Can a file system virtualize processors?

Lex Stein
Microsoft Research Asia
stein@eecs.harvard.edu

David Holland
Harvard University
dholland@eecs.harvard.edu

Margo Seltzer
Harvard University
margo@eecs.harvard.edu

Zheng Zhang
Microsoft Research Asia
Zheng.Zhang@microsoft.com

Abstract

Supercomputers are comprising more and more processors and these processors are increasingly heterogeneous, with differing performance characteristics. The conventional programming models assume that all nodes run in lockstep. Thus, applications run at the speed of the least powerful processor. We introduce DesyncFS, a new programming model based on the block abstraction of traditional file systems. It virtualizes the performance characteristics of processors; this allows their heterogeneity to be hidden. We show that DesyncFS allows cluster throughput to scale with average processor throughput instead of being limited by the slowest processor.

1. Introduction

Increasingly, supercomputers are being constructed as clusters made up of commodity processors, open-source operating systems, and commodity networking. As of June 2006, 3 of the top 10 and 73% of the top500.org[1] supercomputers are considered clusters.

The open standards and loose coupling of clusters encourage organic construction and heterogeneity. For this reason, supercomputers used for HPC applications are becoming increasingly heterogeneous. As well, grids interconnecting clusters introduce network heterogeneity and can further increase processor heterogeneity. Cluster and grid architectures are good because they reduce costs and allow greater and more efficient sharing between research groups.

However, at the same time these architectures create new problems for many typical HPC applications. This paper looks specifically at single-site clusters and the problem of processor heterogeneity. It is possible that some of our ideas can be generalized to grids, but we want to prove and understand our ideas in the cluster first.

Many interesting applications exhibit internal data dependencies that require synchronization at many points during execution. Examples include particle dynamics, gas models, wave equations, Fourier transforms, sorting, cosmology models, chemical reaction dynamics, and many others. These applications are most impacted by cluster heterogeneity, because any synchronization point makes fast nodes stop and wait for slower nodes. Other applications, such as Monte Carlo simulations, need little or no internal synchronization. They are often called *embarrassingly parallel* and require little system or runtime support to run effectively even on a heterogeneous cluster. With heterogeneity, difficulty is caused by the computations that are not embarrassingly parallel, with dependencies between processors.

In order to make these applications perform better on heterogeneous clusters, we need to find some way to *desynchronize* them; that is, to loosen the waits-for requirements imposed on the cooperating processors. One major source of such requirements is the need to read and use intermediate data before it is overwritten. Another is the difficulty, given conventional programming models, of dynamically shifting work away from processors that are falling behind.

To tackle these problems we introduce the concept of a *desynchronizing file system*, or DesyncFS. The DesyncFS programming model has two primary features that address these issues. First, it manages data and computational progress for the application, so that the application itself can be stateless. Second, it supports versioning of that data, so intermediate data can persist even after it is nominally overwritten. We call DesyncFS a file system, even though no disk storage is necessarily involved, because it is based on the fundamental abstraction of file systems: the *block*.

Because DesyncFS makes the individual performance of each processor invisible and irrelevant to the application, we say that it *virtualizes* these performance properties: DesyncFS provides the application with the illusion that it has an arbitrary number of equally fast processors.

Virtualization is an old and general term in computer systems. A system virtualizes if it presents an abstract interface that hides characteristics of underlying resources. Virtualization can occur at many levels in the system software stack. For example, it can be implemented at the machine level, where a virtual machine (VM) emulates the instructions of some physical machine. Or it can be implemented at the API level, where library routines hide details of the underlying resources.

Machine virtualization simulates a processor architecture using a virtual machine monitor (VMM) to implement a VM interface. Work on this form of virtualization began early in the development of modern computer systems with seminal work by IBM [1] and Goldberg [12] [13] on VMMs. Interest in VMMs resurged in the

[1] A global ranking of unclassified supercomputers, by aggregate flops throughput, hosted at `http://www.top500.org/`.

HPCVirt '07 (in conjunction with EuroSys 2007) March 20, Lisbon, Portugal.

recent decade with VMWare and Xen [3]. When clusters are run on virtual nodes, as is already widely the case and will likely increase, heterogeneity can be exacerbated and even fluctuate dynamically.

Virtualization can also be the construction of a new, abstract interface where one did not exist before. This form of virtualization is common in storage systems where many different resources are collected together under one simplifying interface. For example, logical disk volume managers.

All kinds of virtualization solve the same general problem. They add an indirection between system layers to reduce the ability of change and complexity (entropy) at lower layers to affect higher-level components. Change can affect higher layers in a number of ways. It can disrupt performance or even render a system inoperable. When used right, virtualization reduces costs by containing and hiding entropy.

Virtualization is not a new trick for the HPC community. Message passing and shared memory interfaces are designed to provide a kind of virtualization. Programmers of parallel machines became tired of porting their code to every new architecture. So they developed standard interfaces to abstract the details of communication and synchronization. As systems change, this saves tremendous effort by only requiring the port of one library to a new machine. All the programs can follow. For example, once the MPI library is ported, all the MPI programs follow for free.

The processor composition of a large parallel machine is one specific kind of uncertainty and complexity. The goal of DesyncFS is to provide an abstraction layer that hides processor heterogeneity in the same way that MPI, PVM, or OpenMP hide the machine details of communication and synchronization.

