used and simulation techniques.

PIOSIM

PIOSIM was created at UCLA to provide parallel simulation of MPI-IO programs as well as various bench marking utilities. The simulator explores the effects of various caching techniques including cooperative caching [11]. As with other simulation tools and file systems, PIOSIM simulates performance of MPI-IO written code using a trace file input mechanism. However, one of the more interesting features of this simulator is the number of cache management policies available. At the PFS-SIM level, the component used to simulate the specific parallel file system used are LRU cooperative caching techniques including base, greedy, and central caching algorithms [3]. PIOSIM's cache mimics our future goal and some current features that we already have implemented. The write policies and cache properties also exhibit the same level of

flexibility found in our simulator. However, our network and disk I/O modules have a level of completeness that is not present in PIOSIM. By using OMNeT++ as Hecios' simulation kernel, we are able to leverage the entire OMNeT++ communities contributions, such as an extensive network model library, including a newly contributed Infiniband model, and the community provided disk simulation model.

The COMPASS simulator delivers one of the more flexible and adaptable configuration tools to its users. Using PIOSIM's simulation kernel, the COMPASS simulator provides a more system wide approach to cluster simulations through its ability to simulate whole systems and not just specific techniques. It's an execution driven tool that simulates all system components, similar to Hecios, from the interconnection network to the file system. One of the more interesting features of COMPASS is it has the capability to perform caching at both the I/O and compute node levels [2]. As a whole cluster simulation tool, it has proven that it can predict run times and scalability of Sweep3D and NAS benchmarks. But, it lacks a dedicated and flexible network layer similar to Hecios' implementation. We believe that a full network layer allows for greater flexibility in system architecture use as well as evaluation of different network protocols. The caching techniques implemented in both of these simulators could also be implemented in Hecios with little trouble if the need arises.

Simulation of NCAR's MSS

One of the simulators that was specifically built to test caching techniques was NCAR's MSS simulator. The developers note that the simulator's main purpose was to identify optimal cache sizes for the 2 peta-byte MSS [1]. The Java discrete even simulator was developed around a packaged called JavaSim. The system has numerous software components that mimic the actions of the storage devices (tape drives and disk arrays, the system network as well as client nodes. Similar to our model, system delays were set in either a deter-

ministic or probalistic manner by having either static or randomly calculated delays. This simulation is much different then others seen because the replacement policy runs only every 24 hours, and the cache is extremely large with an initial size of 8 TB. However, observing a large system simulator like this allows us to recognize the features that make a large simulation like this possible, features similar to the expansion and caching capabilities found in Hecios. Not only is Hecios built on a faster subsystem, internal C++ simulation kernel of Omnet++, then Java; it can also easily address 8 TB of cache as well as the MSS simulator. Although we implemented FCFS and LRU replacement policies, we also provide a priority field in the cache entry for more complex caching techniques such as those used by the Patsy simulator.

Patsy and Pegasus File System

The Patsy simulation project was created with the same goal we set out to achieve, creating a highly modularized parallel file system simulator [4]. Similar to our development strategy, the Patsy simulator was modeled after a production file system, the Pegasus file system. It uses a custom simulation kernel that allows the systems policies to be replaced through C++'s inheritance feature. However, there is a fixed block size of 4KB per entry whereas our block sizes can be varied or set to a dynamic size property where the block size is exactly the size of each entry. Although we do not discuss our interconnect system in detail here, we feel that by using the Omnet++ simulation tool as our simulation kernel, we have essentially established that our network simulation technique provides an accurate performance model.

Hecios' Contribution

We feel that the Creation of Hecios was fueled by the existence of numerous previous parallel file system and parallel file system simulation tools. We have shown the functionality of I/O middle-ware components of various file systems and various interpretation of how caching protocols should be simulated.

Our cache module implementation, along with the various other simulation modules we have implemented, provide us with the ability of achieving the same functionality as other file system simulators, while giving us the ability to accurately predict performance of production file systems.

CHAPTER 3
SIMULATION TOOLS

Careful consideration was given in choosing a simulation platform. We examined many of the popular network simulation packages such as Opnet and NS2, and also considered creating the simulation kernel from scratch before deciding that OMNeT++ would best suit our needs. Opnet proved to be too fine detailed for our use, as well as non-open source, limiting the potential simulator usage to those who obtain a copy of this expensive software. NS2 is an open-source community developed effort that provides a highly detailed network simulation, but at the cost of being extremely complex and inflexible. One of OMNeT++'s greatest assets is that it easily allows for integration of modules. In fact, on the OMNeT++ website is list of downloadable community contributed modules, of which is a highly detailed disk simulation module. OMNeT++ is also open source, making it possible to distribute our entire simulation package including the simulation kernel.

OMNeT++

OMNeT++ was designed as a highly extensible network simulation tool. It provides a robust simulation kernel coupled with an equally well designed GUI distributed in open source package [26]. The main feature of OMNeT++ is it's ability to include user written source code at any required level of simulation. For example, the third party disk simulator can exist, above, below, or in between multiple TCP/IP protocol simulation layers. OMNeT projects are created by writing an omnetpp.ini file. The omnetpp.ini file specifies the system wide and module specific characteristics of the project, for example, what type of quueing mechanism to use and disk simulator paramaters such as roational speed and delays.

Modules

Costume modules can be written and easily integrated into OMNeT++ using either the C or C++ programing languages. Modules are integreted into OMNeT by specifying module paramaters in the omnetpp.ini file and configuring their communications capabilities and instantiation in the .ned files. They can contain as many source files as needed, allowing the module writer to seperate module components into a logically organized structure.

NED files

Ned files are one of the more esential parts of OMNeT++. The main method of communication between modules in OMNeT++ apart from a global decleration/function, is the gate. Module specific input and output gates are declared and assigned through the .ned files. The gates are links between simulation components that allow for sending and recieving of messages, both built in, and user provided by extending the cMessage class. Also associated with gates are delays and interconnect type (10/100MB/s and 1000MB/s nic card for example). The protocol implementation used with each link is also specified in the .ned file.

INET

One of the most utilized componenents of many simulations is a TCP/IP implementation. The protocol is used in most network environments and is the protocol of the internet. The INET extentions to OMNeT++ provides a thorough TCP/IP implementation as well as a UDP protocol implementation and application models. INET also includes routing capabilities and can model PPP, Ethernet and 802.11 link layers. Many examples are provided with the INET package as well as an online tutorial outlining the necessary steps to implementing a module that includes INET support. The package is also kept up to date by the community, integrating such features as IPV6 support.

