A Load Balancing Framework: Micro to Macro

Wesley Miaw
University of North Carolina, Chapel Hill
Dept. of Computer Science
wesley@cs.unc.edu

January 11, 2004

1 Introduction

The past decade has seen a move away from supercomputer architectures towards collections of powerful microcomputers. This change has been occurring across all industries, from business to government to academia. There have been several reasons for this migration, not the least of which is cost. It is more cost efficient to make use of many small cheap resources than to build and use a single expensive large resource.

However, it is more complicated to coordinate the use of and maximize the efficiency of a group of resources. Load balancing is one concept often used to achieve this goal. With load balancing, jobs are given access to resources (e.g. processing time, bandwidth) out of a pool of resources provided by the system. The objective of load balancing is to maximize the performance of the system. Performance may mean different things depending on the system concerned; minimizing job delay and maximizing resource use are two examples of performance measurements.

This paper presents a general framework describing the concept of load balancing in the context of three different fields of computer science: microprocessor architecture, web server clusters, and grid computing. Each field approaches the problem of load balancing in their own way, but I will show how the underlying structures of all these approaches are consistent with a general framework. Researchers attempting to design new load balancing schemes will benefit from organizing their scheme along the lines described by the framework.

The papers used to present information from each of the three fields are *Hyper-Threading Technology Architecture and Microarchitecture* [16], EquiLoad*: a load balancing policy for clustered web servers* [8], and *On Partitioning Dynamic Adaptive Grid Hierarchies* [13].

Section 2 presents some background knowledge of the three fields covered for readers who are unfamiliar with those areas of work. Section 3 discusses previous load balancing approaches in the three fields and the field-specific issues that must be addressed by any load balancing scheme. Section 4 presents the general load balancing framework, looks at how recent work fits into that framework, and how that work addresses the field-specific issues involved. Section 5 highlights the framework and presents some benefits of designing schemes with the framework in mind.

2 Technology Background

This section provides some background on the technologies used in the general load balancing framework discussion. Section 2.1 describes the basic problem at the root of inefficient processor usage, and looks at the impact caching, context switching, and out-of-order execution have on this problem. Section 2.2 first defines web cluster job requests (i.e. HTTP requests) and the role of server-side caching. Section 2.3 defines the term grid computing, explains its objectives, and also explains how programs can exploit grid technology.

2.1 Microprocessor Architecture

Maximizing the performance and efficiency of a microprocessor centers around ensuring that an instruction is executed each clock cycle. If something is preventing the processor from executing an instruction, then no work will be done that cycle and a gap is introduced into the workflow. This gap is referred to as a bubble. Bubbles are introduced when a processor cannot immediately execute the next instruction of a thread.

To understand why a thread may stall and introduce a bubble, it is important to understand how a process and its threads interact with information. A process is a single executing instance of a program, and may contain many threads that share information. This shared information may be used for communication between threads, but it also includes the program code containing the processor instructions. If for some reason the currently executing thread cannot access the information it needs, the processor cannot execute that thread's next instruction.

For example, the MIPS machine instruction `lw $1, (100)$2` will load into register 1 the data at the address stored in register 2 offset by 100. If the data at that address is located in RAM or on disk, the processor will be idle while the data is fetched. This problem is independent of application level issues such as locks or synchronization, which must be handled by the operating system.

Three approaches used to address this problem and improve processor performance are caching, context switching, and out-of-order execution. Context switching and out-of-order execution are loosely related to load balancing, and are almost like an inverse form of load balancing. Instead of distributing individual jobs (i.e. threads and instructions) over multiple resources, multiple jobs are distributed over a single resource to ensure that resource is always operating at full efficiency. Caching is used to support context switching and out-of-order execution because when the information required by a thread can be retrieved faster, it is easier to keep the processor running at full efficiency.

2.1.1 Caching

The thread executing on a processor may need data in the processor registers to perform a computation. If that data is not currently available in a processor register, the data must be brought in from memory or disk. When this occurs, the processor must suspend execution until the data is fetched. Fetching this data can take an extremely long time with respect to the processor clock speed. Since the processor cannot perform any work during this time, a bubble will be introduced into the workflow and performance will suffer.

Caching data closer to the processor will reduce the delay experienced by the thread while the

data is being fetched. This will in turn reduce the length of workflow bubbles. Caching may also reduce the frequency of workflow bubbles when used in conjunction with out-of-order execution, as described in Section 2.1.3.

Unfortunately, memory capacity and access latency are directly related: the greater the memory capacity, the greater the access latency.

Driving memory at high clock speeds requires fast memory chips and short signal paths. Manufacturing chips for such high performance is expensive because only a small percentage of produced chips can be reliably run at the high speeds required. The high cost and need for short signal paths is the reason faster memory must be used in smaller quantities.

In contrast, magnetic storage can be produced in large quantities. Magnetic storage is orders of magnitude slower than the small caches, but at the same time orders of magnitude greater in terms of storage capacity. A thread requiring data stored on disk will have to wait a very long time, relative to data stored in fast memory, for that data to arrive at the processor.

As a compromise, system architects use several levels of cache memory to keep as much data as close to the processor as possible. A small amount of very fast memory will be placed near the processor, and data will be looked for in this cache first. A second slower, but larger, cache will be looked in next. And so on.

2.1.2 Context Switching

Context switching is one approach used to keep the processor busy during a workflow bubble. The idea is to switch execution from one thread to another. While the first thread is waiting for data, the second thread can execute and keep the processor busy. However, there is an overhead associated with context switching due to processor state management.

When a context switch occurs, the current processor state for one thread must be saved and the processor state for the new thread must be loaded. This processor state contains all of the information relevant to a thread's execution, such as register values and the currently executing instruction. Without this information, the processor will not know where to resume execution of a thread, and it will not know what data was being used by a thread when it was switched out.

If saving and loading the state was not necessary, or could be done at negligible processor clock cycle cost, then context switching would be enough to keep processors busy assuming a sufficient number of threads. Unfortunately, the overhead of context switching is high enough to have a significant impact on performance. The cost can only be lessened by executing many thread instructions between context switches, and this cannot be guaranteed as each thread's work profile[1] is unique.

Context switching also has a negative effect on cache performance. In the majority of cases, the new active thread will reference different data than the previous thread. Since the cache now contains data for the previous thread, the new thread will experience many cache misses. When the next context switch occurs, the cache will again contain data unrelated to the new active thread. Thus, although the goal of context switching is to avoid situations involving lengthly workflow bubbles,

[1] A thread's work profile describes its execution behavior. I/O intensive threads are characterized by short bursts of computation between long periods of data retrieval. CPU intensive threads will be the opposite with short bursts of I/O between long periods of computation. Most threads fall somewhere in between these two extremes and will behave differently at different points in time.

context switching itself may introduce bubbles due to cache misses.

2.1.3 Out-of-order Execution

Another approach used to keep processors busy while waiting for data is out-of-order execution. With out-of-order execution, the instructions of a single thread are reordered so that while one instruction is waiting for data, other instructions that do not need that data can be executed. This is only possible when it is known that those other instructions are independent of the waiting instruction.

