Improving Http-Server Performance by Adapted Multithreading

Jörg Keller
LG Technische Informatik II
FernUniversität Hagen
58084 Hagen, Germany
email: joerg.keller@fernuni-hagen.de

Olaf Monien[*]
Thilo Lardon Computertechnik
Füllenbachstrasse 4
40474 Düsseldorf, Germany
email: olaf@monien.net

ABSTRACT
It is wellknown that http servers can be programmed easier by using multithreading, i.e. each connection is dealt with by a separate thread. It is also known, e.g. from massively parallel programming, that multithreading can be used to hide the long latency of a remote memory access. However, the two techniques do not readily complement each other because context switches do not necessarily occur when latencies would suggest them. We present an http server that combines these two features: as soon as an event with a longer latency is encountered, such as the server cannot send all data with one buffer, the context is switched. The implementation is based on the replacement of threads in the http library Indy by an own implementation that allows explicit context switch as in fibers. We benchmark our http server with the apache benchmark against the original thread-based implementation, with different file sizes and different load levels. We find that if the server's network connection is fully utilized, our implementation needs about one third of CPU time to handle the same throughput. If the network connection is not the bottleneck, then our implementation achieves a 26% higher throughput, given a 100% CPU utilization in both servers. The application of fiber-based multithreading is a technique similar to using assembler: it is not feasible on a large scale, but use in a library provides enormous performance benefits transparently to the user.

KEY WORDS
Multithreading, Parallel Processing, Web Technologies, Performance Evaluation

1 Introduction

An http server is an important software application on todays server (hardware) platforms, which possesses an enormous amount of parallelism. It handles multiple — often thousands of — connections simultaneously. Hence, it is appropriate to program them in a parallel manner: each connection is handled by its own thread. Moreover, the performance requirements (especially concerning the throughput) are high. Hence we concentrate on how to improve web server performance.

In each thread, events with a long latency can occur. An example is a connecting client that requests a file, which can be (almost) arbitrarily large, while it has to be transferred via a send buffer that has a fixed finite size. Thus, the data to be sent will not fit completely in the buffer. In this case the thread must wait until the buffer content has been sent over the network before it can fill the buffer again with the remaining data.

These latencies can potentially be hidden by a context switch to execute another thread while the former thread is waiting. However, for typical thread packages, the application has no possibility to enforce a context switch at a particular instant. In addition, although thread switch time in modern operating systems has become quite small compared to process switch time, the context switch time for threads is still rather long in comparison to the latencies incurred. Therefore, the gain would be quite small.

Latency hiding by multithreading has been explored e.g. in the context of the massively-parallel Cray T3E architecture [1, 2]. There, the calls for a remote memory access were detected during compilation and extended with a context-switch. Also, the standard POSIX thread calls were replaced by inserting code of a re-implementation that delivered a much higher performance. This was achieved by keeping the context as small as possible, avoiding operating system calls, and inlining the context-switch code. The latter measure had the advantage that by a compile-time analysis, only parts of the context that are needed in the next section must be loaded, and only parts of the context that were changed in the last section must be stored.

We try to use this idea in the area of http servers. As we want to initiate a context switch at particular places, we look at fibers[1] as used by the Windows operating system, that are related to coroutines which were popular in the eighties [3]. As the Windows fibers are not fast enough, we do a re-implementation (see Section 2). We integrate our fibers into a multithreaded http server of the open source protocol suite Indy [4] and show that only a handful of code modifications are necessary (see Section 3). We evaluate our concept by measuring performance of our http server and the unmodified http server of the Indy suite (see Section 4). We also compare these results to the popular apache http server.

[*]Work done as part of Master's thesis at FernUniversität.

[1]Fibers are similar to threads, but the user can specify when a fiber shall transfer control to another fiber.

