SWEB++: Partitioning and Scheduling for Adaptive Client-Server Computing on WWW

Daniel Andresen
Dept. of Computing and Info. Sciences
234 Nichols Hall
Kansas State University
Manhattan, KS 66506
dan@cis.ksu.edu

Tao Yang
Dept. of Computer Science
University of California
Santa Barbara, CA 93106
tyang@cs.ucsb.edu

Abstract

The SWEB++ project studies runtime partitioning, scheduling and load balancing techniques for improving the performance of on-line WWW-based information systems such as digital libraries. The main performance bottlenecks of such a system are caused by the server computing capability and Internet bandwidth. Our observations and solutions are based on our experience with the Alexandria Digital Library (ADL) testbed at UCSB, which provides on-line browsing and processing of documents, digitized maps and other geo-spatially mapped data via WWW. A proper partitioning and scheduling of computation and communication in processing a user request on a multiprocessor server and transferring some computation to client-site machines can reduce network traffic and substantially improve system response time. We have proposed a partitioning and scheduling mechanism that adapts to resource changes and optimizes resource utilization. We have developed a software tool which implements and supports the use of our scheduling strategies when programming WWW applications, and have demonstrated the application of this system for on-line DL information browsing. We have conducted a set of experiments to examine the system performance on the Meiko parallel machine and a cluster of workstations.

1 Motivations

The number of digital library (DL) projects is increasing rapidly at both the national and the international levels (see, for example, [2, 11]) and they are moving rapidly towards supporting on-line retrieval and processing of major collections of digitized documents over the Internet via the WWW. Performance and scalability issues are especially important for DLs. Many collection items have sizes in the gigabyte range while others require extensive processing to be of value in certain applications. Critical performance bottlenecks, that must be overcome to assure adequate access over the Internet, involve server processing capability and network bandwidth. Considering that popular WWW sites such as AltaVista already have millions of requests a day, the server performance must scale to match expected demands. While we expect network communication technology to improve steadily, particularly with the advent of ATM and ADSL, we still need to consider the minimization of network traffic in the design of a WWW system.

Our research is motivated by the above situation and develops solutions addressing performance issues of WWW-based applications. In [4, 5], we have studied issues in developing a WWW server cluster 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. 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. However, 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 Internet networking protocols built in. Carefully designed scheduling strategies are needed to avoid imposing too much burden on these clients. At the server site, information on the current system load and disk I/O bandwidth affects the selection of a server node for processing a request. In addition to this, the impact of available bandwidth between the server and a client needs to be incorporated. Thus dynamic scheduling strategies must be adaptive to variations of client/server resources in multiple aspects.

In the SWEB++ project, we have studied a model for characterizing computation and communication demands of WWW-based information access requests and investigated a partitioning and scheduling scheme to optimize the use of multiprocessors, parallel I/O, network bandwidths and client resources. The scheduling decision adapts to dynamically changing server and client capabilities. We have developed a software tool which implements and supports the use of our scheduling strategies when programming WWW applications. The paper is organized as follows: Section 2 discusses our client-server model, examples of client-server task partitioning, and scheduling strategies. Section 3 presents the software architecture of SWEB++ in supporting the use of client-server task partitioning and mapping for a Web server cluster with client resources. Section 4 presents experimental results and verifies our analytical results. Section 5 discusses related work and conclusions. An analysis of our scheduling model in a homogeneous environment is reported in [3].

2 WWW request processing

We first present a model for WWW request processing, give two applications to demonstrate the use of this model, and then briefly discuss our partitioning and scheduling scheme.

2.1 The model of client-server computing on WWW

Our WWW server cluster consists of a set of nodes connected with a fast network and presented as a single logical server to the Internet. User requests are first evenly routed to processors via DNS rotation [4, 12]. Each server node may have its local disk, which is accessible to other nodes via remote file service in the OS. Server nodes in the system communicate with each other and redirect requests to the proper node by actively monitoring the usages of CPU, I/O channels and the interconnection network.