The remainder of this paper introduces the concept of a DesyncFS, outlines the design, then uses a simple and general HPC benchmark to evaluate the performance of one implementation, C-DesyncFS.

2. Background and related work

Two classes of research are related to DesyncFS; parallel programming models and parallel file systems. Programming models abstract synchronization and communication while file systems abstract data sharing across time and space.

2.1 Parallel programming models

A parallel programming model is a set of consistent abstractions to help programmers write portable and efficient programs for parallel machines. A model must help the programmer solve three problems; how to specify parallel threads of execution, how to organize communication between threads, and how to synchronize data between threads. Since these are orthogonal to the problems of sequential programming, models are often implemented as a set of routines or compiler directives tacked on to a popular sequential language. There are many parallel programming models. Over time, several have emerged as the most dominant. These are raw multitasking, message passing, multithreading, and data parallel. They differ in how they use abstraction to solve the problems of parallel execution, communication, and synchronization.

In raw multitasking, there is no runtime independent from the program and it manages all state. In message passing, all communication is through messages and no data structures are explicitly shared, making synchronization implicit in communication. In multithreading, processes communicate through shared data structures and these must be explicitly serialized through mechanisms such as locks or condition variables. In data parallel, the runtime and compiler together eliminate the management of data and control synchronization from the program. However, data parallel programs still have memory state outside the control of the runtime. Finally, DesyncFS virtualizes all communication and synchronization under one abstraction, file blocks.

state	RM	MP	MT	DP	DesyncFS
sockets	●	●	○	○	○
files	●	●	●	●	●
locks	●	○	●	○	○
memory	●	●	●	●	○

Figure 1. virtualization mechanisms for models.
This figure summarizes abstraction forms across raw multitasking (RM), message passing (MP), multithreading (MT), data parallel (DP), and DesyncFS. Solid circles indicate that the application interfaces directly against the state abstraction. Hollow circles indicate that the abstraction is implicit or managed by the runtime. Dynamic load balancing is particularly difficult for any model that does not hide memory.

It is possible to accomplish runtime adaptation by building on top of a message passing layer. However, the solution would not necessarily be general or reuseable and it would require a lot of work for just one application.

Adaptive MPI (AMPI) [16] is a library and runtime system for dynamic load balancing. AMPI is an implementation of the MPI interface on top of the Charm++ [17] runtime system. Charm++ uses object migration for load balancing. However, Charm++ programs are platform dependent and cannot migrate across different architectures. This limits its ability in practice to solve problems of processor heterogeneity. For this reason, the AMPI approach is suitable for balancing heterogeneity higher in the stack, at the application and problem-domain level.

Google's MapReduce [6] is another system for virtualizing at the API level, but goes in a different direction than MPI. It provides a less general interface, but abstracts more. The MapReduce runtime partitions the computation, schedules execution, handles hard failure, and manages communication. The scheduler handles soft failure by distributing work lazily. The cost of all this abstraction is a fairly constrained programming model. All MapReduce computations take a list of key-value pairs as input and transforms these into a final result value list. The details are described in Dean et al. [6]. MapReduce works well for some data processing applications. For example, web indexing and data mining. However, MapReduce is not sufficiently general for HPC applications [18] [6]. DesyncFS provides a more general interface suitable both for data processing and scientific computing, with no less resource abstraction than MapReduce.

Mentat [14] is an object-oriented programming system for medium-grain data-driven computation. However, Mentat lacks a mechanism for adaptive load balancing. Computation in DesyncFS is also data driven, but chunk assignments are adjusted to accomodate heterogeneity. Like the block address space of DesyncFS, Jade [20] and Linda [9] provide globally shared data structures. However, both systems lack a mechanism for dynamic adaptation. DesyncFS combines data-driven computation with a global address space and dynamic adaptation.

2.2 Parallel file systems

A file system has two essential properties. First, it allows one-sided communication. Second, data is addressed through a logical, shared namespace. Logical means that the namespace abstracts to hide details of its underlying resource representation. These two essential properties allow file systems to satisfy their central purpose of facilitating sharing across time and space (between different processes).

Some secondary properties are often associated with file systems, but are less essential. Persistence is one secondary property, though some common file systems lack persistence. For example, memory file systems or UNIX TmpFS [22]. A hierarchi-

cal namespace is another secondary property. Some systems called file systems are not hierarchical. For example, the Semantic File System (SFS [11]) and MogileFS[2], the file system used by LiveJournal.com. Some people claim that flat file systems are properly called object stores and not file systems, but there is disagreement on this point.

There have been mountains of research on distributed and parallel file systems. It is difficult to know where to start, so we will discuss some relevant peaks.

Surprisingly to many, NFS [21] continues to be widely used in HPC environments despite the fact that it has changed little since 1984, lacks any real security, and has a data consistency model that is impossible to formalize. This surprising success is the result of three factors. First, for runtime sharing HPC programs tend to use message passing or shared memory libraries, not file systems. File systems are generally used for very coarse granularity sharing, between jobs. Therefore, the lack of a consistency model does not matter. Second, environments that want security tend to use mechanisms outside of the file system. For example, login control and firewalls. The lack of security in NFS matters less when security is being enforced elsewhere. Third, NFS is mature, stable, well-understood, and relatively inexpensive to maintain and administer. This combination of factors has given the granddaddy of network file systems surprising longevity.

