Profiling Apache Web Server on IA-64 Linux

Course Project
CS 423, Fall 2001

Srikanth Kandula [kandula@uiuc.edu]
Sankalp Singh [ssingh7@uiuc.edu]

December 5, 2001

1 Introduction

Internet has become a part and parcel of our daily lives. Arguably, this great popularity of the Internet is due largely to the explosive growth of the World Wide Web. Web servers are the core components that support web access through HTTP and other protocols. Web servers thus occupy a vital position in the critical path of any web-based activity. More and more corporations and organizations are depending on high performance Web servers for the success of their businesses. Instances of popular Web sites registering tens of millions of hits per day are not uncommon.

There have been three traditional ways for a Web site to handle high volume of traffic, namely *replication (mirroring), distributed caching and improving server performance.* While considerable effort has gone into studying the first two, and many times leading to effective techniques, much lesser attention has been given to improving the Web server performance. It is important as even with replication and caching, a large number of requests may still hit the original Web server for a busy Web site. Also, more and more pages today are dynamically generated and hence cannot be cached or replicated.

The progress in this regard has been hampered due to the publishing of very few results that quantitavely study Web server performance and identify bottlenecks. In this project we try to measure and analyze the performance of the Apache Web server on 2-CPU SMP IA-64 (Itanium) machine running Linux 2.4.9. Apache is by far the most widely deployed Web server on the Internet [1]. And add to it the advantage that Apache is an open-source software, with the source code freely available for modification and instrumentation. IA-64 is well-suited as the architecture of choice for this study for various reasons. It provides an easy, very fine-grained and accurate handle to perform performance monitoring and profiling through hardware support in the form of its Performance Monitoring Unit. The implementation of the monitoring in hardware is a significant step in decreasing the overheads and increasing the accuracy. The Linux-2.4.9 kernel provides access to most of the features of the IA-64 PMU. IA-64 based machines are also expected to establish a significant share in the high-end server market—machines that usually host Web server software.

While several academic and commercial efforts have been made to *benchmark* Web server software [2][3], most of these measure the Web server performance from a client's perspective, giving metrics like throughput and latency. There have not been many profiling efforts on the server-side. This has been due in part to lack of kernel and hardware support for such profiling. The present work tries to precisely measure and analyze the performance of the Apache Web server software using profiling at the server-side. We suggest some improvements based on our observations, but we also believe that this work lays the groundwork for other researchers/developers to try out potential improvements.

This work involved a multitude of tasks. We wrote distributed Web clients for generating realistic workloads to model usual traffic patterns, and also observed server performance at the client-end. We designed appropriate metrics, and then obtained them by profiling Apache Web server under the workloads generated by the clients. This involved instrumentation of the Apache code and experimentation. We further analyzed the profile data generated by aggregating it along various dimensions and visualizing it through graphs and charts.

The rest of this report is organized as follows. Section 2 provides some background on Apache and IA-64 Performance Monitoring Unit necessary to understand the rest of the report. Section 3 describes the overall architecture as well as the architecture of the load generators in some detail. Section 4 describes how we instrumented the Apache code. Section 5 describes the metrics we used to evaluate Apache's performance. Section 6 summarizes our results. Section 7 describes some improvements we suggest for increased Apache performance on Itanium. Section 8 describes some modifications we made to the original design. We describe our experiences with Itanium in Section 9, and Section 10 provides a synopsis and concluding remarks.

2 Background

2.1 Apache Web server

Apache [4] is a freely available, open-source, UNIX-based Web server. It is also the most widely deployed Web server on the Internet—all these reasons contributing to our decision to choose it for our experiment and analysis.

The main Web server binary is called *httpd* and the configuration file is named *httpd.conf*. Apache, though written completely in C, adheres closely to the object-oriented philosophy. It is organized into modules, each of which performs some distinct task. Each module exports various kinds of functions, like event handlers, directive handlers, functions to be called when a new child process is created etc., and the functionality of the module is accessed only through this exported interface. Each module has an associated set of directives which are the configuration parameters for the module, specified in httpd.conf. Each module has a separate source file, which also includes the functions for taking appropriate action when its directives are encountered in httpd.conf.

This modularity has the advantages of the making the code clean, organized, easy to understand and easy to change. It also makes the server extensible—it is not very difficult to integrate a new external module into Apache. The list of the base modules is provided in Appendix A.

We have used Apache Web server version 1.3.22. As of this version, Apache is a multiprocessed program. When Apache starts, the main process forks into several child processes that are responsible for handling the incoming HTTP requests. When a request comes in, one of the child processes gets it and first parses the URL. If the request is for a static file, it checks the file states, performs security checking, opens the file, reads its contents and sends them to the client. If the request is for dynamic content, say CGI script, it sets the environment, starts the script as another process, and redirects its standard output to the client. If on the arrival of a request, there is no free handler, a new one is created. This continues until an upper limit is reached after which new requests are simply queued-up.

2.2 IA-64 (Itanium) Performance Monitoring Unit

Several researchers have shown that the performance of applications can be significantly improved through the use of trace profiling. Ex. Lebeck [5] uses information about low-level events such as data cache misses, branch mis-predictions etc. to make small changes in the source code thereby gaining significant performance improvements. Such low-level information is difficult to extract because of the lack of hardware support and is further complicated by the several hardware optimizations such as pipelining etc. that are present in modern processors. Thus prior to IA-64, the only option was to perform trace-driven simulations of the program upon previously obtained hardware traces. Such methods were both cumbersome and imprecise. IA-64's PMU offers hardware support for profiling such critical parameters which could play a major role in tuning application programs. The importance of such profiling increases further because IA-64 allows applications to "actively and directly" manage resources that are performance critical such as branch-prediction structures. A quick and dirty introduction to the performance monitoring unit of IA-64 follows as this is crucial for understanding of the design and implementation of Apache web-server metrics which is subsequently discussed. The main functionality of the PMU is spread across two sets of 64 bit registers. 14 Performance Monitor Control registers are used to configure process monitoring, the data output of which goes into 17 Performance Monitor Data registers. To support the wide variety of "monitorable" parameters with the limited amount of register space, the following distinct modes of performance monitoring are provided.

- **Workload characterization**: This is the simplest form of monitoring that could be performed. A set of monitorable events such as IA64_INST_RETIRED(Number of IA64 instructions completed) etc. have been defined. These events could occur either once or multiple times per cpu clock cycle. Ex: TLB misses occur atmost once in a clock cycle, but multiple instructions might be retired in a clock cycle due to hardware parallelism. Four pairs of PMC/PMD are available for monitoring data of this type. Specifying the event to be monitored in PMC[i] (i in 4 to 7) would cause a cumulative count of the number of occurrences of the specified event to be recorded in PMD[i].

- **Cycle accounting**: Often a break-up of the percentage of the total cycles that spent in each "phase" such as instruction execution, data access, brance misprediction etc., is more useful than a cumulative count of the number of occurrences of each event. This mode then gives us a global account of the cpu cycles spent and is performed using the same PMC's and PMD's described above in a similar manner.

- **Statistical sampling**: Now that we have got a picture of the entire system, breaking down into sub-parts such as the amount of cpu-cycles that are spent in each part of the application program or the number of cache misses that occur in the various parts of the program are useful to identify "hot-spots" in the program. If D-Cache misses were identified as the major bottleneck on the system through cycle accounting, then we could perform statistical sampling to obtain precise information as to which part of the program causes the most number of D-cache misses. This would greatly ease subsequent attempts to optimize this part of the program. PMU supports this through the provision of "program-counter sampling", thereby allowing us to tie event counts to specific ranges in PC value. The ITC and ITM registers are used to support timer interrupts and event overflow interrupts. The former could be used to sample both the PC and the event counter value after every time slice while the latter could be used to sample the PC whenever a threshold number of events have been seen. The idea is that collecting few representative samples would be a good statistical measure of the whole. The event counters are still obtained using PMC[4-7] and PMD[4-7], apart from the ITC and ITM, PMC[0-3] are used to configure the various interrupts.

- **Miss Event address sampling**: Program counter sampling is at best a coarse estimate of miss events such as I-Cache or D-Cache misses etc, because of the pipelined execution, i.e. the current PC could be about "48 dynamic instructions" farther away from the instruction that actually casued the event. Further precise information as to which instruction caused the miss and which data address access triggered this event would be very helpful in optimization. This is provided through the use of a large fraction of PMC's and PMD's. PMC[10-12] are used to customize I-Cache/I-TLB miss events, D-Cache/D-TLB miss events and branch traces for mis-predicted brances. PMD[0-3] and PMD[17] are used to store the data of these events such as the address of the instruction and/or data that caused this miss. PMD[8-16] are used to trace branch buffers upon mis-predication of a branch.

- **Granularity of monitoring**: PMU offers fine grained control over the events being monitored. One could choose to monitor only the events occurring when an application is executing (a) in userland, (b) within the kernel, (c)in interrupt handlers and (d) any combination of these three. One could also choose to monitor events for a specific process as opposed to those for all the processes on the system. Monitoring can also be restricted to instructions that belong to a (a) specified range in the address of instructions Ex. a method, (b) specified range in the address of data accessed, (c) specified instruction opcodes and (d) specified instruction sets (IA32 or IA64 — as IA32 is emulated on IA64). The composition of these various granularity options occurs in a complex manner. Interested users are referred to [6][7].

3 Architecture

The setup has two major components. The Load Generators, which are the distributed Web clients for generating HTTP requests modelling realistic workloads, and the Apache Web server with added code for instrumentation and profiling.

We first provide an overview of the entire architecture and then provide some details about the load generators. The details about instrumenting the Apache code are provided in Section 4.

3.1 Overall Architecture

Figure 1 shows the high-level design and interactions between various components. The Load Generators are a set of distributed Web clients that generate HTTP requests under the supervision and control of a manager or *webmaster*. They can generate requests for static as well as dynamic content. During the experiments, these were run off the Sun machines in the CSIL lab in the Digital Computer Laboratory.

These requests are received by the Apache Web server running on an Itanium machine. We make several changes to the Web server code. Most of the code for setting up the PMU, reading off the PMU registers, writing profiling information to the disk

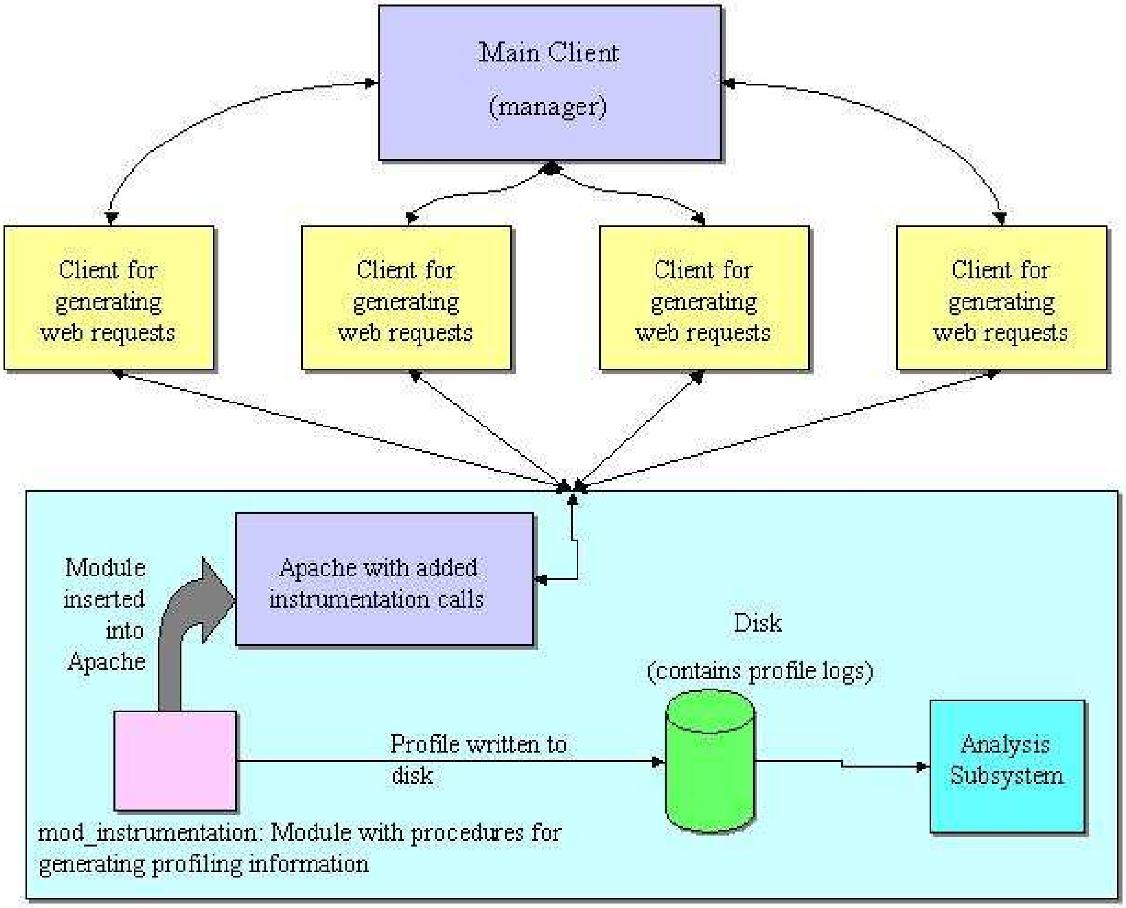

Figure 1: Overall Architecture

was organized as a new Apache module, *mod_instrument*. Simple calls to the handlers defined in this module have been inserted in the sections of the Apache code we intended to monitor. There are also provisions of monitoring the entire code, all the processes, specific PMU events etc. These configuration parameters are passed to the intrumentation module as directives. Once the profile data is generated, it is written to the disk in the form of potentially multiple files. Various scripts, written largely in Perl, aggregate this information and calculate various desired metrics.

3.2 Load Generators

There are various Web server benchmarks available that perform load generation and measure various metrics like server throughput and average response time. The most popular among these is SPECweb99 [2]. Unfortunately, it is a commercial package that comes with a rather high price tag. We didn't have access to this package, hence we developed our own load generator. We have tried to follow the specifications of SPECWeb99 as closely as possible while developing our file set. Writing our load generator also allowed us a lot of flexibility in choosing file sizes, CGI-scripts etc and we could test for non-standard file sets too.

The Load Generator has two parts: the *fileset generator* and the *workload generator*. The fileset generator is run to generate html files with various sizes. These files are then deployed on the Web server. The list of the files to fetch in a particular run are specified in the *filelist* configuration file for the Load Generator. This file contains the list of the filenames of the files to fetch and number of times each file was to be fetched.

The second part, the workload generator, has two further sub-components: the web-clients, and the webmaster which manages the webclients. Firstly, various information like the host on which webserver is running, hosts on which webclients are to be run, number of the client processes to start, number of iterations of fetching the filelist to make, how long to keep generating Web requests etc is put in the configuration file that determines the experiment's testbed.

Next, a script is run to start the Load Generator. It first starts the webclient on the current host, and then starts various webclient processes on the client hosts configured in the testbed config file, using the *rexec* system call. Webclients are distributed equally among the client hosts. The webmaster also starts a thread that listens for connections from the webclients. Upon starting up, the webclients establish a connection to the webmaster and send a READY message. When webmaster has heard from all the webclients, it sends them all a GO signal. This causes the webclients to read the testbed file (shared through NFS), and generate requests to the Web server accordingly. After the stipulated time and iterations are over, the webclients send back their reports to the webmaster, which aggregates these and prints out a synopsis, with metrics like average throughput, average response time, average number of errors etc.

As described above, the fileset which is to be fetched from Web server can be static html files (randomly generated by the fileset generator) or can be calls to CGI scripts with proper arguments. The standard fileset, developed using the SPECweb99 white paper, is desribed in the Table below.

