Master thesis
liblaiogen: a generic LAIO implementation

Olivier Crameri
Assistant: Aravind Menon
Professor: Prof. Willy Zwaenepoel

School of Computer and Communication Sciences
EPFL, Lausanne, Switzerland

February 2005

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Abstract

In this thesis, we introduce a new implementation of the LAIO api, `liblaiogen`. LAIO stands for Lazy Asynchronous I/O. It is an api for performing asynchronous I/O. Among several benefits, one of the most important is that LAIO is lazy, in the sense that it creates a continuation only when an operation actually blocks. LAIO was introduced along with an implementation for FreeBSD using scheduler activations to provide this lazy characteristic.

Our objective is to provide a cross-platform implementation. To achieve this, `liblaiogen` uses threads eagerly instead of relying on scheduler activations to save threads for non blocking operations. By doing this we challenge the argument that kernel threads are inherently expensive, which is the justification for the need of a mechanism such as scheduler activations.

We compare the performance of `liblaiogen` in the scope of event-driven web servers, using the same web server and the same benchmark that was used to benchmark the original FreeBSD implementation of LAIO. We show that on recent versions of Linux with lightweight threading support, the web server using `liblaiogen` performs better than the one using LAIO on FreeBSD. We highlight the different components of the operating system that are responsible for the differences in performance.

Acknowledgment

With these lines I wish to thank all the people that have contributed to this thesis. First, I wish to thank all the LABOS members, who kindly integrated me in their team. This include, of course, my advisor Professor Willy Zwaenepoel. I wish to thank him for his supervision and his advices. I would also like to thank Dr. Steven Dropsho who devoted lots of his time to discuss the different issues related to the project and give me feedback on the thesis. Finally, I would like to thank Aravind Menon for helping me bust out those tricky, nasty bugs that I still cannot entirely avoid.

Contents

List of Figures

List of Tables

1 Introduction

We introduce `liblaiogen`, a new implementation of the *Lazy Asynchronous I/O* (LAIO) api[1]. The original implementation of LAIO was a user-level library for FreeBSD. `liblaiogen` is a new implementation of the LAIO api that is cross-platform.

In order to avoid confusion between the two LAIO implementations, we will use the following convention for the rest of this thesis: LAIO will always refer to the api, whereas `liblaio` and `liblaiogen` will refer respectively to the original FreeBSD implementation and the generic portable implementation.

Asynchronous I/O is the base mechanism used by event-driven servers to perform concurrent processing of multiple requests. An event-driven server performs one basic step associated with the serving of a request at a time, interleaving the processing steps of many requests. For this reason, an event-driven server has to avoid blocking on any type of operation because it would block all the requests.

Most operating systems offer non-blocking I/O to perform network operations in an asynchronous manner. Unfortunately, non-blocking I/O is generally not avaiable for disk. So, in order to execute disk I/O asynchronously, one has to use another API such as AIO. This leads to more complicated programming in an event-driven server and limits its portability since AIO is not available on every operating systems. Moreover, even AIO does not support simple I/O operations such as `stat()`, thus forcing event-driven servers to accept that some operations can block.

LAIO, as introduced by [1] is an API to perform asynchronous I/O. This api provides three main benefits. First, it is general, in the sense that the api is suitable for all types of I/O operations (disk, network,...). Second, LAIO notifies the application when an event completes, and never at some intermediate stage. Those two benefits make programming asynchronous I/O much easier and more concise. The third benefit is that LAIO is *lazy*, in the sense that it creates a continuation only when an operation actually blocks. This way, for non-blocking operations, LAIO acts as a simple wrapper and no significant overhead is introduced. This particular characteristic relies on specific support from the operating system and is for this reason not always possible to implement.

`liblaio` is an implementation of the LAIO api as a user-level library for FreeBSD. When an application uses `liblaio` to perform an I/O operation asynchronously, internally `liblaio` executes it using synchronous (blocking) system calls. This way for operations that do not block, no overhead is introduced. However for operations that do block, `liblaio` uses scheduler activation [2, 3] to spawn a new thread and enable the application to continue. The advantages are obvious. First, it simplifies the implementation by using a simple universal mechanism for everything. Second, it does not waste thread because it creates a new one only for blocking operations and in this way `liblaio` implements the lazy characteristic of LAIO.

The kernel support for scheduler activations, on which `liblaio` relies, is a mechanism present in many but not all operating systems. Linux is an example of an operating system without scheduler activations (although some patches exist, they are not officially supported). This limits the portability of `liblaio`. With `liblaiogen`, we provide a higly portable implementation of LAIO. To achieve this, we explicitly dismiss the lazy characteristic of `liblaio` in favor of an implementation where each I/O operation (blocking or not) is run in a separate thread. Our results show that the lazy characteristic is not always essential. Indeed, under operating system with lightweight threading support we show that `liblaiogen` achieves equivalent or even better performance than `liblaio`.

1.1 Contribution

The contribution of this thesis is two fold. First, we provide a new highly portable implementation of the LAIO api that relies on Posix standards. Second, we analyse the performance of this new implementation and highlight the operating system components responsible for the improved performance over the original lazy implementation in liblaio.

1.2 Outline

The rest of this thesis is organised as follow: Section 2 describes the LAIO api and the two implementations, `liblaio` and `liblaiogen`. Section 3 presents the methodology used to benchmark both implementation, analyses the performances and discusses the results. Section 4 presents the related work. We make concluding remarks in section 5. Appendix A presents a possible alternative for the implementation of `liblaiogen`.

2 Design and Implementation

In this section, we first describe the LAIO api. We then discuss `liblaio`, the original implementation of the LAIO api for FreeBSD. Finally, we introduce our own implementation, `liblaiogen`.

2.1 LAIO's interface

LAIO's interface is very simple and effective. It is made of three different calls. The main call, `laio_syscall()` has the same signature as `syscall()`. `syscall()` is used to perform indirect system call and can therefore be used to perform I/O. If the system call is able to terminate without blocking, the behavior of `laio_syscall()` is identical to `syscall()`. However, if the desired system call is unable to complete without blocking, `laio_syscall()` returns `-1` and sets the global variable `errno` to `EINPROGRESS`. We refer to this case as a *background operation*. A background operation is identified by a unique handle which is returned by `laio_gethandle()`, the second call in the api.

Finally, `laio_poll()` is the call being used to collect completed background operations. This call takes three arguments. The first one is an array of `laio_completion` structure. For each completed background operation, an `laio_completion` structure is used to store the return value and error code of the operation, along with the handle identifying the operation. The second argument of `laio_poll()` is an integer indicating the size of the array of `laio_completion` structures, this is, the maximum number of completed operations to collect. The third argument is a `timespec` structure used to specify a timeout telling `laio_poll()` how much time it will wait to collect terminating operations in the case where none are available at the time `laio_poll()` is called.

Figure 1 and 2 show an example of using LAIO.

```
client_write(struct request *request)
{
    client_socket = request->client_socket;
    client_buffer = request->client_buffer;
    nb_bytes_to_write = request->nb_bytes_to_write;

    return_value = laio_syscall(SYS_write, client_socket, client_buffer,
nb_bytes_to_write);

    if (return_value == -1) {
        if (errno == EINPROGRESS) {
            request_handle = laio_gethandle();
            request->event_handler = client_write_complete;
            register_request(request, request_handle);
            return;
        }
        else {
            error_value = errno;
        }
    }
    else {
        client_write_complete(request, return_value, error_value);
    }
}
```

Figure 1: Event handler using LAIO

```
for(;;) {
    rv = laio_poll(completions, completions_size, timeout) ;
    if (rv == -1) {
      handle_error();
    }
    for (i = 0; i < completions_size; i++) {

        errno = completions[i].laio_errno;
        return_value = completions[i].laio_return_value;
        request_handle = completions[i].laio_handle;

        request = find_coressponding_request(request_handle);
        event_handler = request->event_handler;

        (*event_handler) (request, return_value, errno);

    }
}
```

Figure 2: Event loop using LAIO

2.2 liblaio

LAIO's FreeBSD's implementation relies on the kernel support for scheduler activation [2]. In essence, scheduler activation is a mechanism that allows the kernel to directly notify the application of certain events by means of delivering upcalls. Blocking a thread in the kernel, due to I/O, is an example of such an event.