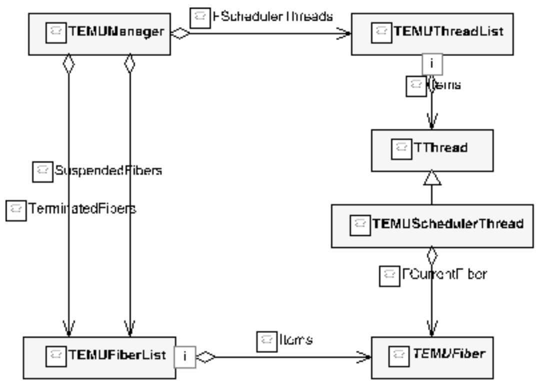


Figure 1. Class concept of EMUfibers

While our concept may seem quite specialized, we once again mention that web applications are a major application area, and that the changes to the http server code happen in the Indy library linked to the application. The changes to Indy are minor. The fiber implementation itself is a library of its own. Our idea in a sense is similar to using assembler: this is not feasible on a large scale, but still done in libraries when performance is a serious issue. The implementation within a library also allows to apply the same idea in other performance sensitive fields as well, may it be in the same library or by applying this technique to another library.

Finally, in contrast to the idea by Grävinghoff, we do not need specialized compiler tools. Therefore we can rely on standard compilers and enjoy all their possibilities for optimization as well.

2 Re-Implementing Fibers

Fibers in the Windows operating system have some disadvantages. First, their documentation is not very good. Second, creating or destructing a larger number of fibers produces a high CPU load, which is not desired because this frequently will happen in an http server, although the concept of thread pools partially alleviates this disadvantage. Third, the fibers are not object-oriented, while the http server we want to use them is written in Delphi, Borland's object-oriented development tool based on the Pascal programming language. Therefore, we decided to implement a fiber class ourselves. We call these EMUfibers in tribute to the EMU system [1] that inspired them.

Necessary was to create code for creating and destructing a fiber, a `yield`-call that switches context from the current fiber so that another fiber can continue, and a manager that schedules the fibers. Additionally, calls for synchronization are implemented, so that the fibers can access the graphical user interface, which in Delphi runs under the MainThread and is not thread-safe. The software construction is shown in Figure 1.

During initialization, we create one Manager instance. The manager maintains two lists: SuspendedFibers and TerminatedFibers. The former contains all fibers to be executed, and the latter contains all fibers that have terminated but are not yet freed, e.g. because another fiber might need some result data from them. The manager also starts and controls one or several threads, onto which the EMUfibers from the list SuspendedFibers are scheduled. See Figure 2 for a complete flow diagramm.

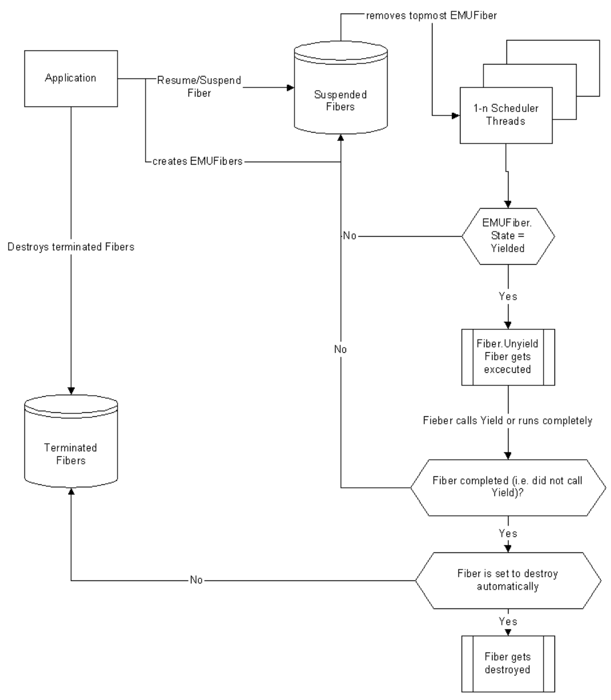

Application
Resume/Suspend Fiber
Suspended Fibers
removes topmost EMUFiber
1-n Scheduler Threads
creates EMUFibers
Destroys terminated Fibers
No
EMUFiber. State = Yielded
Yes
Fiber.Unyield Fiber gets excecuted
No
Fieber calls Yield or runs completely
Terminated Fibers
Fiber completed (i.e. did not call Yield)?
Yes
No
Fiber is set to destroy automatically
Yes
Fiber gets destroyed