File System Simulation (FSS)

A community provided module, the file system simulation uses OMNeT++'s simulation kernel to simulate the latencies associated with a physical hard disk. Included in the addon is the ability to read disk requests from a file, or useing one of the built in generators. Since this module is open source like all the other community provided OMNeT++ components, modification and redistribution are permitted, making it an ideal cadidate for use in our simulator. Also included are disk cache replacement policies such as LRU, priority, and fair share. The disk simulator also provides specifications for an HP hard disk realeased in 1994, and the ability to change hard drive specifications allowing for simulation of a more modern disk drive.

HECIOS

One of the main reasons for choosing OMNeT++ for our simulator, Hecios (High End Computing I/O Simulator), was the ability to manipulate it into a highly configurable and accurate file system simulator. As our aim is to movel real world systems, Hecios is modeled after PVFS2. The implementation consists of a set of client and server simulation modules that reads input from a set of trace files, one for each client node, and passes those generated message requests through the simulator untill eventually generationg a response. Through a configuration system using the omnetpp.ini and the .ned files we are able to specificy such paramaters as:

- Number of Server Nodes
- Number of Client Nodes
- Network link type and speed
- Network transfer protocol

Messages

Since OMNeT++ modules communicate through a message facility, and MPI is a message passing standard, standard MPI messages are passed between each of the modules. Module/layer specific requests and responses are generated at transmition time through the connecting gates, and the messages are transported with TCP accross the network using INET's network simulation capabilities.

Client

The client side software is responsible for generating requests to the servers and caching requested data when the cachiing mechanism is enabled. Although PVFS2 does not implement any form of caching, our simulator, and more specifically our I/O middlware cache, aims at providing a tool that would yield a realistic performance analysis of a cache system for a possible future implementation. The client simulation component is composed of the application module at the very most top of the stack, followed by the cache module, and finally the file system module that handles recieved MPI requests and processes them to produce network messages. The network simulation layer, implemented through the use of the INET facilities, provides a real world model of a typical ethernet network, including connections from each node to the system switch.

Server

When packats are recieved on the client side of the network layer, they are passed to the request scheduler. The request scheduler then processes those messages and passes them to the I/O scheduler which in turn generates a disk read or write message to the hard disk simulated by the community provided FSS. After the FSS has determined that the request is complete, a response message is generated and sent back through the stack. The server also generates invalidation messages that is released from the appliction layer when a write is recieved to simulate the effects of a cache coherent system.

CHAPTER 4
CACHE MODULE DESIGN

The Hecios simulator is designed in a manner that provides component separation for ease of module interchange. Between each layer of the Hecios modules is a communication component that easily allows message exchanging. The application and file system component are separated by the I/O middle-ware component, referred to as Hecios' caching system. Communication to and from the I/O middle-ware component and the application and file system components is done through a series of OMNeT++ standard gates. These gates allow the sending and receiving of pre-defined messages with or without transmission delays that can easily be set at the configuration layer in the .ned file, and can also modified during sending procedures if the simulator determines that a different delay is to be used. The .ned files also specify gate type (such as input or output) and what other gates they are hooked to in the other layers.

The I/O middle-ware component is subdivided to easily allow future integration of other caching techniques using the current underlying system structure. These four components are:

- Request Handler (cache_module.cc) - This module handles communication between the layers and implements a cache type.

- File Cache Module (complex_cache) - This module handles cache insertion, lookup, and deletion commands from the request handler on a per file level, implements a replacement policy.

- Block Cache Module (simple_cache) - This module handles cache insertion, lookup, and deletion commands from request handler on a per block level within each file, this component also implements a replacement policy.

- Replacement Policy Module (replace_policy.cc) - This module decides what the next replacement position should be, handles cache insertions, and cache updates.

As each one of these components can be used with other modules, we will review the implementation techniques for each separately, provide an analysis of the associated data structures, and describe the testing procedures used to ensure accuracy.

Request Handler

As messages are received by both sides of the cache module, both from the application and from the file system layer, those messages must be captured and the proper procedure for handling the message must be initiated. The handleMessage function parses the received message and calls the proper processing function. Below is a list of the messages handled and a short description of each messages purpose and the appropriate caching operation performed:

- MPI_FILE_OPEN_REQUEST - Request from the application layer to open a file given a file name. We do not make a cache lookup in this function, just pass the open request to the file system.

- MPI_FILE_CLOSE_REQUEST - Request from the application layer to close a file given a file handle. We evict the file from the cache and forward the message to the file system.

- MPI_FILE_DELETE_REQUEST - Request from the application to delete a file. We forward this request to the file system, we do not look in our cache because a delete will occur without first seeing a FILE_CLOSE_REQUEST.

- MPI_FILE_PREALLOCATE_REQUEST - Request from the application to allocate a specific amount of space for a file. Since PVFS2 handles all cached meta-data request, the request is propagated down to the file system.

- MPI_FILE_SET_SIZE_REQUEST - Request from the application to change the size of a file. If the file is not cached, we add the file to the cache and forward the message to the file system. If the file is cached, we respond back to the application layer and evict any data past the given resize amount; when the consistency flag is enabled we also forward the request to the file system.

- MPI_FILE_GET_SIZE_REQUEST - Request from the application for the file's size. If the file is cached, a response is sent back; if it is not, the file is added to the cache and the request is propagated through to the file system level.

- MPI_FILE_GET_INFO_REQUEST - Request from the application layer for a files meta-date information. Since PVFS2 handles all cached meta-data request, the file the request is propagated through to the file system level.

- MPI_FILE_SET_INFO_REQUEST - Request from the application layer to change a files meta-data information. Since PVFS2 handles all cached meta-data request, the request is again propagated to the file system.

- MPI_FILE_READ_AT_REQUEST - Request from the application layer to read data from a file at a specific offset [15]. If the file is present in the cache, a response is sent back to the file system; otherwise, the request goes out to the file system and the read data is cached.

- MPI_FILE_READ_REQUEST - We assume all reads will be READ_AT requests and therefore throw an error if one of these messages is seen.

- MPI_FILE_WRITE_AT_REQUEST - Request from the application layer to write at a pre-specified offset. If the file is cached, a response is returned and the request is propagated to the file system depending on if the consistency flag is enabled or not. Otherwise, the request is just

propagated to the file system and the written data is cached for future use.

