Adaptive Scheduling with Client Resources to Improve WWW Server Scalability

Daniel Andresen and Tao Yang
Department of Computer Science
University of California
Santa Barbara, CA 93106
{dandrese, tyang}@cs.ucsb.edu

Abstract

WWW-based Internet information service has grown enormously during last few years, and major performance bottlenecks have been caused by WWW server and Internet bandwidth inadequacies. Augmenting the server with multi-processor support and shifting computation to client-site machines can substantially improve the system response time and for some applications, it may also reduce network bandwidth requirement. In this paper, we model client-server partitionable WWW applications and propose adaptive scheduling techniques that optimize the use of client-server resource by predicting the aggregate impact of I/O, CPU and network capabilities. We present a software system called SWEB++ which implements and supports the use of our scheduling strategies when programming WWW applications. We also provide a performance-analysis framework based on homogeneous client-server assumptions to identify the impact of system loads and network bandwidth and demonstrate the effectiveness of our scheduling strategies. Finally we present several experimental results to examine the system performance and verify the usefulness of the analytic model.

1 Introduction

Recently WWW-based computer technology has had an explosive growth in providing on-line access for images, files, and computing services over Internet. For example, most digital library (DL) systems [16] will be based on the WWW. Research efforts are being made to build a virtual global computing service through WWW servers, e.g. [13]. An important feature of such services is that information can be obtained and processed transparently, independent of its location. The emergence of gigabit information superhighways further enhances the vision of accessible world-wide global computing resources.

The major performance bottlenecks for the advent of WWW technology are server computing capabilities and Internet bandwidth. Popular WWW sites such as *Alta Vista* [1] receive over twenty million accesses a day, where the computational demand for each request is often less than that in many sophisticated WWW systems (e.g. DLs). In [4], we have addressed issues of developing multi-processor Web servers in dealing with this bottleneck using networked workstations connected with inexpensive disks. As the WWW develops and Web browsers achieve the ability to download executable content (e.g. Java), it becomes logical to think of transferring part of the server's workload to clients [19]. Changing the computation distribution between a client and a server may also alter communication patterns between them, possibly reducing network bandwidth requirements. Such a global computing style scatters the workload around the world and can lead to significantly improved user interfaces and response times.

Our main research challenge in the today's global computing environment is the effective management and utilization of resources from multi-processor WWW servers and client-site machines. Blindly transferring workload onto clients may not be advisable, since the byte-code performance of Java is usually 5-10 times slower than a client machine's potential. Also a number of commercial corporations are developing so-called "network computers", with little or no hard drive, and a minimal processor, but with Java and the Internet networking built in. A careful design of the scheduling strategy is needed to avoid imposing too much burden on these Net PCs [20]. At the server site, information on the current system load and the disk I/O bandwidth also affects the selection of a server node for processing a request. In addition to this, the impact of available bandwidth between the server and client needs to be incorporated. Thus dynamic scheduling strategies must be adaptive to the variation of client/server resources in multiple aspects.

In this paper we propose an integrated scheduling model for partitioning and mapping client-server computation based on dynamically changing server and client capabilities. We also present a software system called SWEB++ to support programmers in utilizing our model for their WWW applications. We present a performance-analysis framework on homogeneous client-server environments to examine the impact of client-server resource availability, load migration and scalability issues.

The paper is organized as follows: Section 2 gives a model and an example for client-server web computing. Section 3 presents the software architecture of SWEB++ in supporting the use of client-server task partitioning and mapping. Section 4 discusses adaptive scheduling strategies for a multi-processor WWW server with client resources. Section 5 gives a performance-analysis framework to examine the system performance on a homogeneous environment. Section 6 presents the experimental results. Section 7 discusses related work and conclusions.

2 A Model of Client-Server Web Computing

We first give a general overview of WWW protocols for Internet client-server interaction. We then present a model for client-server Web computing and demonstrate it using a DL application for multi-resolution image browsing.

WWW is based on three critical components: the Uniform Resource Locator (URL), the HyperText Markup Language (HTML), and the HyperText Transfer Protocol (HTTP). The URL defines which resource the user wishes to access, the HTML language allows the information to be presented in a platform-independent but still well-formatted manner, and the HTTP protocol is the application-level mechanism for achieving the transfer of information [7, 8, 21]. An HTTP request would typically activate the following sequence of events from initiation to completion. First, the client determines the host name from the URL, and uses the local Domain Name System (DNS) server to determine its IP address. The local DNS may not know the IP address of the destination, and may need to contact the DNS system on the destination side to complete the name resolution. After receiving the IP address, the client then sets up a TCP/IP connection to a well-known port on the server where the HTTP server process is listening. The request is then passed in through the connection. After parsing the request, the server sends back a response code followed by the results of the query. This response code could indicate the request service can be performed, or might redirect the request to another server. After the contents of the response are sent to the client, the connection is closed by either the client or the server [21]. The results of the request are normally described in HTML for client display. Current client-side browsers such as Netscape support Java, and the request results can be a platform-independent program (called applet) runnable at the client machine to produce the results for display.

2.1 The Model

We model the interaction between client and server as a task chain that is partially executed at the server node (possibly as a CGI [25] program) and partially executed at the client site (as a Java applet if applicable). A task consists of a segment of the request fulfillment, with its associated computation and communication. The task communication costs differ depending on whether the task results must be sent out over the Internet or can be transferred locally to the next task in the task chain. Also, when the Java code accesses the server, some data items residing at the client-site machine may be available for re-use for the new request. The following items appear in a task chain definition: 1) Data items to be retrieved from the server disk. 2) A task chain to be processed at the server and client site machines. 3) Data items that are available at the client site memory.

Our WWW server model consists of a set of nodes connected with a fast network as shown in Figure 1 and presented as a single logical server to the Internet [4]. The user requests are first evenly routed to processors via DNS rotation [22]. Each server node may have its local disk, which is accessible to other nodes via remote file service in the OS.

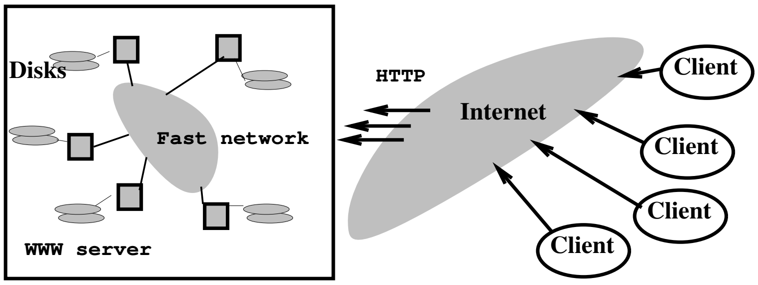


Figure 1: The architecture of a multi-processor WWW server.

Each task chain can be scheduled onto one of the server nodes. Our challenge, then, is to select an appropriate node within the server cluster to process the request modeled by a task chain. Also we need to partition tasks in this chain into two parts, one for the client and another for server such that the overall request response time is minimized (illustrated in Figure 2). For example, if the server nodes are heavily burdened and the client is rather powerful, it might be sensible to shift everything possible to the client. If the client happens to be a 16Mhz 386sx sitting in a high school laboratory or under-powered NetPC [20], then keeping things on the server could be more optimal in terms of request response time. In addition to balancing between client and server machine load and capability, the network bandwidth between client and server also affects the partitioning point. We assume that local communication between tasks within the same partition (client or server) is zero while client-server communication delay is determined by the current available bandwidth between them.

2.2 An example: multi-resolution image browsing

We briefly demonstrate the above model for an application in the Alexandria Digital Library system (ADL) [3]. The current collections of ADL involve geographically-referenced materials, such as maps, satellite images, digitized aerial photographs, and associated metadata. The images can range in size from 10-100MB, with larger files expected in the near future as the scanning resolution increases. It is infeasible to simultaneously transfer many large data files for browsing over the current slow Internet links. The

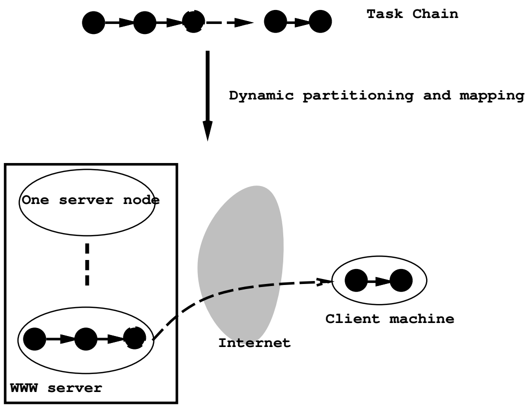


Figure 2: An illustration of dynamic task chain partitioning and mapping.

ADL has adopted progressive multi-resolution and subregion browsing strategies to reduce Internet traffic in accessing map images [5]. This approach is based on the idea that users often browse large images via a thumbnail (at coarse resolution), and desire to rapidly view higher-resolution versions and subregions of those images already being viewed. To support these features, the ADL system is using a wavelet-based hierarchical data representation for multi-resolution decomposition of images.