However, NFS does have drawbacks for HPC. HPC programs tend to use other abstractions for synchronization, but many do make heavy use of file systems for other purposes. For example, for checkpointing and outputting data for visualization. For HPC, the biggest problem with NFS is its performance and poor scalability. This problem is not easy to fix because it derives from the protocol itself. In particular, the NFS protocols couple metadata and data together and assume they are served by one server. The bottleneck is knit into the protocol.

The NASD [10] and xFS [2] file system architectures eliminate the NFS bottleneck by carefully separating control and data. Data is served by many parallel servers and control is only centralized when needed, for consistency and to preserve file system semantics.

Lustre[3] and PVFS2 [19] are working realizations of the NASD architecture. In both systems, data servers are decentralized across nodes of the cluster while the metadata server is centralized. Both systems go farther than just scaling NFS by providing the POSIX file system interface and semantics. POSIX semantics are similar to those traditionally provided by a local UNIX file system. Much of the engineering challenge of Lustre lies in scaling these semantics [4]. In general, a lot of effort has been put into scaling POSIX semantics to large clusters. However, it is unclear that the desire for POSIX has a rational foundation. The POSIX file system interface has strong semantics and there is no compelling evidence that HPC applications want or need these. It is possible that the interest in POSIX comes from a desire to standardize on some interface, any interface. The problem with standardizing on POSIX is that its semantics are tight and hard to scale.

3. Design

The DesyncFS design is described by its data and execution models. The data model describes how application data is structured. The execution model model specifies how data is computed and in what order.

[2] We could not find any formal publications about MogileFS. The main developer maintains information at `http://www.danga.com/mogilefs/`.

[3] There are few publications on Lustre. The system is built by Cluster File Systems, Inc. (CFS) and they manage a website for the project at `http://www.lustre.org`.

3.1 Data model

Blocks are the base unit of both addressing and computation. Every block has a globally-unique address. Each address is a series of non-negative integer components. The first two components are special and owned by the system. The first is the file identifier and the second is the version number. The remaining components address the data within the file. Each block belongs to the file specified by the value in its first component. Every block in that file has the same number of address components.

Every file has a fixed number of components, strictly greater than two. The first component is its number, but all remaining components are integer intervals. The dimensionality of a file is its number of components. The dimensionality of a file's data is the number of file components less two, for the file ID and version. Those components belong to the system and do not address data.

A contiguous rectangular extent of blocks within a file is called a *chunk*. Chunks are the unit of work allocation to nodes. The runtime knows about chunks, but the application does not. The application only knows about blocks.

Figure 2 illustrates with an example.

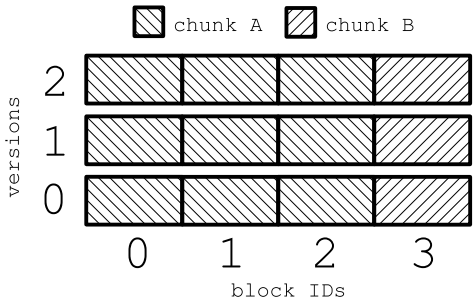

Figure 2. DesyncFS data model.
This example shows a file with ID 5, described by the chunk descriptor (5, [0 2], [0 3]). This file is three-dimensional with one-dimensional data. Chunk A has descriptor (5, [0 2], [0 2]). Chunk B has descriptor (5, [0 2], 3).

Block values do not change, but may be replaced with updated versions. Once made, a binding from block address to block value cannot be modified. With this property, blocks are said to be *immutable*.

3.2 Execution model

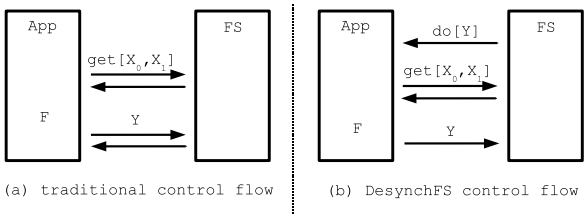

Figure 3. control flow contrast.
This figure contrasts the traditional relationship between the file system and application with this relationship in DesyncFS.

The DesyncFS execution model is inspired by cellular automata [24] and dataflow. Computation is achieved by generating new data blocks. The application defines a stateless compute function that takes a block address as input. This function reads blocks as needed and outputs one or more blocks.

In the traditional relationship between application and file system, control resides in the application and it calls into the file system as needed for storage. This can be described as a push-pull relationship. The file system is passive and the application actively pushes and pulls data. A desynchronizing file system inverts this relationship, making the file system active and the application passive. Scheduling control resides in the file system and it calls into the application as needed to compute blocks. Figure 3 illustrates how DesyncFS inverts the traditional relationship between file system and application. This inversion gives the runtime scheduling control and, with the stateless computation operator, the ability to transparently balance load.

4. Implementation of C-DesyncFS

Like MPI or Linda, DesyncFS is independent of programming language. It has been implemented in Standard ML (SML) and C. This paper reports on C-DesyncFS, the C implementation. Appendix A describes the C-DesyncFS application and system interfaces. The details of the C and SML interfaces differ, but share the general ideas. As well, the two implementations have the same high-level system architecture.

4.1 High-level system architecture

DesyncFS comprises a set of processes distributed across processors and communicating over a network. There are two kinds of processes; compute nodes and the map. Figure 4 illustrates the general architecture.