Two instructions are independent of each other if the register written in one instruction is not used by the second instruction. For example, the sequence of instructions `x = y * 3` and `z = y + 5` are independent because the first does not reference `z` and the second does not reference `x`. Therefore, it does not matter in what order the two instructions are executed. However, the sequence of instructions `x = y * 3` and `z = x + 5` are dependent, and changing the order of execution will result in a different value for `z`.

Caching helps when reordering instructions because the sooner an instruction can retrieve the data it requires, the sooner a dependent instruction can be executed. If it only takes one clock cycle to fetch cached data, only one independent instruction needs to be executed to avoid a workflow bubble. On the other hand, if it takes one hundred clock cycles to fetch the required data, one hundred instructions independent of the waiting instruction must be executed to avoid the bubble.

The problem with out-of-order execution is that most program instructions are dependent on previous program instructions, and register values will differ depending on how you reach an instruction. For example, the following code involves a branch:

if `y = 5` *then* {`x = a; y = b`} *else* {`x = b; p = 7`}; `z = x * y;`

If the first branch is taken, `z = a * b`. If the second branch is taken, `z = b * 5`. Branch prediction is something the processor does to try and predict if the first or second branch will be executed. If the processor's prediction is correct, it may be able to execute instructions out-of-order. In this case, the instruction `p = 7` could be executed independent of the other instructions, but only if the processor correctly predicts the second branch.

Out-of-order execution can be used to improve performance by preventing potential workflow bubbles, but its efficacy will be limited by problems like instruction dependencies and incorrect branch prediction.

2.2 Web Servers

Web server performance is measured in requests satisfied per second and request response time. The more requests a server can process each second, the more users the service can support. Shorter response times mean a higher quality of service. In order to understand how servers process requests, it is important to understand a little bit about the HTTP protocol and document caching.

2.2.1 HTTP Requests

A web client, such as a browser, sends requests to a web server using the HTTP protocol. A typical request contains the path and filename of a document on the server. The contents of that document are sent back to the client. In the case of HTML documents, references to additional documents, such as images, may be included in the content of the requested document. The client will then also request the referenced documents.

The request process described above is a simplistic view of what may actually happen when an HTTP request is serviced. Although everything appears to be a document request to the client, in many cases the document does not exist but is instead dynamically constructed in response to the request. Constructing the requested content may require accessing data from many locations and performing expensive computations. Complicated situations like this obviously require more time and resources and will have a negative impact on web server performance.

2.2.2 Server-side Caching

Regardless of whether a document is dynamically constructed or not, caching the data used to generate the document can have a dramatic improvement on response time and requests per second. Rather than a server retrieving data from slow disk or across the network, data may be kept in memory on the server for swift access. Keeping the commonly requested data items in memory means the majority of requests can be processed without going to disk or the network.

The problem with caching is of course the cache size. Fast memory is orders of magnitude more expensive than comparably sized disks, and smaller amounts of memory are more quickly addressed by the computer. In addition, the majority of web servers today use 32-bit processors which limits memory address space size to 4GB. The recent push towards 64-bit processors increases the address space size, but hardware is still incapable of supporting the several hundred gigabytes, or terabytes, that a 64-bit address space could reference.

2.3 Grid Computing

Grid computing is a relatively new concept that has grown out of widely-distributed research programs such as SETI@home [14], a program that makes use of volunteered computing resources to analyze data in the Search for Extraterrestrial Intelligence. In this case, that data consists of radio signals recorded by the Arecibo dish radio telescope. Rather than using a single, expensive supercomputer to quickly analyze the gathered data, SETI uses thousands of inexpensive computers to analyze independent chunks of the data. The computation time of these computers is donated by people from all over the world who are interested in SETI.

The general idea behind grid computing, implemented by many programs like SETI@home, is to aggregate distributed computational resources and present them as a single service with a common interface. The grid itself is a collection of resources connected by a network and made available to users and programs. On a small scale, a grid may consist of dozens of workstations at a single physical location. A large grid, on the other hand, may be constructed of several smaller grids or clusters at different organizations from across the globe [17].

The performance of a grid itself is somewhat hard to quantify, as performance also depends on

the specific client application. However, any application will generally try to minimize the total wall clock computation time by making efficient resource decisions. In order to understand how wall clock computation time is minimized, one needs to understand how grids differ from other architectures in terms of resources and parallelism.

2.3.1 Computational Resource Sharing

Grid computing is different from clusters or other forms of computer resource sharing and aggregation because a grid computing infrastructure is more arbitrary and ad hoc. Traditional approaches to building a computing resource infrastructure involve specifying hardware, software, and communication requirements. Any such infrastructure is targeted at a specific application or service, and typically under the control of a single entity.

In contrast, the resources used in grid computing may span across many organizations and may be geographically separated. Different portions of the grid will be under the control of different organizations who make their resources available, as a service, to users of the grid.

Access to resources is provided by implementing a grid service interface. Users of the grid must program to that interface to exploit the grid's resources. Globus [10] is one of the more popular grid development environments. The Globus Toolkit provides the framework for resource monitoring, discovery, and management by providing features such as SOAP[2] message security, OGSI[3] protocols, and file transfer.

Since the resources made available by a grid are of such an ad hoc and arbitrary nature, it is necessary to dynamically allocate and manage the available resources based on service requirements and client application needs. However, unlike clusters, each node is independently responsible for managing its own resources. This makes resource allocation and management much more difficult, as there is no central coordination point able to make informed decisions.

2.3.2 Distributed Process Parallelism

Given access to grid resources, the issue then becomes how to optimize computation parallelism. The grid itself has no real knowledge of the applications it will execute. It is the responsibility of the client application to break computations into independent chunks suitable for processing on grid nodes. The grid software will then distribute those computations across the many nodes, in the manner requested by the application.

A real example of process parallelism can be seen in how the NC BioGrid [6] uses NCBI BLAST, a gene sequencing analysis program [7]. Jobs are distributed across the grid by copying the BLAST executable and a gene sequence input file to the nodes. Each node receives a different input file with data independent of all other input files. After a node completes its analysis and returns its results, a new job is sent to the free node.

[2]SOAP is an open XML-based protocol for exchanging information in a decentralized and distributed environment. [15]

[3]OGSI, or the Open Grid Services Infrastructure, provides the features required by a grid service. Some of these features are service invocation and security interfaces. [11]

3 Previous Load Balancing Approaches

Before presenting the load balancing framework, it is important to look at current approaches to load balancing in the three areas this paper is concerned with. The approaches described below are thread-level parallelism in microprocessors, clustered web servers, and manual grid partitioning. While these approaches are important and do have a positive impact on performance, there is still room for improvement.

3.1 Microprocessor Thread-level Parallelism

Thread-level parallelism is the practice of executing multiple threads at the same time. Three different methods used to support thread-level parallelism are multiprocessor systems, chip multiprocessing, and time-slice multiprocessing. In the first two methods, job distribution is fairly simple: individual threads are multiplexed over multiple physical processor resources. The third method ends up separating many threads into discrete blocks of instructions, and multiplexing those blocks onto a single processor.

3.1.1 Multiprocessor Systems

Multiprocessor systems are systems with more than one physical processor. Making use of more physical processors allows a single system to execute multiple threads at the same time. However, supporting multiple processors does require special hardware support. The processor itself may have to undergo minor design changes to support the notion of shared memory. Other changes include additional processor connection pins on the mainboard, communication chips with support for multiple buses and processors, and bus architectures that are shared or duplicated.