This process of multi-resolution browsing is summarized as follows. An image is decomposed hierarchically using the wavelet transform [10]. When a client requests an image, the system searches through a database catalog engine and then delivers a thumbnail of this image. When a client requests higher-resolution versions of these thumbnails, the server retrieves compressed image coefficient data from permanent storage and delivers it to the client as the client machine performs the inverse wavelet to construct the higher resolutions images from the existing thumbnail and new coefficient data. If the client machine does not have the reconstruction capability, the server will perform the reconstruction on the client's behalf. Notice that a client may identify an interesting point in an image and request an enhanced resolution view of the subregion surrounding that point. In this case the system needs to find the appropriate subregion image wavelet data, sending the client site only the data necessary for the subregion.

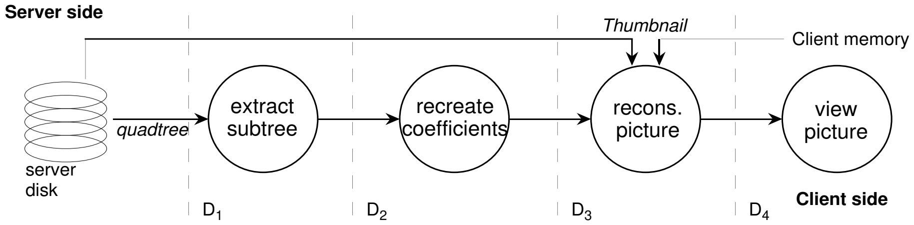


Figure 3: **A task chain for wavelet image enhancement.** Four cutoff points are depicted for possible computation partitioning between client and server.

Figure 3 depicts a task chain for processing a user request in accessing an image or subimage with higher resolution, based on an implementation in [27]. The chain has four tasks, which are summarized as follows:

- *Fetching and extracting the subtree.* The wavelet image data is stored in a combined quad-tree/Huffman encoded form on a disk. These compressed files must be fetched. The appropriate subtree of a quadtree with its associated compressed coefficient data must be extracted in its compressed form for the next stage.

- *Recreating the coefficients.* The compressed coefficients must be expanded to their original form.

- *Reconstructing the pixels.* After the coefficients are available, the inverse wavelet procedure is called to create the new higher-resolution image from the thumbnail image. Notice that the thumbnail image needs to be fetched from the server disk if the reconstruction is conducted on the server. Otherwise, the thumbnail image is already available on the memory of the client machine.

- *Viewing the image.* For our purposes, we assume the viewing of the image takes no computation time, but must be done on the client.

Three of the above tasks can be done on either the client or the server, which leads to four possible split points illustrated in Figure 3. We explain them as follows:

- **D1** Send the entire compressed image to the client and let it do everything.

- **D2** Send over the relevant portions of the compressed image for the client to process.

- **D3** Recreate the coefficients from their compressed representation and send them to the client for reconstruction.

- **D4** Recreate the image requested by the client and send the pixels to the client.

3 The SWEB++ Architecture

The SWEB++ system is designed to assist the programming of distributed WWW applications which can be adaptively scheduled between a multi-processor server and a client, making use of the resources of both to minimize response times. The current client-server computation model is based on task chains discussed in the previous section. The main components of SWEB++ are depicted in Figure 4.

In developing a WWW application, a programmer first creates the task binary executables for the client and server. Then the programmer describes each task and the task chain using the task definition language (TDL). After that, the SWEB++ composer takes the task specification as an input and generates the C++ code to link the necessary modules together used for client and server site libraries. It extracts the task attribute information from the specification and produces C++ stubs for the scheduler's resource requirement estimation. The composer also creates a CGI shell script program for the run-time chain executor to invoke when a split point is provided.

The scheduler at each server node interacts with a httpd daemon for handling HTTP requests. It also has a server *broker* module which determines the best possible processor to handle a given request. The broker consults with two other modules, the *oracle* and the *loadd*. The *oracle* is a module which utilizes user-supplied information to predict the CPU and disk demands for a particular request, as well as determining the appropriate split point. The *loadd* daemon is responsible for updating the system CPU, network and disk load information periodically (every 2-3 seconds).

3.1 The Run-time Protocol

We discuss the run-time protocol between a client and the SWEB++ server in sending and processing requests. The main idea is that the system uses keys provided in the task chain specification to recognize task chains and associated arguments. The details are described as follows.

1. The client launches a request as a URL with the following format:

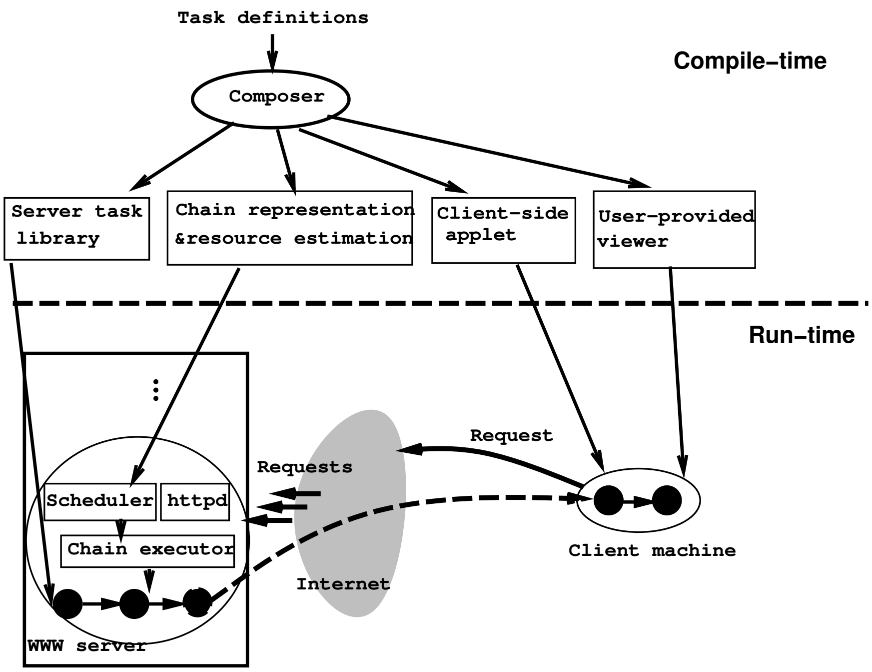


Figure 4: Components of the SWEB++ architecture

`<task chain key> ? <argument list>.`

For example, "/cgi-bin/wavelet?sub_extract="santa-barbara"", where "wavelet" corresponds to a task chain key name. The argument "santa-barbara" is used for one task in this chain with key "sub_extract".

2. After the server receives the URL, it takes the chain key from the URL and compares it to the task chain key collection supplied by user specification. If a match is found, then the corresponding task chain is initiated. If no match is found, then SWEB++ assumes it to be a generic request and schedules it to a server node with the lowest load without attempting any chain splitting operations.

3. For the matched request, the scheduler estimates the cost based on the chain-specific information, decides a split point and an appropriate server node, and then executes the server-side tasks in the chain using arguments contained in the URL argument list. The details on scheduling are in Section 4.

4. The client-site operation is activated by setting the output type of the task before the split point. The client library provides the binary for the execution.

Notice that the system has two types of clients to interact with. The first is a standard HTTP browser without our custom code. The second has our client-side library, and can take an active part in fulfilling its requests. For situations where the client does not have our custom Java applet library, the system defaults to server-only scheduling, and returns only completed request results. This situation arises for Java-less browsers, or the first time Java-enabled browsers access the server prior to downloading the SWEB++ executables. After the client has acquired the custom library, it then can take an active role in contributing its resources and send back with its requests estimated information on the bandwidth to the server, latency for a server connection, and the client's processing capabilities.

3.2 The Task Definition Language

The Task Definition Language (TDL) is similar to the Durra language used by [26], with several extensions to meet our needs. The syntax is straightforward, and many simple tasks can be defined within the language itself. For complex tasks, TDL allows a user to embed a complete C++ function where necessary. A demonstration of the specification for the wavelet example in Section 2 is listed in Figure 5. We explain the key components of TDL below.

The task chain specification started with "CHAIN" has the following format:

```
CHAIN <task chain name> IS <ordered task list> <
    REQUEST_KEY = <chain key>
>
```

The task list contains the ordered task names in the chain. The attribute field "REQUEST_KEY" indicates the chain key appeared in a URL so that the system can recognize the required chain action for a request, as discussed in the run-time protocol. For example, the last line of Figure 5 gives a specification for the wavelet task chain.

Each task specification started with "TASK" has the following format:

```
TASK <task name>  <
    <attribute> = <value>
    ...
    <attribute> = <value>
>
```

The main task attributes are summarized below.

- *SERVER_BINARY and CLIENT_BINARY* define the task's binary to be run by the server or client.
- *ARGUMENT_KEY* gives the label for an argument of this task in the URL argument list, as discussed in the run-time protocol.
- *FILE_IN_SIZE* gives the size of the files to be read from a server's disk. The default is zero. We assume no files can be read from the client disk, conforming to typical current browser security practices.
- *LOCALITY* gives the location of files that the task will retrieve at run-time, allowing the scheduler to determine on which server the files are physically stored. By default we assume remote disk access will be required.
- *EXECUTION_LOCATION* indicates whether a binary can be run on the server only (SERVER_ONLY), client only (CLIENT_ONLY) or either (EITHER).
- *OUTPUT_SIZE* is the size (in bytes) of the output for this task. Notice that it is equal to the input size of the next task in the chain. The input of the first task in the chain is always 0.
- *OUTPUT_TYPE* is the type of the data output (e.g., text/html or image/jpeg). This attribute information can be used to activate appropriate client-site operations if the split point is after this task.
- *COMPUTATION* is the amount of computation required to complete the task. The default is zero.