Figure 2. Flow diagramm of EMUfiber control

A thread which is given an EMUfiber first checks whether the fiber is ready to run, i.e. in state `Yielded`. If so, it is unyielded and executed until the fiber terminates or calls the `Yield` function. In the former case, the fiber is transferred to the list TerminatedFibers, or freed if it is not to provide any result to another fiber. In the latter case, it is transferred to the list SuspendedFibers.

If the application creates a new EMUfiber, it is in state `Suspended`. If the fiber is resumed by the application, then it takes state `Yielded`. In both states, the fiber is in list SuspendedFibers. The difference between these two states is due to compatibility with Windows threads. In our application, a newly created fiber is immediately resumed, and never suspended. With a call to `UnYield`, the state changes to `Executing`. For a complete state diagramm, see Figure 3.

The `Yield` and `Unyield` methods were programmed in Delphi inline assembler for performance reasons. As the methods are normals procedure calls, most of the context, ie. the registers, need not be saved explicitly. As the thread where the fiber is executed in remains the same, this is possible. Additionally, the base and stack pointers of the fiber and the thread and the return address

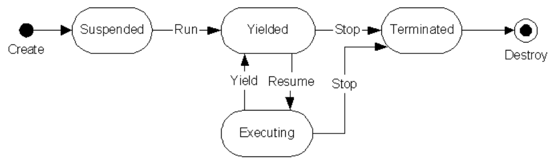


Figure 3. State Diagramm of an EMUfiber

must be stored. Hence, the fiber context is very small. The `Yield` method consists of about 20 assembler instructions, which is much faster than current thread implementations.

The scheduling strategy used is a simple first-in-first-out strategy. Besides the obvious performance advantage, we believe this strategy to be sufficient in our application environment, for the following reasons. First, almost all fibers deal with a client connection, and therefore have identical priorities. If there are many connections, then the probability is quite small that a fiber is scheduled again while its send buffer is not yet cleared. This normally only happens if the communication bandwidth is the bottleneck. In the case that the communication bandwidth is sufficient, then fibers may still experience differing times to clear their buffers because some of them are subject to bandwidth bottlenecks on the side of their client (or in the network). Then a further increase in throughput might be possible if fibers whose connections experience a higher bandwidth get higher priority. As, however, the penalty for scheduling a fiber and immediately yielding it again is quite small, we believe this increase to be small.

The detailed description of the procedures implemented, together with the assembler code listing, can be found in [5], or obtained by contacting the authors.

3 Integration into an http Server

The Indy (Internet direct) library [4] provides implementations for a number of internet protocols, among them http, both for the server and the client side. It is open source and included in Delphi. As Delphi is available for Linux under the name Kylix, our code can also be used for that operating system after a small number of operating-system specific changes.

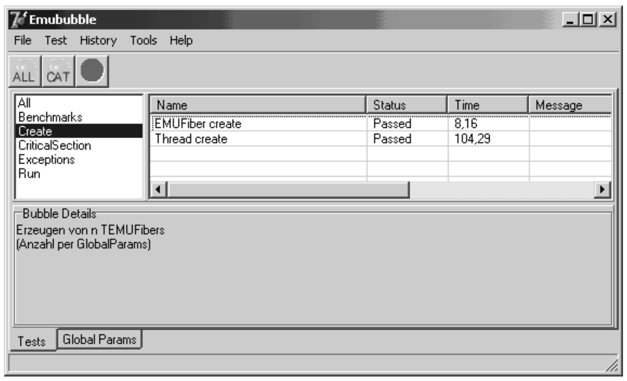
The http server in Indy uses two types of threads: a so-called ListenerThread waits for incoming connections on port 80, and a so-called PeerThread is created for each new connection. The latter handles the communication between the server and the client, and is terminated upon the end of the connection.