File Sizes	Access Frequencies
0-1KB	35%
1KB-10KB	50%
10KB-100KB	14%
100KB-5MB	1%

The CGI script tested took the name of a file as its argument, read it from the disk and wrote it to the standard output. Hence, the CGI script too was used to fetch the same standard fileset, but with an additional level of indirection and causing Apache to use its CGI module.

We had initially intended to implement this entire component in Corba/C++ or Corba/Java. But the current approach seemed to be a neater way as we could do away with problems of running daemons/servers on all potential client machines and other overheads, and the socket programming/rexec model could be programmed without significant difficulties.

4 Instrumenting Apache Source Code

As described in Section 2.1, Apache is organized in the form of modules. This extends the usual object-oriented programming advantages—clean organization of code and very easily achieved extensibility. We decided to conform to Apache's philosophy and organized most of our profiling code as a separate module which we call *mod_instrument* or *instrumentation module*. Though a large part of our time on the project was spent on understanding how Apache was organized, once we understood, it provided a very good interface to adhere to. The module takes the following directives:

- **ProcessProfiling**: this directive is either *On* or *Off*. If it is On, the entire Apache Web server program (all processes) are monitored. Otherwise, the modules listed in the *MonitorModules* directive are monitored.
- **MonitorModules**: this directive specifies the list (can be of arbitrary length) of Apache Web server modules to be monitored. It is effective only if Process-Monitoring is Off or unspecified.
- **MonitorEvents**: this directive specifies the list of PMU events to be monitored. It can be at most of length 4, any further events beyond that length are ignored.
- **LogPath**: this directive specifies the path of the directory where the log files containing the profiling information would be written. These logfiles are of the form *pmulog.[pid of the child]*.

- **PrivilegeLevel**: this directive specifies the privilege-level at which monitoring is to be performed. It can be *user, kernel or both*.

Specifying all this information in the form of directives, which in turn are listed in *httpd.conf*, allows rapid profiling while changing various parameters. No change in the source code and recompiling is required. Just restarting Apache causes it to re-read the configuration file. Appendix B shows a sample section of *httpd.conf*.

Like other Apache modules, this module has various handlers. There is a list of handler functions which has a handler for each of the above directive. This list contains a mapping of the handler functions to the directive they are meant to process. Whenever Apache encounters a directive for *mod_instrument* module, it calls the relevant handler. The handler in turn intializes various internal state variables (which are private to the module).

Similarily, there are other functions and handlers. Figure 2 shows a typical scenario of how and when various functions and handlers are called.

- *instrument_init* which is called whenever Apache web server is started. This initializes various internal data structures.

- *instrument_child_init* which is called whenever a new Apache Web server child process is spawned. We further initialize some data structures here. We also open the logfile for this process: *LogPath/pmulog.[pid of the process]*. Also, if ProcessMonitoring has been enabled, this function calls *start_profiling* with relevant arguments.

- *instrument_child_exit* which is called whenever an Apache Web server child process is about to be killed. We perform various clean up operations here, like dumping any buffered profiling information (for module profiling), and stopping profiling (and dumping the values in the PMD's) in case ProcessProfiling was enabled.

- *start_profiling* is called either by *instrument_child_init* (if ProcessProfiling is *On*) or by handlers in various other Apache modules. It is exported as a handler in *mod_instrument*, hence other modules can call it too. One of its arguments is ID of the calling module. It checks to see if the current module is in the list of modules that were specified with the *MonitorModules* directive (list is already stored in an internal data structure by the handler for this directive.) It further checks for nesting of calls to *start_profiling* and *stop_profiling* and continues further only if profiling has already not been started by the calling Apache module. It then sets up the configured events (which also have been read from the configuration

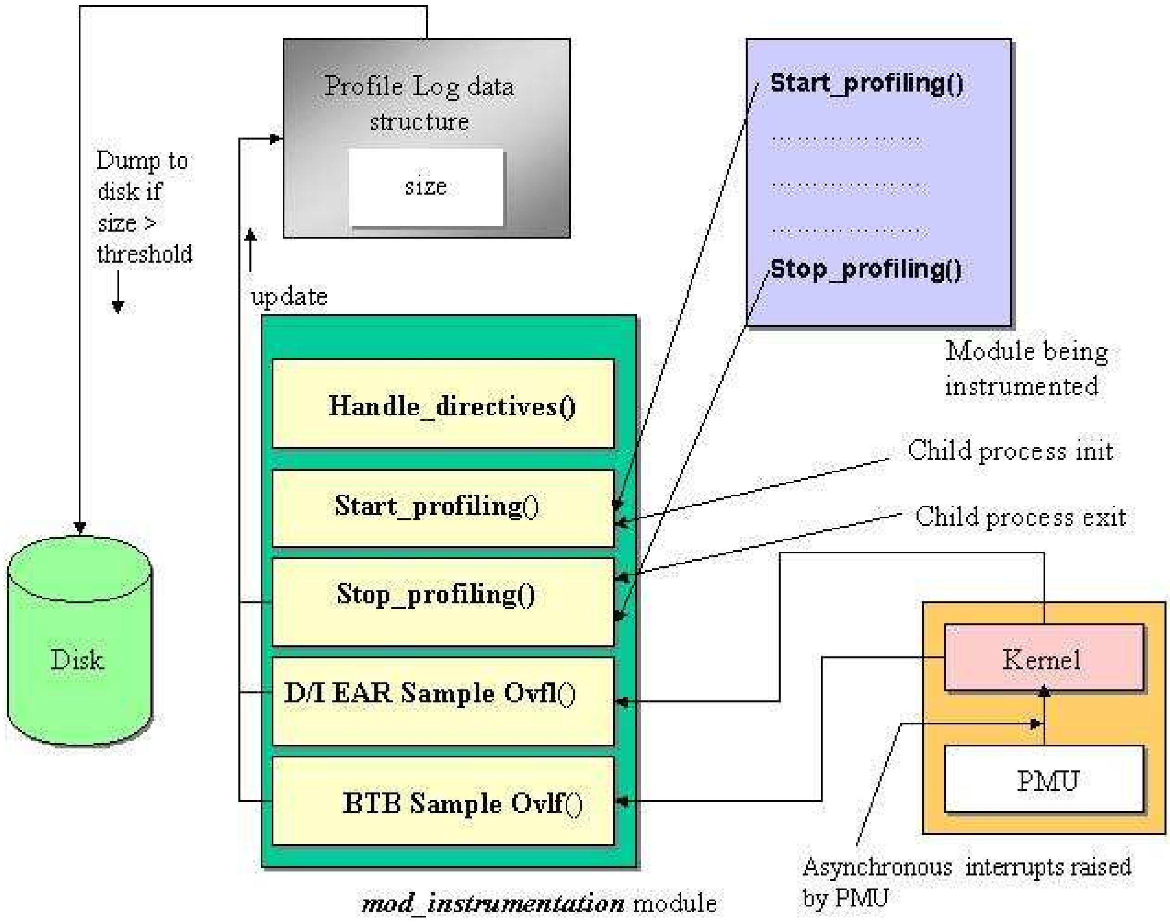

Figure 2: A typical interation between various funtions and handlers

file and stored in an internal data structure) into the PMCs, sets up a context and starts profiling. If ProcessProfiling is enabled, calls from other modules are ignored.

- *stop_profiling* is called either by *instrument_child_exit* or by handlers in various other modules. Like *start_profiling* it too is exported as a handler by *mod_instrument* and thus can be called by code in other modules. It too receives the ID of the calling module as an argument. It checks if the calling module is in the list of modules for which we are doing profiling and if there is a proper nesting of calls to *start_profiling* and *stop_profiling*—there should be exactly one pending *start_profiling*. If the checks are satisfactory, it stops the profiling and reads the information from the PMU registers. This information is then written to the buffered profile data log for the current child process together with an appropriate header (module name, name of events etc.)

There are other functions which act as signal handlers for sampling based events. A sampling rate is specified for these events—whenever this threshold is reached, an interrupt is raised by the PMU and which is passed to the application as a signal. Signal handlers record PMD values at these instances in the profile log.

Additional code has been linked to Apache to provide calls for setting up and reading from PMU. These are largely setup code and libraries used by the *pfmon* utility. We could not use the *pfmon* utility directly as it is used to do user-level profiling of an unchanged *binary*. It cannot be used directly to measure the performance of Apache due to two major reasons: Apache is multi-processed and we also want to monitor selected portions of the server code.

Calls to *start_profiling* and *stop_profiling* are inserted in the major functions of various modules. The calls also include the ID of the calling module as a parameter.

5 Metrics

This section describes in detail the various metrics we devised and measured to quantify the performance of Apache on IA64 machines.

5.1 Benchmarking performance visible to the clients

Benchmarking the performance of Apache, as seen by web clients accessing the web server has been done. This helps us to compare the performance of Apache in the current configuration with the performance of Apache on other architectures/configurations. The manner in which these measurements are performed has already been discussed in the description of load generators(Section 3.2). The following useful metrics have been quantified,

- **Server Throughput**: This is the number of bytes of useful data that the server outputs in unit time.
- **Average Response Time**: This is the delay between the initiation and fulfillment of a request as seen by the client. This metric is complementary to Server throughput since a HTTP server can achieve high throughput but still have a

signficantly high average response time if its scheduling policy is unfair. An unfair scheduling policy might cause some requests to starve while fulfilling other incoming requests with a high throughput. Thus low response time, in conjunction with high throughput conclusively verifies the soundness of the scheduling algorithm.

- **Server Connection Rate**: Another complementary metric would be the rate at which connections could be served. Every web-server has connection initiation overheads. Apache is a multi-process web-server in which the number of live processes at any time are constrained between a user specified minimum (default=5) and maximum (default=16). Child Processes are spawned and killed proportional to the current load on the server. Thus a new connection would not only need to be scheduled to a lightly loaded child process but spawning a new process might also be necessary. If these overheads are large, then the server might have low response times and high throughput but would definitely have a low connection rate. Thus this metric in conjunction with the above two quantifies the average per-connection initiation overhead in Apache.
- **Varying content types**: We broadly define two classes of content. Static content includes HTML pages, images, and any other data items that Apache could deliver by reading off the file system. On the other hand dynamic content includes output of CGI scripts, server side includes and other plug-ins, the delivery of which involves execution of code by Apache. It has been an observed fact that Apache performance greatly varies with the type of the content. Thus each of the above metrics, and those that would be described later in this section have been performed for both static and dynamic content. The manner of creation of this content has been discussed in Section 3.2.

5.2 Server side metrics that are quantified by the aid of PMU

This section describes the server side metrics and details how each of these metrics could be quantified using events that can be profiled using PMU.

5.2.1 Basic Events

This section deals with the events mentioned under "Workload Characterization" in Section 2.2.

- **Instructions Per Cycle**: The number of instructions that execute per cycle is a good measure of the amount of instruction level parallelism in the architecture that Apache is able to achieve. PMU allows us to measure the number of IA64 instructions that were actually executed, the number of IA32 instructions that were actually executed and the number of cpu cycles these instructions were executed in. The corresponding basic events are "ia64_inst_retired", "ia32_inst_retired" and "cpu_cycles" respectively. The number of instructions per cycle is then obtained as:

$$\text{No. of IA64 instructions per cycle} = \frac{ia64\_inst\_retired}{cpu\_cycles}$$
$$\text{No. of IA32 instructions per cycle} = \frac{ia32\_inst\_retired}{cpu\_cycles}$$

The IA64 architecture has an IA32 emulator to support legacy binaries. Since we deployed Apache in a non-root installation and compiled the binaries on IA64 itself, the number of IA32 instructions executed should be zero.

- **Floating point operations**: The number of floating point operations that a program executes is needed in order to determine the type of workload that is being generated by the program. PMU provides two basic events fp_ops_retired_hi and fp_ops_retired_low that can be used in conjunction to estimate the number of floating point operations that are executed by a given program within a given period. The number of floating point operations that are executed (retired) is obtained from the following formula,

  $$fp\_ops\_retired = 4 * fp\_ops\_retired\_hi + fp\_ops\_retired\_low.$$

  The semantics behind this formula and the reason why only fp_ops_retired_hi and fp_ops_retired_low can be computed while fp_ops_retired cannot is beyond the scope of this report as this would involve details of the FPU in IA64.

- **TLB Misses**: TLB's are used to map virtual addresses to physical addresses on the hot path of execution. Since TLB misses usually cause high penalties, most modern operating systems show significantly small percentages of TLB misses. Also the number of TLB misses is an estimate of the degree of locality among the data/instructions being accessed by the application. We used the following PMU event counters to obtain the TLB miss ratios both for I(Instruction)-TLB and D(Data)-TLB. The number of Data TLB misses is the value of the counter *dtlb_misses*, and the total number of data references that are successful is the value of the counter *data_references_retired*. Thus the miss ratio of the Data TLB can be calculated as

  $$\text{DTLB miss ratio} = \frac{dtlb\_misses}{data\_references\_retired}.$$

Similary the number of ITLB misses can be obtained from the value of the counter itlb_misses_fetch whereas the value of total ITLB accesses is obtained from itlb_references. The ITLB miss ratio is then calculated as

$$\text{ITLB miss ratio} = \frac{itlb\_misses\_fetch}{itlb\_references}.$$

Experimentally obtained values of these metrics are described in Section 6.

5.2.2 Events used to perform cycle accounting

This section clearly details how we perform "Cycle Accounting" that has been mentioned in Section 2.2. The goal of the effort here is to account for each cycle of cpu time and pin-point what exactly has caused the cycle to be used up. This is a non-trivial problem in modern architectures that implement pipelining as no operation completes in one cpu cycle and several operations (a data fetch, an instruction execution etc) might occur within a single cpu cycle. PMU support for cycle accounting exports 8 counters. A priority is attached to each counter. During each cpu cycle, events being monitored by more than one of these counters would occur. In such scenarios, this cpu cycle is attributed (counted in) the counter that has the largest priority among those being monitored. The following lists the eight events that help in cycle accounting and their semantics in decreasing order of priority.

- **pipeline_backend_flush_cycle**: Counts the number of cycles lost due to mispredicted branches and pipeline flushes.
- **data_access_cycle**: Counts the number of cycles lost in accessing data from the memory. This includes the latency spent in retrieving data from any of the several levels of cache hierarchy and those involved in retrieving data from the memory.
- **execution_latency_cycle**: Counts the number of cycles that are lost due to latency in execution of instructions due to data dependencies (flow, anti, input).
- **memory_cycle**: In addition to the cycles lost in data access stalls that are counted by data_access_cycle above, this counts the cycles lost due to RSE spills/fills (Registers are all used up.). Thus this counts the total number of cycles lost in accessing data during the period of execution.
- **execution_cycle**: In addition to the cycles lost due to data dependencies that are counted by execution_latency_cycle above, this counts the number of cycles spent in instruction issue, stops and resource oversubscription stalls.

- **inst_access_cycle**: This counts the cycles lost in accessing/fetching instructions from the memory.
- **pipeline_all_flush_cycle**: In addition to the cycles lost due to mis-predicted branches, counted by pipeline_backend_flush_cycle above, this counts the number of cycles lost due to instruction fetch branch bubbles. Thus this counter counts the total number of cycles lost in branch induced latencies.
- **inst_fetch_cycle**: In addition to the number of cycles lost in instruction fetches that are counted in inst_access_cycle above, this counts the number cycles lost due to partial instruction fetch stalls. Such stalls occur when non-instruction cache lined instructions are specified as branch targets. Since data can be brought into the cache only in multiples of cache lines, such a branch would involve bringing in redundant instructions when a cache miss occurs.