- *SPLIT_SERVER_COMPUTATION* is the amount of extra computation required on the server to prepare the data for transfer. For example, some output data can be compressed further before transferring over the Internet.
- *SPLIT_CLIENT_COMPUTATION* is the extra computation required on the part of the client to receive the data transferred (e.g., decompression). This defaults to zero.
- *SPLIT_OUTPUT_SIZE* is the size (in bytes) of the data transmitted if the split occurs after this task. The term is necessary since many network-related operations, such as marshaling, change the size of their input. This term defaults to *OUTPUT_SIZE*.

4 Request Scheduling

After the system compiles a set of request types to be scheduled, the information provided in the specification will be useful in assisting the system in making the selection of server processor and partitioning between client-server for optimal performance. Several factors affect the response time in processing requests, including loads on CPU, disk, and network resources. The load of a processing unit must be monitored so that requests can be distributed to relatively lightly loaded processors. Since data may need to be retrieved from disks, disk channel usage must be monitored. Simultaneous user requests accessing different disks can utilize parallel I/O channels to achieve higher throughput. The local interconnection network bandwidth affects the performance of file retrieval since many files may not reside on the local disk of a processor, so remote file retrieval through the network file system will be involved. Local network traffic congestion could dramatically slow the request processing. We first present the cost model for predicting the response time in processing a request, then we discuss a strategy to select a server node and decide a good split point.

Notice that in [4], we have proposed using URL *redirection* to implement the request re-assignment. The HTTP protocol allows for a response called "URL Redirection". When client C sends a request to server S_0, S_0 returns a rewritten URL r' and a response code indicating the information is located at r'. C then follows r' to retrieve the resulting data. Most Net browsers and clients automatically query the new location, so redirection is virtually transparent to the user. This reassignment approach requires a certain amount of time for reconnection. To avoid the dominance of such overhead, we include the redirection overhead in estimating the overall cost and a reassignment is made only if the redirection overhead is smaller than the predicted time savings. For example, if a request involves a small file retrieval, no redirection occurs. DNS rotation is used to provide an initial load distribution. Load re-balancing is effective under the assumption that our targeted WWW applications (e.g. DLs) involve intensive I/O and computation in the server. Another approach for implementing re-assignment is socket forwarding, which avoids the overhead of re-connection, but requires significant changes in the OS kernel or network interface drivers [2]. We have not used it for compatibility reasons.

4.1 A cost model for processing task chains

For each request, we predict the processing time and assign this request to an appropriate processor. Our general cost model for a request is

$$t_s = t_{redirection} + t_{data} + t_{server} + t_{net} + t_{client} \tag{1}$$

$t_{redirection}$ is the cost to redirect the request to another processor, if required. t_{data} is the server time to transfer the required data from the server disk drive, or from the remote disk if the file is not local. t_{server}

```
#sample task chain definition for wavelets
TASK wav_extract_subtree <
    SERVER_BINARY = "wavelet.subtree_extract"
    CLIENT_BINARY = "wavelet.subtree_extract.clnt"
    ARGUMENT_KEY = "sub_extract"
    EXECUTION_LOCATION = EITHER
    FILE_IN_SIZE = FUNC
                     return dbm_lookup_filesize(ARGUMENT_KEY[1]);
                   FUNC_END
    LOCALITY = ARGUMENT_KEY[1]  #use our path in the first argument for locality
    OUTPUT_SIZE = FUNC
                     return subregion_size(FILE_IN_SIZE);
                  FUNC_END
    OUTPUT_TYPE = "x-wav/compressed_coeff"
    COMPUTATION = FUNC:
                     return 100*OUTPUT_SIZE;
                  FUNC_END
>

TASK wav_recreate_coefficients <
    SERVER_BINARY = "wavelet.recreate_coeff"
    CLIENT_BINARY = "wavelet.recreate_coeff.clnt"
    ARGUMENT_KEY = "rec_coeff"
    EXECUTION_LOCATION = EITHER
    OUTPUT_SIZE = FUNC
                      return 8*INPUT_SIZE;
                  FUNC_END
    OUTPUT_TYPE = "x-wav/uncompressed_coeff"
    SPLIT_SERVER_COMPUTATION = FUNC   #compress the coefficients for transfer
                                 return 20*OUTPUT_SIZE;
                               FUNC_END
    SPLIT_CLIENT_COMPUTATION = FUNC #decompress the coefficients after transfer
                                  return 20*OUTPUT_SIZE;
                               FUNC_END
    SPLIT_OUTPUT_SIZE = INPUT_SIZE
    COMPUTATION = FUNC:
                    return 100*OUTPUT_SIZE;
                  FUNC_END
>

TASK wav_create_ppm <
    SERVER_BINARY = "wavelet.recreate_ppm"
    CLIENT_BINARY = "wavelet.recreate_ppm.clnt"
    ARGUMENT_KEY = "rec_ppm"
    EXECUTION_LOCATION = EITHER
    FILE_INPUT_SIZE= FUNC
                       if (ON_SERVER) {
                               return dbm_lookup_filesize(ARGUMENT_KEY[1]);
                       } else { return 0;}
                     FUNC_END
    LOCALITY = ARGUMENT_KEY[1]
    OUTPUT_SIZE = FUNC
                    return image_size(ARGUMENT_KEY[2]);
                  FUNC_END
    OUTPUT_TYPE = "image/ppm"
    COMPUTATION = FUNC
                     return 100*OUTPUT_SIZE;
                  FUNC_END
>

TASK view <
    EXECUTION_LOCATION = CLIENT_ONLY
    COMPUTATION = 0
>

CHAIN wavelets IS wav_extract_subtree, wav_recreate_coefficients, wav_create_ppm, view <
    REQUEST_KEY = "wavelet"
>
```

Figure 5: The TDL specification of the wavelets task chain.

is the time for server computation required. t_{net} is the cost for transferring the processing results over the Internet. t_{client} is the time for any client computation required. We discuss the above terms as follows.

- $$t_{data} = \begin{cases} t_{lstartup} + \frac{\text{Size of server disk data needed}}{b_{disk} \times \delta_1} & \text{If local,} \\ t_{rstartup} + \frac{\text{Size of server disk data needed}}{min(b_{disk} \times \delta_1, b_{lnet} \times \delta_2)} & \text{If remote.} \end{cases}$$

  If the file is local, the time required to fetch the data is simply the file size divided by the available bandwidth of the local storage system, b_{disk}, plus some startup overhead $t_{lstartup}$. We also measure the disk channel load δ_1. If there are many concurrent requests, the disk transmission performance degrades accordingly. We currently ignore the startup costs for network disk I/O in our implementation, since if the request is large, other the transfer times dominate, and if the request is small, network overhead dominates.

  If the data is remote, then the file must be retrieved through the interconnection network. The local network bandwidth, b_{lnet}, and load δ_2 must be incorporated, plus the startup overhead t_{rstart}. Experimentally, we found on the Meiko approximately a 10% penalty for a remote NFS access, and on the SUN workstations connected by Sparc/Ethernet the cost increases by 50%-70%.

- $t_{server} = CPU_{load} \frac{\text{No. of server operations required}}{CPU_{\text{server speed}}}$.

  The number of server operations required depends on how the subtask chain is partitioned. The cost estimation is based on the speed of the examined server node, the estimated CPU load on a destination node (CPU_{load}), and the estimated number of operations required for the task. CPU_{load} represents the number of active jobs sharing the CPU, thus we multiply the required computation time by this factor to approximate the server CPU time. It should be noted that some estimated CPU cycles may overlap with network and disk time and the overall cost may be overestimated slightly, but this conservative estimation works well in our experience.

  The load estimation of remote processors is based on the periodic updating of information given by those remote processors. It is possible that a processor p_x is incorrectly believed to be lightly loaded by other processors, and many requests will be redirected to it. To avoid this unsynchronized overloading, we conservatively increase the CPU load of p_x by σ. This strategy is found to be effective in [28, 29]. We typically use $\sigma = 30\%$.

- $t_{client} = \frac{\text{No. of client operations required}}{CPU_{\text{client speed}}}$.

  The number of client operations required depends on how the subtask chain is partitioned. Here we assume the speed reported by the client machine includes client load factors.

- $t_{net} = t_{nstart} + \frac{\text{No. of bytes for client-server communication}}{\text{Net bandwidth}}$

  This term is used to estimate the time necessary to return the results back to the client over the network. The number of bytes required again depends on how the partitioning is conducted. For the example in Section 2.2, if the server does image reconstruction, then the entire subregion image needs to be shipped. If the client only does image reconstruction, the server only needs to send the compressed coefficient data. t_{nstart} is the startup time for network connection and is ignored in the current setting for similar reasons to those given for t_{lstart} and t_{rstart}.

4.2 The procedure for dynamic chain partitioning and mapping

Given the arrival of HTTP request r at processor x, the scheduler at processor x goes through the following steps. The time complexity of the algorithm is small since task chains are short in practice.