Using this feature, the `laio_syscall()` call operates as follows. Before making the system call, it saves the current thread's context (i.e., the stack). Then it enables kernel upcalls. The next step is to execute the desired system call. If it does not block, then `laio_syscall()` simply disable upcalls and return. However, if the system call blocks, then an upcall is delivered to the application. The upcall handler uses the saved context to change its stack to turn itself into the blocked `laio_syscall()`.This way, the upcall handler can now simply return with the return value set to `-1` and the `errno` set to `EINPROGRESS` in order for the application to continue its normal execution.

Shortly after executing a `laio_syscall()` function that returned a `-1` and set `errno` to `EINPROGRESS`, the application is expected to call `laio_gethandle()` in order to get the handle identifying the background operation so that the application has a way to link a particular background operation to a continuation function.

Whenever the background `laio_syscall()` unblocks, a second upcall is generated. This upcall fills in an `laio_completion` structure with the correct handle, return value, and `errno` value returned by the just completed system call, and adds the structure to a list. The application will be able to retrieve the list of completed operations using the `laio_poll()` function.

2.3 liblaiogen

2.3.1 Background

One of the motivation behind scheduler activation is the claim that kernel threads are inherently expensive. However, using a pure user-level threading library is not suitable to handle operations that may be blocked in the kernel. Indeed, one blocking operation would cause all the user threads to be stalled. The in-between solution consists in having both kernel threads and user-level threads in which user threads are distributed over kernel threads. This is called a M:N threading model. The advantage of this model is that for most non blocking operations lightweight user threads are available but for blocking operations we can still use kernel threads. Scheduler activations allows the programmer to take advantage of this M:N threading model in delivering events to the application informing it of the state of another user thread.

Scheduler activation is present in many modern operating systems, like Solaris, NetBSD and Tru64. However, other operating systems have deliberately chosen to not implement scheduler activations. This is notably the case of Linux. Indeed, the Linux community claims that it is possible, and even preferable, to provide very lightweight kernel threads, in a one-to-one mapping from kernel threads to user threads. Lots of work has been done in this area recently to improve Linux: NPTL [4], a new lightweight threading library, and a new O(1) process scheduler.

The goal of `liblaiogen` is to provide a cross-platform implementation of the LAIO api. For this reason, we abandon the idea of using scheduler activations, in other words we abandon the idea of lazy spawning of helper threads. Indeed, without scheduler activations is not possible to unblock a thread that is blocked in the kernel. Instead, we provide an eager counterpart to `liblaio` in which I/O operations are always spawned in a separate thread, regardless of whether the operation would block or not. This eager model can be implemented in a highly portable manner and we show that under Linux the performance are good.

2.3.2 Implementation

`liblaiogen`'s basic concept is to use a different thread for each operation. Those threads are called helper threads and are kept in a pool to avoid to recreating threads unnecessarily. Each helper thread is represented by a `helper` structure containing a reference to the thread and fields to pass data to and from the main thread. Those structures are referenced in two different lists, one for the free threads and one for the active threads. The `helper` structure contains a field indicating which system call the thread is going to execute, a pointer to an array containing the arguments of the system call to execute, a `laio_completion` structure to store the results of the system call, and some synchronizations primitives. The helper structure also contains a *finish* flag that indicates whether the helper has completed his job and is ready to be collected, or not.

The library also keeps a global counter which remembers the number of completed background operations which have not yet been collected by the application. This global counter must be protected against concurrent accesses by the main thread and the helper threads, therefore, the library maintains a global mutex. A global condition variable is used by the helper threads to inform the main thread of the completion of a background operation. Condition variables have to be protected by a mutex, to prevent the race where the condition variable is signaled before the thread is ready to catch it. `liblaiogen` use the existing global mutex to protect the global condition variable.

Figure 3 shows the implementation of `laio_syscall()` in `liblaiogen` When `laio_syscall()` is executed it first looks for an entry in the free helper thread list (i.e., an helper structure representing an available helper thread). In the case where the free list is empty, it creates a new `helper` structure and a new thread. The `helper` structure is then filled in with the proper value corresponding to the system call to execute, and then added to the list of active threads (the active list). Depending on the state of the thread, it is either woken up by the mean of signaling a condition variable, or simply launched normally.

Figure 4 shows the behavior of an helper thread. First, it executes the desired system call. Then, it fills in the `laio_completion` structure with the return value of the just finished system call, the current `errno` value. The *finish* flag of the corresponding helper structure is set. After that, the

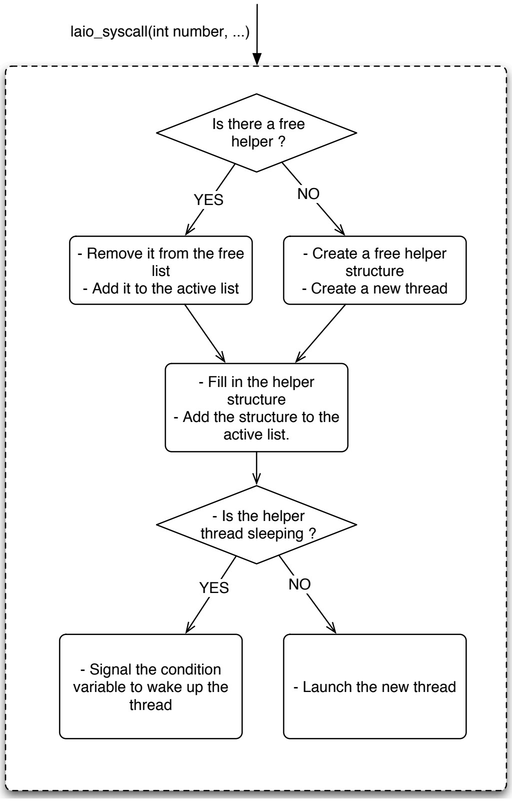


Figure 3: laio_syscall() implementation in liblaiogen

helpert thread locks the global mutex and increments the global counter. Then, it signals the global condition variable to inform the main thread that a system call has completed. Finally, it releases the global mutex, and acquires a local mutex that will allow it to wait indefinitely on a condition variable specific to the thread. This condition variable will be used by a future call to `laio_syscall` to wake up the helper thread so that it can perform another job.

`laio_poll()` is used to collect completed background operations. Figure 5 shows how it is implemented. On entrance, it checks the global counter to determine whether there is already a completed background operation or not. If there is no background operation, `laio_poll()` puts the main thread to sleep by waiting on the global condition variable. It might happen that no operations complete during the time `laio_poll()` is allowed to wait. In this case, `laio_poll()` simply returns. However, if some operations have completed and have to be collected, `laio_poll()` simply browses the list of active helpers to find the helper threads which have their *finish* flag set, indicating that the operation has completed. Those

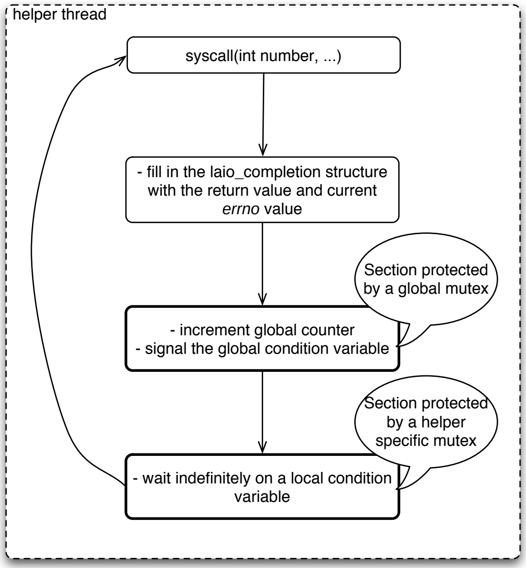


Figure 4: helper thread implementation in liblaiogen

helpers are moved back from the active list to the free list, and the global counter is decremented appropriately. This linear scan might seem expensive, however the list is scanned in the right direction so that older helpers are checked first, this way preventing the scan of the whole list each time `laio_poll()` is called.