All of that means a multiprocessor system will be more expensive and more complex than a single processor system. Overall system performance will improve since multiple threads can execute in parallel, but at the individual processor level, the hardware is not being used more efficiently. Using multiple physical processors does not address the problem of workflow bubbles or reduce the cost of context switching. In fact, the coordination of multiple processors introduces additional overhead so that doubling the number of physical processors will not double the performance.

3.1.2 Chip Multiprocessing

Chip multiprocessing is an attempt to reap the benefits of a multiprocessor system at a reduced cost. Rather than increasing the number of physical processors in a system and duplicating mainboard resources, something that comes at some cost, chip multiprocessing places two microprocessors on a single die. Hardware requirements for chip multiprocessing are similar but less extensive than for multiprocessors, as the hardware must still be aware of the atypical processor configuration, but the two processors share a single connection point to the mainboard.

While chip multiprocessing is arguably a better approach to increasing performance per processing chip than full multiprocessor systems, the cost of designing, fabricating, and using these chips is still higher than that of traditional single processor systems. The single connection point is a performance bottleneck since communication through this point must be shared and managed.

Plus, the problems found in multiprocessor systems and single processor systems, such as workflow bubbles and context switching overhead, are still present under this architecture.

3.1.3 Time-slice Multithreading

Time-slice multithreading approaches the problem of efficiency in a different manner than multiprocessor systems and chip multiprocessing. Rather than changing hardware to improve performance, time-slice multithreading changes software to make better use of available resources.

The idea is to force a context switch between threads at fixed time intervals and also whenever an event occurs that will result in long latencies, such as a cache miss. The benefits and cost of using time-slice multithreading (which is used by almost all modern systems both for efficiency and multiprogramming[4]) are discussed in Section 2.1.2.

3.2 Clustered Web Servers

Clustered web servers group several hosts together to collectively satisfy more requests per second than could be satisfied by a single host. This clustering is transparent to a web client, who is unaware of the cluster and sees only a single host. This illusion is necessary so that the client does not need to be aware of all the participating hosts when sending requests. It also allows an organization to control request distribution to reach a desired level of performance and quality of service.

3.2.1 Multiple Hosts as One

To make the entire cluster function as a single web server, it is necessary to replicate applications and data across all hosts. Each host must run a copy of any applications a client may interact with so a client will not be bound to a specific host when sending requests. This replication scheme may appear at many levels. For example, an application may appear as a single server, but in reality sales functionality might be handled by one set of hosts, while customer support functionality is handled by a different set.

Implementing a replication scheme means that the data needed to process a request must be available to any host. In some cases this is done by replicating the data on each host, just as the applications are replicated. Another option is to keep data in a central repository shared by all hosts, such as a database server. For static data, managing the data is not difficult regardless of the approach. However, if client requests may change data, then changes must be synchronized across all hosts.

Sometimes, it is possible to avoid the complications of mutable data by permanently associating a client with a specific host in the cluster. This is only possible if the mutable data is only relevant to one client and does not need to be shared with the other hosts. If the mutable data does impact behavior for other clients, then this association is not possible. Permanently associating clients

[4]Multiprogramming is a rudimentary form of parallel processing in which several programs are run at the same time on a uniprocessor. Since there is only one processor, there can be no true simultaneous execution of different programs. Instead, the operating system executes part of one program, then part of another, and so on. To the user it appears that all programs are executing at the same time. [12]

with specific hosts sometimes causes a problem for load balancing since it may be very difficult or impossible to accurately predict the load generated by one client. If load cannot be predicted, established associations may end up overloading some hosts while other hosts are left idle.

3.2.2 Multiplexing Requests

Ensuring load is evenly distributed across the cluster's hosts is important for maximizing the performance of a distributed web server. Any distribution scheme must first evaluate the resource requirements of a request, and second identify the best host for processing that request. It is the responsibility of the distributor to use these two bits of information to maximize the cluster's performance.

A distributor is a system that sits between the clients and the cluster and is responsible for associating incoming requests with cluster hosts. Sometimes the distributor truly hides the cluster from client applications by acting as a proxy[5]. Other times, the distributor will associate a client's request to a specific host, and it is the client's responsibility to directly communicate with the assigned host.

There are three primary types of request distribution: DNS-based, dispatcher-based, and server-based [9].

DNS-based This distribution scheme places the responsibility of distributing requests on the DNS server for the cluster hosts. Each time a client requests the IP address of the web server, the DNS server returns one of many possible IP addresses, choosing which host it would like the client to send requests to. Five different clients requesting the IP address of www.cs.unc.edu may be given five different IP addresses. Since the client and the client's local DNS server will cache this returned IP address, this effectively associates that client with that physical host until the cached value expires.

Although the DNS server can be smart about picking the next IP address to return based on its knowledge of current server load, it has no knowledge of the requests the client will end up sending to that address. Even worse, once the DNS server returns an IP address, it has no control over future use of the address since it will be cached at the client and possibly at intermediate servers.

Dispatcher-based Where DNS-based dispatching schemes make use of many IP addresses, a dispatcher-based scheme publicizes only one IP address: the address of the dispatcher. Clients send all requests to the dispatcher, which forwards them to the cluster hosts for processing. The dispatcher acts as a proxy for the cluster.

This proxy scheme must address the issues of connection management and data forwarding. Since all connections between clients and the cluster hosts run through the dispatcher, state must be maintained for the duration of communication. This will limit the number of concurrent connections because there are only so many port numbers available for use by the dispatcher. Likewise, since all data must be forwarded through the dispatcher, it may become a bottleneck in the network. Managing connection state and forwarding data does introduce some overhead.

[5] A proxy manages communication between two systems.

Regardless of those problems, a dispatcher-based scheme does provide an organization with more control than a DNS-based scheme because each request may be managed individually. The dispatcher also has more knowledge than a DNS server because it can have knowledge of both current server load and of the requests, since all requests are forwarded through the dispatcher.

Server-based A server-based distribution scheme is a combination of the DNS-based and dispatcher-based schemes. As with the DNS-based scheme, clients are initially given the IP address of one host in the cluster, but any cluster host may redirect incoming requests to a different host for processing.

Under this scheme, the DNS server may be very ignorant of the cluster servers and requests when returning IP addresses. The cluster hosts, who have knowledge of their current load and the request, and possibly knowledge of other hosts' load, can redirect requests as necessary.

3.3 Grid Partitioning Schemes

Grid computing is a relatively new area of research, and as such there has not been a lot of established work on grid partitioning schemes. Given a set of available grid nodes with different levels of computational power, the problem is to figure out how jobs should be distributed over the grid nodes to minimize the total parallel computation time. Minimizing total computation time also means minimizing the overhead of communication and synchronization between nodes.

The majority of current grid applications are manually partitioned, placing the burden of maximizing performance and minimizing computation time on the developer. Manual partitioning obviously introduces problems. The resulting code will be specific to one grid infrastructure, and will only be as optimized as the developers are able to make it. More experienced or knowledgeable developers may be able to better optimize or use different optimization approaches. Manual partitioning also cannot adapt to changes in the grid infrastructure or changes in grid resource availability during runtime.