- *Preprocess a request.* The server parses the HTTP command, and completes the pathname given. It also determines whether the requested document exists or it is a CGI program/task chain to execute. If it is not recognized as a task chain[1], the system will assign this task to the sever node with the lowest load. Otherwise the following steps will be further conducted.
- *Analyze the request.* Given a task chain r, the system uses the algorithm in Figure 6 to select a partitioning and server node for the minimum response time. Notice that no requests are allowed to be re-directed more than once for avoiding the ping-pong effort.
- *Redirection and fulfillment.* If the chosen server is not x, the request is redirected appropriately. At this point the request is processed in the normal HTTP server manner, with any CGI's executed as determined by the scheduler, and responses for the client returned. The remaining part of a task chain will be further executed at the client machine.

```
Estimate the client CPU load and speed.
Determine the locality of server files to be accessed.
For each available server s
    Estimate the load, CPU speed, local disk available bandwidth and remote disk accessing bandwidth.
    For each possible client-server split point k
        For each task executed on client, delete all server file input edges if those files are available
        in client memory.
        Size of server data = summation of all file input (specified in task attribute FILE_IN_SIZE).
        Size of client-server communication = the output data size of the last server task
          (specified in field OUTPUT_SIZE), plus any server data needed for all client tasks.
        No. of server operations needed = summation of computation for all server tasks
          (see COMPUTATION and SPLIT_SERVER_COMPUTATION).
        No. of client operations needed = summation of computation for all client tasks.
        Evaluate t_s using Equation (1) and its itemized definition.
    EndFor
    Select the best split point with the minimum t_s.
EndFor
Select the best server node with the minimum cost.
```

Figure 6: The procedure of scheduling a task chain.

5 A performance-analysis framework for homogeneous client-server environments

It is difficult to analyze the performance of our scheme for general cases. We make a number of assumptions to provide three facets of analysis: maximum sustained requests per second (MRPS), expected redirection ratio, and predicted task chain split points. In this way, we can analyze the impact of system resource

[1] A simple file retrieval is also listed as a task chain (see Section 5.2.1 for the modeling).

availability from different aspects, for example, the number of server-nodes, available Internet bandwidth and the CPU speed ratio between client and server machines.

We first present our framework, then demonstrate its use for two sample task chains: wavelets processing and file fetches. We will present experimental evidence to support our analysis.

5.1 The framework

Main assumptions. We assume that the system is homogeneous in the sense that all nodes have the same CPU speed and the initial load, and each node has a local disk with the same bandwidth. We assume that all clients are uniformly loaded with the same machine capabilities. All requests are uniform, i.e. involving the same task chain. Each server node processor receives a uniform number of requests, and produces a stable throughput of information requested. In estimating the available local disk or remote disk accessing bandwidth, we assume a linear model such that the bandwidth is uniformly shared among processed tasks. We neglect items such as disk contention and paging at this time.

Request activity models. We examine the performance assuming that each node receives r requests at each second for a period of L. We denote this as model (r, L). Two instances of this model are considered.

- $(r, 1)$. This reflects the system performance in responding to a burst in user requests, which occurs frequently in many WWW sites [6, 12]. All requests are assumed completed in the same length of time and each server processor is dealing with the same number of requests until all finish. All task chains are partitioned uniformly on the same edge.
- (r, ∞). We examine MRPS when $L = \infty$, which represents the maximum sustained performance when the system enters a steady state and receives a stable amount of requests for a long period of time. Thus all requests can be assumed to be completed in the same length of time and each server processor is dealing with the same number of requests. All task chains are partitioned uniformly on the same edge.

We define the following terms:

- p – the number of server nodes.
- R – the total number of requests received for all processors per second.
- r – requests received per processor. $r = \frac{R}{p}$. Notice that we assume that the r requests arriving at each node after the division via DNS are uniform.
- A – the average overhead in preprocessing a request, and deciding a redirection.
- b_1 – the average bandwidth of local disk access.
- b_2 – the average bandwidth of remote disk access.
- c – the average probability of a processed request accessing a local disk.
- S – the average slowdown ratio of the client CPU compared to the server node. $S = \frac{CPU_{\text{server speed}}}{CPU_{\text{client speed}}}$.
- d – the average redirection probability.
- O – the average overhead of redirection.

- B – the average network bandwidth available between the server and a client.
- B_s – the maximum aggregated network bandwidth available from the server to all clients.

Among the r requests arriving at each node, we assume the probability of accessing one of the server disks is equal to 1/p. Then $r * 1/p$ requests are accessing the local disk. Among those r requests, dr of them will be redirected to other node but dr requests will be redirected from other nodes to this node (we also assume that redirection is uniformly distributed because of the homogeneous system). Our experiments show that in such cases, the redirected requests tend to follow file locality. Thus the total number of requests processed at each processor after redirection is r requests per second. Among them, the total number of requests accessing the local disk is r/p from the original arrival tasks, plus an additional d redirected requests. Then the probability of accessing a local disk for those r requests is:

$$c = \frac{\frac{r}{p} + d * r}{r} = \frac{1}{p} + d.$$

Assume the uniform split point is known, we further define:

- H – the average response processing time for each request (the time from when the client launches a request to the time when the client receives the desired data).
- F_s – the average size of server disk files needed.
- F_n – the average size of server-client data needed to be transferred.
- Z – the average number of requests processed simultaneously at each processor. For example, $Z = r$ for model $(r, 1)$.
- E_{server} – the average total task time needed at the server site.
- E_{client} – the average total task time needed at the client site.

Then

$$H = c * \frac{F_s}{\frac{b_1}{Z}} + (1-c) * \frac{F_s}{\frac{b_2}{Z}} + (A + d(A+O))Z + E_{server}Z + F_n/B + S * E_{client}. \tag{2}$$

Notice that $B \leq \frac{B_s}{Z*p}$.

5.1.1 Expected redirection ratios for $(r, 1)$ and (r, ∞)

Note that the scheduler minimizes the processing time for each request. The expected redirection ratio can be drawn by comparing the costs of performing the redirection to the expected benefits.

For H in Equation (2), we can rewrite the formula in the following format:

$$d * Z * (\frac{F_s}{b_1} - \frac{F_s}{b_2} + A + O) + \text{constants independent of } d.$$

We present two scenarios for d:

- $\frac{F_s}{b_1} - \frac{F_s}{b_2} + A + O < 0, i.e.\ \ A + O < F_s(\frac{1}{b_2} - \frac{1}{b_1})$. This corresponds to the case of a large file access with a slow local network. In this case, H is minimized with d at its maximum possible value. Note we have a constraint such that
$$1 - c = 1 - \frac{1}{p} - d \geq 0$$
then $d \leq 1 - 1/p$. Thus $d = 1 - 1/p$.

- $A+O \geq F_s(\frac{1}{b_2}-\frac{1}{b_1})$. This corresponds to the case where we have a fast interconnection network and can fetch a file remotely almost as quickly as a local access; or when the redirection overhead is too high. Then $d=0$, which minimizes H.

5.1.2 MRPS for (r, ∞)

The MRPS is achieved when the entire server system enters a steady state. Then r requests uniformly arrive at each server node at each second and the throughput for each individual processor is also r. For this stage, let H_s be the part of response time spent at the server site for each request. Then each request will be processed at a server node for H_s seconds. Since r new requests come in at each second, the number of requests processed at each node simultaneously is $z = rH_s$.

Based on Equation (2), we have

$$H_s \geq c * \frac{F_s}{\frac{b_1}{rH_s}} + (1-c) * \frac{F_s}{\frac{b_2}{rH_s}} + (A + d(A+O))rH_s + E_{server}rH_s$$

$$r \leq \frac{1}{c * \frac{F_s}{b_1} + (1-c) * \frac{F_s}{b_2} + A + d(A+O) + E_{server}}$$

$$R \leq \frac{p}{c * \frac{F_s}{b_1} + (1-c) * \frac{F_s}{b_2} + A + d(A+O) + E_{server}.}$$

We also notice that since the throughput of p server nodes is R, the total data output per second for the entire system is $R * F_n$. This is restricted by the available output bandwidth of the system. Namely $R * F_n \leq B_s$. Thus $R \leq \frac{B_s}{F_n}$. Using the above two conditions, we have a bound for the MRPS of the entire system as:

$$R^* = \min\left(\frac{p}{c * \frac{F_s}{b_1} + (1-c) * \frac{F_s}{b_2} + A + d(A+O) + E_{server}}, \frac{B_s}{F_n}\right) \text{ where } c = d + 1/p.$$

Naturally the MRPS may be maximized by having the client do everything possible to minimize E_{server}. But such cut point may increase F_n, which also limits the system throughput. With advent of Internet technology, we anticipate that the client-server bandwidth will be improved steadily in the near future, and the MRPS will be achieved by choosing the split point which lets the client do as much as possible. In this manner the maximum possible amount of calculation is spread to the clients.

5.1.3 The expected split point for $(r, 1)$

For this case, $Z = r$, and the H formula in (2) is simplified as follows.

$$H = c * \frac{F_s}{\frac{b_1}{r}} + (1-c) * \frac{F_s}{\frac{b_2}{r}} + (A + d(A+O))r + E_{server}r + F_n/B + S * E_{client}$$

With different split points, all four terms F_s, F_n, E_{server} and E_{client} may change, affecting the optimum choice. We select the split point minimizing H. Partitioning also affects the MRPS that can be reached. In this case a philosophical decision must be made regarding the goal of the server - minimizing individual response times or preserving server resources for future requests. In our current research we assume the former.