The exchange of the threads for our fibers is simplified by the fact that the Indy library does not use the Delphi threads directly but uses a derivation. By changing only a few code lines, our fibers can be used. What remains to be done is the insertion of `Yield` statements at the appropriate places. In fact, only one statement must be added, namely during the write of a data stream. If the data does

```
Creating 1000 EMUFibers (Stack size:  4000
byte)...
1000 EMUFibers created in 8,58 ms.
Destroying 1000 EMUFibers...
1000 EMUFibers destroxed in 9,02 ms.

Creating 1000 WinFibers...
1000 WinFibers created in 12,96 ms.
Destroying 1000 WinFibers...
1000 WinFibers destroxed in 31,91 ms.
```

Figure 4. Comparing EMUfibers and Windows fibers



All
Benchmarks
Create
CriticalSection
Exceptions
Run

Name	Status	Time	Message
EMUFiber create	Passed	8,16	
Thread create	Passed	104,29	

Bubble Details
Erzeugen von n TEMUFibers
(Anzahl per GlobalParams)

Figure 5. Benchmarking EMUfibers against threads

not fit into the buffer, then the context is switched by calling `Yield`. Then the socket responsible for sending the data gets time to transmit the buffer content, until the fiber is scheduled again.

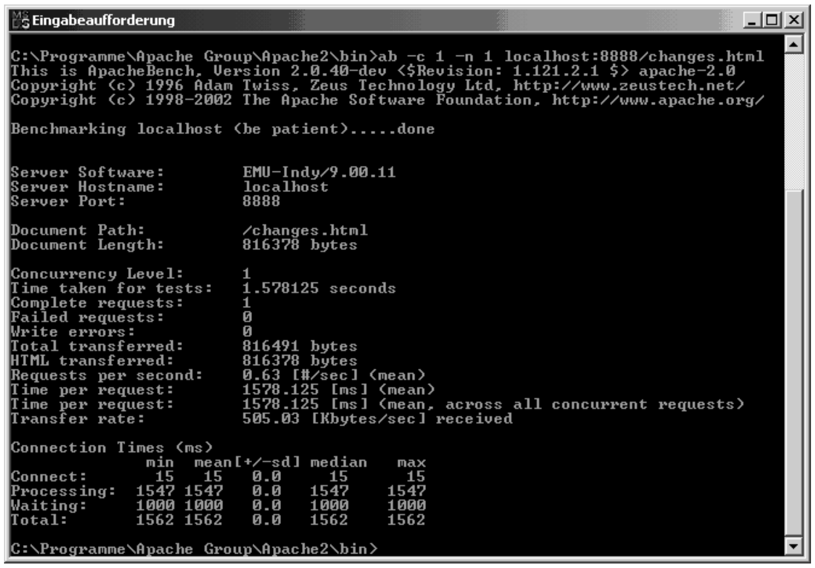
An additional, although minor, problem is that the ListenerThread, which is now a fiber, blocks until a new connection is recognized, and hence no other fibers could be executed. A simple solution is to use two scheduler threads, of which one blocks with the Listener fiber, and the other executes the remaining fibers.

We have created a web server executable, that listens on two ports. Port 80 is used for the http server with our fibers, port 8080 is used for the original http server with threads. The server creates a simple graphical user interface for display in the client browser.

4 Experiments

We first compare the performance of our fiber implementation against the original Windows fibers with the help of a simple test programm that only creates and destroys fibers. We find that creation and destruction of 1000 EMUfibers takes only 66.2% and 28.3%, respectively, of the time that Windows fibers need, see Figure 4.