- MPI_FILE_WRITE_REQUEST - We assume all writes will be WRITE_AT requests, and throw an error when we see this.

- MPI_FILE_XXX_RESPONSEs - Each request sent to the file system has a response associated with it. This response is received from the file system layer when the request action has been completed. In all cases, the response is simply forwarded back to the application layer.

- CACHE_EVICT_RESPONSE - The only action that occurs during a response is when an evict response is received. Upon receiving the response the given, handle, offset, & file extent are evicted from the cache.

Cache Coherence

Although a simple coherence mechanism is fully implemented throughout most of the simulator, we have chosen to include a great number of coherence hooks where possible in the cache to ease the transition to a more complex coherency mechanism when the time is appropriate. In order to simplify debugging and ensure that other parts of the system are functioning before we turn on a component that uses the network as extensively as a coherence protocol, there is a switch in the request handler to enable and disable coherence. When coherence is disabled, data is still stored in the cache; however, write messages are no longer propagated to the file system if they are found in the cache, and therefore, invalid entries exist at other cached locations until they are removed by the replacement policy. As we are initially modeling PVFS2 which currently does not cache file data, the coherence switch allows us to examine the behavior of PVFS2 in a simple non-coherent ideal caching scenario. Although this single feature does not give us an accurate indication of invalidation bandwidth, coupled with an included mechanism that counts

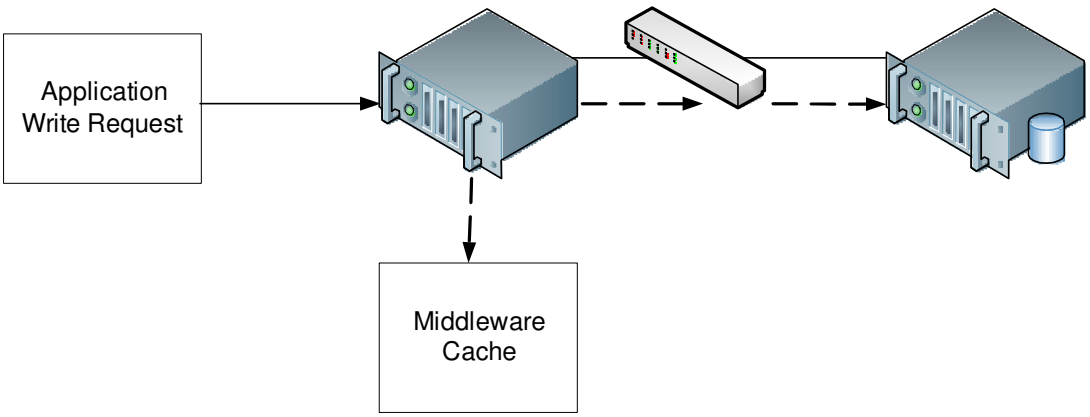

Figure 4.1 Write-through cache

cache evictions, we will be able to fully predict the effects of more complex protocols.

As previously explained, the simple coherence mechanism implemented in the request handler illustrates the ease of implementing a coherence algorithm into the Hecios simulator. When the coherence mechanism is enabled, the cache turns into write-through cache. With a write-through cache, as soon as data is written to the cache, it traverses the network and is written back to the host I/O node (Figure 4.1). This method was initially chosen because of simplicity, write-back coherence algorithms require no state tracking of cached data [10]. This does comes at the cost of an increase in the number of network transactions due to updates occurring at every cached write.

As with many distributed clustering protocols, a trade-off must be made between system complexity and ease of implementation, and network transaction overhead. Although more complex, write-back caching schemes reduce network transactions by only writing back data when necessary, such as in the MSI or MESI protocols [10]. These schemes require the cache to know the current state of each entry; therefore, each cache entry has an entryState that can be left empty, populated by one of the four available states (EXCLUSIVE, VALID, INVALID, UNKNOWN), or a state that is added to the cacheState

enumerator in the cache_entry.h file.

There is also another coherence mechanism which we have not explored, directory based cache-coherence. A caching mechanism used for coherence in the SGI origin, directory-based cache-coherence is based on the idea of setting aside an area of the cache for storing where the home node of each block exists. Depending on whether the directory scheme used is flat or hierarchical, the amount of network transactions needed to invalidate cached data and write back dirty data could be significantly reduced. However, such a schema would easily allow for an implantation of a cooperative caching technique [11] to give a possible performance increase.

Whichever coherence mechanism is chosen for implementation, there must be an easily accessible interface available for searching the cache, cache insertion, and cache deletion. Hecios provides this through the use of a set of helper methods that do just that.

Helper Methods

The cache_module.cc includes a set of 3 public helper methods that provide access to the cache. They allow the request handler to perform insert and remove operations to the file cache.

- cacheAddHandle(int handle, int offset, int extent) - The given handle,offset and extent are added to the cache, handle overlaps are resolved by storing the largest of either the input or currently stored extents. On overlap, a combining feature is present for combining overlapping offsets and extents within the same file handle.

- cacheRemoveHandle(int handle) - Calls the underlying cache remove function with the given handle, removing the file from the cache.

- cacheEvict(int handle, int offset, int extent) - Calls the underlying cache removeOffsetExtent function with the given handle, offset and extent.

```
struct LRUSimpleCacheEntry
{
   int extent;
   int offset;
   int address;
   int state;
   double timeStamp;
   std::list<int>::iterator lruRef;
};
```

Figure 4.2 Cache Entry Data Structure

This function removes the given offset and extent from a stored file handle and removes the whole file handle if the result is an empty entry.

Cache Entry Structure

The cache insertion, update and replacement policies handle entry eviction and insertion through a superclass based policy system. The cache entry data structure contains 6 fields (Figure 4.2). At insertion, the state of inserted cache items is not set as the current coherence mechanism does not use this field. The cache entry class provides public access to all the fields to allow for easier function calls. The first four store the current entry's properties, the timeStamp holds the simulation time the entry was added to the cache, and the lruRef references the corresponding lru entry in the lru list.