5.2 Case analysis

5.2.1 File fetches

We study a sample case, the file fetch, to demonstrate the use of the above framework. The task chain is shown in Figure 7 and has only two tasks, sending the file and viewing it. The computational cost for sending the data is $g * F$ where F is the file size to be fetched. The location of two tasks is fixed, and the split point is fixed between these two tasks.

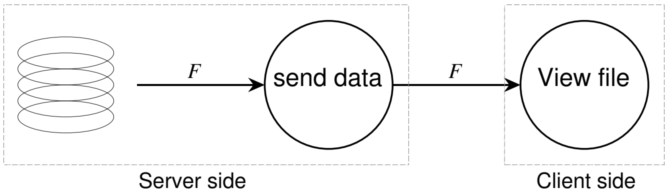


Figure 7: The task chain for a file fetch.

Expected Redirection Ratio for $(r, 1)$ **and** (r, ∞)**.**

- If $A + O < F(\frac{1}{b_2} - \frac{1}{b_1})$, $d = 1 - 1/p$.
- If $A + O \geq F(\frac{1}{b_2} - \frac{1}{b_1})$, $d = 0$.

MRPS for (r, ∞)**.** Then the upper bound for the sustained MRPS of the entire system at the steady state is:

$$R^* = \min(\frac{p}{(\frac{1}{p} + d)\frac{F}{b_1} + (1 - \frac{1}{p} - d)\frac{F}{min(b_1, b_2)} + A + d(A + O) + g * F}, \frac{B_s}{F}).$$

The expected split point. Fixed.

5.2.2 Wavelet chain processing

Here we analyze the wavelet task chain illustrated in Figure 3 for retrieval of a subregion with size $n \times n$. First we define some additional terms:

- F_1 – the average size of the compressed wavelet data (quadtree and coefficients).
- k – the average fraction of F_1 actually needed for a subregion.
- g – the constant ratio for the cost of sending disk files, e.g., gF_1 is the time for sending data F_1.
- F_2 – the size of the thumbnail. The image size is $n/2 \times n/2$.
- E_1 – the average server CPU time for extracting the subtree information.
- E_2 – the average server CPU time for creating the subregion coefficient data.
- E_3 – the average server CPU time for image reconstruction.
- E_c – the average server CPU time for coefficient data compression.
- E_d – the average server CPU time for coefficient data decompression.

There are 4 possible partitions discussed in Section 2.2 and we mark the response time for these partitions as H_1, H_2, H_3 and H_4. We choose the one with the minimum processing time, i.e.,

$$H = \min(H_1, H_2, H_3, H_4). \tag{3}$$

Partition D1:

$$F_s = F_1,\ F_n = F_1,\ E_{server} = gF_1,\quad E_{client} = E_1 + E_2 + E_3.$$

$$H_1 = c * \frac{F_1}{\frac{b_1}{r}} + (1-c) * \frac{F_1}{\frac{b_2}{r}} + (A + d(A+O))r + r * gF_1 + S * E_1 + S * E_2 + S * E_3 + F_1/B.$$

Partition D2:

$$F_s = F_1,\ F_n = k * F_1,\ E_{server} = E_1 + gkF_1,\quad E_{client} = E_2 + E_3.$$

$$H_2 = c * \frac{F_1}{\frac{b_1}{r}} + (1-c) * \frac{F_1}{\frac{b_2}{r}} + (A + d(A+O))r + r * (E_1 + gkF_1) + S * E_2 + S * E_3 + k * F_1/B.$$

Partition D3:

$$F_s = F_1,\ F_n = k * F_1,\ E_{server} = E_1 + E_2 + E_c + gkF_1,\quad E_{client} = E_d + E_3.$$

$$H_3 = c * \frac{F_1}{\frac{b_1}{r}} + (1-c) * \frac{F_1}{\frac{b_2}{r}} + (A + d(A+O))r + r * (E_1 + E_2 + E_c + gkF_1) + S * E_3 + k * F_1/B + E_d * S.$$

Partition D4:

$$F_s = F_1 + F_2,\ F_n = n^2,\ E_{server} = E_1 + E_2 + E_3 + gn^2,\quad E_{client} = 0.$$

$$H_4 = c * \frac{F_1 + F_2}{\frac{b_1}{r}} + (1-c) * \frac{F_1 + F_2}{\frac{b_2}{r}} + (A + d(A+O))r + r * (E_1 + E_2 + E_3 + gn^2) + n^2/B.$$

Expected Redirection Ratio for $(r, 1)$ and (r, ∞). We determine the redirection ratio d for different partitions in order to minimize the response time.

For H_1, H_2, and H_3, $F_s = F_1$. For H_4, $F_s = F_1 + F_2$. According to our analysis in Section 5.1.1, based on the difference between $A + O$, $F_1(\frac{1}{b_2} - \frac{1}{b_1})$, and $(F_1 + F_2)(\frac{1}{b_2} - \frac{1}{b_1})$, d can be selected as either 0 or $1 - 1/p$. We have three cases:

- **Case (a):** When $A + O \geq (F_1 + F_2)(\frac{1}{b_2} - \frac{1}{b_1})$, $d = 0$ for all H_1, H_2, H_3 and H_4.
- **Case (b):** When $A + O < F_1(\frac{1}{b_2} - \frac{1}{b_1})$, $d = 1 - 1/p$ for all H_1, H_2, H_3 and H_4.
- **Case (c):** When $F_1(\frac{1}{b_2} - \frac{1}{b_1}) \leq A + O < (F_1 + F_2)(\frac{1}{b_2} - \frac{1}{b_1})$, $d = 0$ for H_1, H_2 and H_3, and $1 - 1/p$ for H_4.

MRPS for (r, ∞). The expected MRPS depends on the four choices of partitioning. We calculate the MRPS bound R_i^* for each partitioning and choose the maximum one as the bound.

For D1,

$$R_1^* = \min\left(\frac{p}{c * \frac{F_1}{b_1} + (1-c) * \frac{F_1}{b_2} + A + d(A+O) + gF_1}, \frac{B_s}{F_1}\right).$$

For D2,

$$R_2^* = \min\left(\frac{p}{c * \frac{F_1}{b_1} + (1-c) * \frac{F_1}{b_2} + A + d(A+O) + E_1 + gkF_1}, \frac{B_s}{kF_1}\right).$$

For D3,

$$R_3^* = \min(\frac{p}{c * \frac{F_1}{b_1} + (1-c) * \frac{F_1}{b_2} + A + d(A+O) + E_1 + E_2 + E_c + gkF_1}, \frac{B_s}{kF_1}).$$

For D4,

$$R_4^* = \min(\frac{p}{c * \frac{F_1+F_2}{b_1} + (1-c) * \frac{F_1+F_2}{b_2} + A + d(A+O) + E_1 + E_2 + E_3 + gn^2}, \frac{B_s}{n^2}).$$

We assume that $n^2 >> k * F_1$ (the size of the original image is normally larger than that of compressed wavelet data). In comparing R_1^*, R_2^* R_3^* and R_4^*, we need to consider that d is different for Cases (a), (b) and (c). For Cases (a) and (b), d is chosen to be the same among all partitions, thus it is easy to show that $R_4^* \leq R_2^*$ and $R_3^* \leq R_2^*$.

For Case (c), d is the same for D_2 and D_3. Thus $R_3^* \leq R_2^*$. Between R_4^* and R_2^*, we can plug in d values, and we have.

$$R_2^* = \min(\frac{p}{1/p * \frac{F_1}{b_1} + (1-1/p) * \frac{F_1}{b_2} + A + E_1 + gkF_1}, \frac{B_s}{kF_1}).$$

$$R_4^* = \min(\frac{p}{\frac{F_1+F_2}{b_1} + +A + (1-1/p)(A+O) + E_1 + E_2 + E_3 + gn^2}, \frac{B_s}{n^2}).$$

Since, $F_1(\frac{1}{b_2} - \frac{1}{b_1}) \leq A + O$ in Case (c),

$$\frac{F_1+F_2}{b_1} + (1-1/p)(A+O) > \frac{F_1}{b_1} + (1-1/p)F_1(\frac{1}{b_2} - \frac{1}{b_1}) = 1/p * \frac{F_1}{b_1} + (1-1/p) * \frac{F_1}{b_2}.$$

Thus $R_4^* < R_2^*$ for Case (c).

In summary, only partition D1 or D2 could lead to the highest MRPS for all three cases. Then,

$$R^* = \max(R_1^*, R_2^*).$$

Expected Partition Points for (r, ∞)**.** Formula (3) can help us to determine the partitioning points for different scenarios. For example, we can compare H_1, H_2, H_3, H_4 as follows.