WWW applications such DLs involve extensive client-server interaction, and some of computation can be shifted to the client (Figure 1). In this paper we model the interaction between client and server using a task chain which is partially executed at the server (possibly as a CGI program) and partially executed at the client (possibly as a Java applet). A task consists of a segment of the request fulfillment, with its associated computation and communication. Task communication costs differ depending on whether task results must be sent over the Internet or can be transferred locally to the next task in the task chain. The following items appear in a task chain definition: 1) A task chain to be processed at the server and client site machines. A dependence edge between two tasks represents a producer-consumer relation with respect to some data items. 2) For each task, specify the input data edge from its predecessor, data items directly retrieved from the server disk and data items available at the client site memory. It should be noted that if a task is performed at a client machine, some data items may be available at this machine and slow communication from the server can be avoided. One such an example is wavelet-based image browsing, discussed in the next section.

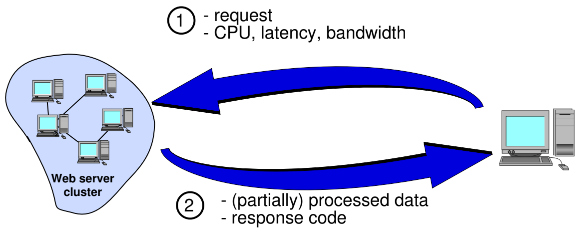


Figure 1: SWEB++ client-server model.

Each task chain is scheduled onto one of the server nodes. Our challenge, then, is to select an appropriate node within the server cluster for processing, partitioning the tasks of the chain into two sets, one for the client and another for server, such that the overall request response time is minimized. In addition to considering the balancing between client and server machine load and capability, the network bandwidth between client and server affects the partitioning point. We assume that the local communication cost between tasks within the same partition (client or server) is zero while client-server communication delay is determined by the latency and current available bandwidth between them.

2.2 Application examples

Text extraction from a Postscript file. We demonstrate the use of above model in Postscript document browsing. The extraction of the plain text from a Postscript-formatted document is an application which can benefit strongly from client resources, but requires dynamic scheduling. This application is useful for content replication, information retrieval for non-Postscript enabled browsers, and other tasks [10]. Figure 2 depicts a task chain for processing a user request which extracts text from a subset of Postscript pages. The chain has two tasks that can be performed either at server or at client. 1) *Select the pages needed.* Eliminating unnecessary postscript pages reduces the computation needed for Step 2. The time to do this is small compared to the rendering time in the next task. 2) *Extracting the text.* A page is rendered via a software Postscript interpreter, and the text is extracted for the client. This step takes a significant amount of processing time. Thus there are three possible split points illustrated in Figure 2. We explain them as follows: D_1 - Send the entire Postscript file to the client and let it do everything. D_2 - Send over the relevant portions of the Postscript file for the client to process. D_3 - Extract the text on the server and send it over to the client for viewing.

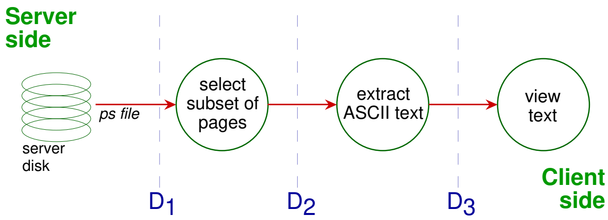


Figure 2: The task chain for text extraction from a Postscript file.

Dynamic scheduling is needed for balancing bandwidth and processing requirements. Postscript files are typically large, and so in most situations require a large amount of time to transmit. Text extraction dramatically reduces the size of the data to be transferred, but imposes a large computational burden. For example, extracting the text from a 25-page, 750KB technical paper takes about 50 seconds on a Sparc Ultra-1 workstation, with an output of about 70KB of text. Thus if the server does the processing, approximately 90% of the bandwidth requirements can be avoided, but this imposes a large amount of work on the server. The scheduler must determine a proper split point as a function of bandwidth and available processing capability.