4 A Load Balancing Framework

Previous approaches to load balancing microprocessor resources, distributed web servers, and grids give us some insight into what functions a general load balancing framework must provide to maximize performance and minimize cost. These functions are job partitioning, job distribution, resource management, and a client interface.

Microprocessor thread-level parallelism partitions jobs by thread, distributes threads across the available resources, includes new logic for managing the resources, and provides physical pathways and hardware instructions for use by the operating system. Clustered web servers are similarly defined, with HTTP requests instead of threads, physical servers for its resources, and a dispatcher as its interface.

Currently there are no established load balancing techniques for grid technology, and client applications and grids depend on customized algorithms for executing work units. However, the proposed framework can be applied to grid computing to create a load balanced grid. Such a load balancing scheme is described by Dynamic Adaptive Grid Partitioning.

This framework outlines the minimal functional requirements that any load balancing scheme must address. Without accurate and efficient job partitioning and distribution, the benefits of load balancing cannot be maximized and in some cases attempting to load balance with poor job management will have a negative impact on overall system performance. The same applies to management of the physical resources used by the system. The last consideration, a client's interface with the load balanced system, does not have as much to do with performance as it does with usability. if a load balanced system is not easy to use, it will have limited application and users will be slow to adopt it.

4.1 Job Partitioning

Classifying job requests is important for grouping jobs into different groups representative of the type and amount of resources the job requires. At one extreme, each job can be associated with the specific set and quantity of resources it needs to complete. However, such fine-grained partitioning can introduce prohibitive overhead and may itself require a significant amount of dedicated resources. At the other extreme, all jobs can be placed into the same group. This will basically result in random distribution since no job is seen as any different from another, and the benefits of load balancing cannot be exploited to their full effect.

Hyper-threading, EquiLoad, and Dynamic Adaptive Grid Partitioning partition job requests in increasingly complex ways.

4.1.1 Hyper-threading Partitioning

A hyper-threading capable processor simply treats each thread individually, and performs no special grouping of threads. Multiple threads are then executed simultaneously on a single physical processor without performing any context switching. Since the processor resources are shared among the multiple threads, this is similar to chip multiprocessing with the benefits of time-slice multithreading and without the costs associated with context switching.

4.1.2 EquiLoad Partitioning

EquiLoad is a dispatcher-based scheme for distributed web servers that assumes document content size is highly correlated with the request processing time, regardless of the specific resources required to generate the content. This assumption is based on research in web server workload characterization[6]. Given knowledge of the response size distribution, EquiLoad will partition the the response size histogram into intervals of equal load. Each interval is assigned to one of the cluster hosts.

This partitioning is illustrated in Figure 1, which is based on workload traces from the 1998 World Soccer Cup web site[7]. In this example, there are four hosts in the cluster. Since it is assumed that the load generated by a request is highly correlated with the response size, the goal is to partition the requests into four intervals that will generate equal total load. This is done using phase-type distributions. A phase-type distribution represents random variables through a mixture

[6]The EquiLoad paper cites [1, 2, 3, 4].

[7]Data available at http://researchsmp2.cc.vt.edu/cgi-bin/reposit/search.pl?details=YES&detailsoffset=135

of exponentials and is consistent with a Markov chain. (It is the exponential nature that results in the curves seen in Figure 1.) A Markov chain is a sequence of random values where the probability of a value at a time t depends on the value at time *t-1*. Each of the resulting intervals is assigned to one of the four hosts. All requests that generate a response size in interval i are satisfied by host i.

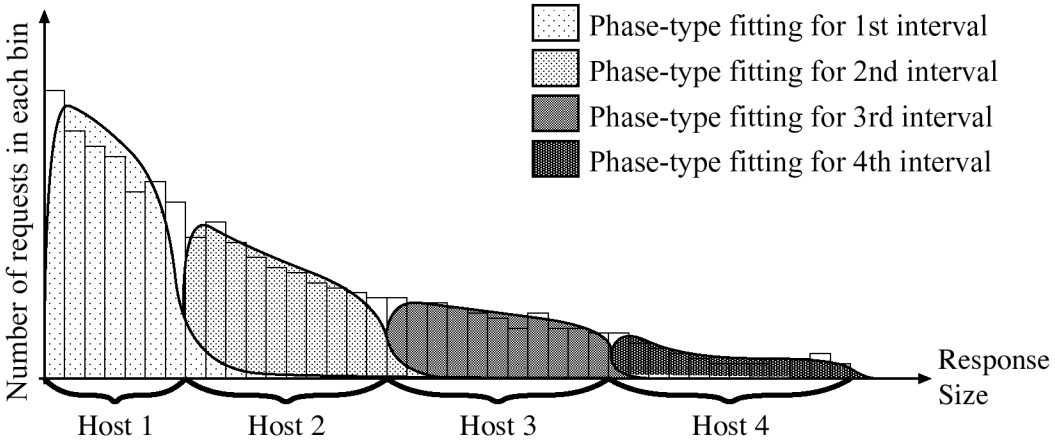


Figure 1: Fitting N = 4 intervals of the size distribution with individual phase-type distributions.

This behavior introduces some request size homogeneity into the request queues and will improve response time, but this partitioning scheme does present some problems. Choosing the partition boundaries is important because bad choices will create unbalanced load on the cluster servers. Additionally, as workload characteristics change over time, the partitions must be changed to match. Ensuring that partitions are up-to-date and accurately represent equal load can be very difficult.

4.1.3 Dynamic Adaptive Grid Partitioning

This grid partitioning scheme, DAGP, represents a multi-dimensional grid by a one-dimensional recursive ordered index space. In simpler terms, a two-dimensional map representing an arbitrary organization of the grid would be partitioned into squares of equal area. The partitions are then numbered 1 to n, where n is the total number of partitions. If a partition requires additional refinement because it is capable of performing additional computations in a single time step, then this process is repeated within that partition. A partition will be marked for refinement or for recombination with its neighbors if its performance falls outside some margin of error. The numbered partitions are then listed in some order.

Figures 2 and 3 illustrate the conversion from a two-dimensional representation of a grid to its corresponding one-dimensional representation. Recall that the two-dimensional map of grid nodes is some arbitrary arrangement that may or may not have a relationship to the nodes physical or logical locations in the grid. The two-dimensional map shown in Figure 3 was first partitioned into sixteen squares enumerated 0 through 15. The four center squares were further partitioned and recursively enumerated 0 through 15. This refinement means those center squares are contributing more resources to the computation than the unrefined edge squares.

Once the two-dimensional representation has been partitioned, those partitions must be rearranged into a one-dimensional representation. There are many possible ways to order the 28 final partitions in the one-dimensional representation. Morton order and Peano-Hilbert order are two space-filling curves[8] that might be used to construct the ordering. These space-filling curves are shown applied over a 4x4 map in Figure 2.

Applying the Morton order curve to the partitions in Figure 3 builds the one-dimensional linear representation {0 1 4 5 2 3 6 7 8 9 12 13 10 11 14 15}. The four refined partitions have the Morton order curve applied to them as well, and the resulting order is recursively inserted into the above representation. The final one-dimensional linear representation is {0 1 4 {0 1 4 5} 2 3 {2 3 6 7} 7 8 {8 9 12 13} 12 13 {10 11 14 15} 11 14 15}.