Based on experimental results, we have identified that all the cpu cycles could be broken down into those in which unstalled execution occurs, those lost due to dependencies within the code, those lost due to memory stalls (instruction fetch, data read and data write) and those that are lost due to pipelining stalls caused primarily by execution of branch instructions. Using the above eight events that are supported by PMU, we obtain this useful information using the following formulae

$$\begin{aligned}
\textbf{Unstalled Execution} &= execution_cycle - execution_latency_cycle \\
\textbf{Dependency Stalls} &= execution_latency_cycle \\
\textbf{Memory Stalls} &= memory_cycle + inst_fetch_cycle \\
\textbf{Pipeline Stalls} &= pipeline_all_flush_cycle.
\end{aligned}$$

The rationale behind these formulas is obvious from the description of each of the events. We performed cycle accounting of the cycles spent at the user level during execution of Apache, those spent at the kernel level and those spent in each individual module. Further we also quantified the variation in the cycle breakdown for static content as compared to the cycle breakdown for dynamic content. Experimentally obtained values of these metrics are described in Section 6.

5.2.3 Events to measure the performance of memory hierarchy

Several applications have different performance on the same machine because of the variance in the performance that they extract from the memory hierarchy — primarily due to different access patterns. Specifically retrieval of static content and retrieval of dynamic content from Apache would greatly differ in the performance they obtain from the Itanium memory hierarchy. Thus quanitization of the losses at each level of the hierarchy would be interesting.

Memory Hierarchy Background

Before we describe which metrics we obtain and how we obtain the values of these metrics, we briefly discuss the memory hierarchy on the current Itanium architecture in HP i2000 (the machine running the web server). Barring some differences in the sizes of each level of cache, the Itanium architecture has the following memory hierarchy.

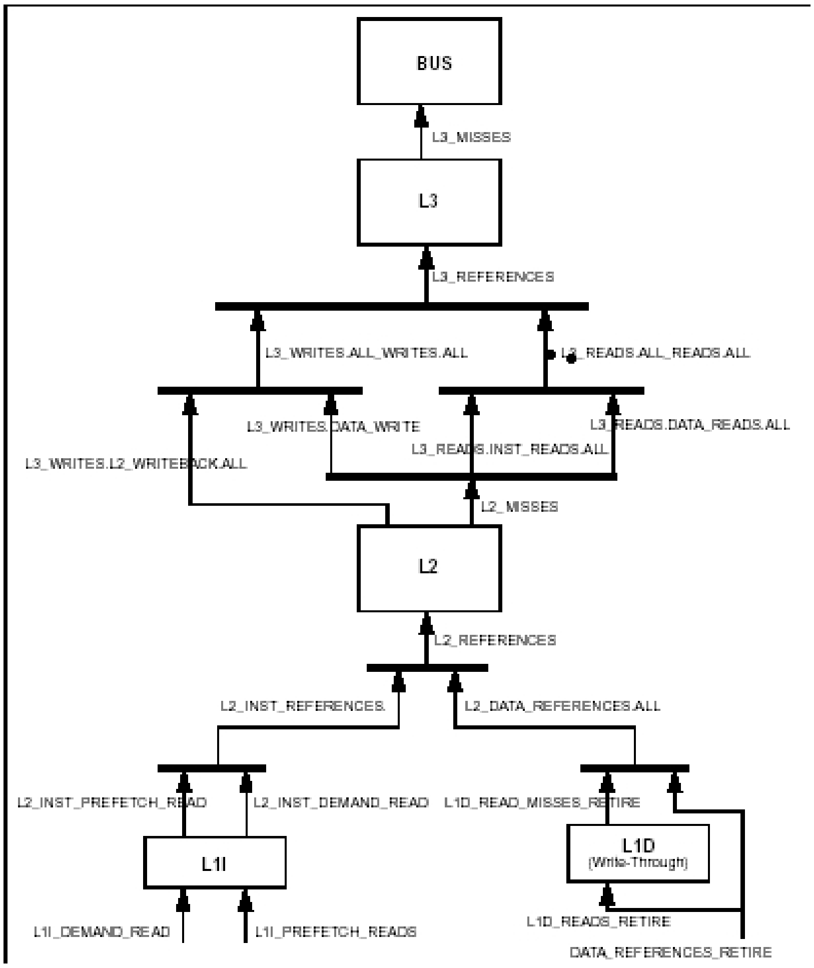

Figure 3: Memory Hierarchy in IA64

The Itanium architecture has 3 levels of cache progressively increasing in size and the latency involved in servicing a miss. Caches at the L2 and L3 level are unified and support both integer and floating point data. However distinct instruction cache and data cache are maintained at the L1 level. Further the L1 data cache is optimized primarily for integer data access, all floating point reads cause forced misses in the L1 data cache.

PMU events and formulae for metrics

Since we have quanitified the number of floating point operations to be a small percentage of the total data accesses (as shown in Section 6.3), we completely ignore floating points in calculation of latencies. Further, though we identify the miss ratios at each level of the memory hierarchy, we plot the product of number of satisfied accesses and the latency at each level. The list of PMU events that are used is too long and has been ommitted for brevity. Interested readers can refer to the Figure 3 or read up [6]. The following are the formulas used to obtain the cache miss ratios from PMU event counters.

$$\begin{aligned}
\textbf{L1 I-Cache Total Accesses} &= l1i_demand_reads + l1i_prefetch_reads \\
\textbf{L1 I-Cache Misses} &= l2_inst_prefetch_reads + l2_inst_demand_reads \\
\textbf{L1 I-Cache Miss Ratio} &= \frac{L1ICacheMisses}{L1ICacheTotalAccesses} \\
\textbf{L1 I-Cache Satisfied Requests} &= (1 - L1ICacheMissRatio) * (L1ICacheTotalAccesses) \\
\textbf{L1 D-Cache Total Accesses} &= l1d_reads_retired \\
\textbf{L1 D-Cache Misses} &= l1d_read_misses_retired \\
\textbf{L1 D-Cache Miss Ratio} &= \frac{L1DCacheMisses}{L1DCacheTotalAccesses} \\
\textbf{L1 D-Cache Satisfied Requests} &= (1 - L1DCacheMissRatio) * (L1DCacheTotalAccesses) \\
\textbf{L2 Total Accesses} &= l2_references \\
\textbf{L2 Cache Misses} &= l2_misses \\
\textbf{L2 Cache Miss Ratio} &= \frac{l2_misses}{l2_references} \\
\textbf{L2 Satisfied Requests} &= (1 - L2CacheMissRatio) * l2_references \\
\textbf{L3 Total Accesses} &= l3_references \\
\textbf{L3 Cache Misses} &= l3_misses \\
\textbf{L3 Cache Miss Ratio} &= \frac{l3_misses}{l3_references} \\
\textbf{L3 Satisfied Requests} &= (1 - L3CacheMissRatio) * l3_references \\
\textbf{Memory Total Accesses} &= l3_misses \\
\textbf{Memory Satisfied Requests} &= l3_misses
\end{aligned}$$

Now it remains to estimate the latencies involved in satisfying the request at each level of the memory hierarchy. As has been stated before, we assume that all data operations involve integer data. The latency in accessing data from each of the caches is a straightforward processor specific value. For HP i2000, these values are

$$\begin{aligned} \mathbf{L1I} &= \ \ 2 \text{ cycles} \\ \mathbf{L1D} &= \ \ 2 \text{ cycles} \\ \mathbf{L2} &= \ \ 6 \text{ cycles} \\ \mathbf{L3} &= \ \ 21 \text{ cycles.} \end{aligned}$$

The latency at a cache deeper in the memory hierarchy also includes the cost due to misses at earlier levels. Calculation of memory latency is however a non-trivial problem. This has been estimated as follows,

Memory is PC100 SDRAM → Clock frequency is **100 MHz**.
Addressing Delay is **2** memory cycles.
Data transfer Delay is **3** memory cycles.
Component Latency is **4** memory cycles.
Total latency is **9** memory cycles, which is about $\frac{9*733}{100} = 66$ cpu cycles.
Latency due to misses in earlier levels of cache = 20 cpu cycles (1 less than the latency of a L3 satisfied request).
Thus accesses that are satisfied off the memory suffer a latency of about **86** cycles.
Experimentally obtained values of these metrics are described in Section 6.

5.2.4 Statistical sampling, Miss-event address sampling and Other events

This section deals with other metrics that we had suggested in the design document. We have implemented the functionality that allows us to monitor these metrics but have not performed experiments due to lack of time. In this section we present the rough steps of how this can be achieved. Broadly speaking, this involves the initialization of sampling rates in a set of sampling registers and setting up of corresponding signal handlers. The idea is that we would not be interested in all occurrences of an event (say a branch miss) but would be more interested in analyzing statistical samples of this event. Thus whenever an overflow occurs, i.e. enough events have been generated to satisfy the sampling rate, the kernel interrupts to the signal handlers which can then observe this event. Both D-Cache misses and I-cache misses could be more clearly identified using this mechanism. However, a significant amount of effort would be needed to perform such fine-grained profiling. Other interesting profiling experiments could be done using the opcode tagging support of PMU which allows us to profile only those instructions that match an instruction opcode.

6 Experimentation and Results

Both user level instructions and kernel level instructions that are executed in a run of Apache have been profiled. Apache has been profiled for different types of content, static and dynamic. Each of the metrics described in Section 5, unless otherwise stated, has been experimentally measured. We present graphical representations of the values we obtained for each of these metrics and comment on each of these results in the rest of this section.

6.1 Client Side Results

6.1.1 Server Throughput

Figure 4 compares the server throughput visible to the client for static content vs that achieved for dynamic content. The server throughput for static content is about 87Mbits/sec. Such a high throughput is obtained because the memory available on our system is about 2GB. The entire set of files either lies in the cache or in the memory. The total size of the cache in Itanium is about 2.5MB. Thus most of the smaller files that are retrieved as static content are cached. Though our load generation involves accessing files as large as 5MB, such accesses are few in number. Thus most of the accesses are smaller files and are usually serviced from cache hits. This conjecture is later proved when we look at the satisified accesses at each layer in the memory hierarchy. On the other hand, all the dynamic content is generated on the fly and thus has to be served off the memory. Thus while static content retrieval benefits from caching of smaller files, dynamic content retrieval has low throughput as all accesses are from the memory (pipe output). Some points worth mentioning are:

- The SpecWeb99 benchmark does not have a large enough data set to stress the 2GB memory available. Since the dataset tested for is largely based on the SpecWeb99 dataset, we have not tested the overheads involved in paging of Apache's memory. These would arise if the total footprint of a run is larger than the size of the memory on the web server.
- The improved performance of static content retrieval is largely due to the locality in file access. Instead if a large set of files is accessed sequentially, the performance obtained would be similar to that in the dynamic content retrieval phase. However SpecWeb99 conjectures that the access patterns on most web servers include a small set of files which motivates our initial assumption.

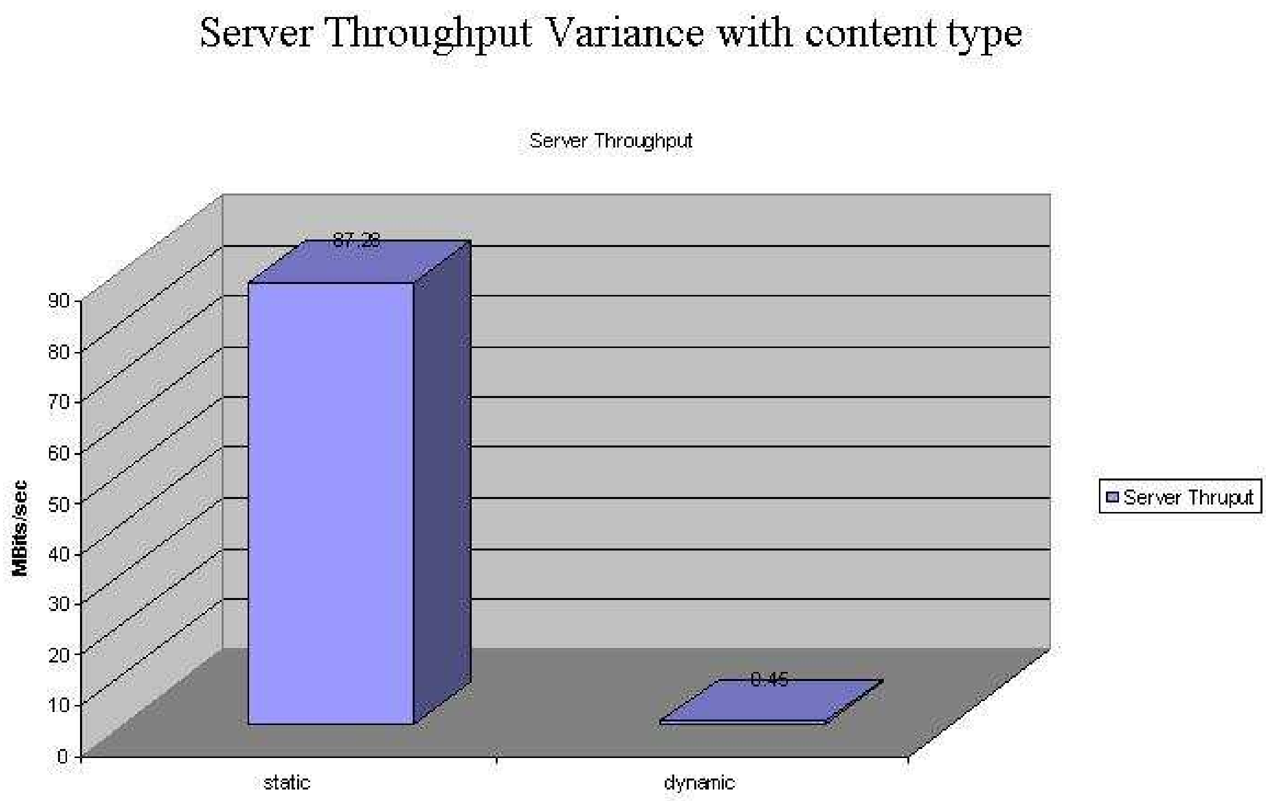

Figure 4: Server throughput variance with content type

6.1.2 Average Response Time

Figure 5 compares the average response times as seen during the retrieval of static content with those seen during the retrieval of dynamic content. As has been stated in Section 5.1, high throughputs accompanied by low average response times verify that the scheduling algorithm implemented by Apache is fair. As is clearly seen in the results, the average response time for static content is far less than the average response time for dynamic content. An interesting issue is that while the static content run has an average throughput that is 170 times that of the dynamic content run, it's average response time is only 0.5 times the average response time for the dynamic content. This can be explained as follows.