The map starts a computation, distributes the work, and periodically adjusts the work distribution based on feedback from compute nodes. The map is the only centralized component and a DesyncFS application has only one map.

The map allocates block chunks to nodes. Nodes send lookup requests to the map to resolve block locations. Chunk mappings can be safely cached with no cache coherency protocol. Nodes do this to reduce the lookup load on the map. If there are no load balanced file, nodes will only communicate with the map at the beginning of the computation. If a file is load balanced, the amount of communication increases with the frequency of adaptation.

DesyncFS presents the application with a logically-named, global block store. The interface provides location transparency. That is, the application can read any block without knowing where that block is stored. The same holds for writes. In some cases, the application programmer may choose to generate extra blocks beyond the block it was explicitly directed to compute by the file system. This can happen if the computation of one block yields other blocks as a side effect. The file system also provides location transparency for writes, shipping the block to its remote home if its surrounding chunk is assigned to another processor.

The nodes do the computation and store and share blocks. Each node is a process with two threads; the block processor (`bproc`) and the block server (`bserv`). Chunks are assigned to nodes so that each coupling of block processor and server are responsible for exactly the same set of blocks. The block processor generates blocks, scheduling execution and calling the application compute routine. The block server listens on the network for block requests. Both threads share a block cache interface so that the processor can generate and store blocks directly without memory copies or network transfers.

The runtime uses pipelined prefetching to smooth out heterogeneity without any load balancing. At any point, the runtime can call the `appDepList` routine for forward or backward dependencies. This functional representation of data dependencies can be used to prefetch data blocks from fast to slow processors, thereby eliminating I/O stalls from the compute paths of slow processors. The current implementation uses this approach and its effects are shown in section 5.1. This approach is limited to narrow processor speed variance. For greater levels of heterogeneity, dynamic adaptation must be used to make a structural change in the assignment of blocks to processors.

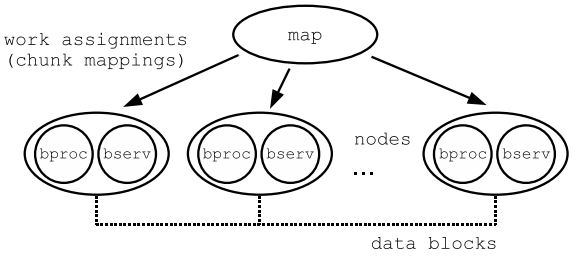

Figure 4. DesyncFS system architecture.
This figure shows the high-level architecture of DesyncFS. There is one centralized map server and many nodes. Each node contains a block processor (`bproc`) and block server (`bserv`). The map distributes work to nodes in the form of chunks. Nodes form a global block space and share blocks directly amongst one another. With load balancing, later chunk assignments will be delayed so that the map can collect information on processor throughputs to adjust for heterogeneity.

4.2 Dynamic adaptation

Adaptation can be automatic or programmers can provide hints to help. If programmers know the heterogeneity in advance, the job initialization file can be written to assign chunks to balance load. If they lack such knowledge, or heterogeneity may be unexpected, the initialization file can contain a naïve initial mapping and ask the system to adapt the file after some number of versions.

Dynamic adaptation uses feedback to lazily map chunks to balance load. Files are the unit of load balancing. That is, every file is or is not load balanced. If a file is not load balanced, all of its chunks are assigned to processors when its job begins. If a file is load balanced, chunks are assigned at the beginning, but for a number of versions that does not span all of the file's versions. Later versions are grouped into chunks when the system has more information on the relative speeds of its component processors.

Every block request from a remote processor is either a hit or miss at the target block server. For each of its assigned chunks, the block server counts all hits and misses from remote processors. If a block server has many misses and few hits, its colocated processor is not satisfying its dependencies quickly enough. If a block server has few misses and many hits, it is doing a good job of satisfying its dependencies. This miss rate provides information on how well a processor is satisfying its dependencies.

Periodically, each processor sends the map server a count of the hits and misses associated with each of its chunks. The map server uses this information to rebalance blocks across chunks. The current map implementation uses a simple recursive bisection on the miss rate for load balancing.

The map processor will send out new chunk mappings when one of two events happens. First, if it detects that some processor will soon exhaust its allocated chunks and need more work. The map will detect this because it receives periodic updates from each node on that nodes hit and miss count for each chunk and current version position in each chunk. Second, if a node requests the mapping of a block that has not been allocated to a chunk.

4.3 Crash recovery

How DesyncFS handles processor heterogeneity is a focus of this paper. We will explain crash recovery briefly, but not explore it deeply.

Crash recovery is easier for DesyncFS than many other systems. There are two reasons for this. First, DesyncFS only runs deterministic computations. This excludes interactive computer games, but does not limit its applicability for HPC. Methods for simulating randomness with pseudorandomness are well-established. Second, blocks are immutable. Together, these two properties mean that failure detection need not be perfect. If a processor is assumed to have crashed but was just slow and rejoins the computation, the worst that can happen is that there are exact duplicates for some blocks. Crash recovery is handled through a mix of replication and reexecution. The job initialization file specifies the period of replication. A crash loses blocks and these blocks are reconstructed through reexecution from the checkpoints established by replication.

5. Evaluation

This section evaluates the DesyncFS approach by comparing C-DesyncFS with OpenMPI [8] on a simple, but general HPC benchmark.