Then, we compare EMUfibers against threads. There, creation of 1000 EMUthreads needs only 7.8% of the time for threads, see Figure 5.



```
C:\Programme\Apache Group\Apache2\bin>ab -c 1 -n 1 localhost:8888/changes.html
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.121.2.1 $> apache-2.0
Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Copyright (c) 1998-2002 The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient).....done


Server Software:        EMU-Indy/9.00.11
Server Hostname:        localhost
Server Port:            8888

Document Path:          /changes.html
Document Length:        816378 bytes

Concurrency Level:      1
Time taken for tests:   1.578125 seconds
Complete requests:      1
Failed requests:        0
Write errors:           0
Total transferred:      816491 bytes
HTML transferred:       816378 bytes
Requests per second:    0.63 [#/sec] (mean)
Time per request:       1578.125 [ms] (mean)
Time per request:       1578.125 [ms] (mean, across all concurrent requests)
Transfer rate:          505.03 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       15   15   0.0     15      15
Processing:  1547 1547   0.0   1547    1547
Waiting:     1000 1000   0.0   1000    1000
Total:       1562 1562   0.0   1562    1562

C:\Programme\Apache Group\Apache2\bin>
```

Figure 6. Example output of the Apache benchmark

Now, we compare a http server based on our fiber implementation with the original multithreaded http server of the Indy suite. The platform uses an AMD K6-III processor at 400 MHz with 512 MByte memory, running under the Windows NT4 Server operating system. We use the *Apache Benchmark* `ab` [6]. For our tests, we use two files of sizes 10,541 bytes (`Haewelmann.html`) and 4 bytes (`echo`, generated by the server's echo function) respectively, to reflect the varying file sizes on the web. The first file reflects a typical html page, while the second reflects the answer to a generated request. Note that the transferred amount of data is about 100 bytes more because of the `http` protocol overhead. We also experimented with a file of about 800 KBytes (`changes.html`), but there the data transfer times dominate. Consequently, the differences between the two http servers diminish.

For each server and each file, we performed the following tests: Either a local client or a 10 MBit network connection was used. Concurrency levels of 1, 10, and 50 were used. In the Apache Benchmark, the concurrency level denotes the number of requests that are posed in a very short time before a response is expected. This is used to simulate multiple clients. Figure 6 depicts a sample output of `ab`.

If a local client is used, the communication bandwidth is not the limit. Hence, if the concurrency is high enough, the server CPU will be fully utilized. In this case, we compare the throughputs achieved by the two servers. If a network connection is used, it will saturate when the concurrency is high enough. In this case we compare the CPU utilizations, and in addition the throughput to check whether a lower CPU utilization sacrifices performance.

For the situation of a local client, we find that our throughput is 26% and 50% higher, respectively, for the two files, if the concurrency is at least 5, to achieve 100% CPU utilization; see Figure 8.

For the situation of a network connection, we find that our CPU utilization is 40% and 80%, respectively, of the other http server's, if the concurrency is at least 5, to achieve 100% network utilization; see Figure 9.

At the same time we find that our throughput is identical for the larger file and about 30% higher for the short file; see Figure 10. This indicates that the original http server was not able to fully utilize the network bandwidth in this case due to local performance problems (its CPU utilization is about 90%). It hence seems that our implementation may be especially suited for large numbers of short requests.

Last, we compare our implementation against the Apache web server 2.0.45, which is one of the most popular open source products for web servers. As server platform, we use an 1.3 GHz Celeron processor with 1 GByte memory, running under the Windows 2000 professional operating system. We apply the apache benchmark with the file `haewelmann.html` and concurrency[2] 5 for a local client, because this file represents a typical web page size, and the differences between the servers should be most prominent without the influence of network bandwidth.

Figure 7 depicts that the EMUfiber-based web server achieves a throughput which is slightly higher than the Apache web server. The original Indy web server achieves about 3,000 KByte/sec, as to be expected in similarity to Figure 8.