Multi-resolution image browsing. In the Alexandria project, a progressive multi-resolution and subregion browsing strategy is adopted to reduce Internet traffic in accessing map images. 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 images [5, 7]. Figure 3 depicts a task chain for processing a user request to access an image or subimage at a higher resolution, based on the implementation in [14].

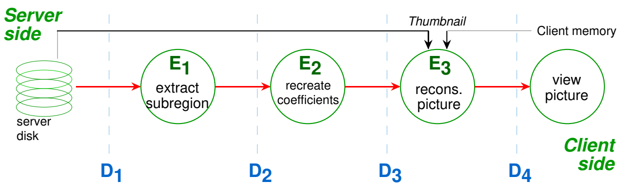


Figure 3: A task chain for image reconstruction.

Our model of wavelet computation and communication uses the chain of tasks (E_{1-3}) depicted in Figure 3, which are: 1) Fetch and extract the subregion indexing data when a user wants to review a subregion. 2) Use the indexing data to find coefficient data to be used in constructing a higher resolution subregion image. 3) Use the coefficient data and thumbnail image to perform a wavelet inverse transform in producing a higher resolution image. 4) View the image. Thus there are four possible split points between a client and a server in Figure 3. D_1 : Send the entire compressed image to the client and let it do everything. D_2 : Send over the relevant portions of the compressed image for the client to process. D_3 : Recreate the coefficients from their compressed representation and send them to the client for reconstruction. D_4 : Recreate the image requested by the client and send the pixels to the client. For split point D_3, the thumbnail file is available at the client machine, thus the data access from the server disk for thumbnail data is eliminated.

2.3 Partitioning and scheduling for request processing

Given the arrival of HTTP request s at node x, The server parses the HTTP command, and expands the incomplete pathname. 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 request to the server node with the lowest load. Otherwise the system will analyze the request, select a partitioning point and server node for the minimum response time using the cost function discussed below. If the chosen server node is not x, the request is redirected appropriately. Otherwise, a part of this chain is executed at this server node and the remaining part of a task chain will be further executed at the client machine. No requests are allowed to be re-directed more than once, to avoid the ping-pong effect.

The predicted cost for processing a request on a node is:

$$t_s = t_{redirection} + t_{data} + t_{server} + t_{net} + t_{client}.$$

$t_{redirection}$ is the cost to redirect the request to another processor, if required. t_{data} is the server time to transfer the required data from a server disk drive. If data files are local, the time required to fetch is approximated as the file size divided by the available bandwidth of the local storage system. If data files are remote, then each file must be retrieved through

[1] A file fetch can also be modeled as a chain[4].

the interconnection network. Then t_{data} is approximated as the file size divided by the available remote accessing bandwidth. If there are many concurrent requests, local data transmission performance degrades accordingly. At run-time, the system needs to monitor the disk channel load of each local disk and also available remote data accessing bandwidths. t_{server} is the time for required server computation. t_{net} is the cost for transferring processing results over the Internet, which is the size of client-server communication volume divided by available network bandwidth. t_{client} is the time for any client computation required, which is the number of client operations required divided by client speed. Here we assume the speed reported by the client machine includes client load factors.

In the implementation, we need to collect three types of dynamic load information: CPU, disk, and network. CPU and disk activity can be derived from the Unix *rstat* utility, as well as some network information. A daemon (*loadd*) periodically updates the above information between the server nodes. Latency to the client can be approximated by the time required for the client to set up the TCP/IP connection over which the request (with the latency estimate) is passed. Client bandwidth is determined by the client, which measures the number of bytes per second received from the server for messages over a minimum size. The client passes both the latency and bandwidth estimates to the server as arguments to the HTTP request.

3 The SWEB++ Architecture

The SWEB++ system is designed to assist the programming of WWW applications which can be scheduled between a multi-processor server and a client, making use of the resources of both to minimize response times. The current client-server computational model is based on task chains, as 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.