The simple benchmark is an equation solver based on Jacobi iteration, called the solver benchmark. The MPI implementation is adapted from the code described in Gropp et al. [15]. This benchmark is interesting because it it simple and its pattern of parallelization is shared by a large number of numerical programs and more complicated PDEs [5] [7].

The experiments were run on the University of Utah Emulab [23] cluster, using up to 81 of the nodes. Each node has an 850MHz single core PentiumIII processor with a 256KB cache, 256MB memory, and they are connected by flat, switched 100Mbps ethernet. All nodes run Linux 2.4.20 with OpenMPI version 1.1.3. All experiments were run on this cluster. The homogeneity of the nodes allowed us to isolate and vary processor heterogeneity. Processor heterogeneity is emulated by timing the critical compute regions and stalling computation by a multiplier of the compute time at the end of these regions. The laggard factor term used in this section is equal to one plus this multiplier. For example, if there is no stall, the laggard factor is one. If there is a stall equal to the compute time, then the laggard factor is two. The experiments vary the laggard factor to measure the effects of varying heterogeneity.

Note that OpenMPI has had more time and resources for tuning than C-DesyncFS. C-DesyncFS has been built and tuned by one programmer over a period of just over one year. OpenMPI is built by a team of programmers and the version used in our experiments is a product of several years' effort and tuning. Therefore, it is possible that some of the performance gap between the two systems is not fundamental, but due to the greater resources applied to the development of OpenMPI.

The experiments are in three groups. In section 5.1, we run on a homogeneous cluster where all processors run at the same rate and all chunks are of uniform size. This investigates the base cost of virtualization across increasing scale. In section 5.2, we run on a heterogeneous cluster, varying the size and shapes of the chunks. This investigates how performance varies when chunks are varied and their size is proportional to processor throughput. In section 5.3 we run on a heterogeneous cluster and evaluate how dynamic load balancing can improve application performance.

5.1 Uniform chunks

DesyncFS virtualizes more than message passing and therefore adds runtime overhead that does not pay off when the system is homogeneous. To investigate the extent of this overhead, we ran an experiment with homogeneity and uniform chunks. Figure 5(a) shows the result over an increasing number of nodes. The ratio of C-DesyncFS to OpenMPI compute time varies little from one node to 81 nodes. The ratio is highest at 9 nodes, with C-DesyncFS taking 13.3% longer. The ratio is lowest at 81 nodes, with C-DesyncFS taking 11.1% longer. This provides an upper bound on the cost of the increased virtualization, for this workload, but it is not clear how much lower this cost can be pushed with better implementation. It is not clear how much of the C-DesyncFS overhead is due to fundamental inefficiencies from the higher levels of abstraction or if OpenMPI is just a tighter implementation. Regardless, our C-DesyncFS implementation costs about 10% more than OpenMPI when there is no heterogeneity.

Increased heterogeneity will reduce the performance of both message passing and DesyncFS, unless DesyncFS load balances to correct the allocation of work. Figures 5(b) and 5(c) show the results of experiments with processor heterogeneity, but uniform C-DesyncFS chunk sizes. The chunk sizes are uniform at four chunks per processor. Figure 5(b) varies the speed of only one of the 16 processors. The ratio of C-DesyncFS to OpenMPI drops monotonically from 1.10 at homogeneity (a laggard factor of one) to 0.71 at a laggard factor of eight. Figure 5(c) varies the speed of 15 of the 16 processors. The ratio of C-DesyncFS to OpenMPI drops monotonically from 1.08 at homogeneity (one across all 15 laggards) to 0.76 when all the laggards have a factor of eight. These two cases are for extreme cases of heterogeneity at opposite ends of the spectrum. In the first case, one increasingly slow procesor. In the second case, one increasingly fast processor. Both OpenMPI and C-DesyncFS take longer, but C-DesyncFS performs better with increasing heterogeneity. This is due to block prefetching. The block dependency callback gives the file system information to prefetch blocks from fast nodes to slow nodes in the background. This reduces the likelihood of a slow node stalling on network I/O waiting for a block from a fast node, helping the slow processors.

5.2 Nonuniform chunks

Figure 6 shows four different chunk mappings; (i) to (iv) and the performance under these different mappings. The benchmark is exactly the same as for the results shown in figures 5(b) and 5(c). However, the cluster is more heterogeneous than in those experiments. Instead of only having slow and fast processors, ten processors have laggard factor two, two nodes have laggard factor one, and the remaining four nodes have laggard factor four.

This replicates the kind of heterogeneity we would expect to see in an organically constructed cluster; a few slow old processors, a few fast ones, and the majority in between.

Note how performance is flat across mappings (ii) to (iv) that preserve area to throughput. In each of these, chunks are assigned to processors so that chunk area is proportionate to processor throughput. The four slowest processors get the small rectangles, the two fastest processors get the big rectangles, and the remaining ten processors get the squares. The bar graph shows that the chunk mapping does not affect throughput. This is good news because it means that a load balancer can assign chunk sizes based on induced processor throughputs without worrying too much about the details of the assignment topology.

The difference between C-DesyncFS and OpenMPI on mapping (i) is due to the prefetching effect, described above in 5.1. The difference from C-DesyncFS on the uniform mapping of (i) to C-DesyncFS on the nonuniform mappings of (ii) to (iv), shows how performance improves by allocating blocks to processors in proportion to their throughput.