Applying the Peano-Hilbert order is done in a similar fashion, and results in the one-dimensional linear representation {0 1 {0 1 5 4} 4 8 12 13 {8 12 13 9 10 14 11 15} 14 15 11 7 {7 6 2 3} 2 3}.

Once the grid has been partitioned, jobs can be distributed over the grid by assigning a computation to each partition (i.e. by assigning a computation to each 'number' in the one-dimensional representation).

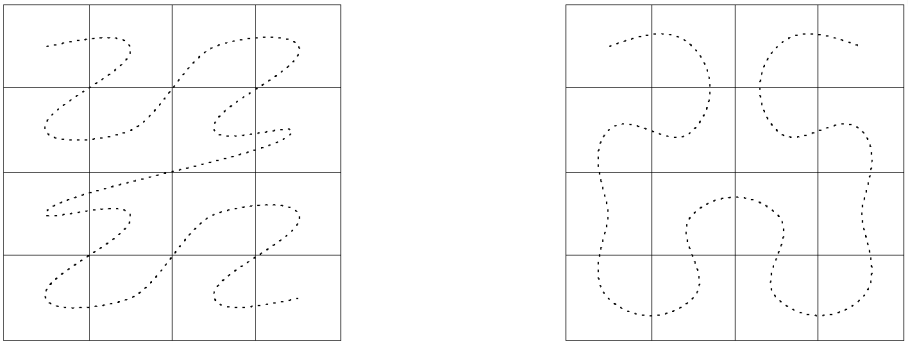


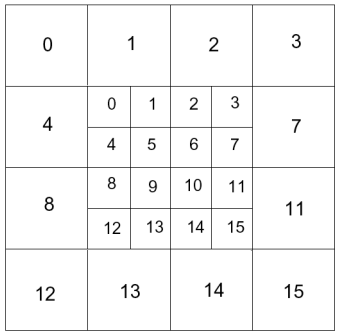
Figure 2: Two possible space-filling curves.

4.1.4 Job Partitioning Analysis

There is very little involved in partitioning jobs for hyper-threading because threads naturally lend themselves to the idea of a job. This is partially because the operating system already views threads as jobs and can distribute them over physical processors. But also because hardware has very limited knowledge of the task it is trying to complete since its view of threads is so low-level (i.e. at the instruction level). If the processor hardware tried to maintain more detailed information about the threads, the added overhead and circuitry would impair performance.

EquiLoad, on the other hand, does maintain information about its requests by continually monitoring and adjusting its partitions. This is possible because EquiLoad has a high-level view of all the requests coming in and all of the responses being returned. Unlike hyper-threading, where

[8]Space-filling curves perform a one-to-one locality-preserving mapping from multi-dimensional space to one-dimensional space.



{0 1 4 {0 1 4 5} 2 3 {2 3 6 7} 7 8 {8 9 12 13} 12 13 {10 11 14 15} 11 14 15} (Morton)

{0 1 {0 1 5 4} 4 8 12 13 {8 12 13 9 10 14 11 15} 14 15 11 7 {7 6 2 3} 2 3} (Peano-Hilbert)

Figure 3: Composite representation of space-filling curves over the DAGH.

the hardware forgets a thread as soon as it is switched out, EquiLoad can remember everything it has previously seen. All of this state is much easier and cheaper to manage in software than in hardware.

DAGP has the the most complicated partitioning management system due to the distributed and unpredictable nature of the grid and its applications. After each time increment in the parallel computation, a regriding process is done where partitions are refined or rejoined to update DAGP's knowledge and partitioning of the grid. The decision to refine or rejoin partitions will be based on whatever criteria is applicable to the computation. Example criteria might be total computation time or total inter-node data communication. A partition that was refined in a previous time increment may now need to be rejoined at the current time increment, and vice versa. This might result from changes in grid hardware, other computations competing for the same resources, or even something as basic as fluctuations in communication bandwidth and delay.

Any load balancing scheme must have some approach towards job partitioning to maximize performance when distributing jobs over the available resources. The complexity of job partitioning depends a great deal on the variability and range of job requirements and also on the knowledge available to the partitioning technique. Hyper-threading, EquiLoad, and DAGP partitioning are increasingly complex for this reason. Threads and machine instructions are the most simple and homogeneous types of jobs, while DAGP computations can vary a great deal. Processor hardware has extremely limited job knowledge and limited memory capabilities. DAGP has a very high-level view of both the grid and the impact a computation has on grid resource use.

Besides the complexity of job partitioning, there is also a question of how often to repartition. Hyper-threading always places each job, or thread, into its own partition and so never really undergoes any sort of partitioning process. DAGP performs its repartitioning after each time increment. There is, however, a lot of choice as to when EquiLoad may repartition its request to response size histogram. Experiments with the World Cup 1998 trace data showed the best performance when limiting the histogram data to the previous day. But I would argue this result has a lot to do with the event-driven nature of requests and time-limited relevance of the site's content.

4.2 Job Distribution

Given a method of identifying job resource requirements, the next step is to distribute jobs over the available resources in the most efficient manner, maximizing the performance of the system. The easiest form of distribution is random or round-robin. If the variability of job resource requirements is very low, this may not be a bad approach since there is very little overhead associated with a random distribution implementation. However, when job resource requirement variability is high, and covers a wide range, then random or round-robin distribution can easily create load imbalances. This will reduce the system's efficiency and performance.

The distribution approaches used by hyper-threading, EquiLoad, and DAGP are increasing complex. This is related to the complexity of job partitioning because the number and characteristics of partitions are used to make distribution decisions.

4.2.1 Hyper-threading Distribution

Since hyper-threading does not attempt to group threads together into partitions, it uses round-robin to distribute jobs. The steps of this process are enumerated below and illustrated in Figure 4. The two threads executing in parallel on this hyper-threading processor are distinguished by the colors yellow and red. Limiting the maximum number of resources used by each of the two logical processors ensures fairness and prevents deadlock.

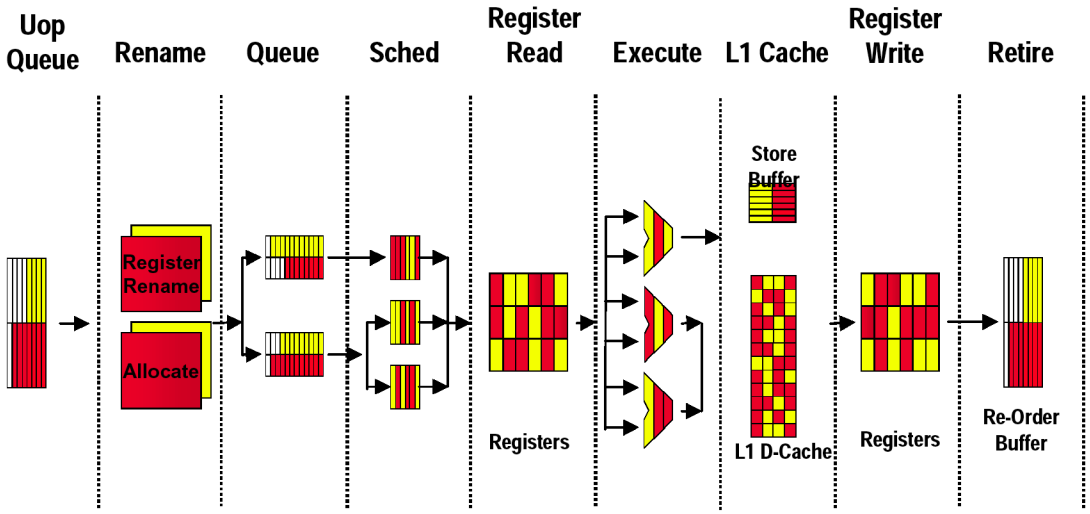


Figure 4: Hyper-threading execution engine pipeline.