At run-time, the scheduler at each server node interacts with an HTTP server (NCSA httpd) 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:

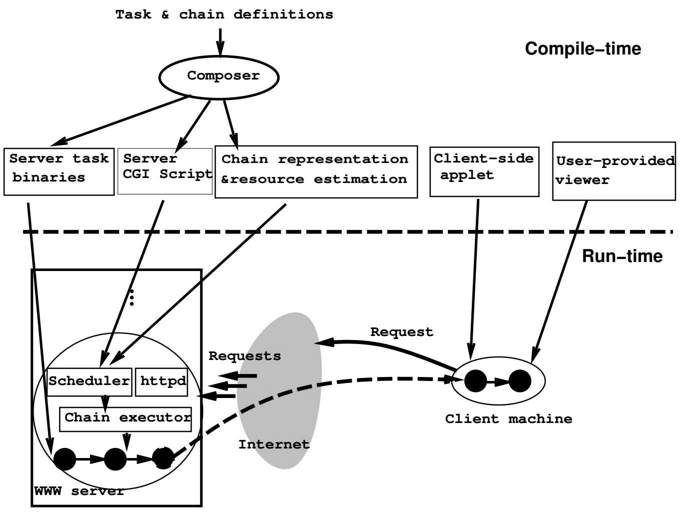


Figure 4: Components of the SWEB++ architecture

<task chain key> ? <argument list>.

For example, in the URL "/cgi-bin/wavelet?sub_extract=pacific", "wavelet" corresponds to a task chain key name. The argument "pacific" 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 2.3.

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 used to characterize the tasks and task chains. TDL is similar to the Durra language proposed in [15], 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. We explain the primary TDL elements for defining task chains and tasks below and give several small examples.

```
#sample task chain definition for wavelets
CHAIN wavelets IS wav_extract_subtree,
     wav_recreate_coefficients, wav_create_ppm,
     view <
    REQUEST_KEY = wavelet
    MINIMUM_SPLIT = 0
    MAXIMUM_SPLIT = 2
>
```

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

```
#task definition for expanding compressed
#wavelet coefficients
TASK wav_expand_coefficients  <
  EXECUTION_LOCATION = EITHER
  SERVER_BINARY = "wavelet.recreate_coeff"
  CLIENT_BINARY = "wavelet.recreate_coeff.clnt"
  OUTPUT_TYPE = "x-wav/uncompressed_coeff"
  ARGUMENT_KEY = REC_COEFF
  MULTI_DEF = FUNC
    wavelet_req_info* t = analyze_wavelet(rq);
            s->cpu_cycles = t->pixels * WAV_EXP_COEFFS;
            s->out_bytes =  4 * t->pixels;
    delete t;
    FUNC_END
  SPLIT_SERVER_COMPUTATION = FUNC
          SPLIT_SERVER_COMPUTATION = 20*OUTPUT_SIZE;
          FUNC_END
  SPLIT_CLIENT_COMPUTATION = FUNC
          //decompress the coefficients after transfer
          SPLIT_CLIENT_COMPUTATION = 20*OUTPUT_SIZE;
          FUNC_END
>

TASK view <
    EXECUTION_LOCATION = CLIENT_ONLY
    COMPUTATION = 0
>
```

Figure 6: Specification of two tasks in the wavelet chain.

Task chains. Expressing a simple task chain in TDL is straightforward. For example, in Figure 5, we have the task chain as implemented corresponding to the abstract chain listed in Figure 3.

A generic task chain specification is started with keyword "CHAIN" and has the following format:

```
CHAIN <task chain name> IS
    <ordered task list> <
    REQUEST_KEY = <chain key>
    MINIMUM_SPLIT =
        <earliest possible split point>
    MAXIMUM_SPLIT =
        <last possible split point>
>
```

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.

The elements of TDL specific to task chains are:

- *REQUEST_KEY* defines the string which, if it appears in the URL, denotes are request of this particular chain type. Usually this would be the name of the CGI program associated with the task chain.
- *MINIMUM_SPLIT and MAXIMUM_SPLIT* define the first and last tasks, respectively, which can be accomplished on both the client and the server, and hence can serve as potential split points.