For H_1, H_2, and H_3, d is always the same. Notice that $r = R/p$. Thus we have

$$H_2 - H_1 = (R/p - S)E_4 - (1-k)F_1(\frac{1}{B} + g)$$

$$H_3 - H_2 = (R/p - S)E_5 + E_c * R/p + E_d * S.$$

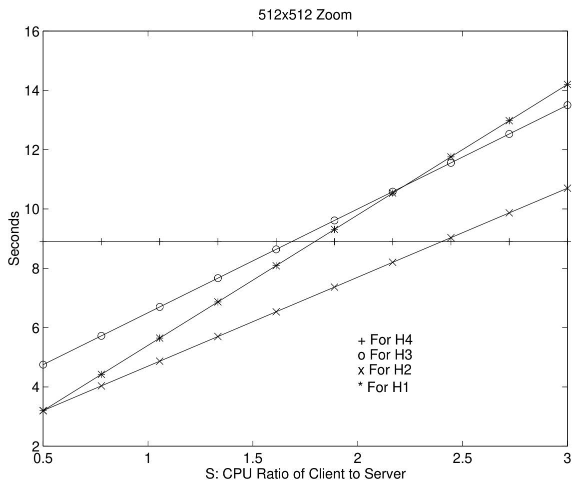
Between H_4 and H_2, if d is the same for Cases (a) and (b):

$$H_4 - H_2 = (R/p - S)(E_3 + E_2) + (n^2 - kF_1)(\frac{1}{B} + g) + F_1(\frac{(1-c)}{b_2} + \frac{c}{b_1}).$$

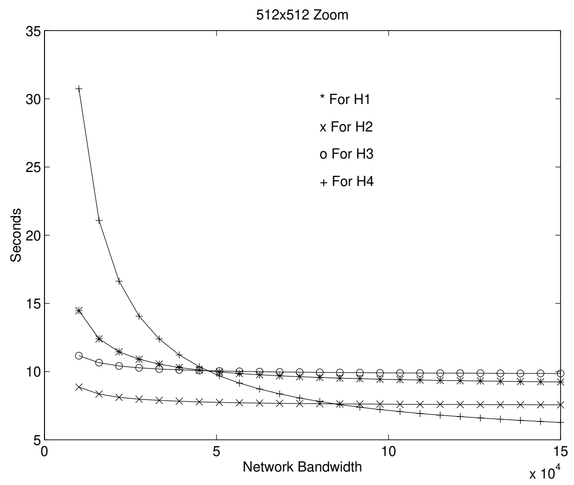
For Case (c):

$$H_4 - H_2 = (1-1/p)(A + O - F_1(\frac{1}{b_2} - \frac{1}{b_1}))R/p + (R/p - S)(E_2 + E_3) + (n^2 - kF_1)l(\frac{1}{B} + g).$$

If $R/p \geq S$ but B is very small, then $H_3 \geq H_2$, $H_4 \geq H_2$, and $H_1 \geq H_2$. Namely, H_2 will be the minimum and D2 will be chosen. Intuitively, if the server is very busy, it is always better for the client to do as



(a)



(b)

Figure 8: Zooming on a subregion. (a) Client power varying, (b) Network bandwidth varying; each server node is twice as powerful as a client.

much as possible. But since the Internet is slow, D2 is better than D1 since D2 involves less client-server communication overhead.

We graph the values of H_1, H_2, H_3, and H_4, illustrating their relationship to server load, client capabilities, and network bandwidth. We utilize the following parameters based our experimental data: n=512, k = 0.25, R = 6, p = 6, $g = 2.5 \times 10^{-7}$, $E_1 = 1.4$ sec, $E_2 = 0.4$ sec, $E_3 = 2.6$ sec, $b_1 = 1100000$ byte per second, $b_2 = 1000000$ bytes per second, A = 0.001 sec, O = 0.1 sec, $F_1 = 56000$ bytes, $F_2 = (n/2)^2$, $E_c =$ 0.9 sec, and E_d= 0.9 sec. Figure 8 shows two results of H_1, H_2, H_3, and H_4, when B ranges from 10000 to 150000 bytes/sec and S is set to range from 0.5 to 3. For points on the left side of the figure, it is advisable to send over the compressed subtree data for the client to process if the server is too busy, otherwise the server should send over the completed image. On the right, the conclusion is similar when the client-server bandwidth increases steadily.

6 Experimental Results

We have implemented a prototype of SWEB++ on a Meiko CS-2 distributed memory machine in about six thousand lines of C/C++ code. The Meiko CS-2 can be viewed as a workstation cluster connected by the Elan fast network. Each node has a scalar processing unit (a 40Mhz SuperSparc chip) with 32MB of RAM running SUN Solaris 2.3 [23]. The theoretical peak performance of Elan network is 40MBytes/sec; however, the TCP/IP layer communication on the Meiko can achieve approximately 3-15% of the peak performance. Our primary experimental testbed consists of six Meiko CS-2 nodes as our server, each of which is connected to a dedicated 1GB hard drive on which the test files reside. Disk service is available to all other nodes via NFS mounts.

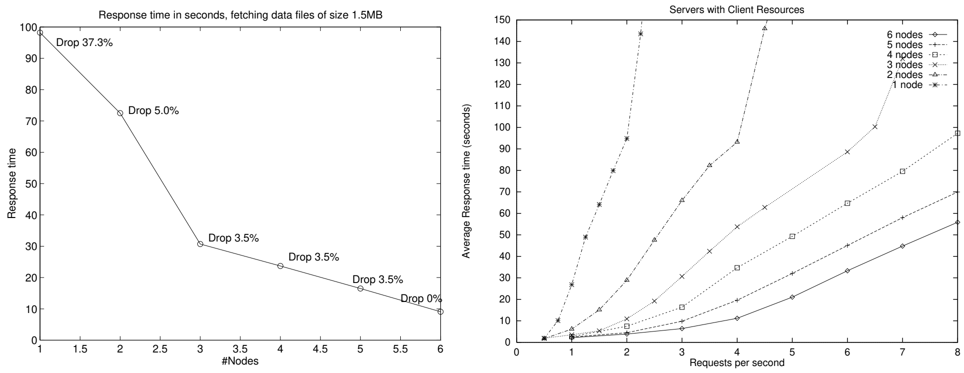
We report our experiments to examine the performance of our system in the following aspects. We first examine the performance impact of utilizing multiple servers and client resources. We proceed to discuss how our scheduling algorithm deals with "hot spot" situations to achieve proper load balancing. We also present experiments supporting the analysis model presented in Section 5. All results are averaged over multiple runs, and the test performance is affected by dynamically-changing system loads since the machines are shared by many active users at UCSB. The client machines are loaded with our custom Java applet library implementing some of the basic operations, including wavelet reconstruction. To avoid

Internet bandwidth fluctuations, clients are located within the campus network to assist in experimental stability over multiple runs.

We primarily examine the SWEB++ performance of wavelet subimage retrieval and file fetches. The wavelet code used for our experiments was developed in [27]. The wavelet operation we choose is to extract a 512×512 subregion at full resolution from a $2K \times 2K$ map image, representing the user zooming in on a point of interest at a higher resolution after examining at an image thumbnail. Examining file fetching helps us to understand the impact of server disk I/O and file size on the system behavior.

The overhead of our scheduling and load monitoring is quite small for all experiments. Analyzing a request takes about 2-4ms, and load monitoring takes about 0.1% of CPU resources. These results are very similar to those reported in [4].

6.1 The impact of adding multiple servers



(a) (b)

Figure 9: Request response times as p and/or RPS changes. (a) File fetches, 1.5MB files, 8RPS, (b) Wavelet subimage retrieval. The period is 30 seconds.

We examine the scalability of the multi-processor server with client resources. We run a test for a period of 30 seconds, at each second R requests are launched from clients. Figure 9(a) shows the average response times for 1, 2, 3, 4, 5 and 6 server nodes for fetching a set of 1.5MB files. Figure 9(b) shows the average response times with client resources for wavelet subimage retrieval. We can see from the experiment results that the response times decrease significantly by using multiple server nodes. Notice that for a heavily-load WWW server, the new requests may be dropped since the server may not be able to handle more requests. We mark the drop ratios for the file fetch experiment on the left of this figure and observe the drop ratios decrease substantially with multi-processor support.

We more closely examine the scalability issue by computing the relative scale-up ratio of performance from $i-1$ nodes to i nodes. This ratio is defined below where $H(i)$ is the response time using i nodes.

$$\Delta_{i-1}^{i} = \frac{\frac{H(i-1)}{H(i)}}{\frac{i}{i-1}}$$

The denominator indicates the server resource's increasing speed. If $H(i-1)$ is too large, then we use ∞ as the result. If $\Delta_{i-1}^{i} = 100\%$, it indicates a perfect performance scale-up. $\Delta_{i-1}^{i} > 100\%$ indicates a super-linear scale-up. This is possible because increased memory and disk resources reduce paging. Table 1 shows the relative scale-up ratio which clearly demonstrates the scalability achieved when the RPS varies for wavelets. The scale-up results are similar for file fetching.

	Δ_1^2	Δ_2^3	Δ_3^4	Δ_4^5	Δ_5^6
RPS=2	250%	200%	93%	106%	125%
RPS=4	∞	109.1%	125.3%	133.3%	175.9%
RPS=6		∞	102.9%	118.2%	117.2 %

Table 1: Relative scale-up ratios with client resource for wavelets.

6.2 The impact of utilizing client resources

We compare the improvement ratio of the response time $H(i)$ with client resource over the response time $H'(i)$ without using client resource (i.e. all operations are performed at server), which is $H'(i)/H(i) - 1$. The comparison result for p=6 is shown in Table 2 for wavelets. As the server load increases steadily, the response time improvement ratio increases dramatically.