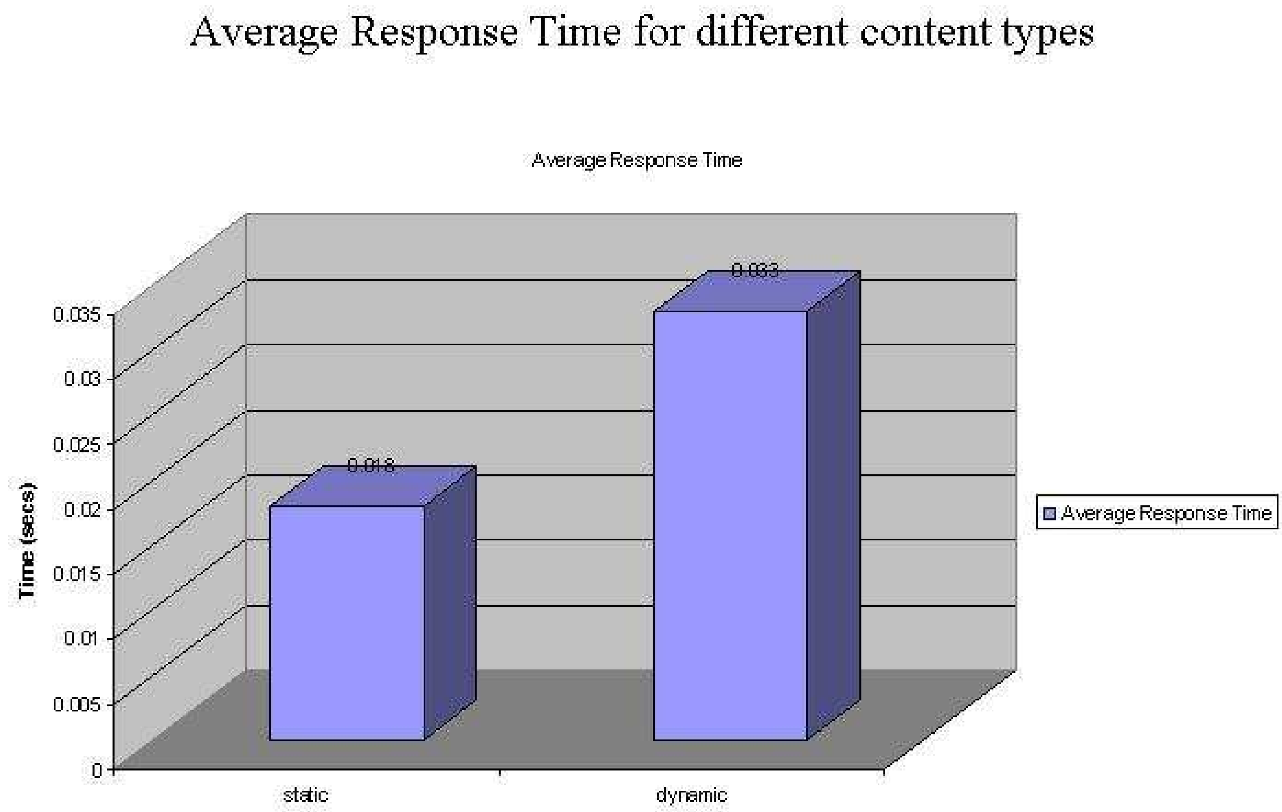

Figure 5: Average response time for various content types

Apache performs some degree of load balancing by spawning new processes until a specified maximum has been reached. Thus the response time for a connection not only involves the time the server takes to respond to this request but also involves the time the request is queued up waiting for a server process to become free. Since the server connection rate is significantly higher in the case of static content than in the case of dynamic content and the maximum number of processes that could run simultaneously is same for both, overheads in response time due to queuing of requests would be greater for the case of connections in the static content run. This clearly argues the case for increasing the upper bound on the number of processes that can be run simultaneously on HP i2000 from the default value of 16.

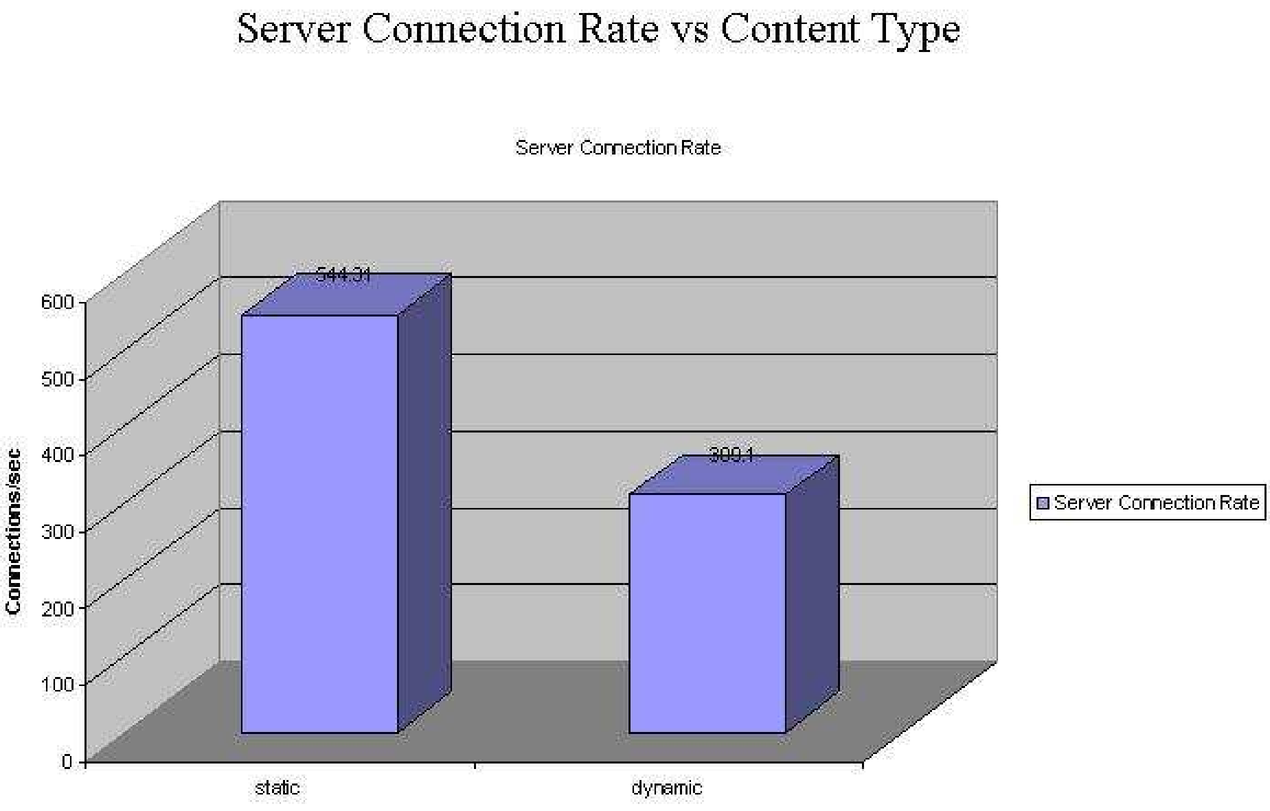

Figure 6: Server Connection rate for various content types

6.1.3 Server Connection rate

Figure 6 compares the server connection rates seen in the static content run with those seen in the dynamic content run. The greater server connection rate for static content run is clearly a fallout of our load generator implementation. While the number of load generators is constant for both runs, each load generator initiates a new connection as soon as the earlier request has been responded to. Thus a lower response time clearly leads to the issue of greater connections. Further since the overheads involved in responding to a static content request are far less than those in responding to dynamic content requests, the server is able to handle greater number of connections within the same period. Each run in our experiments is bounded by time elapsed and not by the

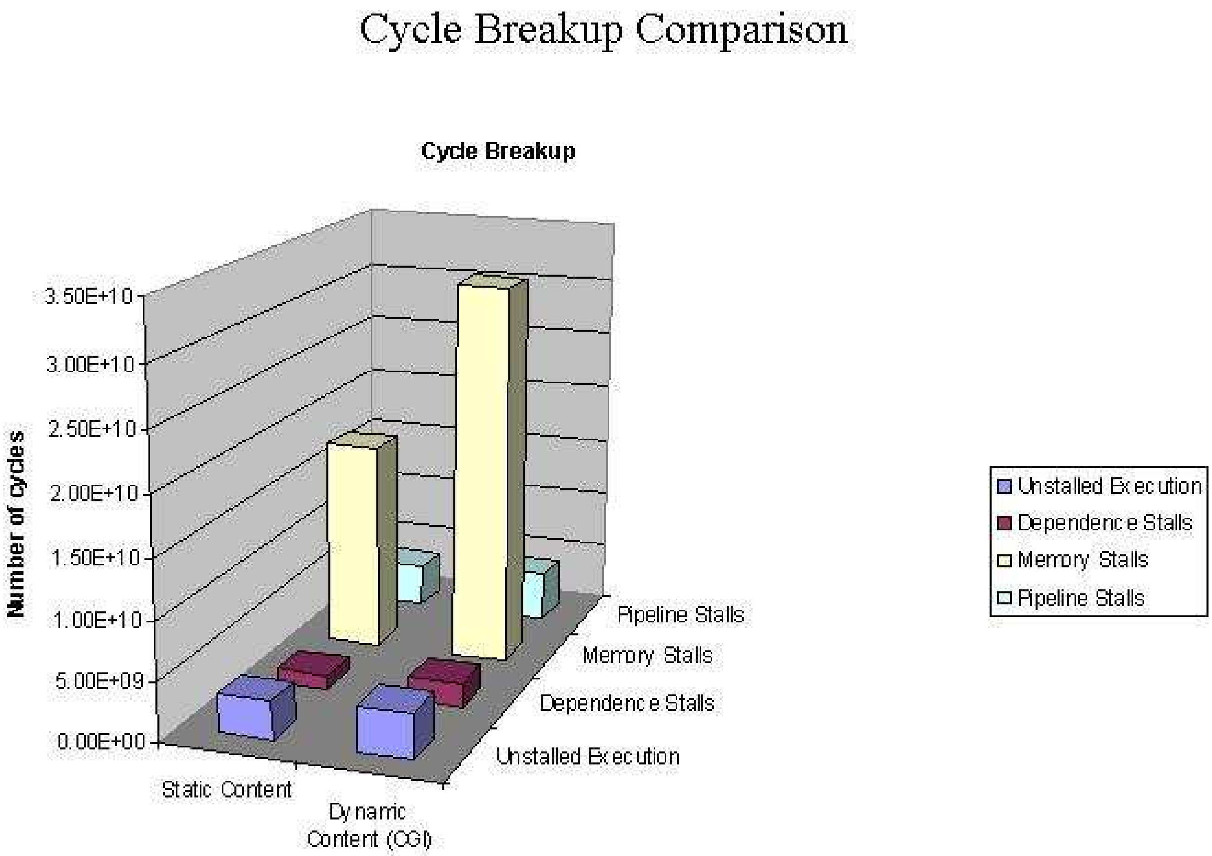

Figure 7: Apache's Cycle breakup comparison: 3D view

total number of connections completed.

6.2 Server Side Metrics

6.2.1 Apache Cycle Breakup

Figures 7 and 8 show the breakup of cpu cycles during both the static content run and the dynamic content run. Figures 9 and 10 portray different views of these graphs which are scaled up to show the percentages rather than the actual weights. The following observations are in order.

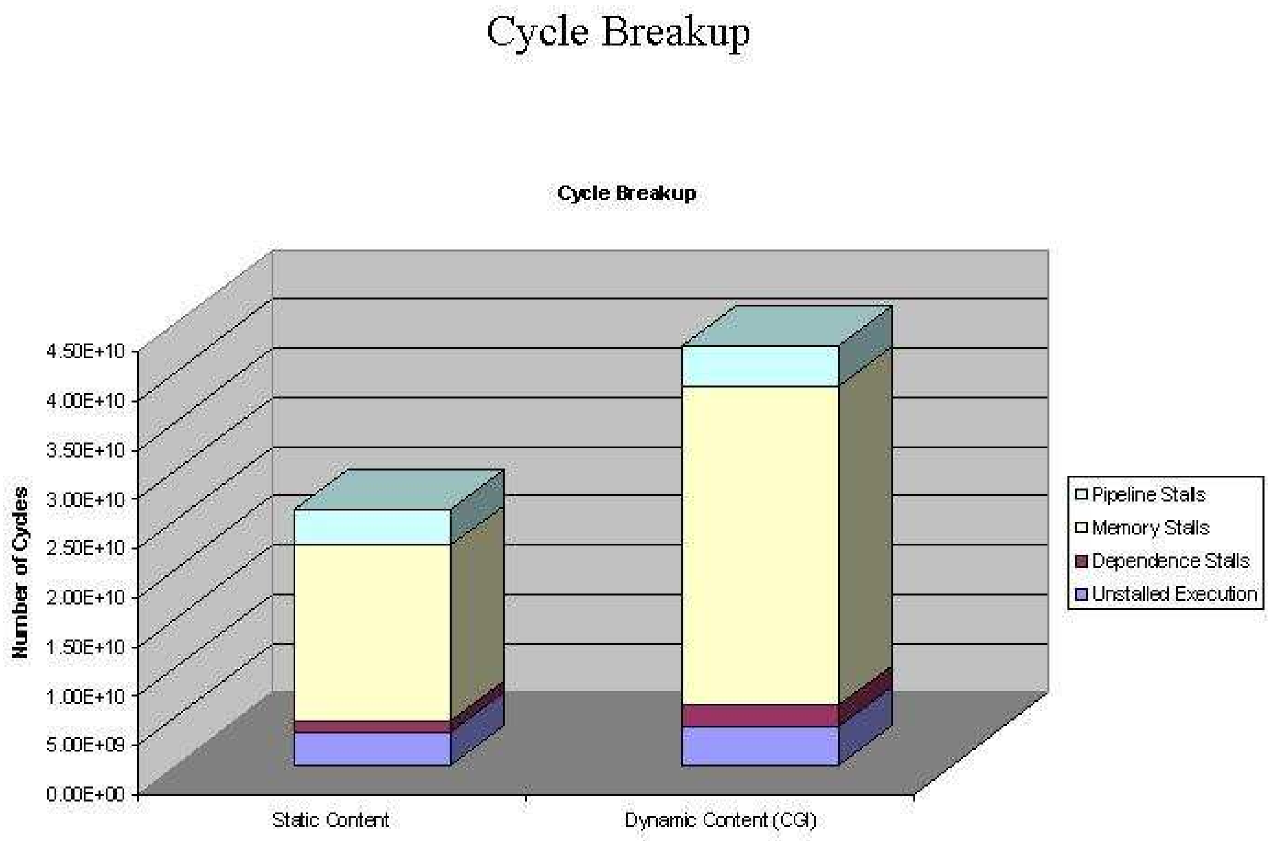

Figure 8: Apache's Cycle breakup comparison: another view

- As can be seen in Figures 9 and 10, the largest chunk of cycles in each run goes into memory stalls. This clearly categorizes Apache as a data intensive application. Such a categorization is intuitiveley correct as the main activity of Apache, connection request handling, has to do with reading data off the local disk and sending it out onto the network.
- From Figures 9 and 10, it can be seen that a greater percentage of static run goes into unstalled execution. This is because the execution of the dynamically executed process is not profiled by our monitor and clearly the remaining part of Apache has far fewer instructions to execute during the dynamic run. Pipeline stalls are a direct fallout of having a large number of branches. A reasonable

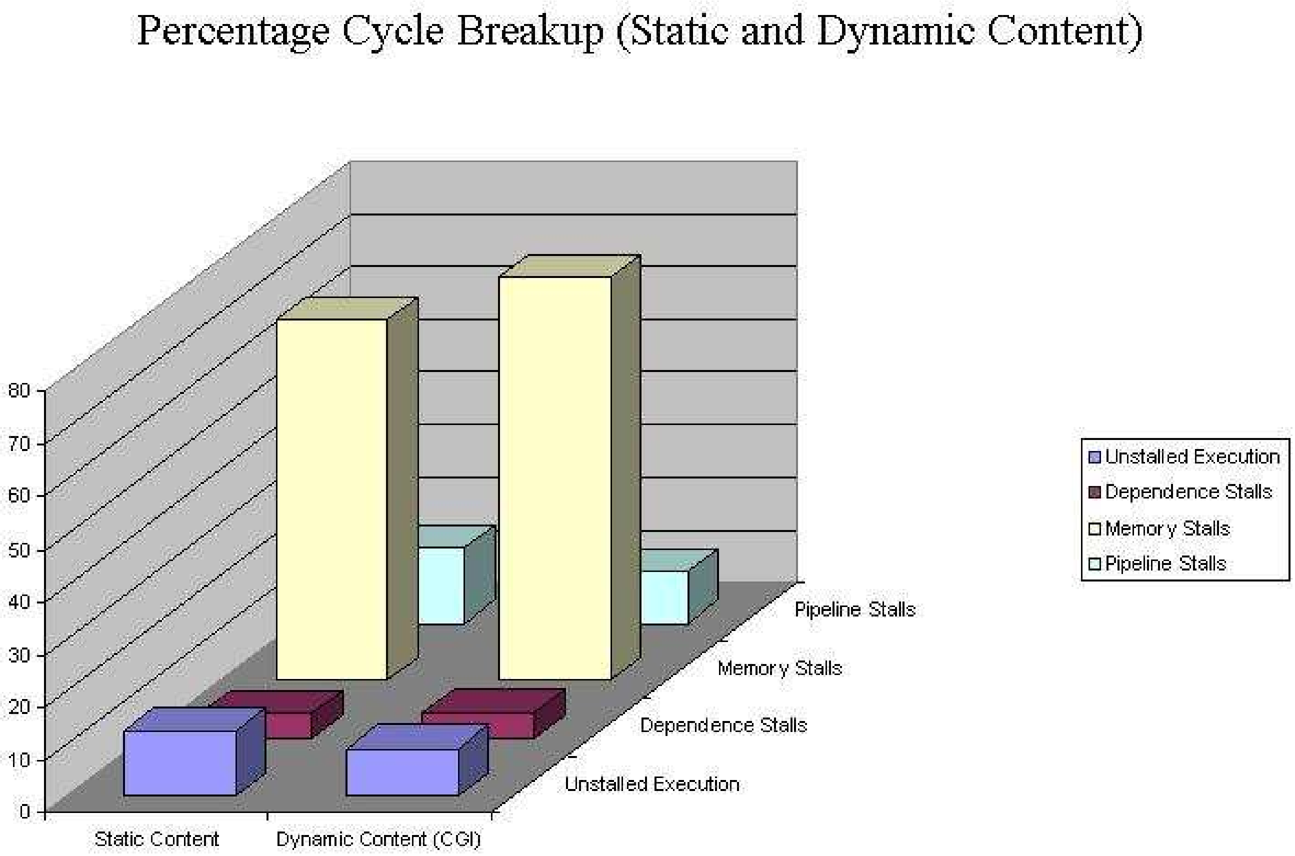

Figure 9: Percentage cycle breakup: 3D view

assumption is that the number of branch instructions that are executed is proportional to the number of instructions that are executed. Since more instructions are executed in the static content run, more branches would occur resulting in a greater percentage of cycles that are lost in pipeline stalls.