Notice that this implementation requires many context switches to perform one I/O operation. The minimum number of context switches is two in the best case. Indeed, the first context switch is required in order to switch from the main thread to the helper thread that will perform the I/O operation. In the best case, where the operation does not block, another context switch is required to switch back to the main thread so that it can continue its execution. However, if the operation does block, then the helper thread will be scheduled out until the operation unblocks, inducing two more context switches.

3 Performance analysis

In this section we present the performance analysis of `liblaiogen`. We first use micro-benchmarks to validate the correctness of `liblaiogen` and to take a look at the overhead that `liblaiogen` introduces. We then analyse the performance of `liblaiogen` in the scope of event-driven web servers using two different web servers. We also compare the performance of the web server using both `liblaiogen` and `liblaio` on Linux and FreeBSD.

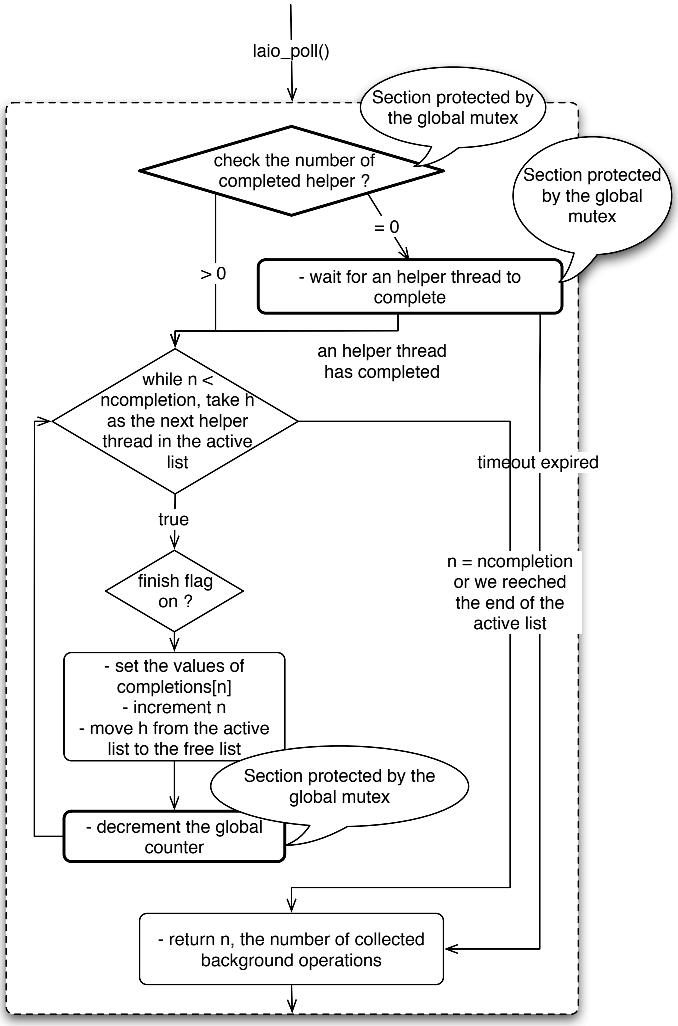


Figure 5: laio_poll() implementation in liblaiogen

3.1 Environment

Our benchmark machines are 2.4 Ghz Pentium IV XEON machines with 1GB of memory, ultra ata hard drives, and gigabit network cards. The operating systems are Linux 2.6.10 and FreeBSD 5.3.

On Linux, we run the benchmarks with two different I/O scheduler, the anticipatory I/O scheduler and the CFQ I/O scheduler. We will refer to those configurations respectively as `linux-as` and `linux-cfq`. Both Linux versions run with the *Native Posix Thread Library* [4], a lightweight threading library for Linux that provides a 1:1 threading model, this is, each user thread is mapped to a kernel thread.

On FreeBSD, we use the `libthr` [5] library to run with `liblaiogen`. `libthr` is a threading library that also provides a 1:1 threading model with a Posix compliant interface. To benchmark `liblaio` on FreeBSD, we use `kse` for the threading system as this is required.

3.2 Background

As described in section 2.3, `liblaiogen` uses threads eagerly for each I/O operations. As we will see in the results, the performance impact of using `liblaiogen` varies greatly between the different operating systems and architectures.

Under Linux, we do the benchmarks with two different I/O schedulers. The I/O scheduler is the part of the kernel that tries to optimize disk I/O throughput by reducing the disk head moves. Disk head moves are responsible for large delays between disk accesses, this is why reordering the disk access in order to minimize the moves generally improves the disk performance significantly. Concurrent disk accesses provide an opportunity for the I/O scheduler to optimize disk I/O even more, because there are more requests over which to optimize the disk head moves. By using asynchronous I/O we will generate a large number of concurrent disk I/O.

The Linux community has been working on completely rewriting the I/O subsystem prior to releasing Linux 2.6. Several new I/O schedulers have been implemented. Eventhough the 2.6 version has now reached is tenth stable version (2.6.10), developers are still improving the I/O subsystem. The anticipatory scheduler [7] is now the default I/O scheduler for most Linux distribution, this is why we started our benchmarks with this one (i.e. with `linux-as`). The idea behind the anticipatory scheduler is to introduce a small delay between the scheduling of subsequent requests. This avoids scheduling of a request from another process before the current process has had a chance to issue its next request. Thus, it allows a better optimization of the disk head moves by preserving locality. Indeed subsequent I/O requests from one process are more likely to be located close to each other on the disk than subsequent requests belonging to two different processes.

Since the results under `linux-as` did not met our expectations we decided to also run the expermint with another I/O scheduler, the time-sliced CFQ scheduler (i.e. `linux-cfq`). The big idea behind this scheduler is to allocate time-slice to each process in order to distribute disk access fairly in a similar way the kernel distribute CPU time among processes.

Linux does has some other advantages over FreeBSD to handle threads efficiently. The new threading library NTPL [4] is very lightweight and scales well to a large number of threads (several thousands). The process scheduler of Linux 2.6 has a complexity of O(1) meaning that the time needed to schedule a process is constant and independent of the number of processes present in the system. Since we are using kernel threads with `liblaiogen` we benefit directly from the O(1) scheduler.

3.3 Workload

In order to benchmark our web servers, we use the same workload and the same procedure that was used in the LAIO paper [1]. This workload was obtained from real web servers at Rice University. Table 1 shows the workload characteristics.

For the micro-benchmarks, we use the trace files as a sequence of requests to perform different I/O operations.

To benchmark the web servers for our macro-benchmarks, we use the trace files with a program simulating concurrent clients sending requests. We vary the number of clients in order to vary the pressure that is put on the server system. The sequence of requests is kept in order, this means that each simulated client takes the next request in the trace file. The program terminates when the trace file is exhausted and reports overall average throughput and response times.

Workload	Nb. of requests	Dataset size	Total data transfers
Rice	245'820	1.1. Gigabytes	8 Gigabytes

Table 1: Workload characteristics

3.4 Micro-benchmarks

We created several micro-benchmarks in order to validate the execution of `liblaiogen` and to evaluate the overhead it creates under Linux.