1. Each thread is allocated half of the uop queue, which stores the decoded machine instructions for execution. Example uops are add, load, or store.

2. The allocator is responsible for allocating buffers needed to execute a uop, and will select uops from the queue by alternating between threads. The register rename logic, which maps the registers referenced by IA-32 instructions to physical registers, executes in parallel on the same uop as the allocator. Each logical processor uses its own Register Alias Table to perform the mapping.

3. The two instruction queues (one for memory operations and one for all other operations) are also partitioned into halves: one half for each logical processor.

4. Instructions are pumped from the instruction queues into the multiple schedulers as quickly as possible, alternating between the two logical processors. The schedulers manage out-of-order execution by deciding when uops are ready to execute, based on memory dependencies and resource requirements.

5. The next four sections (register read, execute, L1 cache, and register write) perform the actual uop.

6. After execution completes, the uop is placed into the re-order buffer for use in the retirement stage. Retirement reorders uops that were executed out-of-order back into the original program order. The retirement logic alternates between logical processors, retiring uops for one thread first, then doing the same for the other thread.

4.2.2 EquiLoad Distribution

The distribution scheme proposed by EquiLoad is a two-stage process that adapts over time to changes in the requested document size histogram. The goal is to forward the documents in each phase-type fitting (see Figure 1) to a different cluster host. In Figure 5, the dispatcher attempts to determine which back-end host an incoming request belongs to. If the dispatcher is unable to make this decision, then the request is forwarded to a random host. If the randomly chosen host was the incorrect choice for the request, that host will redirect the request to the correct host. This is possible because the document size information will be available to the back-end host.

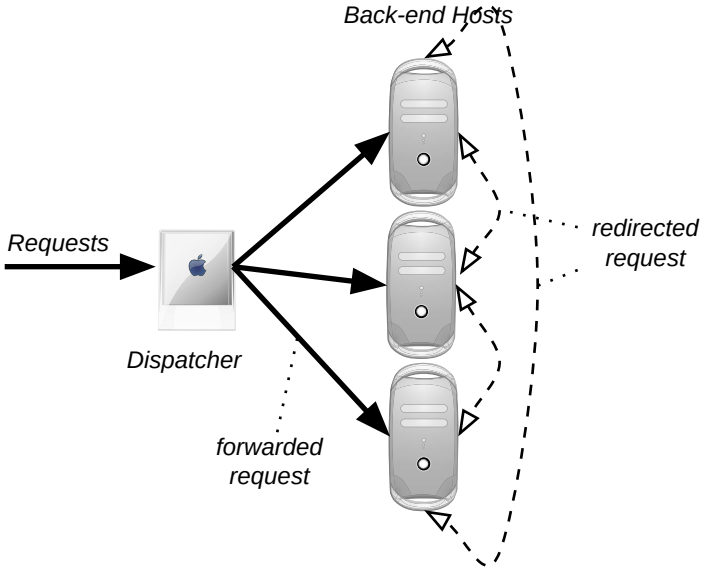


Figure 5: Model of a clustered web server using EquiLoad.

This two-stage process provides some important advantages:

- The dispatcher does not need to know the request sizes or the current load on cluster hosts.
- Requests will need to be redirected at most once, and only if the dispatcher does not have knowledge of the request size or its knowledge is incorrect.
- Cache behavior will be close to optimal since the same requests will be processed by the same cluster hosts.

4.2.3 Dynamic Adaptive Grid Partitioning Distribution

A process called adaptive mesh refinement (AMR) is used to create the grid hierarchy of computational regions. As mentioned earlier, the goal is to partition the grid into units over which the parallel computation can be distributed. Using an AMR process designed by Berger & Oliger [5], we begin with a single partition G_1^0. As the computation proceeds, regions of the grid will be further partitioned or rejoined to build a tree hierarchy and its corresponding grid partitioning like the one shown in Figure 6. The partition label G_j^i indicates that the partition represents a i refinements and is the jth subgrid at that level of refinement. Partitions at the same refinement level are siblings in the tree.

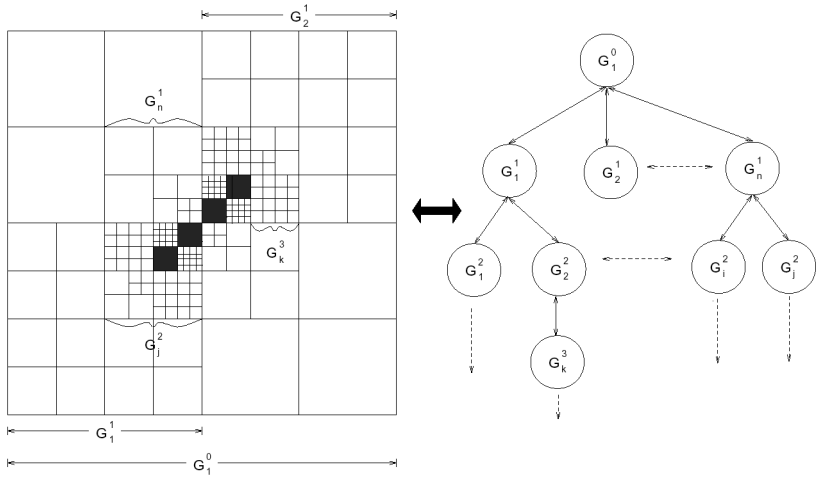


Figure 6: Adaptive grid hierarchy constructed with the Berger-Oliger AMR scheme.

The goals of the grid hierarchy are to expose data-parallelism[9], to minimize communication overhead, to balance load, and to enable reparitioning with minimal overhead. Data-parallelism is exploited by maximizing data reference locality within the grid. Communication overhead is minimized in the same manner, by arranging for communication within local areas of the grid.

The regriding operation is executed on the one-dimensional linear Dynamic Adaptive Grid Hierarchy (DAGH) representation described in Section 4.1.3 as follows:

1. Each processor refines its portion of the linear space in the DAGH.

[9]Data-parallelism refers to parallel computations with respect to data structures, rather than with respect to the computations. For example: performing the same operation on two halves of a structure versus performing two different operations on the entire structure.

2. The refined linear space is communicated to all processors, and the new DAGH is built.

3. Each processor determines which portion of the new DAGH it is responsible for.

4. Data is moved between processors to match the new hierarchy.

4.2.4 Job Distribution Analysis

As with job partitioning, job distribution is the least complex for hyper-threading and the most complex of DAGP. In many ways, this is a direct result of the complexity of job partitioning.

Hyper-threading has very little by way of job partitioning, so the distribution scheme can be very simple: in this case round-robin. If hyper-threading attempted more intelligent partitioning, perhaps by criteria like shared data references, then its distribution scheme could take that into account.