Tasks. Figure 6 gives a specification for a subset of the wavelet tasks in Figure 3. Each task specification started with keyword "TASK" has the following format:

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

The various task attributes include setting the output type (*OUTPUT_TYPE*), location of execution (client or server; *EXECUTION_LOCATION*), amount of computation required, and others. The value given to an attribute can also take the form of a C++ function, providing additional flexibility. The details can be found in [1].

3.3 Application integration

To integrate an application with SWEB++ after creating a TDL specification and compiling it via the Composer, several steps need to be completed. There are four major elements which need to be manually put into place: the Composer-generated shell script and oracles, the server binaries, and client binaries.

The CGI shell script generated by the Composer controls the execution of the server's portion of the chain. Consisting of a case statement for each possible breakpoint in the chain, it halts chain execution at the split point chosen by the scheduler and passes the results so far to the client with the appropriate content type label.

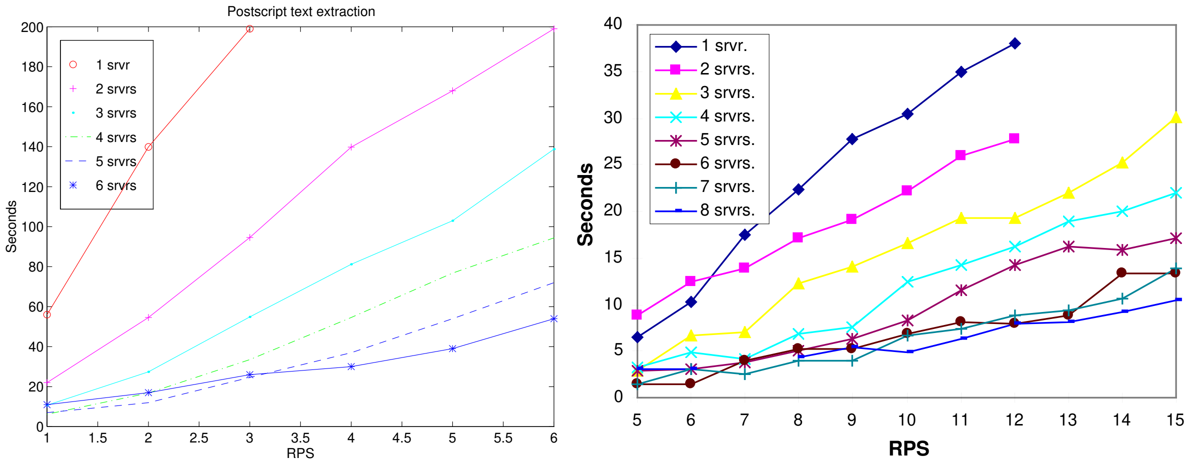
Integration with the SWEB++ client-side libraries is done manually, generally requiring the replacement of calls to the *URL* classes be replaced with calls to the *swebURL* class. This provides for the automatic addition of the SWEB++ arguments to the URL, as well as informing the server that the client is capable of properly handling partially-completed requests. The *swebURL* classes should must be made available in the server download directories.

4 Experimental Results

We have implemented a prototype of SWEB++ on a Meiko CS-2 distributed memory machine and a cluster of SUN Ultra workstations linked by 100Base-T Ethernet. The Meiko CS-2 can be viewed as a workstation cluster connected by the Elan fast network. Each node has a a 40Mhz SuperSparc chip with 32MB of RAM running SUN Solaris. Our primary experimental testbed on Meiko consists of six CS-2 nodes as our distributed server.