Cache Structure and Policy Design

As we have previously stated, for ease of future implementation of cache protocols into our system we have separated the different components of our I/O middle-ware. The replacement policy portion of the cache module is outlined using a C++ superclass. The superclass for all replacement policies is defined in the replacement_policy.h file. As a subclass replacement policy, replacement techniques must implement the three pure virtual policy functions, GetEvictIndes(), PolicyUpdate(), and PolicyInsert(). Included in our policy

development are full implementations of LRU and FIFO policies. These replacement techniques can be used in either the file level caching structure, complex_cache, or the block level caching structure, simple_cache. A complex cache item is created for each handle/file, each complex cache item stores a list of corresponding cache blocks in the form of simple cache items.

Initially, as a way of illustrating a fully integrated cache; insertion, removal, and eviction were built into one module, simple_cache.h. However as the need grew to add more replacement policies into the cache, the replacement policy portion of the cache was separated from the other portions. Because of this, the data structures that comprise the cache, a standard STL (Standard Template Library) list and a standard STL map, are created and initialized in complex_cache.h module which will be further referred to as the data cache file (Figure 4.3).

Cached data structures

File Cache (complex cache item)

The data cache file incorporates two structures that provide efficient access and storage for cache items, a C++ standard template library map and a list. Although one of these data structures is sufficient, in order to cut the overhead associated cached data accesses, both structures are required. The manner in which a cache is normally accessed is one of two ways:

- Searching for a specific address/entry - In which case map traversal is faster due to the tree based implementation found in the GNU STL library implementation [22].

- Finding which address/entry should be evicted - In which case list traversal is faster because of the nature of the STL list. The STL list allows insertion of items in an order that corresponds to the implemented replacement policy.

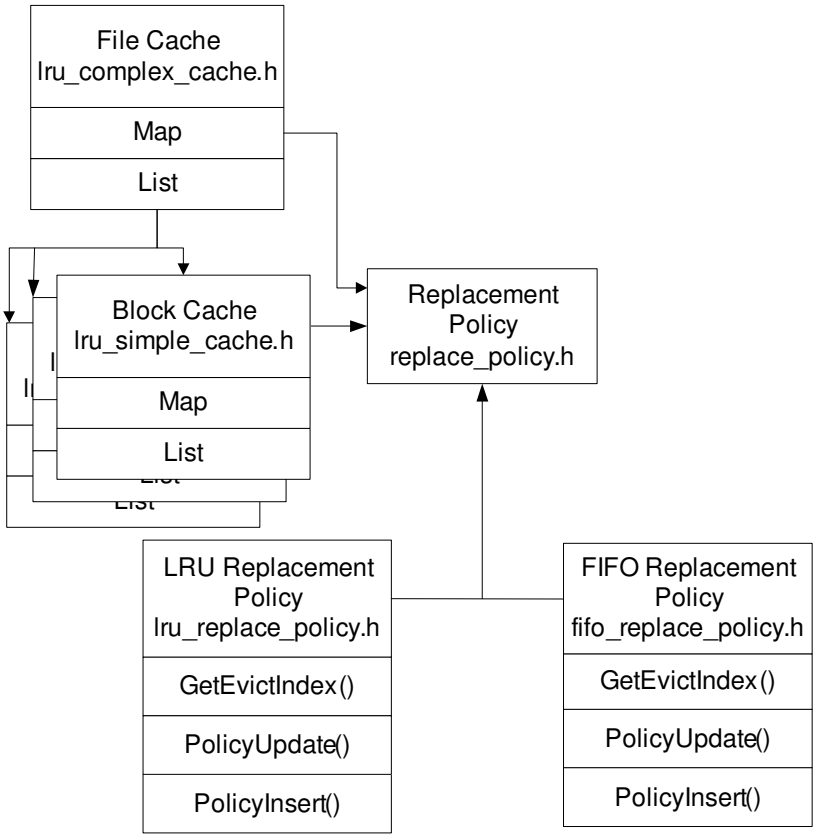

Figure 4.3 Cache Module Layout

The map stores two structures, a reference and a cache entry. The cache entry stores a handle, and a reference to an lru_simple_cache item that stores each individual entry block (i.e., entry and extent). When a search for a specific address is requested, the map's find function traverses the tree and returns the object with an equal handle, or an iterator to the end of the map if none exists. A quick mathematical operation is then performed to determine if the searched for handle is within the range of the found handle up to the stored extent. This list is a simple list structure and only stores the order of handles as determined by cache policy order.

Block cache (simple cache item)

The block cache entry data structure works in a similar manner to the file cache entry. There is a map and list associated with the block cache, and simple cache item associated with each individual block. Each block is comprised of an offset and an extent, when a new entry is added a series of comparisons occur as described below to correctly insert the block in its proper location

Insertion Policy for Block Cache

The insertion policy implementation located in the insert() function in lru_simple_cache.h, handles all block insert requests made from the file cache. The function takes in two arguments, an entry offset, simply identified as the key argument, and an entry extent, referred to as the value argument. First, iterators are initialized for map updating and traversal . A call is then made to the maps upper_bound command with the given key, and an offset is returned to one of the iterators. If an entry is found in the list and there is more then one item in the cache, an update procedure occurs to the cached entry, and the entry is moved to the proper location depending on the insertion method chosen, FIFO or LRU. This is done by calling the specific policy's PolicyUpdate() function.

The second scenario occurs if there are no items in the cache, a proper push_back or push_front call is made, depending on which insertion algorithm is chosen, and the entry count is incremented. This and the previous scenario are perfect examples of why the insertion and replacement policies are separated. Policy separation allows insertion and replacement to be independent, allowing a multitude of cache control options once more policies are implemented. Another scenario occurs when there is only one item in the cache, in this case, the proper policy dependent cacheInsert() function is called, an update to the list is made and the cache size is updated. In order to provide limits on cache size and provide usage statistics, the size of the data kept in the cache is updated whenever an insertion or removal occurs. If an entry will overfill the predetermined single entry size, blocks are evicted until the empty space is sufficient. If all blocks are evicted, the entry is simply inserted into the cache, as the file cache also performs a global size limitation. This soft limit feature allows for entries greater then individual block sizes yet still allows the entire cache to maintain it's size limitation.

The final cache insertion scenario occurs when the item to be inserted does not match any previously stored entries. First, items are evicted using the implemented replacement policy's eviction procedure if the cache size is too big, taking into account both the physical cache size and the maximum number of entries parameters (both easily configured). The correct map insertion position is found through results of the initial map upper_bound function call. Finally, cached data is added to the list at the policy-specific position and the entry count is updated. One of the more unique features of this cache is also apparent in this scenario. If overlapping entries exist at this step, the cache combines the entries, keeping the handle of the lowest entry, and extending the extent to include both entries. As each scenario of cache insertion varies in the amount of steps taken, this effects the overhead associated with insertions depending upon the state of the cache. An analysis is presented later in the chapter. A typical insertion procedure is seen in Figure 4.4.