Both EquiLoad and DAGP require continuous monitoring of the system's performance and workload characteristics to dynamically optimize their job partitioning and distribution policies. Between the two, the job load and communication characteristics associated with DAGP cover a much wider range with greater variability. Correspondingly, DAGP's distribution scheme is more complicated. In addition, EquiLoad gets around the dispatcher's lack of knowledge by allowing the back-end hosts to make independent decisions and communicate among themselves. This is not possible in a grid environment because smart distribution requires global knowledge of both the grid and the parallel computation. A single grid host does not have this knowledge. The highly distributed nature of the grid also means any two-level approach towards job distribution could easily introduce high communication overhead.

4.3 Resource Management

Of course, any distribution scheme needs a set of resources over which load will be distributed. Simply duplicating or increasing the amount of available resources does not guarantee improved performance. Poor or inefficient management of resources can in fact degrade overall system performance below that of a system with fewer resources. Minimizing the overhead associated with managing and employing system resources will maximize each resource's contribution to the system performance.

Unlike job partitioning and framework, the resource management is the most complex with hyper-threading and the least complex with dynamic adaptive grid partitioning. This is due to how resources are added to and used in the different systems.

4.3.1 Hyper-threading Resources

Perhaps the reason resource management is most complicated for hyper-threading is because by its nature, hyper-threading is a solution that almost entirely resides in the hardware domain. Very little modification is required on the software end, but the necessary modifications to hardware are fairly extensive and require duplication or sharing of many hardware resources.

Some of the resources duplicated for each logical processor are the interrupt controller; the ITLB[10] and DTLB[11]; and instruction pointers for the trace cache[12] and TLBs. Shared resources include caches, execution units, branch predictors, and queues. The bus is shared and bus requests are served on a first-come-first-served basis.

Processor architecture state is also duplicated. Each logical processor maintains its own version of the architecture state, just as a non-hyper-threading physical processor would maintain its state. The information stored in the architecture state consists of all the registers, and this information is presented to the operating system to provide the illusion of multiple physical processors.

The processor can also operate in three different modes to optimize the use of shared resources. When both logical processors are being used for two different threads, the processor is in MT-mode where all resources are shared. In MT-mode, the processor executes with hyper-threading enabled as described above. When there is only one active thread, the processor can execute in ST0-mode or ST1-mode instead. In one of the STX-modes, the processor behaves exactly as if without hyper-threading. This allows a single thread to execute with maximum performance.

4.3.2 EquiLoad Resources

Constructing an EquiLoad distributed web server requires very little specialized effort since all special considerations can be addressed in software. The setup is almost identical to existing dispatcher-based distributed web servers, where several back-end servers form a cluster with a front-end distribution server. The front-end distribution server will use the EquiLoad distribution algorithm, and the back-end servers must be modified to redirect requests that belong to a different server.

Swapping in EquiLoad on the dispatcher is not very difficult. If the original arrangement consists of a single random-policy dispatcher and two back-end cluster hosts as shown in Figure 7a, then the same cluster with EquiLoad might look like Figure 7b. In this case, a simple HTTP proxy has been added to the dispatcher to forward requests while keeping track of document content sizes to continuously tune the forwarding policy. Additional HTTP proxies have been added to the back-end servers to redirect requests incorrectly forwarded by the dispatcher.

Some additional communication is required between the dispatcher and back-end servers in an EquiLoad cluster to coordinate the assignment of request partitions to the different servers. The dispatcher needs feedback from the back-end servers to evaluate how well it forwarded requests using its current partitioning. If a significant percentage of requests were redirected by the back-end servers, the dispatcher's partitioning must be adjusted.

4.3.3 Dynamic Adaptive Grid Partitioning Resources

No special resource management is required for DAGP, since the very nature of the grid requires the management of resources.

[10]The Instruction Table Lookaside Buffer caches the mapping of next-instruction pointer virtual addresses to physical addresses.

[11]The Data Translation Lookaside Buffer caches the mapping of virtual addresses to physical addresses.

[12]The trace cache caches uops decoded from program instructions.

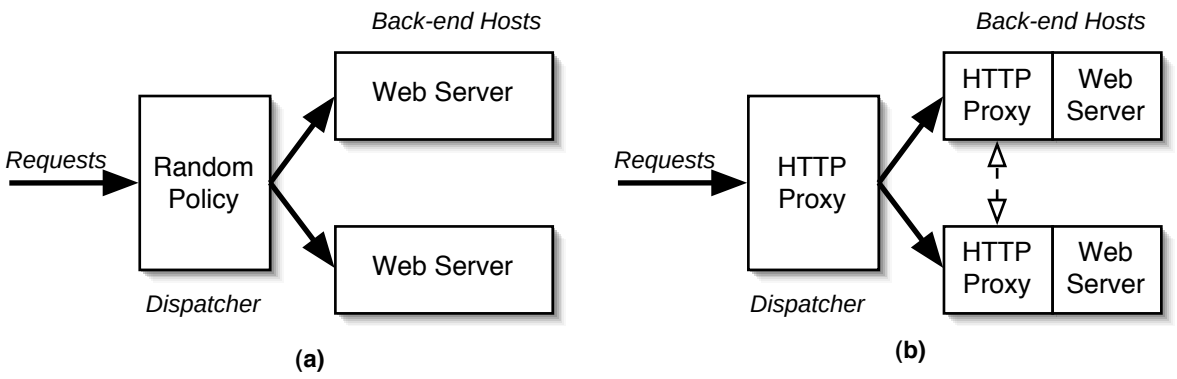


Figure 7: Adaptive grid hierarchy constructed with the Berger-Oliger AMR scheme.

4.3.4 Resource Management Analysis

The most complicated resource management needs result from specialized hardware, because implementing logic in hardware is both more expensive and more integrate than the same logic in software. New software can be installed very easily. Installing new hardware is a physical process and hardware algorithms cannot be tweaked afterwards. Plus, any errors in the hardware can only be fixed by installing new hardware.

Unlike job complexity, resource management complexity is greatest for hyper-threading because of the hardware focus. Designing and constructing a hyper-threading processor requires the addition and rearrangement of physical resources on the processor die, as well as new hardware logic to manage those resources. In contrast, EquiLoad only requires the inclusion of some HTTP forwarding software on the dispatcher and back-end hosts and DAGP requires no special hardware management software.

It could be argued that the additional machines and the switches and network interfaces used to connect them, in both EquiLoad and DAGP, are analogous to the additional hardware introduced by hyper-threading. This is correct to some extent, but a big difference comes from the minimal impact that hardware has when changing the job partitioning and distribution policies of EquiLoad and DAGP. Making changes to hyper-threading policies would require a different hardware configuration and new management logic in hardware. Making changes to EquiLoad or DAGP would only require logic changes in software; the hardware configuration can remain the same.

4.4 Client Application Interface

The final component of the load balancing framework is the client application interface. The simpler the client interface, the easier it is to make use of load balancing in the target system. Ideally, load balancing should take place without any special knowledge on the part of the client application. If a client can send job requests to a load balanced system in a manner identical to how it would send those same requests to a non-load balanced system, then the client can be used in both environments without modification and reap the benefits of the load balanced system whenever possible.

Both hyper-threading and DAGP could benefit from an easier and more transparent client interface.

EquiLoad, on the other hand, has an entirely transparent interface that is easy to use.