Note that this comparison still is somewhat unfair towards our implementation, as the Apache web server is a product designed solely for that purpose, i.e. a stand-alone standard web server, while the Indy suite contains a custom http server, i.e. an http server that is easily embeddable into a larger application (see also section 5 for an example).

[2]For Windows 2000 professional, there seems to be a limitation of the allowed number of requests to 5. Higher concurrency values lead to request rejection.

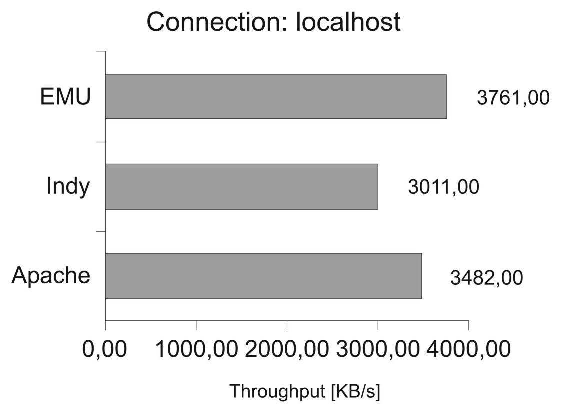


Figure 7. Comparison with Apache web server

Yet, in the latter case, efficiency of the http server software may be more important because the hardware for an application containing a web server often is less powerful than the hardware solely bought for running a web server.

5 Conclusions

We presented a re-implementation of Windows fibers, usable under the Windows and under the Linux operating systems. We presented how to easily modify the open source http server library `Indy` to replace threads for our fibers. We benchmarked our server against the original Indy server. We found that it achieves lower CPU utilization in a setting where the network connection is the bottleneck, and that it achieves higher throughput in a setting where the processing power is the bottleneck. For short requests, it even achieves both. We also compared our solution to the apache web server and achieved comparable throughput.

While our solution is not the method of choice for use on a large scale, it is well suited for use in a software library setting. In this sense it resembles the use of assembler: use it in libraries where performance counts.

The proposed solution is to be included as an optional module in the next revision of the Indy suite, and on that basis, in IntraWeb [7], a commercial software package for the construction of web-based applications.

References

[1] A. Grävinghoff, On the Realization of Fine-Grained Multithreading in Software (PdD Thesis) (Hagen: FernUniversität, 2002).

[2] A. Grävinghoff and J. Keller, Fine-Grained Multithreading on the Cray T3E, in *High-Performance Computing in Science and Engineering*, (Berlin: Springer-Verlag, 2000) 447-456.

[3] C.D. Marlin, Coroutines, (Berlin: Springer-Verlag, 1980).

[4] Indy Pit Crew, Indy, `http://www.indyproject.org/indy.html`, 2003.

[5] O. Monien, Implementation of an Http Server with Increased Latency Tolerance on IA32 Plattform (Master's Thesis in German), (Hagen: FernUniversität, 2003).

[6] Apache Software Foundation, Apache http Server Project, `http://httpd.apache.org`, 2003.

[7] Atozed Software Ltd., IntraWeb, `http://www.atozedsoftware.com/intraweb`, 2003.