- Figures 7 and 8 depict the actual values rather than the percentages. It is clear that not only are the total number of cycles more in the case of dynamic content, but the number of cycles that go into each individual phase are also greater. This has to be expected due to overheads in spawning out, executing and reading data from a dynamic process.

- The single largest chunk of increase from the static content run to the dynamic

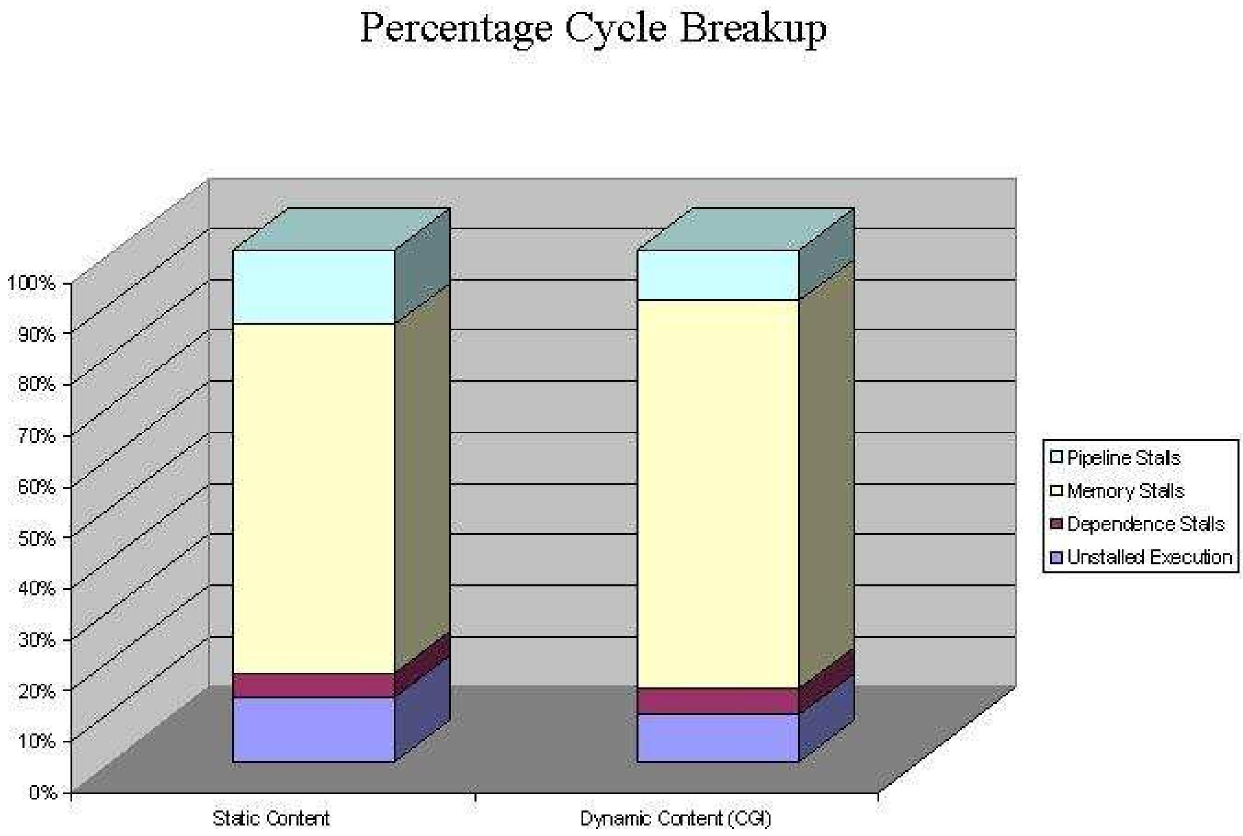

Figure 10: Percentage cycle breakup: another view

content run is in the number of cycles that go into memory stalls. As further experiments clearly show, this is because a greater percentage of data reads, during retrieval of dynamic content, result in cache misses and are only satisfied at the memory. Thus for the same amount of data being retrieved, a greater number of cycles are necessary. Slight increases in each of the others are marginal fallouts of this problem, more cycles to block and read data etc. This has been explained in detail when analyzing Apache's memory hierarchy performance later in the next sub-section.

6.2.2 Memory Hierarchy Performance

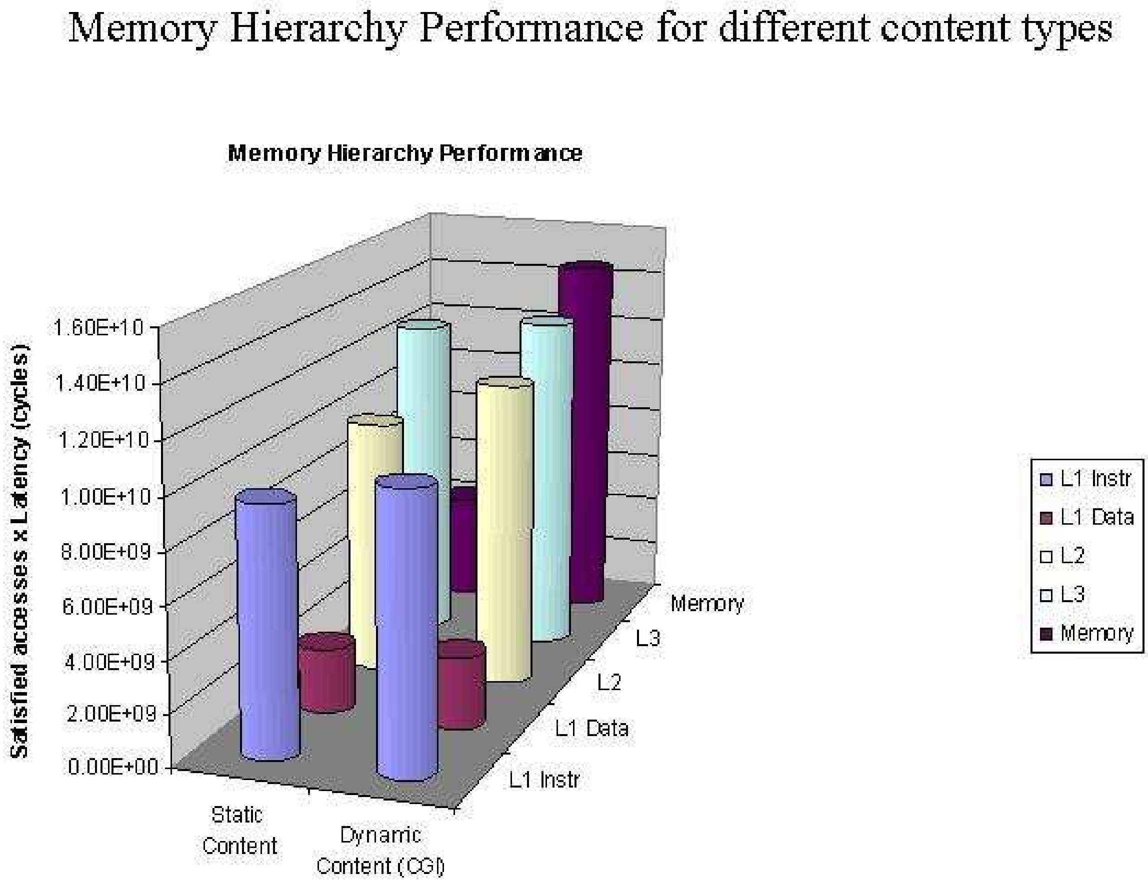

Figure 11: Memory hierarchy performance for different content types: 3D view

Figures 11 and 12 compare the actual weights of data accesses satisfied at each level in the memory hierarchy. Figures 13 and 14 compare the percentages of these weights. Each weight is calculated as a product of the number of requests that are satisfied at this level and the latency involved in satisfying a request at this level. From Figure 11, it could be easily inferred that while the weights of each level of cache are almost similar, the weight of memory is greater for the dynamic content retrieval run. Figure 12 clearly shows that the total of all these weights is significantly higher for the dynamic content run than that for the static content run. These results clearly validate our earlier conjecture that the excess overhead in dynamic content retrieval is primarily due to

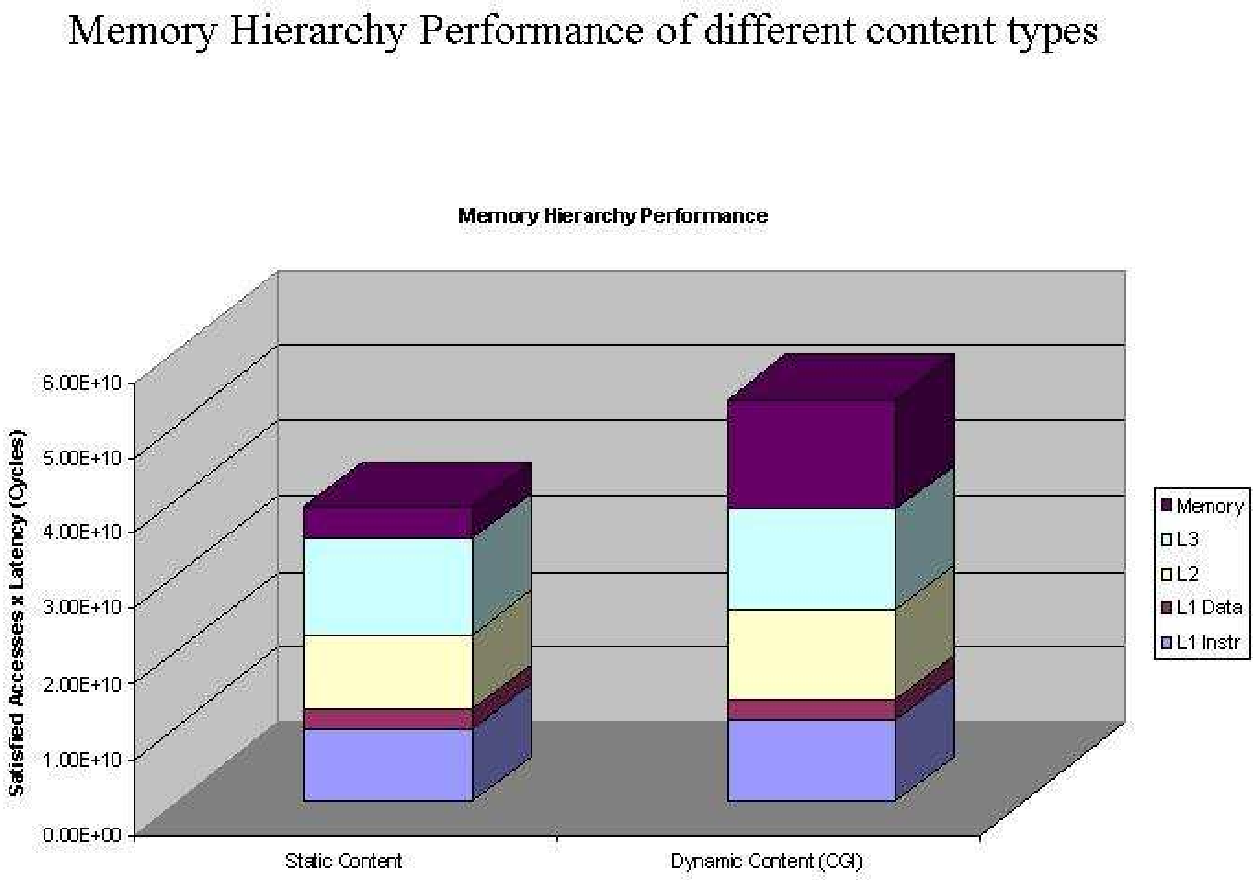

Figure 12: Memory hierarchy performance for different content types: another view

added latency in retrieving the dynamic (low spatial locality) data from the memory. Figure 13 compares the percentages of weights of the request that are satisfied at each level in the hierarchy. Clearly, in the case of static content retrieval, the percentages of total that are satisfied at the caches are far higher than in the case of dynamic content retrieval. Similarly the percentage of weight resolved through accesses in the memory are far higher for dynamic content retrieval. This clearly verifies earlier hypothesis. The largest weight in static content retrieval is attributed to the L3 cache, which is the largest among the caches. This implies that increasing the sizes of the L1 and L2 caches would result in greater performance for the static content retrieval. On the other hand the largest chunk in dynamic retrieval goes into memory accesses and this situation could only be improved by providing for quicker memories.