3.4.1 The getpid() micro-benchmark

The first micro-benchmark is very simple. It consists of executing one million `getpid()` through `liblaiogen`. The reason of choosing `getpid()` is that this is a very simple system call which doesn't execute any I/O, allowing us to evaluate the cost of the context switches induced by the use of `liblaiogen`. We limit the number of parallel operations that can be executed simultaneously, thus limiting the number of helper threads in the system.

Number of helper threads	Time (s)
1	6.12
10	6.10
100	6.12
Single thread, no LAIO	0.80

Table 2: Results of the getpid() micro-benchark under linux-as

Table 2 shows the results of this micro-benchmark. While it takes only 0.8 second to execute one million `getpid()` calls in a single threaded application without using `liblaiogen`, it takes 6.2 seconds to execute them through `liblaiogen` using one helper thread. This huge difference in performance can be explained by the context switch overhead induced by using `liblaiogen`. Indeed, as explained in section 2.3.2, `liblaiogen` introduces at least two context switches per executed operation. This way, while we may have as few as 0 context switches in the single threaded application, we have 2 million context switches using `liblaiogen`. Compared to `getpid()`, a context switch is much more expensive this is why the difference of performance is not surprising. Also not surprising is the fact that increasing the number of helper threads in this benchmark does not improve the performance. However, it does not hurt the performances, probably thanks to the new O(1) scheduler and the NPTL library of Linux.

3.4.2 The open-stat[-read] micro-benchmark

The aim of this micro-benchmark is to simulate a similar amount of disk I/O that is performed by a web-server. The principle is simple, the workload's trace file is read and on each request a sequence of I/O operations is executed. For each request, the first variant of the workload opens the file (`open()` system call) and fetches its size (`stat()` system call). The second variant adds a third phase where the file is read into memory (`read()` system call). As in the `getpid()` micro-benchmark, we limit the number of helper threads in `liblaiogen()` by limiting the number of parallel operations that can execute simultaneously. The results are compared with a program that uses threads to execute the same sequence of operations for each request. The requests are distributed equally among the threads so that each thread proceeds the same number of requests. Each thread processes a request in its entirety, processing the I/O operations sequentially. This is similar to the way a thread-based web server works.

	No liblaiogen		liblaiogen	
Nb. of threads	Cold (s)	Warm (s)	Cold (s)	Warm (s)
1 thread	6.12	2.09	12.47	8.41
10 threads	5.54	1.88	11.66	6
100 threads	5.06	1.92	11.03	7.94

Table 3: Results of the open-stat micro-benchmark under linux-as

Table 3 shows the results of the variant that do not read the files. This table shows clearly that `liblaiogen` introduces overhead compared to using only threads. This overhead can be explained by the context switches that are much more frequent using `liblaiogen`. We see that the overhead introduced is of the same order as with the `getpid()` micro-benchmarks (a few seconds for hundreds of thousands requests). Even-tough we are now performing I/O, increasing the number of threads to perform I/O does not seem to help much for either versions.

	No liblaiogen		liblaiogen	
Nb. of threads	Cold (s)	Warm (s)	Cold (s)	Warm (s)
1 thread	117.5	49.41	126.9	62.83
10 threads	125.9	46.82	102.1	50.81
100 threads	111.9	54.08	82.21	38.88

Table 4: Results of the open-stat-read micro-benchmark under linux-as

Table 4 shows the results of the variant that read all the files of the trace. Here again, we notice a small performance hit on the version that uses 1 helper thread with `liblaiogen` over the version that does not use `liblaiogen` with 1 thread. This performance hit is consistent with our previous micro-benchmarks. Finally, we notice that the version using `liblaiogen` provides a performance boost proportional to the number of helper thread showing that `liblaiogen` is able to exploit concurrency to improve I/O performances. Surprisingly, the version that does not use `liblaiogen` fails to provide better performances with more threads. The only difference between the two versions is the sequence of the operations. While with the version using `liblaiogen` each request is treated in an event-driven fashion, the version that does not use `liblaiogen` serves each request in a different thread processing all the I/O operations sequentially. This difference of behavior is actually similar to what we observe between event-driven web servers and thread-based web servers and suggests that the performance boost provided by the I/O scheduler is sensitive to the order in which requests are processed.

3.5 Macro-benchmarks

This section presents the results of the macro-benchmark used to test and compare the performances of `liblaiogen` and `liblaio`. Those macro-benchmarks consist of two web servers, `ohttpd` and `thttpd`, described in the following sections. The results are explained in the last section.

3.5.1 ohttpd web server

ohttpd is a tiny event-driven web-server that we developed for the purpose of testing `liblaiogen` and understanding which I/O operations have a major impact on the overall system performances. Indeed, with `ohttpd` it is possible to select which operation will be executed through LAIO and which will be executed synchronously. `ohttpd` is very simple, it only understands the subset of the HTTP protocol needed to process the request of our benchmarks. `ohttpd` is small, less that one thousand lines of code. `ohttpd` was useful to test and debug `liblaiogen` because even-though it is small and simple it is functionally correct and provides relevant results in the experiments.

The actual sequence of operations needed to handle a request is the following: `accept()` to accept the connection, `read()` to read the request from the connection, `open()` to open the requested file, `stat()` to fetch the requested file's size, `write()` to send back the HTTP headers, and finally `sendfile()` to send the requested file. The `write()` is not needed under FreeBSD because `sendfile()` takes an extra argument to send the headers. However, under Linux, before executing the `write()` we set the TCP_CORK socket option in order to ask the kernel to send the packet only when there is a minimum amount of data to send, this prevents the kernel from sending tiny packets just for the headers. This option has the same effect as using the extra `sendfile()` parameter under FreeBSD. `sendfile()` is a system call present in many modern operating systems such as FreeBSD and Linux. It reads from a file descriptor and writes the contents to another one without the need for mapping the file in user-space.

3.5.2 thttpd web server

The thttpd [6] web server is a well known event-driven web server. It uses non-blocking I/O for network and blocking I/O for disk. `thttpd` is a good example of a server where the developer has chosen to tolerate the performance penalty induced by the use of synchronous I/O in order to improve the portability and decrease the complexity of the code.

The version we use is the 2.25b version that has been modified by the authors of LAIO [1]. The first modification introduces the `sendfile()` system call instead of the standard `write()` from a mapped file. The second modification consists in introducing `LAIO` for every I/O operation (including disk I/O). This way, we have two similar versions of `thttpd`. The first one, that we will call `thttpd-nb-b` is the version that uses non blocking I/O for network (including `sendfile()`) and blocking I/O for disk. The other version, uses LAIO for every operations. We will call this version `thttpd-liblaio` for the one runnning `liblaio` under FreeBSD, and `thttpd-liblaiogen` for the one running `liblaiogen` under Linux and FreeBSD.

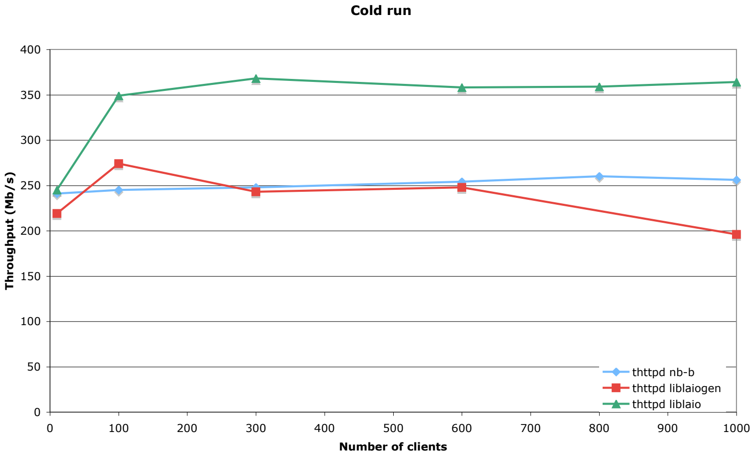
3.5.3 Results

In this section we show the results obtained using `liblaiogen` and we compare them to the results obtained using `liblaio`. We start by showing our reference result, `thttpd` using `liblaio` and comparing it with `thttpd` using `liblaiogen`, both running under FreeBSD. We then use `ohttpd` to highlight the differences of the two versions of Linux we tested. The reason of using `ohttpd` and not `thttpd` for this benchmark is that `ohttpd` is more flexible. Finally we use `thttpd` to compare the performances of `liblaiogen` under Linux and `liblaio` under FreeBSD. Our objective is to determine if `liblaiogen` is able to compete with the lazy implementation.