4.4.1 Hyper-threading Interface

Hyper-threading requires explicit support by the operating system, which can be considered the actual client making requests of the processor. Although the actual jobs being executed by the processor are application threads, it is the operating system that is the hyper-threading client since it manages the execution of the threads.

Intel's engineers minimized the changes necessary for an operating system to support hyper-threading. The minimal requirement is for the operating system to treat a single physical processor as two logical processors, and perform all of its other operations normally. Any operating system that currently supports multiple physical processors could be easily modified to support multiple logical processors if a hyper-threading capable processor is detected.

Two optimizations recommended by Intel are the use of a new HALT instruction and to schedule across physical processors before scheduling across logical processors. The HALT instruction allows the operating system to inform the processor that only one thread is active, and will force a transition from MT-mode to STX-mode. This allows the single active thread access to all processor resources. The preference to schedule across physical processors allows threads to use different physical resources whenever possible.

4.4.2 EquiLoad Interface

From a client standpoint, communicating with an EquiLoad system is entirely transparent. This is mostly a result of how clustered web servers must behave in order to work with HTTP clients and DNS. If clients were required to have special knowledge when communicating with clustered web servers, web clients would either have to be server-specific or speak another protocol that would inform them of the remote cluster's configuration. To ensure compatibility and scalability, all clustered web servers must act identically to a non-clustered web server.

4.4.3 Dynamic Adaptive Grid Partitioning Interface

Unfortunately, making use of grid computing still requires the manual identification of independent computations. The distribution of those computation blocks may be handled by software, as shown by DAGP, but advances have not yet brought the type of out-of-order execution and independent instruction optimizations found in compilers up to the algorithm level. Perhaps it will be possible in the future to identify block- and algorithm-level dependencies.

Given independent computational blocks, it is possible to make use of DAGP in two different ways. Some grids have a front-end interface for making computational requests. For those grids, DAGP could be implemented on the grid and the client application would send requests to the grid as a whole. If no such front-end interface is available, and the client must distribute jobs across the grid itself, then DAGP can be used as its distribution algorithm.

4.4.4 Interface Analysis

It seems clear from the interfaces provided by hyper-threading, EquiLoad, and DAGP that the simplicity and ease-of-use of an interface depends a lot on the hardware changes due to the load balancing scheme and also the maturity of the technology and architecture it depends on.

Adding hyper-threading support to a processor does require modifications to the operating system so it will treat the physical processor differently. But at the same time, a lot of the special hardware added to the hyper-threading processor are not visible to the OS. To a large degree, this is because the additional hardware manages itself and presents the OS with an illusion of multiple physical processors, which operating systems already know how to deal with. Just as application developers no longer need to worry about memory management, it seems as though we're approaching a point where operating systems may not need to worry about hardware management.

EquiLoad is a special case when it comes to compatibility because its interface must be backwards compatible with existing HTTP clients. If all clustered web servers equipped with EquiLoad required clients to speak a different protocol, all clients would need to be updated. That is a very expensive requirement and EquiLoad is more likely to be thrown away before everyone updates their clients.

DAGP and grid computing is heading in the same direction as EquiLoad because standard protocols and toolkits are emerging. However, there are currently many technologies, protocols, and standards competing for dominance in the field. Grid computing currently looks a lot like the earlier Internet where you might retrieve files through a variety of protocols such as Gopher[13], UUCP[14], or FTP. Today almost all document transfer is done via HTTP. It's likely that a single web services standard and a single grid management protocol will eventually be used for all grid computing.

5 Conclusion

Hyper-threading, EquiLoad, and Dynamic Adaptive Grid Partitioning provide a modern look at how research is addressing the framework requirements in different fields of computer science. All three technologies improve performance of systems in their respective fields, but an analysis of the technologies in view of the framework illustrates how their effectiveness varies in the different framework components.

Hyper-threading approaches job partitioning and distribution in a very job independent manner to maximize processor performance while presenting the illusion of multiple physical processors. Minimal job management by the processor allows the operating system to retain control over job management. Appearing as multiple physical processors allows the new hardware to work with existing operating systems.

EquiLoad and DAGP, on the other hand do much more when it comes to job partitioning and distribution because web servers and grid computing systems can take liberties in how they behave and process requests. A processor must always conform to the strict behavior expected by an

[13] Gopher is a client-server protocol used to access documents via a text interface.

[14] Unix to Unix CoPy is a protocol used to store-and-forward files such as email or other files. It was commonly used over dial-up modems.

operating system, but web servers and grids are not bound by strict rules. As long as the document is eventually returned by the web server, and the computation is eventually completed, users do not care how their request was processed.

With the presented framework, and insight provided by current approaches, researchers in any area of computer science can approach the problems of load balancing from a structured standpoint.

References

[1] Martin Arlitt and Tai Jin, *Workload characterization of the 1998 World Cup web site*.

[2] Martin F. Arlitt and Carey L. Williamson, *Web server workload characterization: The search for invariants*, Measurement and Modeling of Computer Systems, 1996, pp. 126–137.

[3] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella, *Changes in web client access patterns: Characteristics and caching implications*, Tech. Report 1998-023, 4, 1998.

[4] Paul Barford and Mark Crovella, *Generating representative web workloads for network and server performance evaluation*, Measurement and Modeling of Computer Systems, 1998, pp. 151–160.

[5] *Berger-Oliger AMR-algorithm*, http://www.math.tu-cottbus.de/~deiter/amroc/html/amr.htm.

[6] *NC BioGrid*, http://www.ncbiogrid.org/.

[7] *BLAST*, http://www.ncbi.nih.gov/BLAST/.

[8] Gianfranco Ciardo, Alma Riska, and Evgenia Smirni, *EQUILOAD: a load balancing policy for clustered web servers*, Performance Evaluation **46** (2001), no. 2-3, 101–124, http://citeseer.nj.nec.com/ciardo01equiload.html.

[9] *Clustered web servers architectures*, http://www.control.auc.dk/~henrik/undervisning/trafik2/riska/www.cs.wm.edu/~riska/main/node53.html.

[10] *The globus alliance*, http://www.globus.org/.

[11] *Grid service specification*, http://www-unix.globus.org/toolkit/documentation.html.

[12] *multiprogramming*, http://searchvb.techtarget.com/sDefinition/0,,sid8_gci212615,00.html.

[13] Manish Parashar and James C. Browne, *On partitioning dynamic adaptive grid hierarchies*, HICSS (1), 1996, http://citeseer.nj.nec.com/parashar96partitioning.html, pp. 604–613.

[14] *SETI@home: Search for extraterrestrial intelligence at home*, http://setiathome.ssl.berkeley.edu/.

[15] *Simple Object Access Protocol*, http://wwww.w3.org/TR/SOAP/.

[16] D. Marr; F. Binns; D. Hill; G. Hinton; D. Koufaty; J. Miller; M. Upton, *Hyper-threading technology architecture and microarchitecture*, Intel Technology Journal **6** (2002), no. 1, 4–15, http://www.intel.com/technology/itj/2002/volume06issue01/art01_hyper/p01_abstract.htm.

[17] R.J. Allan; J.M. Brooke; F. Costen; M. Westhead, *Grid-based high performance computing*, UKHEC Collaboration (2000), http://www.ukhec.ac.uk/.