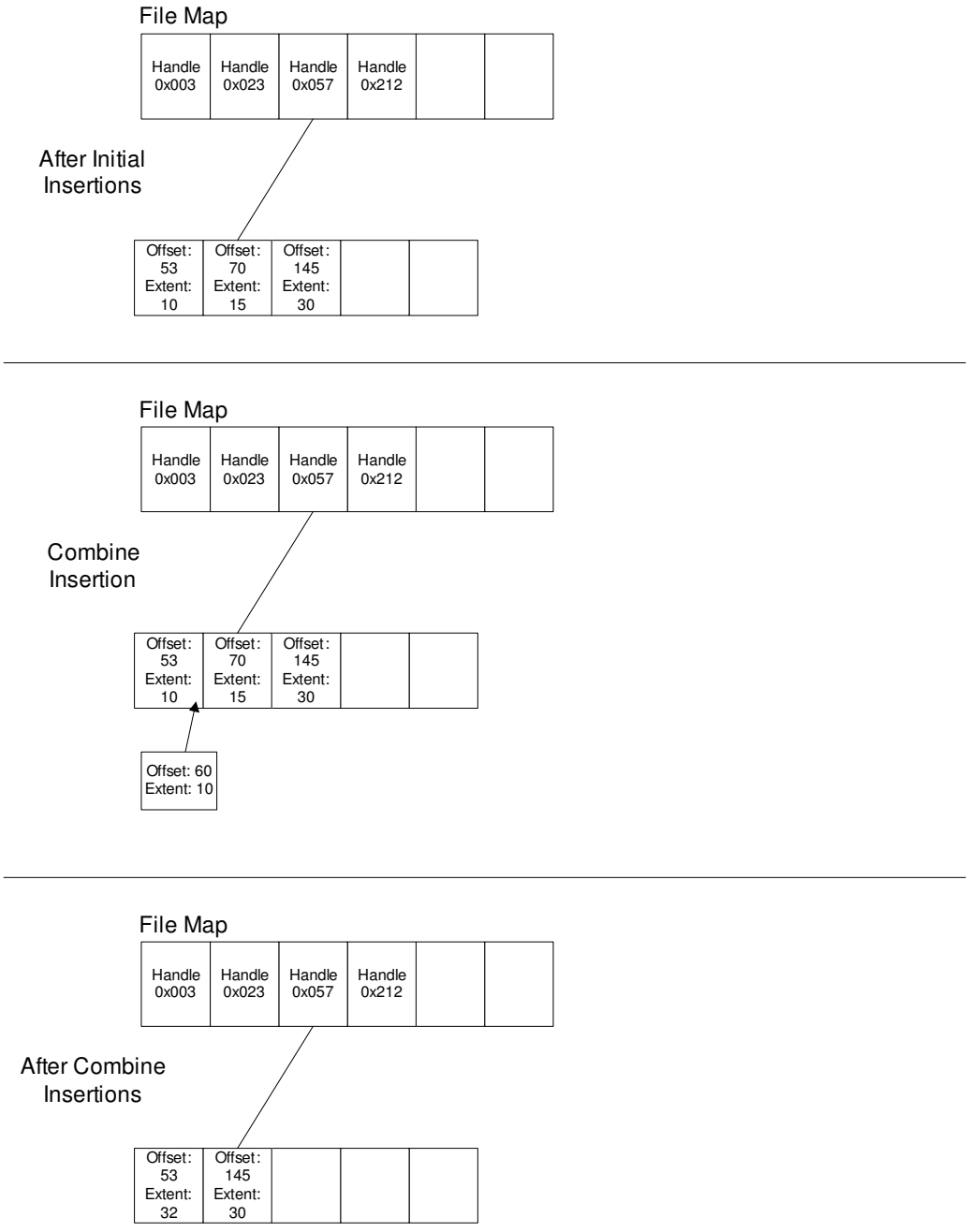

Figure 4.4 Entry Insertion Combining

Insertion Policy for File Cache

The insertion policy for the file cache is a simplified version of the block cache. Instead of upper_bound function calls for map search, a map find call is made, and if the entry does not exist, a new simple cache item is created corresponding to the newly inserted file handle. If the entry does exist, the underlying block cache insert function that is associated with the specific file handle is called, and execution proceeds as stated above. A hard limitation on entry size exists at this level. If the entry to be inserted overfills the cache, it is simply not inserted. Other methods of handling this would be to simply insert a portion of the entry that would not overfill the cache, this might however produce a portion of the cache that is never used again, while evicting all entries in the cache that might have been previously accessed many times.

Replacement Policy

As discussed previously, one of the simplest ways of modifying a caching technique is varying the way in which cache evictions occur. Our cache framework provides a standard replacement policy super class that allows a new eviction policy to be created by overwriting the virtual GetEvictIndex function with another implementation. Both the FIFO replacement policy and the LRU replacement policy are subclasses of the replacement policy class and each implements a unique version of the GetEvictIndex function. A GetEvictIndex() call is made from the insert() function to get the address of the element of the list that is to be removed. As these functions are not very complex, simply returning a pointer to either the beginning or end of the list, they provide a solid example for creating a new standard cross policy functions that could handle insertions or lookups, or possibly a more complex caching mechanism.

Testing

As with any development process, a great portion of ensuring system correctness involves thorough testing of each individual component. For validating individual Hecios modules, we use the well known unit testing approach. Unit testing consists of isolating a specific system component and feeding it an extensive test set of common and boundary condition inputs. In order to guarantee correct operation, the component outputs are compared to expected system outputs, if a non-match occurs, an error message in printed to the screen indicating the point of testing failure.

The cache testing phase occurred in three distinct iterations. The first iteration occurred when the initial cache module, lru_simple_cache, was completed. The module properly inserted and removed data, however, did not include the capability of combining cache entries. The test suite for the initial cache was comprised of constructor testing, insertion testing, removal testing, lookup testing, cache size testing, and LRU policy testing.