We first examine the performance impact of utilizing multiple servers and client resources, then demonstrate that our scheme can successfully balance processor loads when several nodes receive more requests compared with others. We primarily examine the scheduling performance on two applications: text extraction for postscript documents, and wavelet-based subimage retrieval. Each text extraction request consists of extracting one page of text from a 45-page Postscript file of size 1.5MB. The Postscript code for the single page is approximately 180KB, and the extracted text is about 2.5KB. The wavelet operation 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. The client machines are loaded with our custom library implementing some of the basic operations, including wavelet reconstruction. Clients are located within the campus network to avoid Internet bandwidth fluctuations over multiple experiments. The overhead for monitoring and scheduling is quite small for all experiments. Analyzing a request takes about 2-4ms, and monitoring takes about 0.1% of CPU resources [4]. A complete set of experiments is reported in [1].

The impact of adding multiple servers. We examine how average response times decrease when the number of server nodes (p) increases for a test period of 30 seconds, and at each second R requests are launched from clients ($RPS = R$). RPS stands for requests per second. Figure 7 shows the average response times in seconds with client resources for processing a sequence of wavelet and Postscript



(a) (b)

Figure 7: Request response times as RPS changes. (a) Postscript text extraction on the Meiko, (b) wavelet image generation on a SUN cluster. The period is 30 seconds.

RPS	0.5	1	2	3	4
Without (sec)	73.3	132.5	160	294	407
With (sec)	12.8	13.4	15.2	20.6	32.6
Imp. ratio	617%	889%	953%	1327%	1148%

Table 1: Average response time with and without client resources for text extraction on Meiko. $p = 6$.

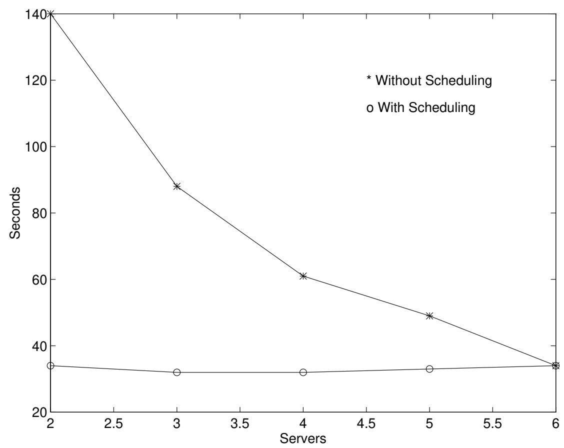
requests. We can see from the experimental results that response times decrease significantly by using multiple server nodes, and this is consistent across all types of loads. The extreme slope for the one-node server is due to the non-linear effects of paging and system overhead for a very high system load. We can also see the effect of limited network bandwidth in Figure 7, where after approximately five servers are used the remaining speedups become marginal due to the saturation of the 10Base-T Ethernet network link to the clients.

The impact of utilizing client resources. We compare the improvement ratio of 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). This ratio is defined as $H'(i)/H(i) - 1$. The comparison result for $p = 6$ on Meiko is shown in Table 1 for processing a sequence of Postscript text extraction requests for a period of 30 seconds. Table 2 is for wavelets. As the server load increases steadily, the response time improvement ratio increases dramatically.

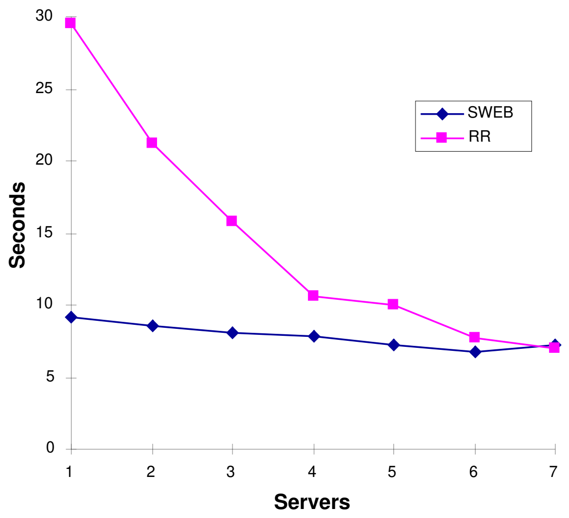
RPS	1.0	1.5	2.0	2.5
Without	15.97	57.24	123.76	213.4
With	4.78	5.61	6.45	7.73
Imp. Ratio	234%	918%	1818%	2660%

Table 2: Average response time with and without client resources for wavelets on Meiko. $p = 6$. Times in seconds.