RPS	1.0	1.5	2.0	2.5
Without client resources (sec)	15.97	57.24	123.76	213.4
With client resources (sec)	4.78	5.61	6.45	7.73
Improvement	234%	918%	1818%	2660%

Table 2: Average response time with and without client resource for wavelets. 30s. period, $p = 6$.

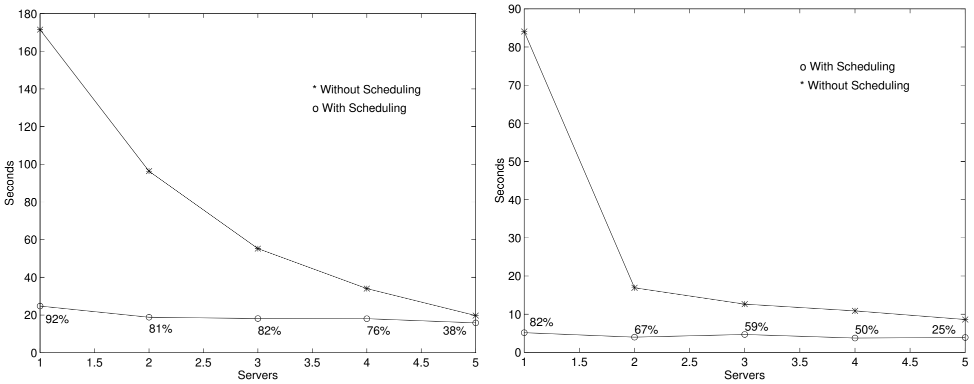
We also note a significant increase in the number of requests per second (RPS) a server system can complete by using client resources. If we consider a response time for more than 60 seconds as a failure in the case of wavelets, then the maximum RPS for the system with and without client resource is summarized in Table 3. Using client resources improves the maximum RPS for a server by approximately 5 to 6 times.

	p=1	2	3	4	5	6
With client resources	1.5	3.0	4.5	6.0	7.5	9.0
Without	0.3	0.6	0.8	1.2	1.4	1.5

Table 3: Approximated maximum RPS with and without client resources for wavelets.

6.3 Load balancing with "hot spots"

"Hot spots" is a typical problem with DNS rotation, where a single server exhibits a higher load than its peers. Various authors have noted that DNS rotation seems to inevitably lead to load imbalances [22, 24]. We examine how our system deals with hot-spots by sending a fixed number of requests to a subset of nodes in our server cluster, giving a wide range of load disparities. Without our scheduler, the selected nodes would have to process all of those requests. The SWEB++ scheduler can effectively deal with temporary hot-spots at various nodes by redirecting requests to other nodes in the cluster. The result is shown in Figure 10 for extracting a 512×512 pixel wavelet subimage (left) and extracting a 64×64 pixel image thumbnail (right). The X axis shows the range of processors which receive requests. The upper curves of Figure 10(a) and (b) show the average response time in seconds when no scheduling is performed and the request processing is limited to the fixed subset of nodes. The lower curves show the average response time with presence of our scheduler. We also mark the redirection rate for each case. We note that redirection rates drop dramatically as load disparity goes down. This trend matches our expectation because where the system tends to be homogeneous and the load is evenly balanced, our analytic model in Section 5 indicates the zero redirection ratio for both of the above experiments in which the file sizes they are dealing with are not large.

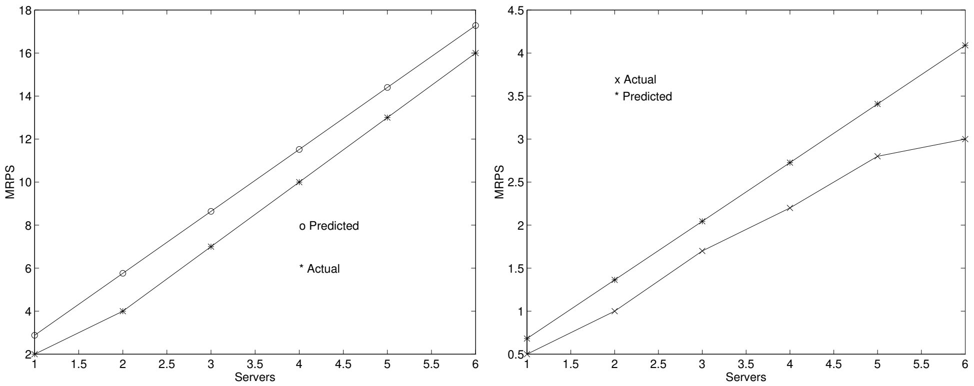


(a) 512×512 wavelet subimage. (b) 64x64 thumbnail.

Figure 10: System performance with request concentration at fixed server subsets. Tests for a period of 30 sec., 4 RPS. The indicated percentage is the redirection rate.

6.4 Verification of analytical results

We further conduct experiments on how the theoretical results presented in Section 5 match the system behavior under the specified assumptions.



(a) File fetch (1.5MB) (b) Wavelets.

Figure 11: Sustained MRPS, experimental vs. predicted results. The test period is 120s.

Sustained MRPS bound. For fetching a set of files of size 1.5MBytes and processing wavelet task chains, we have theoretical MRPS predictions in Section 5 using Meiko performance data. We ran experiments to determine the actual MRPS by testing for a period of 120 seconds and choosing the highest RPS such that the server response times are reasonable and no requests are dropped. The results are shown in Figure 11. We can see that the predicted MRPS bounds are reasonably tight compared to the actual MRPS. The actual numbers are somewhat lower than predicted due to paging and OS overheads which are neglected in our model.

Notice that for this experiment, the clients are simulated within the Meiko machine. The aggregated bandwidth (B_s) of the 6-node server to the other client nodes is high, and does not create a bottleneck. Thus when $p = 6$, the actual MRPS can reach almost 16 for fetching 1.5MB files.

Expected redirection ratio. Experiments in Section 6.3 already indicate that the trend of redirection matches the theoretical prediction for the 512×512 wavelet extraction and small file fetches. We further examine the impact of varying file sizes on the redirection ratio. Figure 12 shows the predicted and actual redirection ratios when the fetched file size varies. Utilizing the formula presented in Section 5.2.1, the difference between $\frac{F}{b_1} - \frac{F}{b_2}$ and $A + O$ determines the theoretical switching point. We use the following data: $A + O \approx 0.1s$, $b_1 = 1.1MB/s$, $b_2 = 1.0MB/s$. Then we can determine that at $F \approx 1.1MB$, d will shift from 0 to $1 - 1/p$. We find that the actual redirection rate curve is quite close to that predicted, although slightly shifted due to background system activity affecting algorithm parameters.

For wavelets, at the present time we do not have compressed wavelet files large enough to approach the predicted switch point. Thus the predicted redirection is a flat line and matches the actual ratio very well.

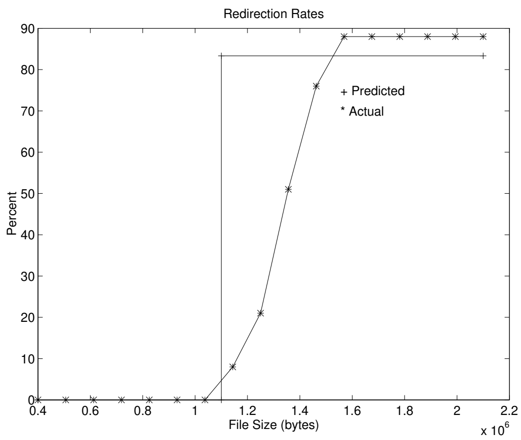


Figure 12: Redirection rates for file fetching. Experimental vs. predicted results.

Expected Split Points. In Figure 13, we compare the theoretically predicted split points with the actual decision made by SWEB++ in the wavelet experiments. The system is processing R concurrent requests (R=6 and 18) when we artificially adjust the server/client bandwidth and CPU ratio reported by the client. Each coordinate entry in Figure 13(a) and (b) is marked with the decision of scheduler. We overlaid the theoretical predictions from the formula of Section 5 on the system results. For each entry, if the choices for all requests agree with the theoretical prediction, we mark the actual selected split decision, otherwise we mark the percentage of disagreement. For example, in Figure 13(a), when the bandwidth is 10,000 bytes/sec. and server/client CPU ratio $S = 2$, D2 is the selected processing option for all requests, matching the analytical model's result. In Figure 13(b), when the bandwidth is 100,000 bytes/sec. and $S = 4$, the percentage of disagreement with the theoretical model is 6% among all processed requests.

We can see the theoretical model closely matches the system's selections. As bandwidth decreases, the scheduler would increase the percentage of requests who make use of client CPU for data decompression and image reconstruction, minimizing the data sent over the network. On the other hand, as client CPU speeds decrease, we expect the server to do more processing.