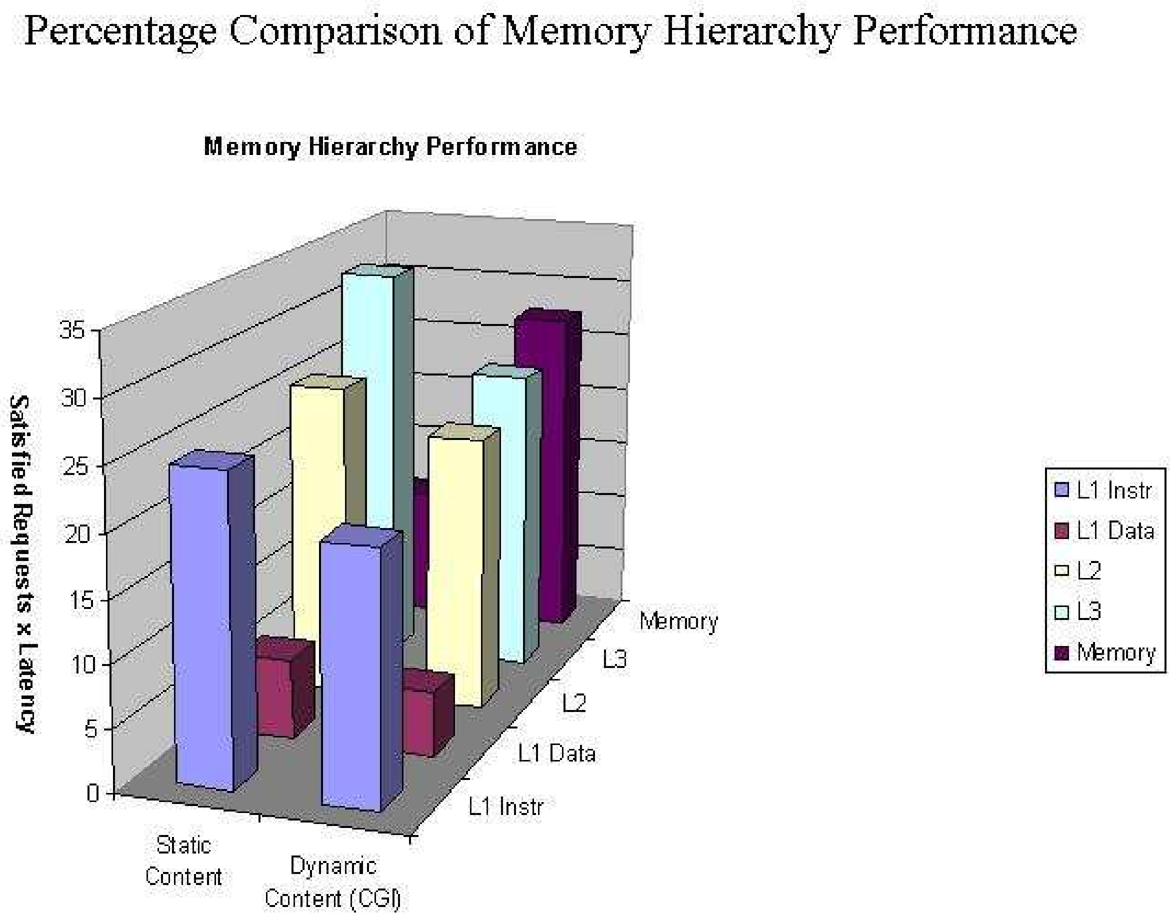

Figure 13: Percentage comparison of memory hierarchy breakup: 3D view

6.2.3 Cycle Breakup among Modules

Figures 15 and 16 provide the per-module break up of cycles when static content is being fetched from the server. While Figure 15 plots the actual values of cpu cycles, memory stall cycles and execution cycles that are spent in each module, Figure 16 plots the percentage distribution of these cycles among the modules. It could be clearly inferred from Figure 15 that the modules, in total, spend more cycles in memory stalls than in unstalled execution, which re-validates the fact that Apache is data intensive. Figure 16 clearly shows that the "core module" accounts for a major chunk of each of cpu cycles, memory cycles and execution cycles. The other major contributors are "config log module", "userdir module" and "alias module" in that order. The core

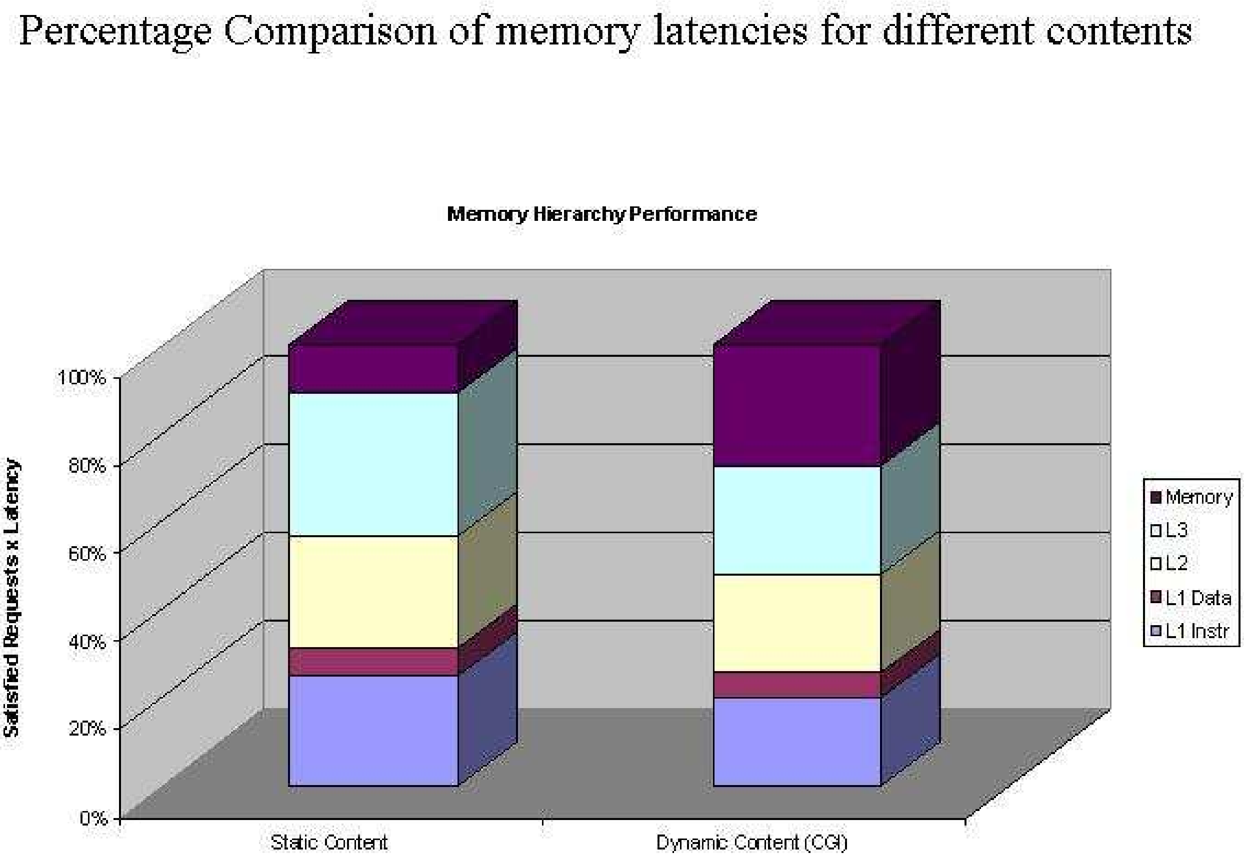

Figure 14: Percentage comparison of memory hierarchy breakup: another view

module is the main module that defines the two handlers which handle incoming static requests and respond to them. The config log module performs logging of entries into the apache access log and error log. The user dir module performs an access of the metadata information of each file and is responsible for opening and closing of files. The "hot path" of execution for static content includes these three modules. Consequently, optimizations in these modules would have the greatest positive impact on Apache performance.

Figures 17 and 18 provide the per-module break up of cycles when dynamic content is fetched from the server. While Figure 17 plots the actual values of cpu cycles, memory stall cycles and execution cycles that are spent in each module, Figure 18 plots the

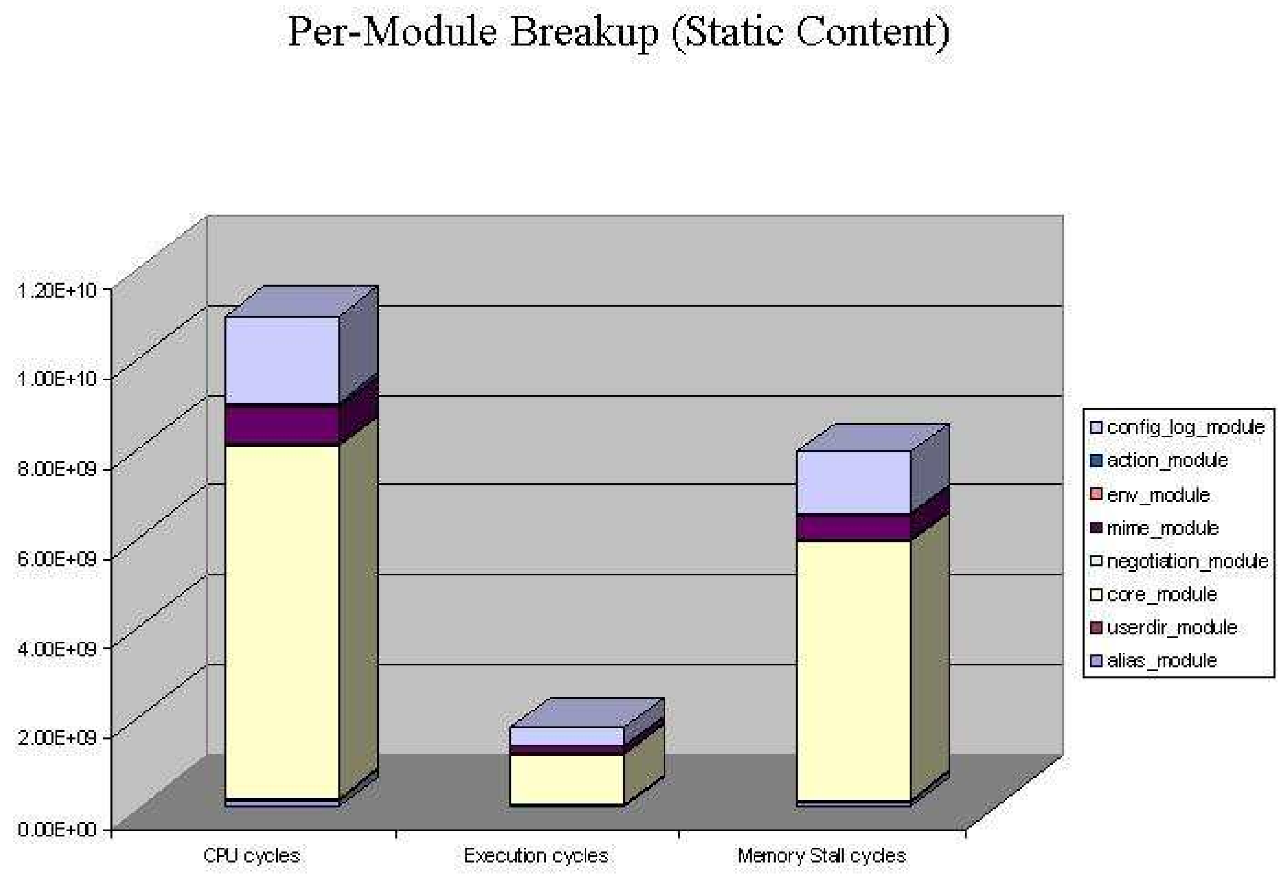

Figure 15: Per module cycle breakup when fetching static content

percentage distribution of these cycles among the modules. It could be clearly inferred from Figure 17 that the modules, in total, use up more memory cycles than execution cycles, which again re-validates the fact that Apache is data intensive. Figure 18 clearly shows that the "cgi module" accounts for a major chunk of each of cpu cycles, memory cycles and execution cycles. While the config log module is a distant second, all the other modules contribute a very small chunk to the total cycles. The cgi module is the main module that defines the handlers which handle incoming dynamic content requests. This module is responsible for spawning processes (using popen) and reading the output of each of these processes (from the pipe created above). The config log module has already been described above. Thus the "hot path" of execution for dynamic content proceeds through the cgi module and the config log module. While

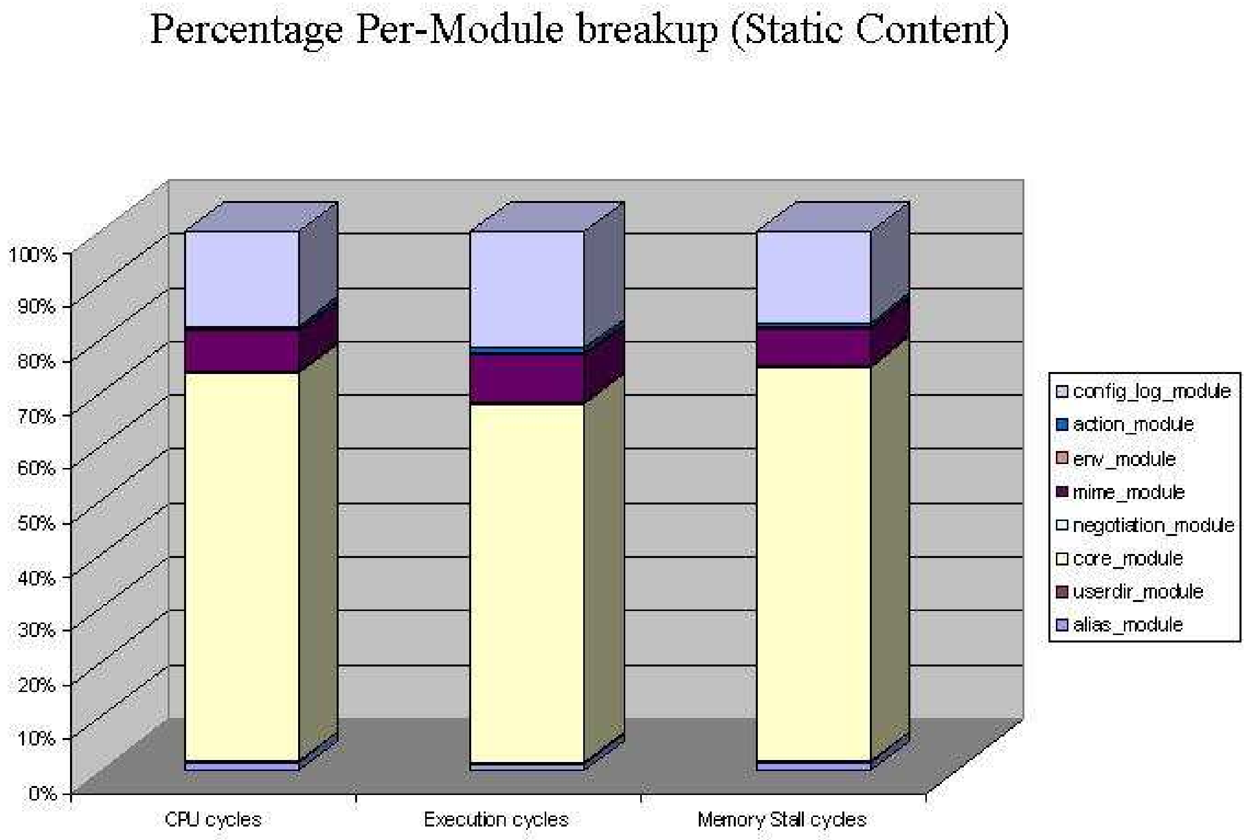

Figure 16: Percentage per module cycle breakup when fetching static content

some of the overheads involved in the config log module can be reduced, overheads in the cgi module are significantly difficult to handle. These would probably need more Operating System support such as efficient inter-process communication (pipes), compiled code instead of interpreted code (mod perl), greater modular support for server side includes so as to minimize the overheads in allocating a new process, with a new memory map of its own etc.

Some optimizations for the config log module have been discussed later in Section 7. However a more detailed discussion of optimizations for the cgi module, apart from those mentioned above, are beyond the scope of this report.