Figure 6 shows the throughput results of the benchmark for the three variants of `thttpd` under FreeBSD. As expected, `thttpd-liblaio` performs much better both for throughput and response time than `thttpd-nb-b`. The overall results have smaller throughput and larger response time, compared to the results obtained in the LAIO paper [1]. However, this is not disturbing because the machines are different and the relative results of `thttpd-liblaio` and `thttpd-nb-b` are consistent.

We see here that `thttpd-liblaiogen` performs very badly. Not only is its throughput smaller than `thttpd-liblaio`, it is actually smaller than `thttpd-nb-b` for the cold run and approximatively equivalent for the warm run. The difference between the cold run and the warm run can be explained easily. During the cold run, most disk I/O operations are blocking. For this reason, the threads created by `liblaiogen` stay active longer in the system with the consequence that more threads need to be created in order to queue all the requests that comes in. However, during the warm run, the number of blocking disk I/O operations is much smaller because a big part of the workload is already in the memory. This way, the number of threads that stay active in the system is also much smaller (because I/O operations complete faster), thus, the overhead is smaller. What is surprising, is that even-though we are avoiding stalling the server on disk I/O, the results are worse. This means than the overhead introduced by `liblaiogen` is quite high.

Figure 7 shows the response time for the same experiments. Usually, we would expect to see the response time increase linearly with the number of requests. This is not the case for the version that uses `liblaiogen`. We have no direct explanation for this, however, as we will see later with the Linux results, the poor results we get using `liblaiogen` might indicate that there is a design or implementation flaw in the kernel leading to some strange behavior under heavy load using lots of threads. Anyhow, these results clearly confirm that `liblaio`, the lazy implementation of LAIO, is a better design for FreeBSD.



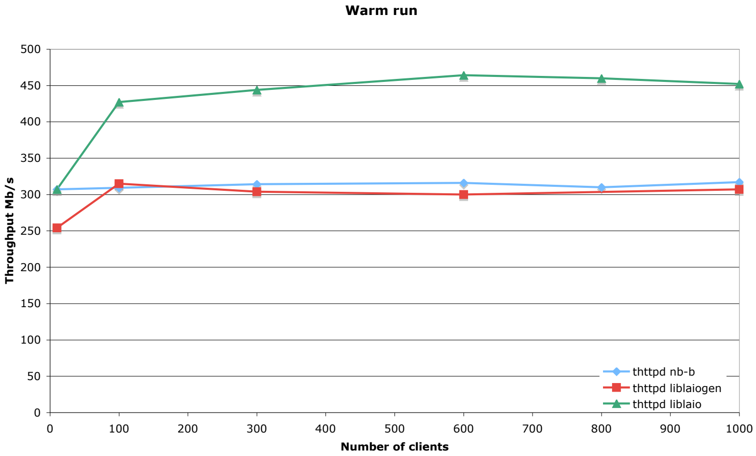
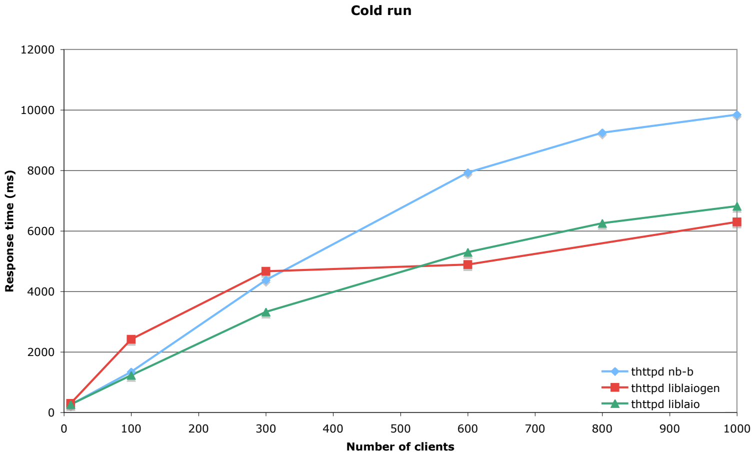


Figure 6: Throughput for the different versions of thttpd running under FreeBSD



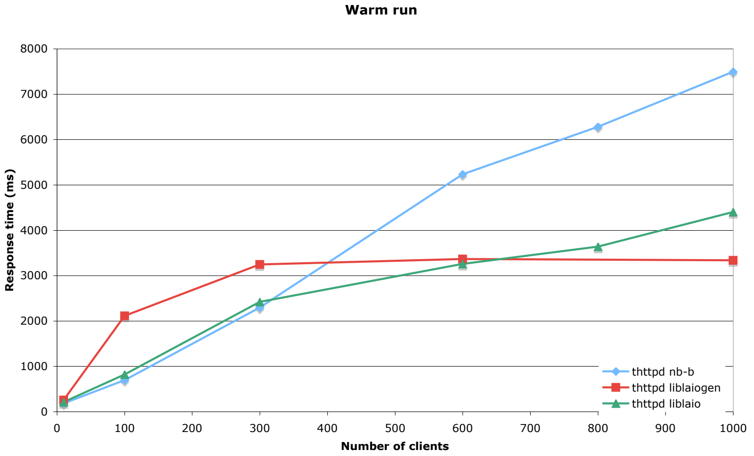


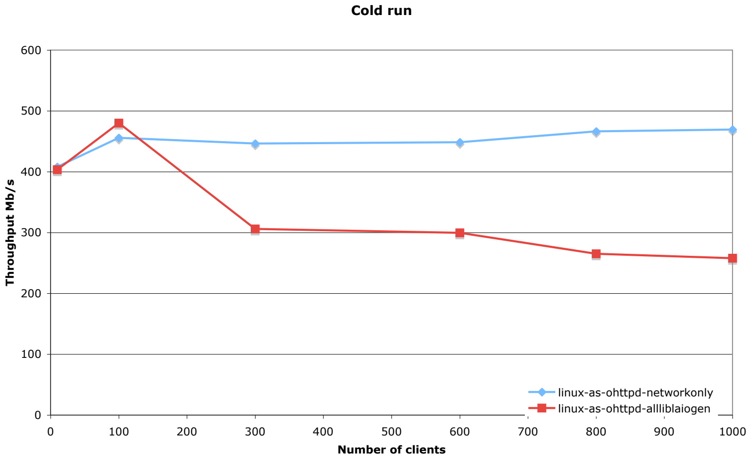
Figure 7: Response time for the different versions of thttpd running under FreeBSD

We will now have a look on the Linux results. Figure 8 shows the throughput results for two different versions of ohttpd with the Rice workload. The first version, that we call `ohttpd-networkonly` is a version that uses `liblaiogen` for everything but disk-only operations . Those operations, namely `open()` and `stat()` are executed using synchronous I/O.[1] The second version, which is called `ohttpd-allliblaiogen` executes all the I/O operations through `liblaiogen`. At first, we did all the measures with `ohttpd-allliblaiogen` only, and because we obtain such bizarre results, we tried to modify `ohttpd` in order to understand which operations where the most expensive to execute through LAIO. The results are quite interesting. Indeed, using `ohttpd-allliblaiogen`, starting from 100 clients we get a smaller throughput than with `ohttpd-networkonly`. This is true both for the cold and the warm run. Interestingly, for the warm run the gap between the two versions decreases progressively with the increase of the number of clients and the gap finally disappears with 1000 clients. The fact that the versions that uses `liblaiogen` for `stat()` and `open()` perform worse is not totally surprising considering the results of our open-stat micro-benchmark. Indeed, we could see that there is no benefit to execute `open()` and `stat()` through `liblaiogen`, probably because those operations do not block long enough. Nevertheless, here we do execute many other blocking operations through `liblaiogen`, so we were disappointed to not get a performance improvement comparable to the one of the open-stat-read micro-benchmark.