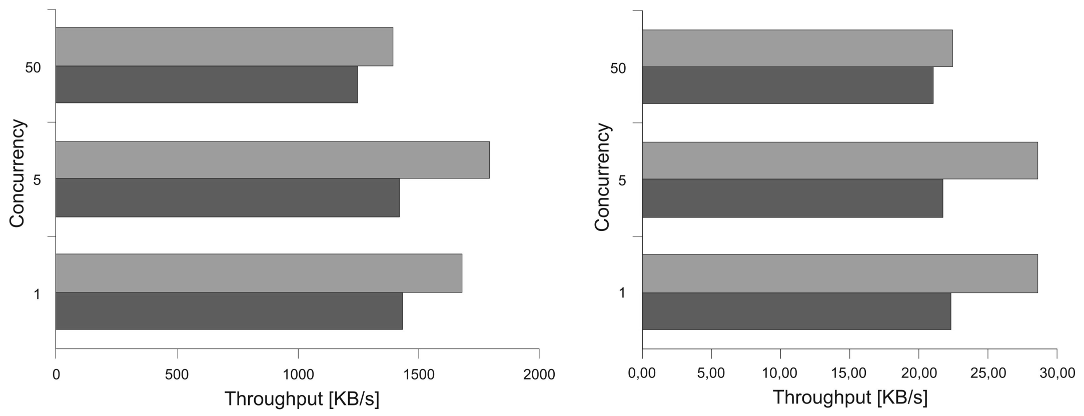


Figure 8. Throughput for `Haewelmann` (left) and `echo` (right) with a local client

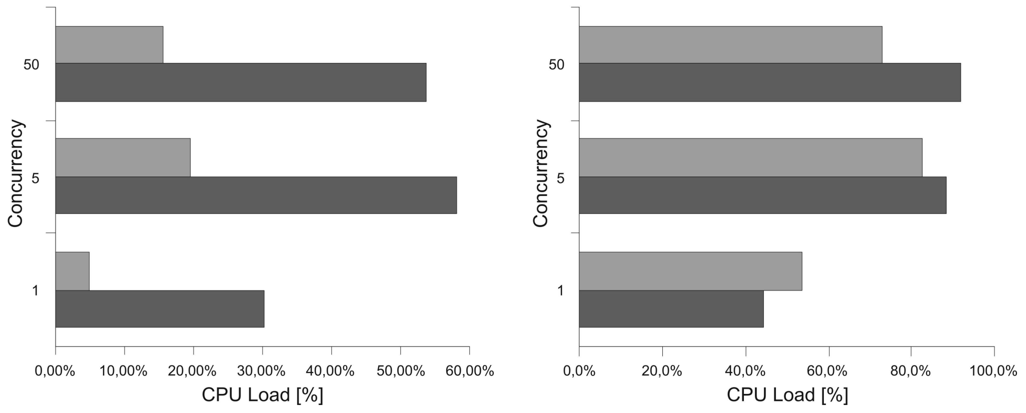


Figure 9. CPU load for `Haewelmann` (left) and `echo` (right) over a 10MBit connection

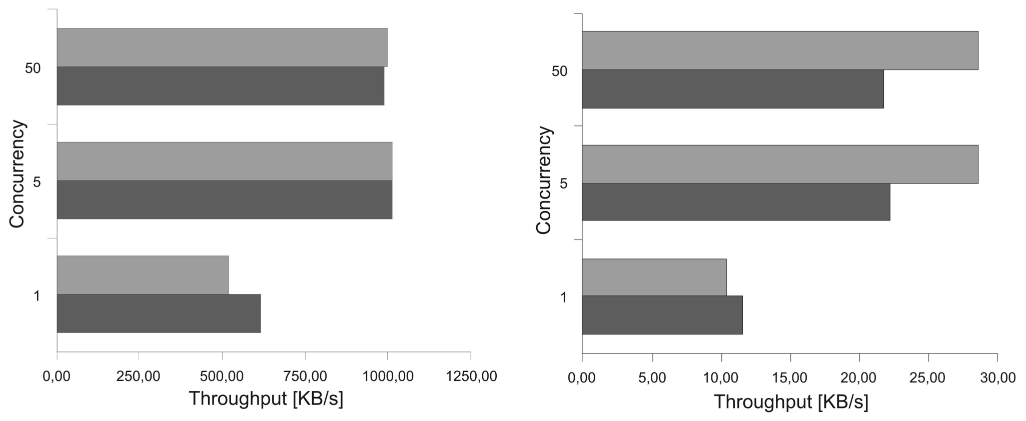


Figure 10. Throughput for `Haewelmann` (left) and `echo` (right) over a 10MBit connection