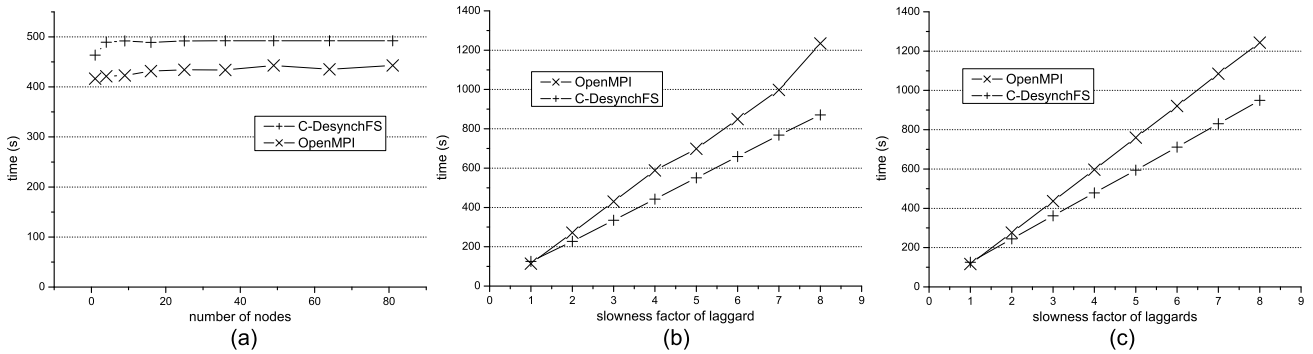

Figure 5. solver benchmark, C-DesyncFS uniform chunks.
These graphs compare the performance of OpenMPI against C-DesyncFS with uniform chunks. All three figures are solver with 16M points per node. C-DesyncFS runs with 4 blocks per chunk. Figure (a) runs for 200 iterations across varying cluster size, while (b) and (c) run for 50 iterations on a cluster of 16 nodes. Figures (b) and (c) show the effects of varying the laggard slowness. Figure (a) has only one laggard while (b) has 15 laggards.

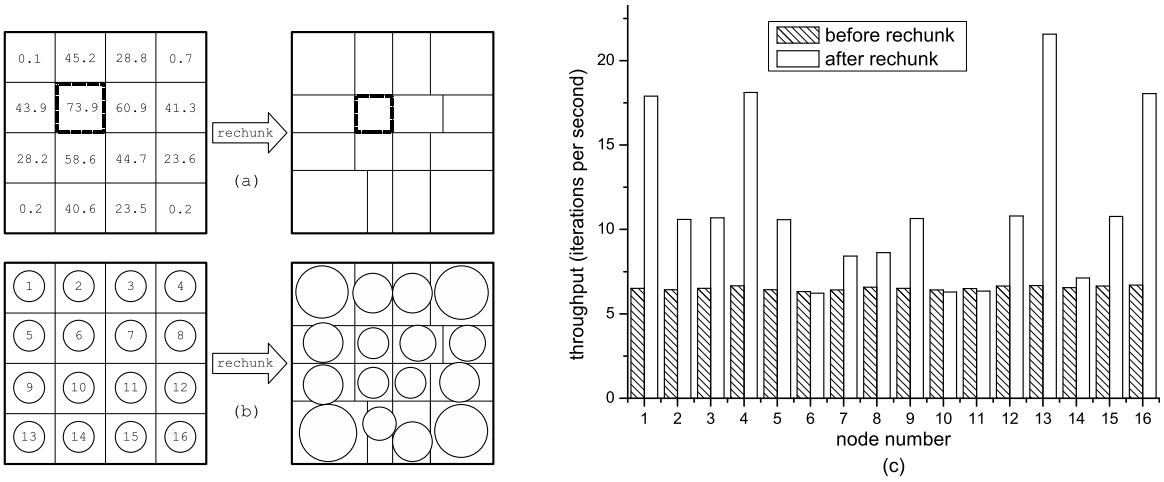

Figure 7. solver benchmark, dynamic chunks for load balancing.
This figure shows the effect of a rechunking on program throughput. There is one slow node with a laggard factor of four. The slow processor (labeled 6 in (b) and highlighted in (a)) has a miss rate of 73.9 before rechunking. The numbers in the left chunk map of (a) are the miss rates before rechunking. The right chunk map of (a) shows the shape of the chunks after rechunking under these miss rates. The left chunk map of figure (b) names the processors with numbers. The area of the circles around each processor corresponds to the throughput of that processor before rechunking. Likewise, the area of each circle in the left mapping of (b) corresponds to the throughput of each processor after rechunking. Figure (c) shows the changed throughput per processor of figure (b) in bar graph form.

5.3 Dynamic adaptation

The user may not know the exact heterogeneity of the machine they are using. In these cases, DesyncFS adapts the computation to the machine for higher throughput. The mechanism for achieving dynamic load balancing is called *rechunking*.

Figure 7 shows how a rechunking can improve application performance. The experiment runs the solver for 300 iterations on 16 nodes where one of the nodes is slow with a laggard factor of 4. The initial layout is naïve. The computation is rechunked once, in the middle, after 150 versions. The first 150 versions take 360s and the final 150 versions take 209s. Therefore, before rechunking, the computation runs at 105 blocks per second and after rechunking, it runs at 180 blocks per second. The rechunking increases throughput by approximately 70%.