Figure 9 shows the response time for both versions. Here again, we get very surprising results. The only linear curve is the one corresponding to `ohttpd-networkonly` for the warm run. This is actually the version that spends the least time executing disk I/O concurrently (since `open()` and `stat()` are executed synchronously and it is the warm run so most of the workload is already loaded in memory). This seems to indicate that performing disk I/O concurrently is a key problem for the kernel and introduce considerable overhead. The fact that the curve for the response time of `ohttpd-networkonly` for the cold run is not linear is explained similarly. Here also, we have an indication that the order in which we process the requests influence greatly the performance. Indeed, the order in which the I/O operations are processed is influenced by the number of clients because it determine the number of concurrent operations that will be executed through `liblaiogen`. Thus, with a different number of concurrent operation the order in which the operations complete might be different because the I/O scheduler might make different decisions. In `ohttpd-networkonly` this phenomena is limited because the two synchronous calls executed for each request reduce the possibility of having a request overtaking another, thus limiting the variability of the order.

---

[1] Actually, the name ”networkonly” is a bit misleading since this version includes sendfile() and sendfile() does perform disk I/O.



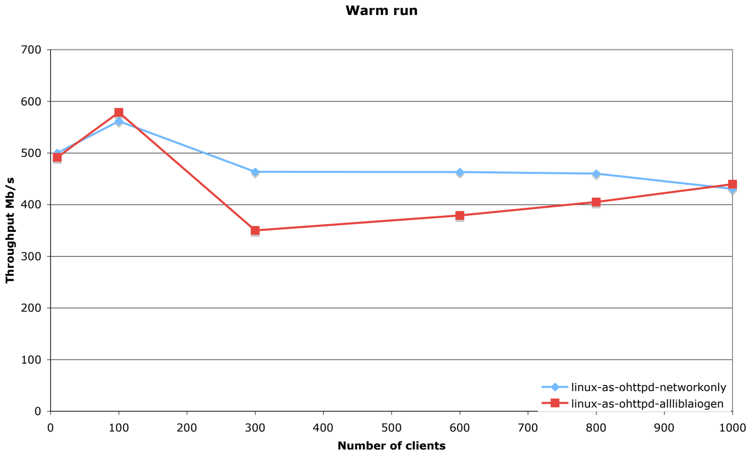
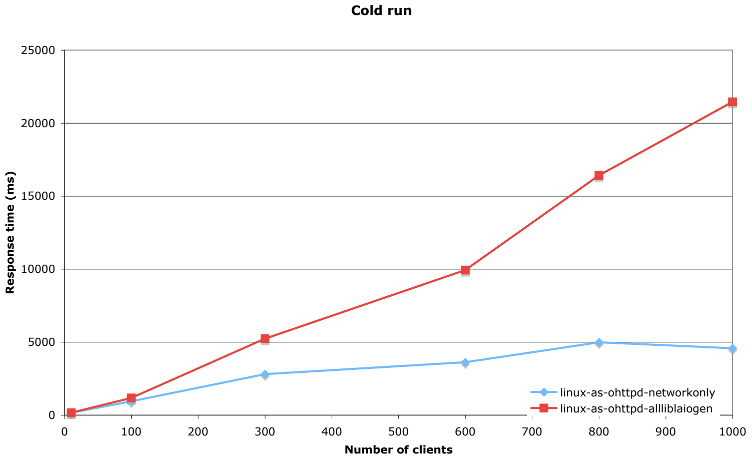


Figure 8: Throughput for the different versions of ohttpd running under linux-as



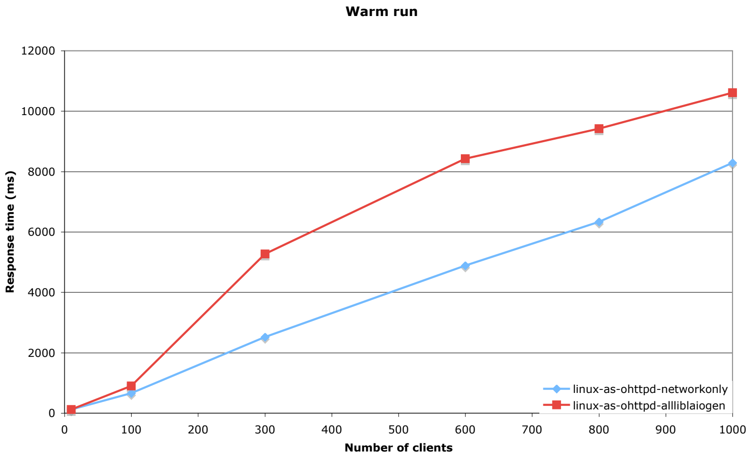
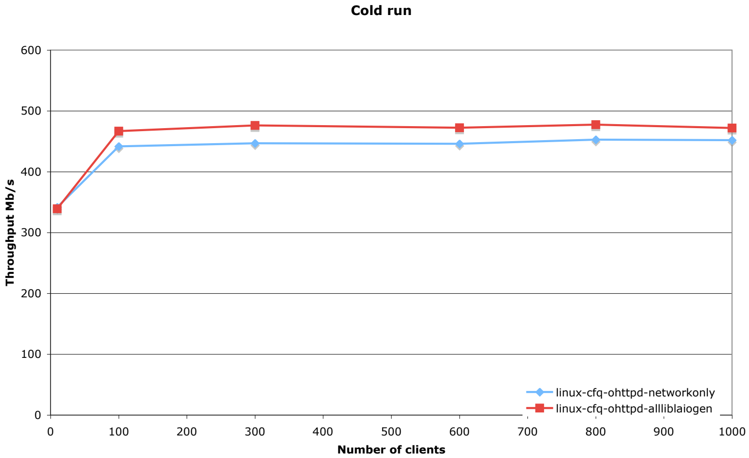


Figure 9: Response time for the different versions of ohttpd running under linux-as

Figure 10 shows the results for the same versions of `ohttpd` and the Rice workload but this time under `linux-cfq`. The results are much better and consistent. The version that uses `liblaiogen` for everything performs uniformly better for the cold run, and for the warm run, results for both versions are more or less equivalent. This is actually the results we expected. `linux-as` and `linux-cfq` have different I/O schedulers, and so the results may be largely influenced by the design of their respective I/O scheduler. The two Linux I/O schedulers are works in progress. The linux time sliced cfq scheduler has already seen 4 major revisions, and the anticipatory scheduler has also been revised several times. Figure 11 shows the corresponding response times. We see here almost linear results for both cold and warm runs. The results are also better than those obtained with `linux-as`. The comparison of the graphs under `linux-as` and `linux-cfq` suggest that the implementation of the anticipatory scheduler [7] under Linux is not yet completely stable and that there might still be some bugs that influence our results. This clearly shows that the I/O scheduler can have a significant impact on the overall system performance in such an environment.