The constructor test simply asserted that creation of a cache object correctly resulted in properly initialized values, for example that the cache size was zero right after creation. The insert test tested if the correct number of items were recognized to be held in the cache. For example, three items were inserted, and the cache size was checked to correctly indicated that three items were in fact in the cache. The removal test explicitly looked for correct removal of specific items after they were correctly inserted. Similarly, the lookup test performs a lookup operation on the cache with items that should be in the cache. For example, an insertion of offset 2018, with extent 2009 provides a cache hit for lookups of 2018, 3000, but not 8000. Cache size testing involved multiple inserts and deletions with checks in between each to correctly determine that the size variable of the cache was being correctly updated. Finally LRU policy testing involved inserting multiple cache entries to fill up the cache, and ensuring that the last used item was being correctly evicted.

The second cache testing iteration occurred during and after the cache

ComplexCache Function	Runtime
insert()	$O(4 \log n)$
remove()	$O(2 \log n)$
removeOffset()	$O(3 \log n)$
removeOffsetExtent()	$O(5 \log n)$
lookup()	$O(\log n)$
findOnlyHandle()	$O(\log n)$
findOnlyHandleOffset()	$O(2 \log n)$
findOnlyHandleOffsetExtent()	$O(2 \log n)$
mapPrint()	$O(n)$
size()	$O(1)$
physSize()	$O(1)$

Table 4.1 ComplexCache Runtime

modifications made for entry combining. The entry combining technique used was combining overlapping entries on insertion. The insert method would check entries after the passed in file handle, and ensure that those entries were not overlapped by the new insertion. If there was overlap, entries were combined. The above tests were re-run; however, the entry offsets and extents were changed to allow for overlap and proper test bench output was generated by all tests in this iteration.

The third testing iteration occurred after the file cache, lru_complex_cache, was completed. Similar tests were run, also testing the file cache's ability to insert files and keep track of the number of files inserted. LRU implementation correctness was also tested along with total cache size eviction. Combining was also tested at this level as was the get size function and the map print functions for both the file cache and the block cache. All tests indicated system correctness and a sample trace was run without error.

Run Time Analysis

An important part of any system implementation, and especially in a complex a simulation systems such as this one where the goal is to simulate of hundreds of attached nodes, careful consideration must be given to each function implementation to ensure optimal system design. We now examine the

SimpleCache Function	**Runtime**
insert()	$O(2 \log n)$
remove()	$O(2 \log n)$
removeRange()	$O(4 \log n)$
lookup()	$O(\log n)$
findOnlyKey()	$O(\log n)$
findOnlyKeyValue()	$O(2 \log n)$
findOnlyKeyValueOffset()	$O(2 \log n)$
mapPrint()	$O(n)$
returnEvict	$O(1)$
size()	$O(1)$
physSize()	$O(1)$

Table 4.2 SimpleCache Runtime

Evict Function	**Runtime**
GetEvictIndex()	$O(1)$
PolicyUpdate()	$O(1)$
PolicyInsert()	$O(1)$

Table 4.3 FIFO and LRU Policy Runtimes

cache module functions and determine the run time for each function. We first consider the cache insert function. All insert operations start with a map find to look for the cache item to be inserted and another map upper_bound call to search for the specific offset/extent range, this gives a runtime of $O(\log n)$. Although the exact run time differs by a couple of constants depending on the number of items in the list and whether the item to be inserted overlaps any other entries, map traversal is only done once, as the STL list provides flexible object ordering. Replacement algorithm getEvictIndex() calls yield a lookup of $\omega(1)$. Cache removals, are also $O(\log n)$. The same above mentioned map find function is called on removals to search for the entry to be removed. List and entrymap deletions are done in constant time as the found entry's LRU and map addresses are retrieved from the find operation, either directly through the return of find or indirectly through a lookup of the list reference variable stored within each cache entry. And for the lookup and the three finds, they also have a run time of $O(\log n)$ because of the use of the map's built in find. Again, no other traversals are needed and only constant time operations such as setting the one or two pointers to be returned are performed. Finally, the cache size() and physSize() functions return in $O(1)$ time as they only return the constantly updated numEntries private variable. In tables 4.1 through 4.3, we see these and all the caching function runtimes. Note the overall efficiency of the algorithms and that only the printing functions ever approach highly undesirable linear time, while all replacement policy functions run in constant time, and all others run in logarithmic time.

Through our analysis of the cache module, we have shown how the flexibility of our middle-ware enables it to be modified and extended to model other caching protocols and techniques. We have also shown with our run time analysis that the included caching functions should scale well due to the efficient nature of their implementations. In large parallel file system structures similar to the ones we are simulating, a lightweight cache package also provides us with the ability to reproduce some of the more complex caching

environments and schemes.

CHAPTER 5
CONCLUSIONS

As the size of parallel clusters and scientific applications has grown, the need for a highly scalable and consistent file system has been meet largely by research developments such as PVFS2. With any software package the need arises to implement ideas that might help system performance, or increase scalability. These research ideas can be explored in many ways, as we have discussed before, three main research vehicles of system design exist. Ordered from least to greatest complexity are analytical system models, system simulations, and system prototypes. Due to the complexity of parallel file systems, most analytical models fail to accurately capture all of the systems characteristics, while simply implementing a new research idea in a production file system could involve a system-wide redesign. In most cases, a thorough simulation, while providing performance indicators, takes considerably less time to implement than a thorough prototype, making it a cost-effective alternative.

This paper describes the process of creating a highly configurable parallel file system simulator cache module. Our system's modular system design allows for system extensibility and reconfiguration based on the simulator user's desired configuration. Our usage of the OMNeT++ simulation package as our simulation kernel, provides accurate modeling of networking hardware and protocols. In our attempt to simulate PVFS2 and other file systems, we organize Hecios' modules in a style similar to the most common parallel file systems layered structure . This provides us with a model that allows us to easily translates studied component specific implementations to their respective production system counterparts. As the main research focus of parallel file systems is to increase I/O performance, many caching mechanisms have been studied in order to reduce the amount costly network transactions. Hecios' included client level data cache provides an easily configurable cache that can

be adapted to emulate those of common parallel file systems, or as a research tool for studying the effects of adding a new caching technique to an existing system.

After establishing the importance of parallel file system research and looking at a variety of production caching techniques and parallel simulators, we introduce Hecios as a candidate for simulating most parallel system while focusing our attention to our I/O middle-ware client cache component. In Chapter 3, we provide a brief overview of OMNeT++'s facilities and mechanism, and continue by exploring the structure of the Hecios simulator.