The rechunking is agnostic to the underlying cause of the performance imbalance. In fact, the solver benchmark has heterogeneity at the application-level. The data space is flat with boundaries and some blocks need more communication for computation than other blocks. In particular, a corner block needs only two blocks, other edge blocks need only three blocks, and interior blocks need four blocks. A toriodal surface would eliminate this, but often flat surfaces represent something real. Figure 7(a) shows how the rechunking adapts for both network and processor heterogeneity. The corner chunks grow because their processors read and compute less. The rechunking balances for both network and processor heterogeneity.

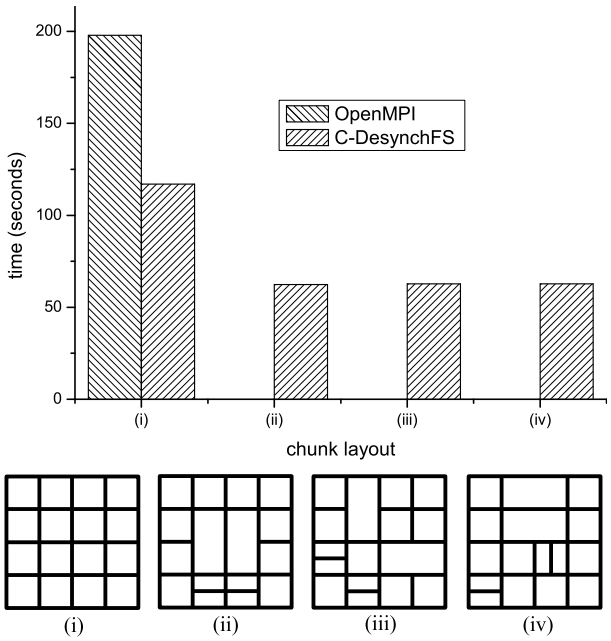

Figure 6. solver, C-DesyncFS nonuniform chunks
This figure shows how performance varies from uniform to nonuniform chunks and for nonuniform chunk mappings of different shapes. Figure (i) shows a uniform chunk allocation. Figures (ii) – (iv) show allocations that preserve area to processor throughput, but with different layouts. These results show how balancing load to throughput increases aggregate throughput under heterogeneity.

6. Conclusions

This paper has introduced desynchronizing file systems, a new abstraction to virtualize processors for high-performance computation on heterogeneous machines. Virtualized processors protect synchronization from slow processors. DesyncFS uses immutable block versioning and stateless operators to give the runtime flexibility to transparently adapt computation.

Our experiments with C-DesyncFS and OpenMPI show that C-DesyncFS adds about 10% overhead across various cluster sizes. With processor heterogeneity, C-DesyncFS performance exceeds OpenMPI, even with naïve chunk allocations. For example, with one slow processor with a laggard factor of four, C-DesyncFS completes the solver benchmark 25% faster. This is due to the runtime's use of dependency information for prefetching. With dynamic load balancing, C-DesyncFS can pull much farther ahead.

A. DesyncFS interfaces

A.1 DesyncFS API

C-DesyncFS is implemented as a runtime library statically linked with the application. The interface is small, only six routines. Figure 9 describes the routines in C. The application calls these routines.

desynchBlockMake. To eliminate memory copies between the file system and application, data is shared by passing pointers rather than copying buffers. Allocating and freeing memory in the same component is a pragmatism of multimodule programming that simplifies memory management. For this reason, `desynchBlockMake` is a simple block factory that wraps heap malloc to keep the details of memory management contained within the C-DesyncFS library.

```
struct range {
    int max;
    int min;
};
typedef struct chunkdesc {
    int numdims;
    struct range *dims;
} chunkdesc;
typedef struct blockaddr {
    int numdims;
    int *dims;
} blockaddr;
```

Figure 8. DesyncFS interface types, in C.
This figure lists the types that describe chunks and block addresses, used in figures 9 and 10.

```
typedef void * rd_handle;
void * desynchBlockMake (size_t blocksize);
int desynchBlockExists (const blockaddr *baddr);
rd_handle desynchBlockRead (const blockaddr *baddr,
    const void **bufp, size_t *lenp);
void desynchBlockWrite (const blockaddr *baddr,
    void *data, size_t len);
int desynchBlockMostRecent (const blockaddr *baddr1,
    blockaddr *baddr2);
void desynchBlockFree (rd_handle dp);
```

Figure 9. C-DesyncFS system call prototypes.
These are the routines used by the application to create and share data.

The application can read and write to any location inside the returned block buffer.

desynchBlockExists. This routine checks if a block exists. If the block does, then the routine returns true (1), otherwise false (0). Block existence can also be checked with `desynchBlockRead` but that routine may fetch the block remotely. This routine will not fetch and is therefore guaranteed to be more lightweight for simple existence checking.

desynchBlockRead. This routine reads a block value. The `bufp` and `lenp` are both out parameters. The block value pointer is assigned into `bufp` and the length is assigned into `lenp`. The routine returns a handle of opaque type `rd_handle`. This handle is later used to free the block.

desynchBlockWrite. This routine binds a block address to a block value. The `data` parameter points to the block value and must be a buffer returned by `desynchBlockMake`.

desynchBlockMostRecent. This routine checks on the most recent version of a block. The first parameter, `baddr1` contains the address of any block. The second parameter, `baddr2` is an out parameter containing the address of the most recent version of this block that exists. The return parameter is true (1) if some version of this block exists. Otherwise, it is false (0).

desynchBlockFree. This routine frees a block reference. This only decrements the number of references. The block will only be unlinked from the file system and its memory freed when two conditions are satisfied. First, all of its forward dependencies exist. Second, it has no references.