We also note a significant increase in the maximum number of requests per second (MRPS) a server system can complete over short periods by using client resources. If we consider a response time of more than 60 seconds as a failure in the case of wavelets, then the MRPS for the system with and without client resources in pro-



(a) Postscript text extraction (Meiko).



(b) 512×512 wavelet subimage (SUN cluster).

Figure 8: System performance with request concentration at fixed server subsets. Tests for a period of 30 sec., 4 RPS.

cessing a sequence of wavelet-based requests is summarized in Table 3. Using client resources improves MRPS of a server by approximately 5 to 6 times.

	p=1	2	3	4	5	6
With	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: Bursty MRPS with and without client resources for processing wavelets on Meiko. Times in seconds.

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 [12, 13]. 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 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 8 for extracting the text from a Postscript page on the Meiko (left) and extracting a wavelet subimage on the SUN cluster (right). The X axis shows the range of processors which receive requests. The upper curves of Figure 8(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.

5 Related work and conclusions

Several projects are related to our work. Projects in [8, 9] are working on global computing software infrastructures. The above work deals with an integration of different machines as one server and does not have the division of client and server. Our current project focuses on the optimization between a server and clients and currently uses tightly coupled server nodes for a WWW server, but results could be generalized for loosely coupled server nodes. Addressing client configuration variation is discussed in [10] for filtering multi-media data but it does not consider the use of client resources for integrated computing.

Load balancing for Web servers is addressed in [4, 6], the main contributions of our SWEB++ work are adaptive partitioning and scheduling for processing requests by utilizing both client and multiprocessor server resources and a software tool prototype for WWW programmers to incorporate this model in developing their applications. The assumptions in the scheduling algorithm are simplified, but the results help us understand the performance impact of several system resources and corroborate the design of our techniques. The experimental results show that properly utilizing server and client resources can significantly reduce application response times.

Acknowledgments

This work was supported in part by NSF CCR-9702640, IRI94-11330, CDA-9529418, and a Navy NRAD grant We would like to thank David Watson and Vegard Holmedahl who helped in the implementation,Terry Smith, and the Alexandria Digital Library team for their valuable suggestions.

References

[1] D. Andresen, Distributed Scheduling and Software Support for High Performance WWW Applications. PhD Thesis, University of California at Santa Barbara, Oct, 1997.

[2] 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, and Q. Zheng, "The WWW Prototype of the Alexandria Digital Library", *Proc. of ISDL'95: International Symposium on Digital Libraries,* Japan, 1995.

[3] D. Andresen and T. Yang, Multiprocessor Scheduling with Client Resources to Improve the Response Time of WWW Applications, *Proceedings of the 11th ACM/SIGARCH Conference on Supercomputing (ICS'97)*, Vienna, Austria, July, 1997.

[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 IEEE ADL'96 (Advances in Digital Libraries)*, pp. 139-148, May, 1996.

[6] M. Colajanni, P. Yu, and D. Dias, Scheduling algorithms for distributed Web Servers, *Proc. of Inter. Conf on Distributed Computing Systems*, 1997. pp. 169-176.

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

[8] H. Casanova, J. Dongarra, "NetSolve: A network server for solving computation science problems", *Proc. of Supercomputing'96*, ACM/IEEE, Nov., 1996.

[9] K. Dincer, and G. C. Fox, "Building a world-wide virtual machine based on Web and HPCC technologies", *Proc. of Supercomputing'96*, ACM/IEEE, Nov., 1996.

[10] 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.

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

[12] 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.

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

[14] A. Poulakidas, A. Srinivasan, O. Egecioglu, O. Ibarra, and T. Yang, "A Compact Storage Scheme for Fast Wavelet-based Subregion Retrieval", in *Proc. of 1997 International Computing and Combinatorics Conference (COCOON)*, Shanghai, China, August, 1997.

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