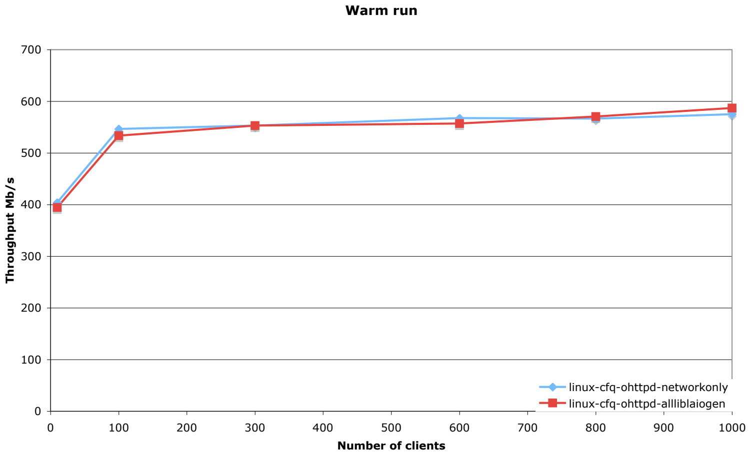
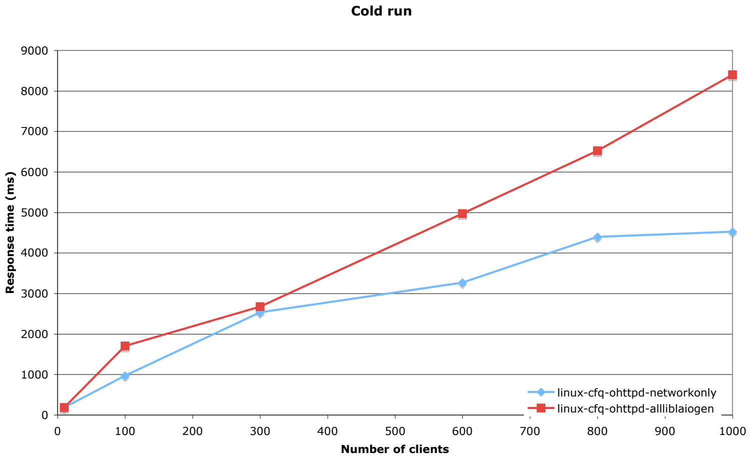


Figure 10: Throughput for the different versions of ohttpd running under linux-cfq



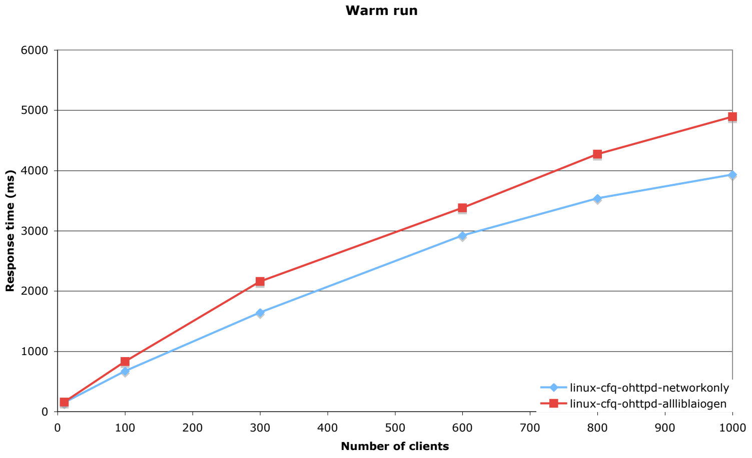
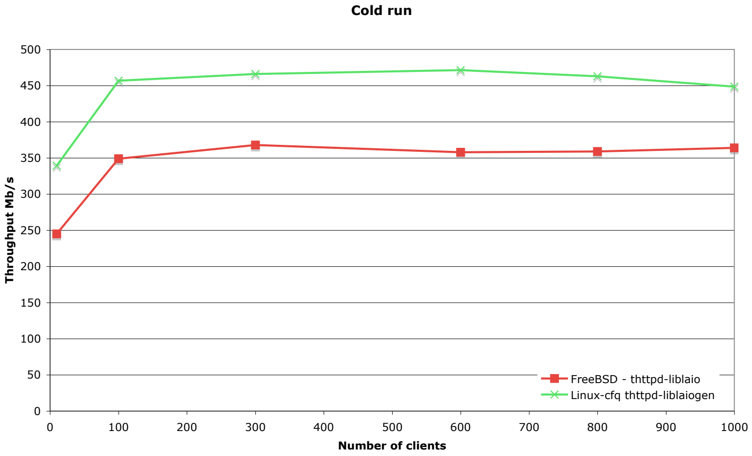


Figure 11: Response time for the different versions of ohttpd running under linux-cfq

We now compare the performances of `thttpd` using LAIO under Linux and FreeBSD. We use `linux-cfq` for this benchmark because we already know that it is more stable and better performing than `linux-as`. On FreeBSD, we run `thttpd` using `liblaio` and not `liblaiogen` because it performs the best.
Figure 12 shows the throughput results. It is clear that `liblaiogen` provides better performances under `linux-cfq` than `liblaio` under FreeBSD in terms of throughput. Figure 13 shows the response time. We see that `liblaiogen` provides a smaller response time than `liblaio` except when the number of clients is greater than 1000. Overall, those results show that with a proper kernel `liblaiogen` is able to perform comparably or even better than `liblaio`.



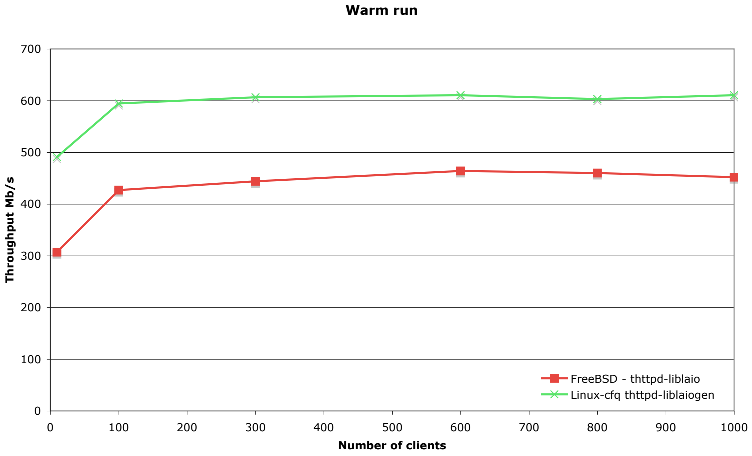
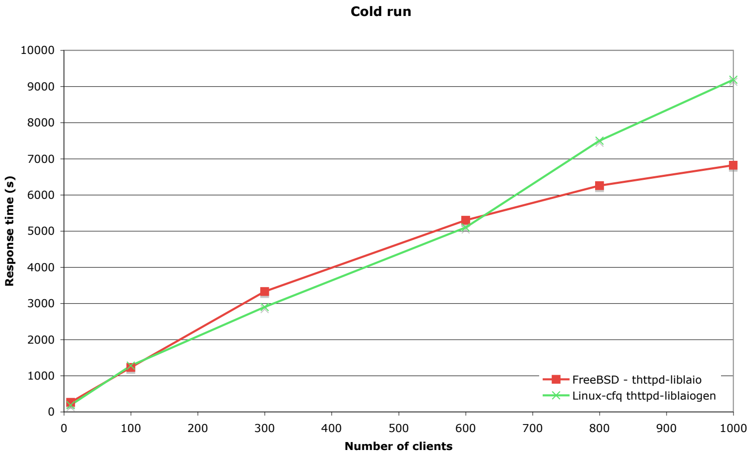