A.2 Application interface

The file system calls into the application for information to make scheduling decisions and for computation services. There are only five callbacks; one to compute forward and backward block dependencies, another to compute block values, and the final three work together to form an iterator. All these routines are implemented by the application. Figure 10 describes the routines in C.

```
typedef struct baddrslist {
    int num;
    int maxnum;
    blockaddr *baddrs;
} baddrslist;
int appDepList (const blockaddr *baddr,
    const chunkdesc *file, baddrslist *list, int dir);
void appCompute(const blockaddr *baddr,
    const chunkdesc *file);
void *appIterInit(const chunkdesc *chunk);
int appIterNext (void *iter, blockaddr *baddr);
void appIterDone (void *iter);
```

Figure 10. C-DesyncFS application callback prototypes.
These are the routines the runtime calls to organize and schedule execution. Together, these routines form the application.

appDepList. For a given block and containing file, this routine returns a list of block dependencies. It returns either forward or backward dependencies. If forward, it returns the list of blocks depending on this block. If backward, it returns the list of blocks this block depends on.

There can be insufficient information to compute the block dependencies. This occurs if the dependencies are dynamic and a needed block value does not yet exist. In such a case the function returns false, otherwise true.

There are cases where the application uses the size of the file to treat blocks differently. For example, boundary conditions. For this reason, the file chunk descriptor, `file`, is one of the in-parameters.

appCompute. For a given block, this routine takes a block address and attempts to compute its value. The routine takes the file chunk descriptor for its second parameter for the same reason as `appDepList`. Like dependencies, computations can vary based on the file shape. The computation on an edge of the file may differ from an internal computation.

The compute and dependency functions both need knowledge about the data dependencies; the compute function to read or write the data to fulfill the dependencies and the dependency function to return the block addresses.

appIterInit. This is the first of the three routines that form the chunk iterator. This iterator follows the style of the C iterator idiom. For a given block, this routine takes a chunk descriptor and returns an opaque iterator. This can point to state initialized by the application, given the chunk value.

appIterNext. This takes the iterator state and returns false (0) if iteration is complete. Otherwise, it returns true (1) and the next chunk address in the second parameter. The application modifies the iterator's internal cursor representation to move it forward through the chunk space.

appIterDone. The system calls this routine when it is finished using the iterator. The application cleans up any allocated memory.

Acknowledgments

The experiments described in this paper were performed on the Emulab [23] cluster at the University of Utah.

References

[1] R. Adair, R. Bayles, L. Comeau, and R. Creasy. A virtual machine system for the 360/40. Technical report, IBM Corporation, Cambridge Scientific Center, May 1966.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proc. of 15th ACM SOSP*, December 1995.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of 19th ACM SOSP*, 2003.

[4] P. Braam, 2005. private communication.

[5] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kauffman, 1999.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of 6th ACM/USENIX OSDI*, December 2004.

[7] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[8] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, September 2004.

[9] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel Programming in Linda. In *Proc. of the International Conference on Parallel Processing*, 1985.

[10] G. A. Gibson, D. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Measurement and Modeling of Computer Systems*, 1997.

[11] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. James W. O'Toole. Semantic file systems. In *Proc. of 13th ACM SOSP*, 1991.

[12] R. P. Goldberg. *Architectural principles for virtual computer systems*. Ph.D. thesis, Harvard University, 1972.

[13] R. P. Goldberg. Architecture of virtual machines. In *Proc. of the Workshop on Virtual Computer Systems*. ACM Press, 1973.

[14] A. S. Grimshaw, J. B. Weissman, and W. T. Strayer. Portable run-time support for dynamic object-oriented parallel processing. *ACM Trans. Comput. Syst.*, 14(2), 1996.

[15] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[16] C. Huang, O. Lawlor, and L. V. Kale. Adaptive MPI. In *Proc. of the 16th International Workshop on Languages and Compilers for Parallel Computing*, 2003.

[17] L. V. Kaleé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proc. of OOPSLA'93*, September 1993.

[18] R. Lämmel. Google's MapReduce Programming Model – Revisited. Draft: Online since 2 January 2006; 26 pages, 22 Jan. 2006.

[19] N. Miller, R. Latham, R. Ross, and P. Carns. High Performance I/O: PVFS2 for Clusters. In *ClusterWorld magazine*, April 2004.

[20] M. C. Rinard, D. J. Scales, and M. S. Lam. Heterogeneous parallel programming in jade. In *Proc, of ACM/IEEE Supercomputing*, 1992.

[21] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, 1985.

[22] P. Snyder. TmpFS: A virtual memory file system. In *Proc. of the Autumn 1990 EUUG Conference*, October 1990.

[23] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of 5th ACM/USENIX OSDI*, Dec. 2002.

[24] S. Wolfram. *A New Kind of Science*. Wolfram Media, 2002.