In chapter 4, we describe the cache module's integration within the Hecios simulator by going over the 4 main cache components. First we look at the request handler and saw how our implementation handles passing MPI I/O operations and decides which specific requests to cache. We then examine a series of cache coherence techniques illustrating the process involved in achieving a fully coherent cache subsystem implementation within Hecios, with a focus on the provided helper methods and each cache entries efficient data structure.

Further into the chapter, was an observation of the implementation of our cache structure and insert and evict policy designs, noting the overall cache module layout and its importance in providing system flexibility. We then analyze our testing procedures to ensure proper cache operation and provide a run time analysis of the cache modules' built in functions.

We have shown that our simulator along with the I/O middle-ware cache component provides the community with a highly scalable simulator. We have also shown how our simulator's easily understandable component structure can help facilitate the development of other caching technique implementations.

Contribution

We believe that our work provides the parallel file system community with a modular cache simulation tool embedded into the highly configurable Hecios simulator with the following:

- An MPI I/O middle-ware that can easily switch between coherent caching, non-coherent caching, and non-caching modes.
- Simplistic and efficient cache data structures, with $O(2 \log n)$ insertions, and lookups, and $O(2 \log n)$ remove and single eviction-inserts.
- Cache data access methods with low run-time.
- LRU and FIFO cache replacement policies with a highly configurable framework for adding other policy types.
- A cache testing unit that ensures proper operation, on a per file and file block basis, including instances of block combining.

Simulation Usage and Usability

One of the main goals of Hecios is to allow the system to easily mimic a wide range of system components and techniques. The modular nature of the OMNeT++ discrete event network simulation framework coupled with the use of the .ned file mechanism, allows components such as network links, switching devices, and specific Hecios components such as the I/O middle-ware, to be easily interchangeable with a single edit. For example, an I/O middle-ware stand-in module exists that simply forwards all requests and responses to the next layer, allowing for a non-caching middle-ware implementation

The process of running and compiling Hecios is straight forward and outlined in Appendix A. The OMNeT++ simulation package can be obtained from the OMNeT++ web site located at www.omnetpp.org. The Hecios simulator including the INET package can be obtained from the PARL CVS repository, under the project name hecios.

Future Work

While we provide a sufficient framework for simulating caching effects in a parallel system, we have not provided the community with a wide assortment of caching protocols or replacement policies. The provided framework allows such implementations to be easily incorporated into system simulations. Although a large number of the Hecios modules are in some form complete, the request scheduler component is yet to be fully implemented; but a sufficient stand in module exists to provide system pass-through operation.

One of the more interesting possibilities for future exploration would be to provide different caching techniques for different portions of the file blocks. Also, as PVFS2 is non-redundant, a possible avenue for future exploration would be a look at redundancy techniques and their effects on overall performance, possibly incorporating coherently cached data. While our simple write through cache provides us with an easily and realistically implementable cache mechanism, other coherence techniques exist that provide better performance, but at the cost of increased implementation complexity on the client and server sides. Even though our chosen write-through cache technique has a large amount of overhead, it provided us with the ability to quickly integrate a coherent caching mechanism, allowing us to fully realize the expandability of Hecios and OMNeT++. In the future, looking at more complex techniques such as directory or write-back caching would be simpler because of the outline provided by the implemented protocol.

Our policy based cache replacement technique, implemented at both the file level and block level, allows for code re-use and simpler system manageability. As the middle-ware cache is also loosely based off of the server level lru_timeout_cache, with a few simple modifications, it too will be able to take advantage of the standard replacement policy structure implemented for the middle-ware. Since all the caching components are independent, not only will we be able to determine which replacement policy yields the best results, it may turn out that a combination of different policies at each cache level pro-

vides optimal system performance.

We have already established that local caches can offer large performance gains by keeping local copies of those files which are accessed either most frequently or recently. With the increase in network bandwidth of today's high speed interconnects, it becomes possible to think that going so far as reading other client caches instead of actually going all the way to the physical I/O node disk, would provide even greater speedups. Research simulations at UC Berkley [11] have looked into this unique mechanism and noticed large performance gains with an effective implementation. Given the current Hecios infrastructure, a list mechanism could be easily implemented that allows each client to know what the other clients are caching. While this would add a great amount of network overhead, OMNeT++'s network oriented nature, would easily allow for an implementation of a second low speed cache network.

APPENDICES

Appendix A

Installing and running Hecios

Installing Omnet++

The following is a tutorial on installing Omnet++ in Linux (FC 5):

```
   Before omnet++ is installed, there are a few required packages
that must also be installed.
   --Install and configure Tcl/Tk packages for your linux distribution.
   For Fedora Core, the Tcl and Tk packages and the devel packages
are required:
   yum install tk.i386
   yum install tk-devel.i386
   yum install tcl.i386
   yum install tcl-devel.i386
   --Also need to install graphviz:
   yum install graphviz.i386
   --Next install blt:
   Although it can be installed through yum, omnet++ does not recognize
it when it is installed that way, it must be installed by hand:
   tar -zxf BLT2.4z.tar.gz
   cd blt2.4z/
   ./configure
   make
   make install
   --And finally install giftrans:
   Download the source rpm, then follow the following instructions
for Fedora Core to setup giftrans, these may differ for other distros:
   rpm -i giftrans-1.12.2-20.src.rpm
   cd /usr/src/redhat/SOURCES/
   tar -zxf giftrans-1.12.2.tar.gz
```

patch -p0 < giftrans-1.12.2-operator.patch

cd giftrans-1.12.2

gcc giftrans.c -o giftrans

cp giftrans /usr/bin/

gzip giftrans.1

cp giftrans.1.gz /usr/share/man/man1/

Now that all the required packages are installed, download the source files from the omnet website for omnet++:

http://www.omnetpp.org/filemgmt/viewcat.php?cid=2

Extract the source files to the desired install directory, for illustration, we installed Omnet++ to /sandbox

--Extract using the command below:

tar -zxf omnetpp-3.2p1-src.tgz

--Now add the omnet++ paths to your .cshrc file:

set path = ($path /sandbox/omnetpp-3.2p1/bin)

setenv LD_LIBRARY_PATH .:/sandbox/omnetpp-3.2p1/lib

--Then go to the newly extracted directory and configure omnet++ with the defualt options:

cd omnetpp-3.2p1

./configure

Notice any messages that configure gives you, if one of the above required packages is not installed properly, it will give a warning. For example, when we ran it we got the following at the end of the configure:

*WARNING: The configuration script could not detect the following packages:

*

* MPI (optional) Akaroa (optional)

```
   *
   *Scroll up to see the warning messages (use shift+PgUp key), and
see config.log
   *for more details.  While you can use OMNeT++/OMNEST in the current
   *configuration, please be aware that some functionality may be
unavailable
   *or incomplete.
   *
   *Your PATH contains /sandbox/omnetpp-3.2p1/bin.  Good!
   *Your LD_LIBRARY_PATH is set.  Good!
   *
   Since we will not be using these two packages now, it is safe to
ignore this warning.
   --Now, make the source files:
   make
   ------Omnet should now be configured and working properly,
   ------go to the samples directory and run a couple of the samples
   ------to make sure.
```

Installing Hecios

Hecios' build system is well maintained and handles cross compiling very well. All that is needed is a working installation of Omnet++ and the Hecios package. From the root directory of Hecios the following commands are executed to configure and make Omnet++:

```
./configure ---with-omnet=/sandbox/omnetpp-3.2p1/lib (where this
is the lib folder of the Omnet++ installation)
make
```

To run the executable simply execute:

```
../bin/hecios
```

BIBLIOGRAPHY

[1] B. Anderson, "Mass Storage System Performance Prediction Using a Trace-Driven Simulator," msst, pp. 297-306, 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05), 2005.

[2] R. Bagrodia, E. Deelman, S. Docy, and T. Phan. "Performance Prediction of Large Parallel Applications using Parallel Simulations," Proc of the 7 th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, Atlanta, May 1999.

[3] R. Bagrodia, S. Docy, and A. Kahn, "Parallel Simulation of Parallel File Systems and I/O Programs," In Proc. of SuperComputing'97.

[4] P. Bosch, and S. Mullender, "PFS and Patsy: a new file system design methodology," Pegasus paper 95-1, Nov. 1995.

[5] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel File System For Linux Clusters," in Proceedings of the 4th Annual Linux Showcase and Conference.

[6] P. Corbett, and D. Feitelson, "The Vesta parallel file system," ACM Transactions on Computer Systems 14, 3 (August), 225–264, 1996.

[7] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J-P. Prost, M. Snir, B. Traversat, and P. Wong, "Overview of the MPI-IO parallel I/O interface," In IPPS '95 Workshop on I/O in Parallel and Distributed Systems, pp. 1-15, Apr 1995.

[8] P. Crandall, R. Aydt, A. Chien, and D. Reed. "Input-Output Characteristics of Scalable Parallel Applications," In Proceedings of Supercomputing '95. ACM Press, December 1995.

[9] T. W. Crockett, "File concepts for parallel I/O," Proceedings of the 1989 ACM/IEEE conference on Supercomputing, p.574-579, November 12-17, 1989, Reno, Nevada, United States.

[10] D. Culler, J. Singh, and A. Gupta, "Parallel Computer Architecture: A Hardware/Software Approach," Morgan Kaufmann Publishers, San Fransisco, CA, 1999.

[11] M. Dahlin, R. Wang, T. Anderson, and D. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," In Proc. of the First Symp. on Operating Systems Design and Implementation, pages 267– 280, November 1994.

[12] The NFS FAQ. http://nfs.sourceforge.net/.

[13] I. Foster et al. "Remote I/O: Fast Access to Distant Storage," Proceedings of the Fifth Annual Workshop on I/O in Parallel and Distributed Systems, 1997.

[14] W. Gropp, E. Lusk, and R. Thakur, "Usng MPI-2 Advanced Features of the Message-Passing Interface," MIT Press, Cambridge, MA, 1999.

[15] MPI Programing Guide. Table 16: MPI subroutine and function summary. http://www.nersc.gov/vendor_docs/ibm/pe/am106mst64.html.

[16] D. James, and D. Pai, "Multiresolution green's function methods for interactive simulation of large-scale elastostatic objects," ACM Transactions on Graphics (TOG), v.22 n.1, p.47-82, January 2003.

[17] B. Juurlink, and H. Wijshoff, "A quantitative comparison of parallel computation models," ACM Transactions on Computer Systems (TOCS), v.16 n.3, p.271-318, Aug. 1998.

[18] M. Kallahalla, and P. J. Varman. "Optimal prefetching and caching for parallel I/O sytems," Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures, p.219-228, Crete Island, Greece, July 2001.

[19] M. Kallahalla, and P. Varman, "Optimal prefetching and caching for parallel I/O sytems," Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures, p.219-228, July 2001, Crete Island, Greece.

[20] J. Kubiatowicz, et al. "OceanStore: An Architecture for Global-Scale Persistent Storage," ASPLOS, December 2000.

[21] E. Lee, and C. Thekkath, "Petal: Distributed virtual disks," In Proc. of the 7th Conf. on Arhcitectural Support for Programming Languages and Operating Systems, October 1996.

[22] The GNU C++ STL class list. Map Implementation. http://www.aoc.nrao.edu/php/tjuerges/ALMA/STL/html/.

[23] J. Liu, B. Chandrasekaran , J. Wu , W. Jiang , S. Kini , W. Yu , D. Buntinas , P. Wyckoff , D K. Panda, "Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics," Proceedings of the 2003 ACM/IEEE conference on Supercomputing, p.58, November 15-21, 2003.

[24] S. Moyer, and V. Sunderam, "PIOUS: a scalable parallel I/O system for distributed computing environments," In Proceedings of the Scalable High-Performance Computing Conference, pages 71–78, 1994.

[25] J. Oly, and D. A. Reed, "Markov Model Prediction of I/O Request for Scientific Application," In Proc. of the First Conference on File and Storage Technologies (FAST), Jan. 2002.

[26] OMNeT++. Homepage. http://www.omnetpp.org/.

[27] R. Shah, P. Varman, and J. Vitter, "Online algorithms for prefetching and caching on parallel disks," Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, June 27-30, Barcelona, Spain, 2004.

[28] The MPI Standard. http://www-unix.mcs.anl.gov/mpi/.

[29] The PVFS2 Parallel File System. http://www.pvfs.org/pvfs2.

[30] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: A Scalable Distributed File System," In Proceedings of the 16th ACM Symposium on Operating Systems Principles, Oct. 1997.

[31] Y. Wang, and D. Kaeli, "Execution-driven Simulation of Network Storage Systems," In Proceedings of the 12th IEEE Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.