Figure 12: Throughput comparison of the best version under each OS



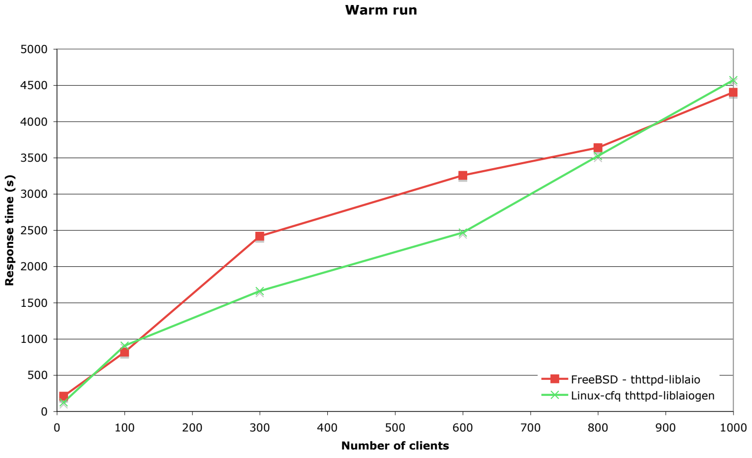


Figure 13: Response time comparison of the best version under each OS

4 Related work

This section present some prior work that exhibit similar characteristics with `liblaiogen`

Flash [8] is a web server that uses the asymmetric multiprocess event driven (AMPED) architecture. It uses non blocking I/O for networking, and helper processes for operations that may block on disk I/O. `liblaiogen` uses the same idea of separate threads[2] for operations that may block, however, there are some important differences. First, AMPED is a server architecture, whereas `liblaiogen` is a library to perform asynchronous I/O. Then, in the AMPED architecture, helper processes are used only for disk. Moreover, the Flash web server tries to guess whether or not a disk operation will block. The result is the equivalent of the lazy characteristic of `liblaio`, helper processes are saved. As already explained, `liblaiogen` does not implement this behavior.

SEDA [9] is an example of another design for highly concurrent Internet services. A SEDA application is constituted of different stages. Each stage of the application is responsible for the processing of one operation (like for instance a network read) and is constituted of a thread pool and a queue. The different stages are interconnected using their respective queues. The stages use their respective thread pool to process multiple requests concurrently. If there is more requests than thread avaiable, the requests stay in the queue until more threads are available. In SEDA, they implement a mechanism in order to adapt dynamically the size of the different thread pools associated which each stage. This mechanism is intended to adapt resource usage to observed server performance in order to prevent the system to fall under heavy load. Although much more simple, we can think of `ohttpd-networkonly`[3] as of something similar to a SEDA server which would have a thread pool size of one for the `open()` and `stat()` operations, and a thread pool of unlimited size for all the other operations. Our results show that this can effectively prevent the performance to drop significantly under Linux with the anticipatory scheduler.

5 Conclusion

We have introduced `liblaiogen`, a new implementation of *Lazy Asynchronous I/O*, an api to perform asynchronous I/O.

`liblaiogen`, by opposition to its cousin `liblaio`, is not lazy. `liblaiogen` uses threads eagerly for each I/O operation without making any distinction between operations that actually block and operation that do not.

We measure with three different micro-benchmarks the overhead introduced by `liblaiogen` under Linux. We show that this overhead is very significant for operations that do not block at all, but that this overhead is

---

[2] Here we make no distinction between threads and processes because for our purpose we use kernel threads which are conceptually very close from processes, even-though the implementation is different.

[3] Reminder: ohttpdnetworkonly is the version of ohttpd that uses liblaiogen for every operations except open() and stat() which are executed synchronously

quickly compensated by the performance speedup obtained for operations that do block.

We then experiment with `liblaiogen` in the scope of event-driven web servers. By using `ohttpd`, a web server developed to test and debug `liblaiogen`, we show that the performance of the web server are directly related to the performances of the I/O scheduler used. We also highlight the very strange behavior of the Linux anticipatory scheduler which might indicate that this scheduler suffers from an implementation flaw.

Finally, we show results using `thttpd`, a well known event driven web server. The results show that under FreeBSD, `liblaiogen` underperforms `liblaio` significantly, confirming the assumption that kernel threads are expensive under FreeBSD. We then show that the same web server using `liblaiogen` in Linux 2.6.10 (CFQ) performs much better than the one using `liblaio` in FreeBSD. The lightweight threading package of Linux, the O(1) scheduler and the high performance CFQ I/O scheduler are the keys to the performance improvement. Thus, this give rises to the question: Would it be possible to achieve even better performance with a kernel that would provide those features and also scheduler activations ?

A A liblaiogen implementation alternative

Another idea for a cross platform implementation of `liblaiogen` consist in using as much as possible the asynchronous api provided by the operating system. In other words, instead of using a thread for each and every operation, we wrap non-blocking I/O every-time it is possible, and we use helper threads for the rest of the operations. We have implemented such a variant.

The implementation of `laio_syscall` doesn't change much. The only difference is that before doing anything, it has to determine whether the desired call to be executed is network related or not. `getsockname()` can be used on file descriptors to check if the file descriptor is a socket or not. If it is not the case, `getsockname()` will return an error. This way `laio_syscall()` knows if it has to invoke non blocking I/O or if it has to use helper threads.

The helper threads stay semantically identical, however there is an important difference. Indeed, we do not use condition variable any more to signal the completion of an operation. This is because in `laio_poll()` we will need to use the `select()` system call to check for completion of non blocking I/O and that this call to `pselect()` will have to be interrupted whenever an helper thread completes (this replace the wait on the condition variable in `laio_poll()`). For this purpose, the helper thread use a signal instead of a condition variable. In `laio_poll()` we now have to check the file descriptor returned by `pselect` in order to catch completed operations in addition to browsing the list of helpers.

By using signals to inform the main thread of the completion of some operation in an helper thread, we have to make sure that the main thread won't miss the signal. To do that `pselect` atomically changes the signal mask so that the signals are blocked until the very moment where `pselect` is able to handle them. Unfortunately, under Linux, `pselect()`'s specification, is not yet fully respected. Indeed, Linux doesn't have a system call for `pselect` so it is implemented in glibc and the signal mask is not changed atomically. This way, the Linux version still contain the race condition where a signal might be sent just after the signal mask has been changed but just before it is ready to handle it. So, our current implementation of `liblaiogen` that wraps non blocking I/O contains this race condition that leads to performance degradation. This is why this implementation is not yet complete and we do not present result with it.

References

[1] Lazy Asynchronous I/O For Event-Driven Servers, Khaled Elmeleegy, Anupam Chanda, and Alan L. Cox, Rice University; Willy Zwaenepoel, EPFL, Lausanne. *Usenix 2004 Annual Technical Conference*.

[2] Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. *ACM Transactions on Computer Systems*.

[3] The FreeBSD Project. FreeBSD KSE Project. `http://www.freebsd.org/kse`.

[4] The Native Posix Thread Library for Linux, Ulrich Drepper and Ingo Molnar, Redhat Inc. `http://people.redhat.com/drepper/nptl-design.pdf`.

[5] FreeBSD: libthr, 1:1 Threading Implementation. `http://kerneltrap.org/node/624` provides references to mailing list archives introducting libthr.

[6] thttpd - tiny/turbo/throttling HTTP server. `http://www.acme.com/software/thttpd/`.

[7] Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O, Sitaram Iyer, Peter Druchsel, Department of Computer Science, Rice University. *ACM Symposium on Operating Systems Principles 2001 (SOSP 2001)*.

[8] Flash: An efficient and portable Web server. Vivek S. Pai, Peter Druchsel and Willy Zwaenepoel, Department of Computer Science, Rice University. In *Proceedings of the Usenix 1999 Annual Technical Conference*.

[9] SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. Matt Welsh, David Culler and Eric Brewer, Computer Science Division, University of California, Berkeley. In *Symposium on Operating Systems Principles, pages 230-243, Oct. 2001*.