Client/Server Bandwidth (bytes/sec) \ Server/Client CPU Speed Ratio	0.25	0.5	1	2	4	10	20	40
100	D2	D2	D2	D2	D2	D2	D2	D2
500	D2	D2	D2	D2	D2	D2	D2	D2
1,000	D2	D2	D2	D2	D2	D2	D2	D2
10,000	D2	D2	D2	D2	D2	D2	D2	D2
100,000	D2	D2	D2	100	100	100	**D4**	**D4**
150,000	D2	D2	D2	**D4**	**D4**	**D4**	**D4**	**D4**
250,000	D2	D2	D2	**D4**	**D4**	**D4**	**D4**	**D4**
500,000	D2	D2	D2	**D4**	**D4**	**D4**	**D4**	**D4**
750,000	D2	D2	D2	**D4**	**D4**	**D4**	**D4**	**D4**
1,000,000	D2	D2	D2	**D4**	**D4**	**D4**	**D4**	**D4**

Predicted D2 region *Predicted D4 region*

(a) $R = 6$

Client-server bandwidth (bytes/sec) \ Server/Client CPU Speed Ratio	0.25	0.5	1	2	4	10	20	40
100	D2	D2	D2	D2	D2	D2	D2	D2
500	D2	D2	D2	D2	D2	D2	D2	D2
1,000	D2	D2	D2	D2	D2	D2	D2	D2
10,000	D2	D2	D2	D2	D2	D2	**D4**	**D4**
100,000	D2	D2	D2	76	**6**	**D4**	**D4**	**D4**
150,000	D2	D2	D2	77	**D4**	**D4**	**D4**	**D4**
250,000	D2	D2	D2	93	**D4**	**D4**	**D4**	**D4**
500,000	D2	D2	D2	95	**D4**	**D4**	**D4**	**D4**
750,000	D2	D2	D2	88	**D4**	**D4**	**D4**	**D4**
1,000,000	D2	D2	D2	98	**D4**	**D4**	**D4**	**D4**

Predicted D2 region *Predicted D4 region*

(b) $R = 18$

Figure 13: Effects of CPU speed and network bandwidth on wavelet chain partitioning decisions.

7 Related work and concluding remarks

The evolution of the WWW brings tremendous changes for interactive client-server Web-based global computing. Compared to the previous SWEB work [4], the main contributions of SWEB++ are an adaptive scheduling model for processing requests by utilizing both client and WWW server resources, a software tool prototype for WWW programmers to incorporate this adaptive scheduling model in developing their applications, and an analytic framework for identifying the performance impact of various system resources. The experimental results with SWEB++ show that proper monitoring and utilizing of server and client resources significantly enhance the scalability of WWW-based services.

Several projects are related to our work. The WWWVM project [13] is aimed at providing meta-computing services by integrating resources of several WWW servers. There are other projects [11, 18, 26] also working on global computing software infrastructures. Our current project focuses on the optimization for single server/client pair and the nodes in the WWW server are relatively tightly coupled. The AppLeS project [9] addresses the scheduling issues in heterogeneous computing and also uses network bandwidth and load information. Their work deals with an integration of different machines as one server and does not have the division of client and server. Addressing client configuration variation is discussed in [15] for filtering multi-media data in order to reduce network bandwidth requirements but does not consider the use of client resources for integrated computing. The classical work on load balancing for distributed systems [14, 28, 29] usually assumes that task resource requirements are unknown, and the criteria for task migration are usually based on a single system parameter, i.e., the CPU load. In WWW applications, we need to consider multiple parameters that affect the system performance, including CPU loads, interconnection network performance and disk channel usages. In [17], multiple resource requirements are predicted and suggested to guide the load sharing, our work provides a more comprehensive scheme focusing on client-server computing environments.

Several issues still remain to be explored. For example, heterogeneous clusters with nonuniform machine architectures remain to be tested and the task chain model needs to be generalized.

Acknowledgments

This work was supported in part by NSF IRI94-11330, NSF CCR-9409695 and a grant from NRaD. We would like to thank David Watson who helped in the implemention of SWEB++, Thanasis Poulakidas for assisting us in using the wavelet code, Omer Egecioglu, Oscar Ibarra, Terry Smith Cong Fu and the Alexandria Digital Library team for their valuable suggestions.

References

[1] The AltaVista: Main Page, http://altavista.digital.com/.

[2] E. Anderson, D. Patterson, E. Brewer, "The Magicrouter, an Application of Fast Packet Interposing", submitted to the Second Symposium on Operating System Design and Implementation (OSDI96), 1996.

[3] D.Andresen, L.Carver, R.Dolin, C.Fischer, J.Frew, M.Goodchild, O.Ibarra, R.Kothuri, M.Larsgaard, B.Manjunath, D.Nebert, J.Simpson, T.Smith, T.Yang, Q.Zheng, "The WWW Prototype of the Alexandria Digital Library", *Proceedings of ISDL'95: International Symposium on Digital Libraries,* Japan August 22 - 25, 1995.

[4] D.Andresen, T.Yang, V.Holmedahl, O.Ibarra, "SWEB: Towards a Scalable World Wide Web Server on Multicomputers", *Proc. of 10th IEEE International Symp. on Parallel Processing (IPPS'96)*, pp. 850-856. April, 1996.

[5] D. Andresen, T. Yang, O. Egecioglu, O.H. Ibarra, T.R. Smith, "Scalability Issues for High Performance Digital Libraries on the World Wide Web", Proc. of the 3rd Forum on Research and Tech. Advances in Digital Libraries (ADL96), pp. 139-148, May, 1996.

[6] M. Arlitt, C. Williamson, Web Server Workload Characterization: The Search for Invariants, Proc. SIGMETRICS Conference, Philadelphia, PA, May, 1996.

[7] T. Bemers-Lee and D. Connolly, Hypertext Markup Language - 2.0, http://www.w3.org/hypertext/WWW/MarkUp/html-spec/html-spec_toc.html, June 16, 1995.

[8] T. Bemers-Lee, Uniform Resource Locators, http://www.w3.org/hypertext/WWW/Addressing/URL/, 1996.

[9] F. Berman, R. Wolski, S. Figueira, J. Schopf, G. Shao, "Application-Level Scheduling on Distributed Heterogeneous Networks", Submitted to Supercomputing '96.

[10] E.C.K. Chui, Wavelets: A Tutorial in Theory and Applications, Academic Press, 1992.

[11] H. Casanova, J. Dongarra, NetSolve: A network server for solving computation science problems, To Appear in Supercomputing'96, ACM/IEEE, Nov 196.

[12] M. Crovella, A. Bestavros, "Self-Similarity in World Wide Web Traffic Evidence and Possible Causes", *Proc. SIGMETRICS96*, Philadelphia, PA, May, 1996.

[13] K. Dincer, and G. C. Fox, Bulding a world-wide virtual machine based on Web and HPCC technologies. To Appear in Supercomputing'96, ACM/IEEE, Nov 196.

[14] D. L. Eager, E. D. Lazowska, and J. Zahojan, Adaptive load sharing in homogeneous distributed systems, IEEE Trans on Software Engineering, 12(5):662-675. 1986.

[15] A. Fox, E. Brewer, "Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation", Computer Networks and ISDN Systems, Volume 28, issues 711, p. 1445. May, 1996.

[16] E. Fox, Akscyn, R., Furuta, R. and Leggett, J. (Eds), Special issue on digital libraries, *CACM*, April 1995.

[17] K. Goswami, M. Devarakonda, R. Iyer, Prediction-based Dynamic Load-sharing Heuristics, *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 638-648, June, 1993.

[18] A. S. Grimshaw, A. N. Tuong, W. A. Wulf, Campus-Wide Computing: Results Using Legion at the University of Virginia, UVa CS Technical Report CS-95-19, March 27, 1995.

[19] M. Hamilton, "Java and the Shift to Net-Centric Computing", IEEE Computer, pp. 31-39, August, 1996.

[20] R. Hof, "These may really be the PCs for the rest of us", *Business Week*, pp. 76-78, June 24, 1996.

[21] Hypertext Transfer Protocol(HTTP): A protocol for networked information, http://www.w3.org/hypertext/WWW/Protocols/HTTP/HTTP2.html, June 26, 1995.

[22] E.D. Katz, M. Butler, R. McGrath, A Scalable HTTP Server: the NCSA Prototype, *Computer Networks and ISDN Systems.* vol. 27, 1994, pp. 155-164.

[23] Meiko Corp., Computing Surface CS-2 Product Description, Meiko, Waltham, MA, 1994.

[24] D. Mosedale, W. Foss, R. McCool, "Administering Very High Volume Internet Services", *1995 LISA IX*, Monterey, CA, September, 1995.

[25] NCSA development team, The Common Gateway Interface, http://hoohoo.ncsa.uiuc.edu/cgi/, June, 1995.

[26] Ninf, A network based information library for global world-wide computing infrastructure, http://hpc.etl.go.jp/NinfDemo.html, 1996.

[27] A. Poulakidas, A. Srinivasan, O. Egecioglu, O. Ibarra, and T. Yang, Experimental Studies on a Compact Storage Scheme for Wavelet-based Multiresolution Subregion Retrieval, Proceedings of NASA 1996 Combined Industry, Space and Earth Science Data Compression Workshop, Utah, April 1996.

[28] B. A. Shirazi, A. R. Hurson, and K. M. Kavi (Eds), Scheduling and Load Balancing in Parallel and Distributed Systems, IEEE CS Press, 1995.

[29] M. Willebeek-LeMair and A. Reeves, Strategies for Dynamic Load Balancing on Highly Parallel Computers, *IEEE Trans. on Parallel and Distributed Systems,* vol. 4, no. 9, September, 1993, pp. 979-993.