Figures 19, 20 and 21 compare the actual values of per module cycle breakup between

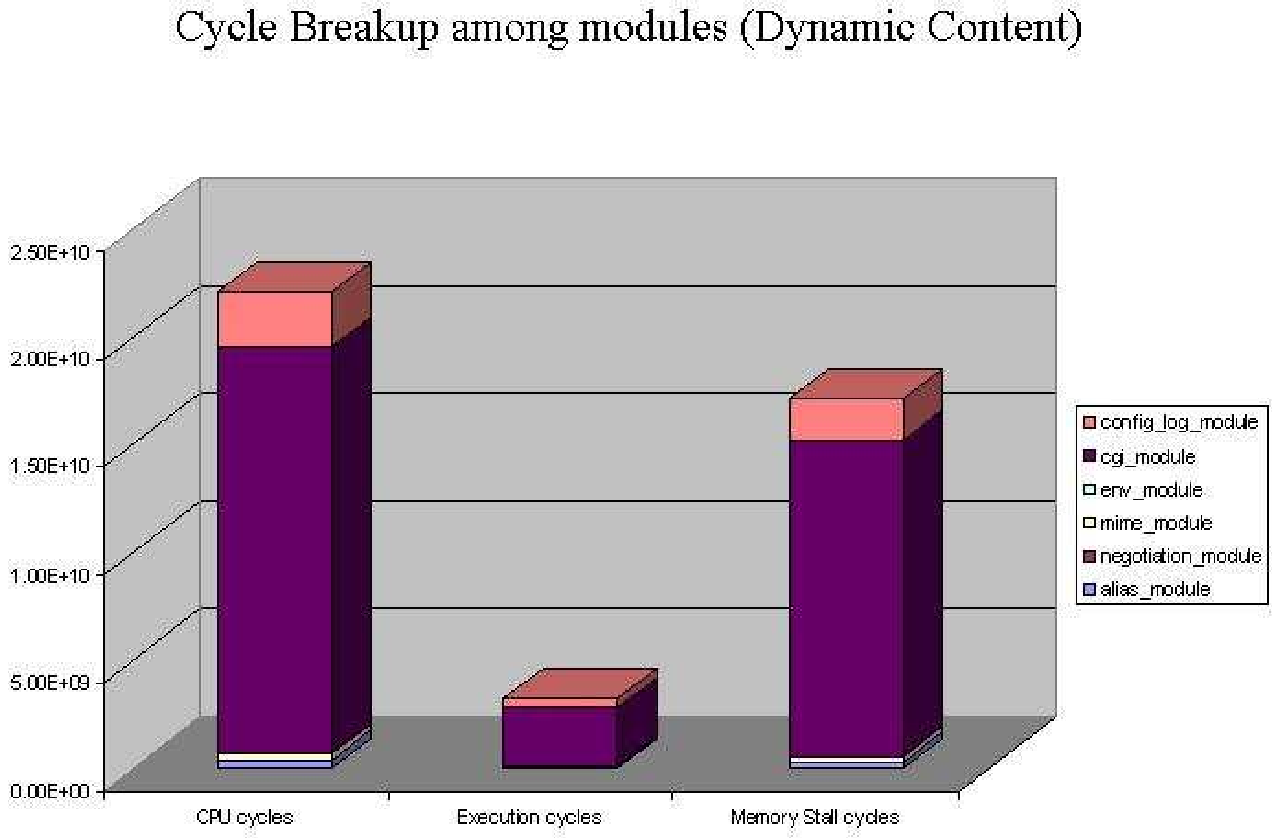

Figure 17: Per module cycle breakup when fetching dynamic content

the static content retrieval run and the dynamic content retrieval run. These graphs serve the informative purpose of more clearly illustrating the differences between the module breakup of cycles for static content and that for dynamic content. Figure 19 clearly captures the idea that while the core module accounts for a major chunk of cycles during static content retrieval, the cgi module performs a similar role during dynamic content retrieval. Contributions of each of the other modules are comparable across the two types of contents. The actual values are compared in Figures 19 and 20. Figure 21 compares the percentages. While dynamic content retrieval has greater total overhead, the largest percentage of time is accounted for by the cgi module. Infact this is so great that it pales out the cost contributions of other modules. On the other hand, static content retrieval has a far smaller cost dominated mostly by core module,

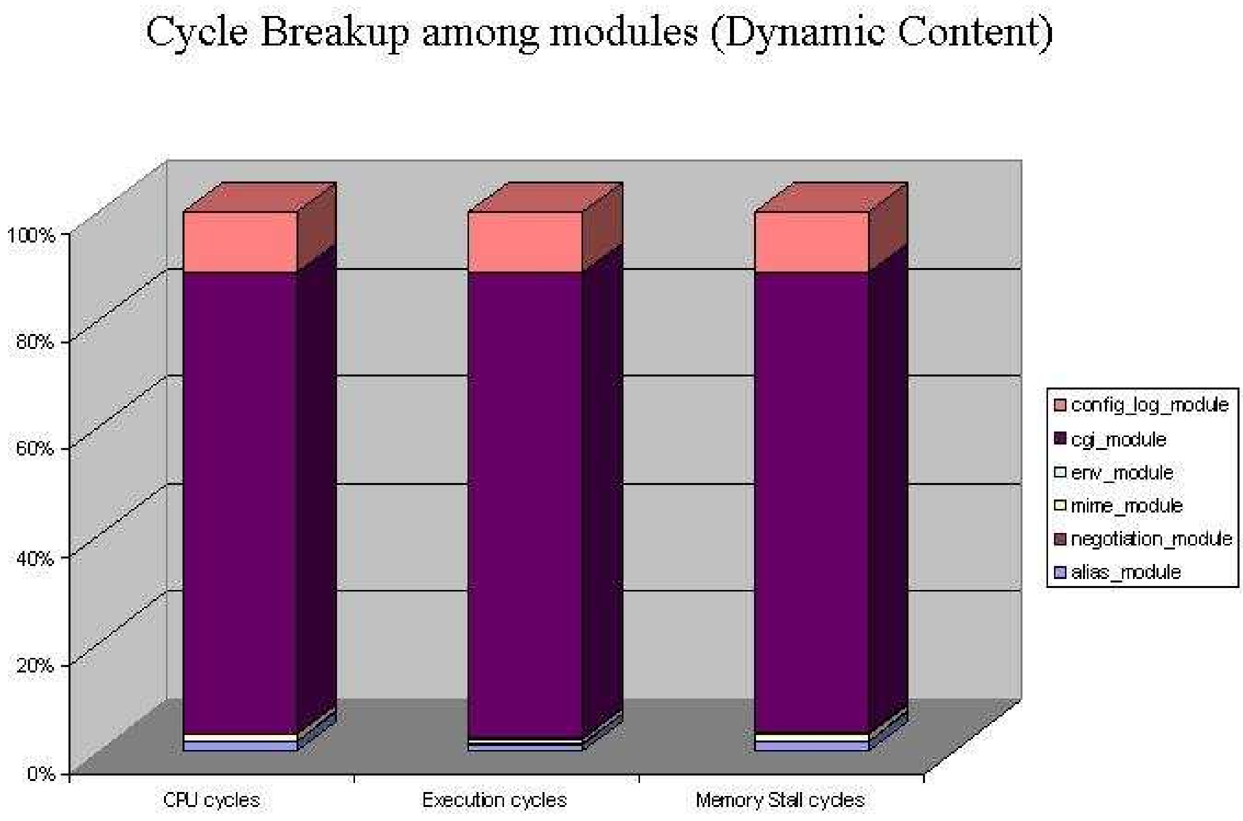

Figure 18: Percentage per module cycle breakup when fetching dynamic content

the config log module and the user dir module.

6.2.4 Cycle breakup across execution levels

Figures 22 and 23 compare the cycle breakup between the instructions that execute in the user level and instructions that execute at the kernel level. Comparing the amount of time that Apache spends in userland with the amount of time it spends in execution of system calls is interesting. We have not only compared the total cpu cycles spent in each privilege level but have also accounted for each individual cycle spent by Apache in each privilege level for static content retrieval. While Figure 22 compares the actual values, Figure 23 compares the percentages. Though the total number of cycles are roughly shared equally between user and kernel level, a greater proportion of the total execution time is spent in user level and a greater proportion of the total memory stall cycles are spent in the kernel level. The latter could be explained as the kernel has to read in the required data information from the kernel space to the user space. The former might be because of the way Apache's application is structured, more user level code with lesser system calls.

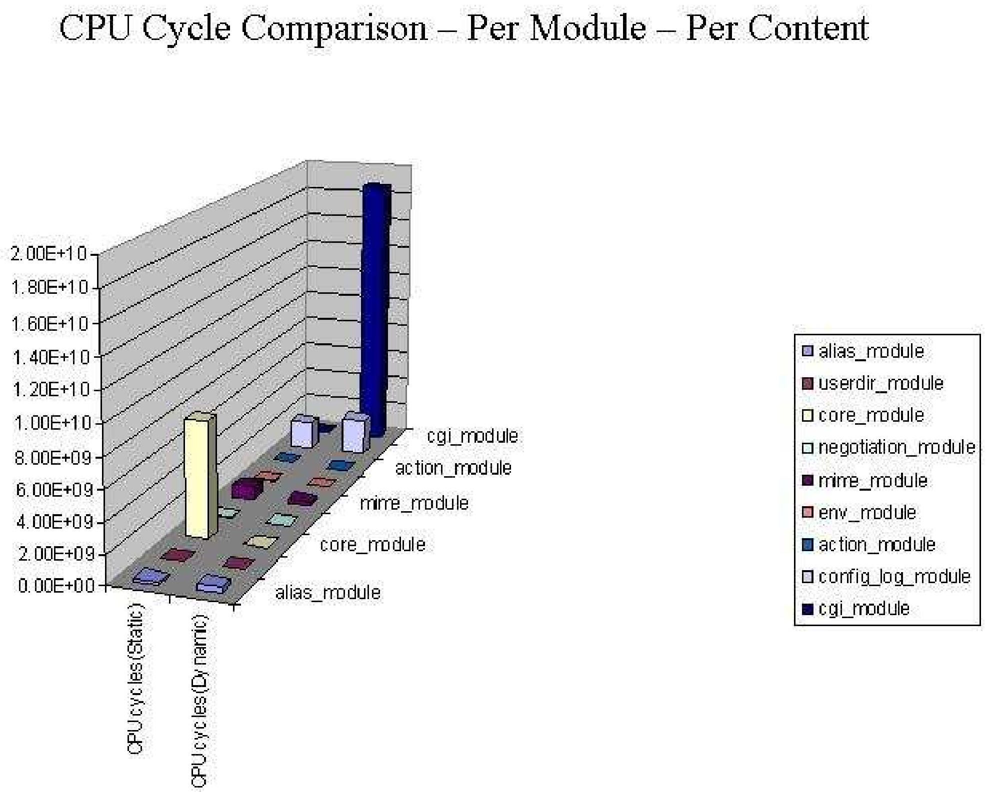

Figure 19: Comparison of per module cycle breakup for different content types

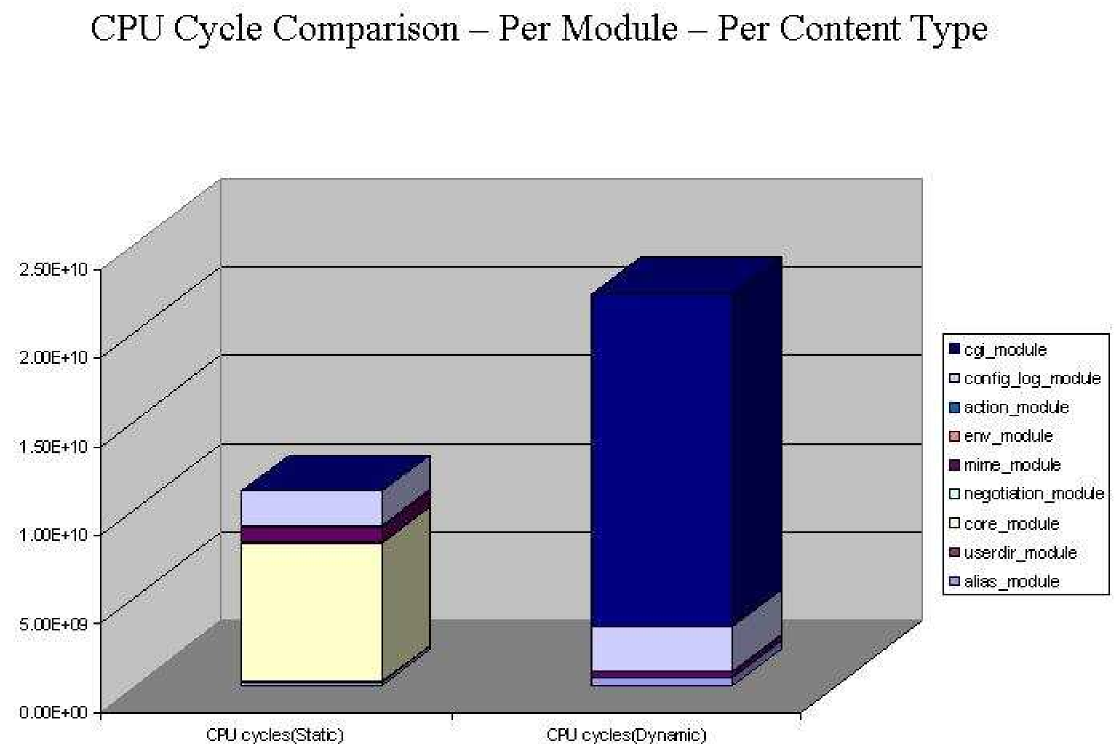

Figure 20: Comparison of per module cycle breakup for different content types: another view

6.3 Miscellaneous Results

The following table summarizes the results we obtained for some of the basic metrics that we had described in Section 5.

Metric	Static Content	Dynamic Content
IA64 IPC	0.5294	0.3834
#Floating point Operations per instruction	0.00041	0.00266
ITLB Miss Ratio	1.4569%	1.3531%
DTLB Miss Ratio	0.5669%	0.6641%

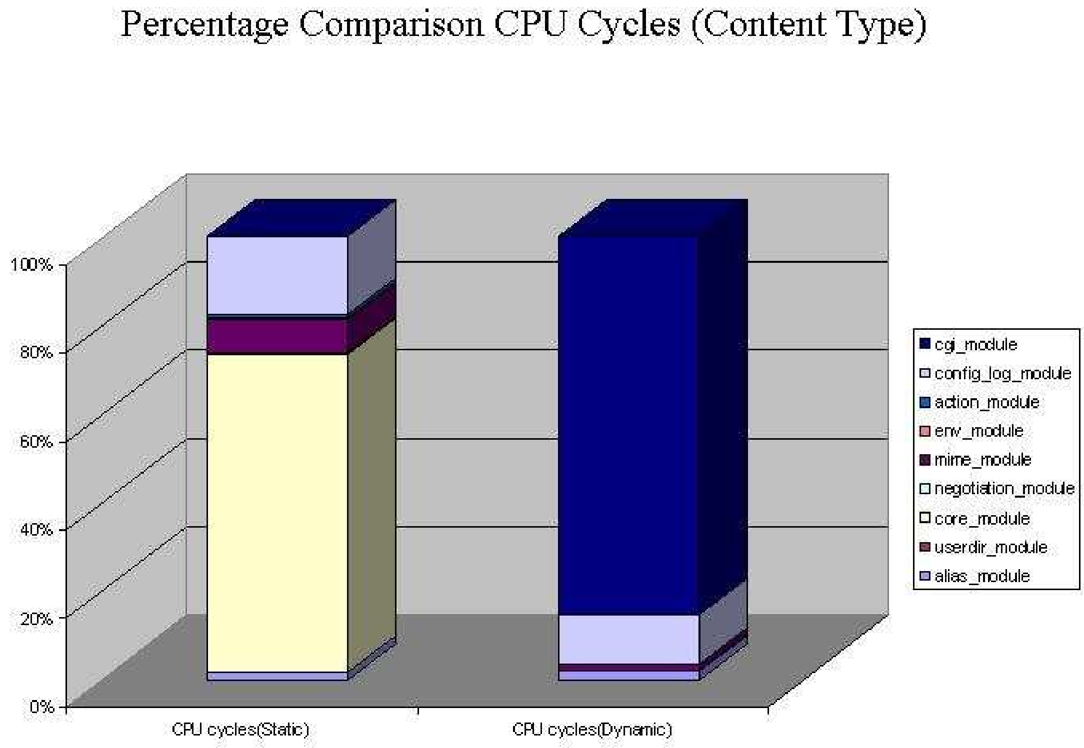

Figure 21: comparison of percentage per module cycle breakup for different content types

6.3.1 Instructions Per Cycle

Since we compiled Apache on the IA64 machine, the total number of IA32 instructions executed were zero. This also makes the number of transitions between IA64 and IA32 architectures, as obtained by the value of *isa_transitions*, zero. Since retrieval of the dynamic content spends a major portion of its cycles in memory stalls, it achieves a low IPC of about 0.38. On the other hand, efficient caching allows retrieval of static content to get an IPC of 0.53. Note that since, the issue width of Itanium is 6, both these numbers are well below the ideal value(6). This is because Apache is a data intensive process leading to a large number of stall cycles.

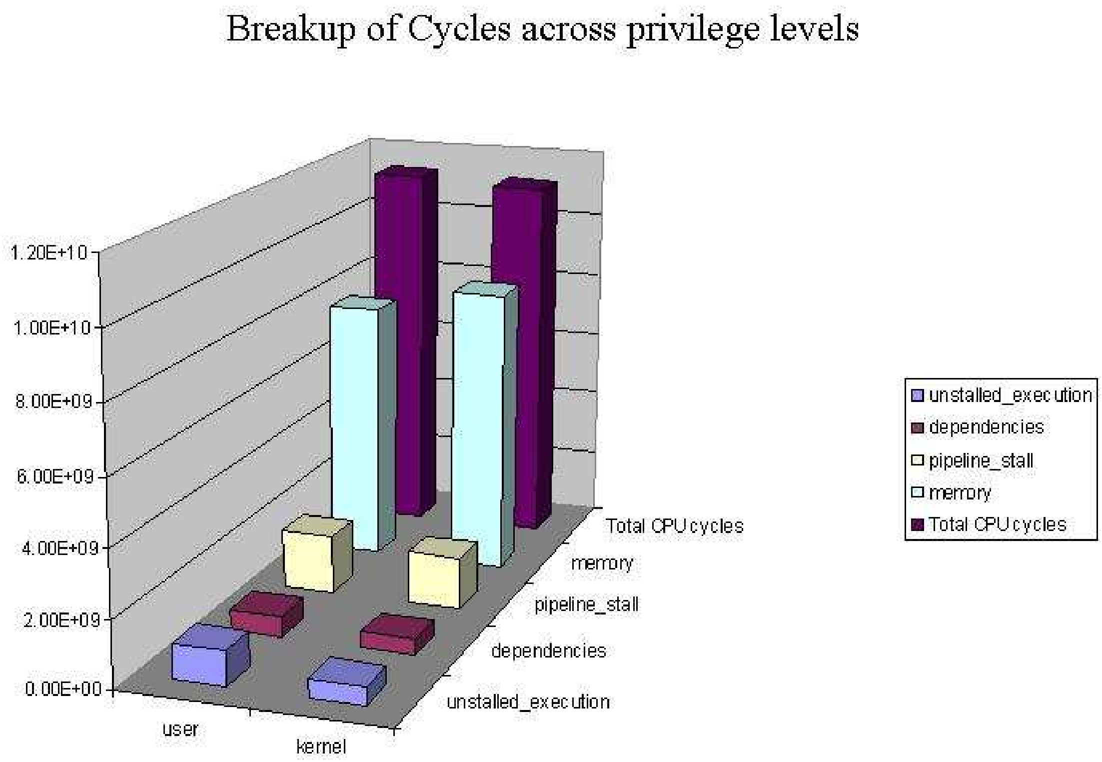

Figure 22: Cycle breakup across privilege levels

6.3.2 Floating Point Operations per instruction

As has been conjectured earlier, the number of floating point operations that are retired are very few. Retrieval of Dynamic content has a slightly larger proportion of floating point operations, perhaps due to the different family of instructions that are executed in reading from pipes.

6.3.3 TLB Miss Ratios

As has been explained earlier, greater number of instructions are executed in the retrieval of static content, thereby a greater proportion of branch instructions are executed. Thus it can be intuitively reasoned that the static content retrieval would have a greater ITLB miss ratio. On the other hand, dynamic content retrieval forces reading from an OS pipe. Since there is possibly a greater variance between the virtual mem-

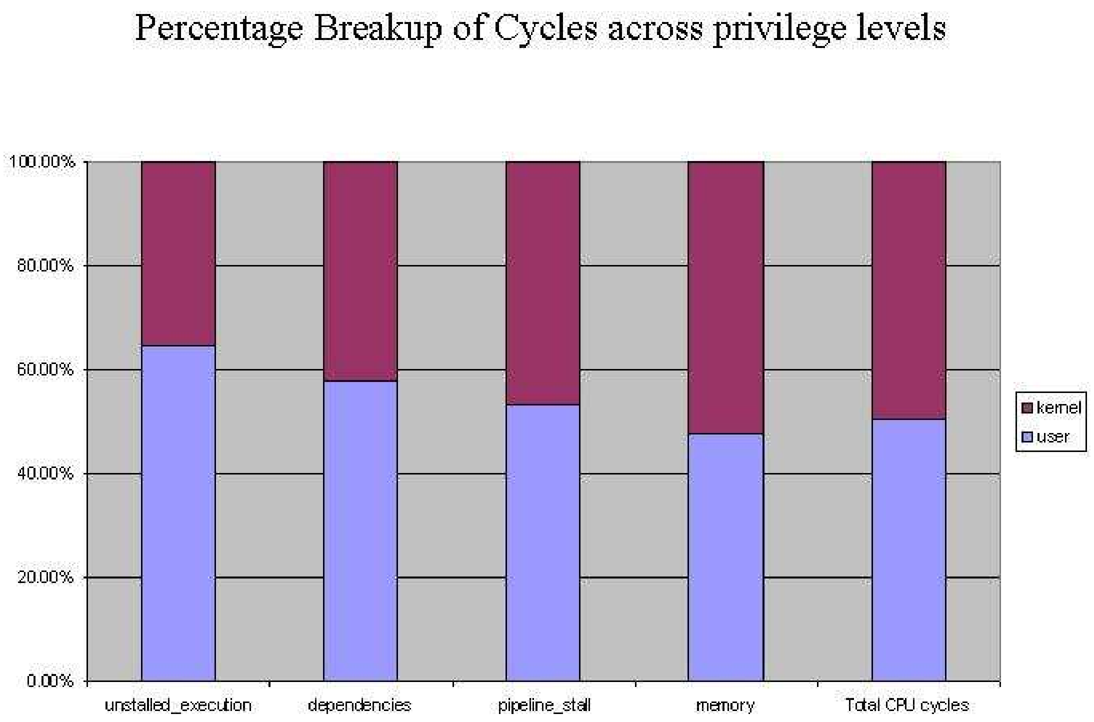

Figure 23: Percentage cycle breakup across privilege levels

ory regions accessed and very few repetitions in the access patterns, dynamic content retrieval has a greater DTLB miss ratio.

7 Suggested Improvements to Apache

While we intend our work to lay a groundwork for other researchers and developers to try out potential improvements in Apache, we ourselves suggest the following improvements. These are based on insights from the analysis of Apache's performance as described above.

- **Decreasing memory stalls in core module**: As we can see, Apache spends a lot of cycles stalling for memory access. When fetching static content, maximum time (and maximum memory stall cycles) are spent in the core module. A sizeable fraction of this is spent in copying data from the kernel address space (file system cache) to the user address space. We believe that this problem can be alleviated to some extent by mapping file system data in the file system cache to the user address space using the *mmap* system call—thus avoiding a data copy. The tradeoff is that it is slightly more expensive to read mmap'd data from the user address space, but we believe that this would be more than offset by the gain in avoiding a memory copy.

- **Improving the performance of config_log_module**: As shown in Figures 15 and 17, a significant amount of time is spent in the config_log_module, both while retrieving static as well as dynamic content. Log writes are already buffered, hence we have not found any major optimizations to reduce the memory stalls caused by *writes* in this module. But there are some other optimizations possible. Firstly, the default log format for the *access_log* has a *%h* entry in the beginning of each log line—this means that the web client's DNS host name has to be logged for each request. This entails performing a reverse DNS lookup. Unfortunately, Apache performs this lookup for *each* request. Since, access of a Web page casues several HTTP requests (for html content, images etc.), there is a significant amount of temporal locality in the client access patterns. Thus, a **user-level caching of reverse DNS lookup results** can lead to better performance. Another optimization is possible by **user-level caching of status strings and timestamps**. Apache logs a human understandable string with each status code that is sent back in the HTTP response (like 404 - Not found, 200 - Success etc.), and also logs the timestamp (in human understandable format) with a least count of a second. New strings are generated (through sprintf) for each log line. However, a lot of this is redundant. Typically, Apache generates hundreds of log lines per second, every time regenerating the same timestamp string, and most requests are successful, again regenerating the same status string again and again. Hence, timestamps can be cached as can the status strings (together with a mapping from status codes), leading to the removal of several sprintf operations.

- **User-level caching of files**: This may be a better alternative to the *mmap* approach described above, though with a lot more development effort required. This would largely obviate the need of the copying from file system cache to the user level buffers. However, this would also involve caching of meta-data information like access times etc, and regularly syncing them with the file system. This approach has been shown to be effective in a recent study [8].

8 Modifications to the design presented in the design document

As has been noted in Section 3.2, we have diverged from our initial desing of building load generator applications in Corba, and have instead built them in C. As has been noted in Section 5.2.4, all of the suggested metrics have not been profiled. This was inherently because of lack of sufficient time and due to the fact that the initial design was fairly ambitious.

9 Experiences with Profiling on IA64 machines

PMU has 4 PMC/PMD registers. This allows us to sample atmost 4 events in a single run. Each run of the experiments involves stress testing the Web Server for a considerable amount of time with the "right" kind of traffic. Further to retain statistical correctness, we performed multiple runs of the experiments and used the averages over these runs. Profiling to quantify interesting parameters requires us to obtain the values of several PMU events. Unfortunately, all these events cannot be measured off the same run. This further increases the need to ensure that there are no significant transient deviations in results obtained across runs so as to be able to correlate results across runs. On the whole, while PMU offers a great deal of flexibility and functionality, profiling significantly large systems such as Apache involves a significant amounts of time and effort.

Instrumenting a multi-process system such as Apache, prior to profiling, involved a moderate amount of effort. However access to the source code was necessary and a fair degree of exposure to the internals of Apache was needed before the actual instrumentation could be done.

10 Conclusion

A quantitative study of Apache's performance has been performed using the fine-grained support of Itanium's Performance Monitoring Unit. Existing performance monitoring support was in-adequate to profile a multi-process application such as Apache. A new Apache module was built to support profiling. Very few changes were made to the handlers among the other modules, so as to reduce the overhead due to performance monitoring. The profiling module *mod_instrumentation* is guided by several directives that could be specified through Apaches configuration file *httpd.conf*. These directives expose greater flexibility in profiling by allowing users to specify the PMU events they want to monitor, the Apache modules they are interested in and the privilege level of instructions they want to monitor, among other things. A distributed load generating application was built which spawns load generating clients on the specified list of machines, synchronizes the load patterns of each of these clients and accumulates the results seen by each of these clients. Values of data throughput and average response time, as seen by the clients were analyzed. A per-cycle breakup of Apache's execution time shows that memory stall cycles form the most significant part. This is intuitively expected since Apache is a data intensive program. A comparison between the static content retrieval and dynamic content retrieval modes of Apache shows us that, dynamic content retrieval involves larger latencies, primarily because reading data off OS pipes involves a significantly larger number of cache misses. This has been clearly documented in the comparision of the memory hierarchy performance obtained by these individual modes of operation of Apache (static, dynamic content retrieval). A per-module breakup of cpu cycles was performed so as to identify the *hot modules* for each of the above modes of operation. Results suggest that the largest chunks of cpu cyles are spent in the *core_module* and *cgi_module* for static and dynamic content retrieval respectively. The *config_log_module* which appends entries to Apache's access log and error log was observed to be the second largest contributor. Several suggestions have been made to optimize the performance of each of these *hot modules*. We believe that future work could build on the implemented *mod_implementation* module of Apache to perform statistical sampling of branch events, Data cache misses and Instruction Cache misses. Another future work could involve the implementation of some of the suggestions for improvement mentioned above and measurements of the performance improvements that could be obtained.

References

[1] Netcraft Web Server Survey, http://www.netcraft.com/survey/

[2] Design Document of SpecWeb99. http://www.specbench.org/osg/web99/docs/whitepaper.html

[3] WebBench 3.0 from eTestingLabs http://etestinglabs.com/default.asp

[4] Apache Server Documentation. http://httpd.apache.org/docs/

[5] Alvin Lebeck et. al "Cache Profiling and the SPEC benchmarks: A Case Study", Tech Report, CS, Univ. of Wisc-Mad.

[6] Intel IA-64 Architecture Software Developer's Manual. Volume 4. Itanium Processor Programmer's Guide (Detailed description of PMU etc.)

[7] Intel IA-64 Architecture Software Developer's Manual. Volume 2. IA-64 System Architecture (More on IA-64)

[8] Yiming Hu et. al, "Measurement, Analysis and Performance Improvement of the Apache Web Server,", to be published Dec. 2001

A Apache Web server Base Modules (by type)

Note that this is the complete list. Not all of these are compiled into Apache by default.

Core

core
Core Apache features

Environment Creation

mod_env
Passing of environments to CGI scripts

mod_setenvif
Set environment variables based on client information

mod_unique_id
Generate unique request identifier for every request

Content Type Decisions

mod_mime
Determining document types using file extensions

mod_mime_magic
Determining document types using "magic numbers"

mod_negotiation
Content negotiation

URL Mapping

mod_alias
Mapping different parts of the host filesystem in the document tree, and URL redirection

mod_rewrite
Powerful URI-to-filename mapping using regular expressions

mod_userdir
User home directories

mod_speling
Automatically correct minor typos in URLs

mod_vhost_alias
Support for dynamically configured mass virtual hosting

Directory Handling

mod_dir
Basic directory handling

mod_autoindex
Automatic directory listings

Access Control

mod_access
Access control based on client hostname or IP address

mod_auth
User authentication using text files

mod_auth_dbm
User authentication using DBM files

mod_auth_db
User authentication using Berkeley DB files

mod_auth_anon
Anonymous user access to authenticated areas

mod_auth_digest
Experimental MD5 authentication

mod_digest MD5 authentication

HTTP Response

mod_headers
Add arbitrary HTTP headers to resources

mod_cern_meta
Support for HTTP header metafiles

mod_expires
Apply Expires: headers to resources

mod_asis
Sending files which contain their own HTTP headers

Dynamic Content

mod_include
Server-parsed documents

mod_cgi
Invoking CGI scripts

mod_actions
Executing CGI scripts based on media type or request method

mod_isapi (WIN32 only)
Windows ISAPI Extension support

Internal Content Handlers

mod_status
Server status display

mod_info
Server configuration information

Logging

mod_log_config
User-configurable logging replacement for mod_log_common

mod_log_agent
Logging of User Agents

mod_log_referer
Logging of document references

mod_usertrack
User tracking using Cookies (replacement for mod_cookies.c)

Miscellaneous

mod_imap
The imagemap file handler

mod_proxy
Caching proxy abilities

mod_so
Support for loading modules (DLLs on Windows) at runtime

mod_mmap_static
Experimental file caching, mapping files into memory to improve performace

B Sample excerpt from httpd.conf

```
....
#
# Instrumentation
#
<IfModule mod_instrument.c>
# Process Profiling is either On (enabled) or Off (Disabled)
    ProcessProfiling On

# Set of PMU events to monitor
    MonitorEvents "memory_cycle" "data_access_cycles" "inst_fetch_cycle"
                  "inst_access_cycle"

# List of modules to monitor (effective only if ProcessProfiling is Off)
    MonitorModules "core_module" "env_module" "mime_module" "config_log_module"
                   "negotiation_module" "status_module" "includes_module"
                   "autoindex_module" "dir_module" "cgi_module" "asis_module"
                   "imap_module" "action_module" "userdir_module" "alias_module"
                   "access_module" "auth_module" "example_module"
                   "setenvif_module"

# Path to the log files
    LogPath "/home/cs423/ssingh7/data"

# Privilege level of monitoring (user, kernel or both)
    PrivilegeLevel "both"
</IfModule>
....
```