3.2 Software

3.2.1 Linux Kernel

Linux kernel was chosen as the OS for the system. This is the pure core Linux OS, in comparison to enriched Linux distributions (like *Ubuntu*, *Debian*, *openSUSE*, *Fedora* and many more), which contain additional libraries, applications and configuration. The Linux kernel version used in this project contains further additions and bug fixes specific to the *MicroBlaze* architecture by *Xilinx*.

To enable correct recognition of the latest *MicroBlaze* processor versions with enabled Processor Version Register (PVR), it might be necessary to apply a patch included in the appendix of this report (B.1.2) to the Linux kernel sources. The patch was extracted from the *PetaLogix* Linux kernel fork.

Building Linux kernel sources to an executable image file requires a set of tools (called toolchain) for compiling source code files and linking binary output to one single image. *Xilinx* provides toolchains, based on the widely used GNU Compiler Collection (GCC) and *binutils*, for cross compiling from Linux x86 and x86-64 architectures to *MicrobBlaze* systems as target architecture. Depending on the used development system, the corresponding toolchain has to be selected.

Configuration

Linux kernel consists of many optional sub-parts for target architectures, device drivers, special features, etc. Which of these parts are compiled and linked into the Linux kernel binary image needs to be configured in the `.config` file in the Linux kernel root directory. This file is "the configuration blueprint for building a Linux kernel image" [10][sec. 4.3.1] and contains all (required) settings.

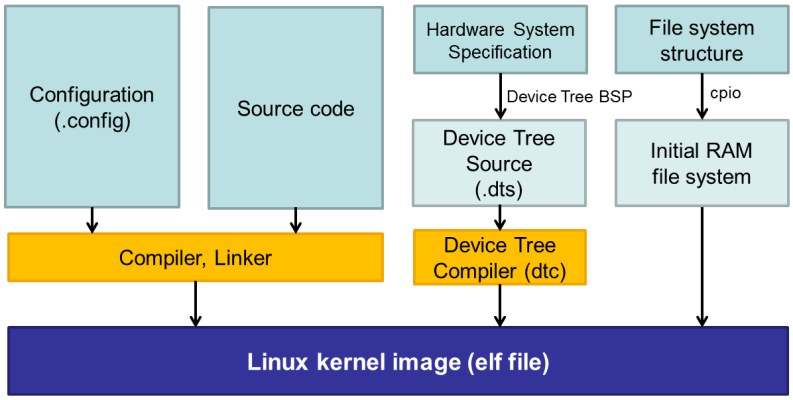

Figure 3.1.: Components of a Linux kernel build.

The *MicroBlaze* processor can be configured with different feature sets (multiplier, barrel shifter, etc.). Therefore the GCC compiler needs to be parameterized for matching the provided features of the target system [27][sec. "Kernel Configuration Details"]. These need to be specified in the `XILINX_MICROBLAZE0_*` settings inside of the `.config` file. The `KERNEL_BASE_ADDR` is also an imported setting which must match the base address of the system's main memory.

The `XILINX_LL_TEMAC` driver is used for the Ethernet interface.

Another part of system configuration is the *Device Tree*. It is an abstraction layer for accessing hardware information, to avoid compiling everything into architecture specific assembler code [8]. It is accessed by the Linux kernel during the boot process for configuring itself and on lookup of hardware information. Therefore the *Device Tree Source (dts)* file is compiled by the *Device Tree Compiler (dtc)* during the Linux kernel build process to an *Device Tree Blob (dtb)* and linked into the final Linux kernel image. It can be generated using the *Device Tree Generator*, a Tool command language (TCL) script reading a system specification file generated by XPS.

3.2.2 The File System

It is possible to use the Linux kernel without a file system, but the makes little sense in practical use. Therefore an *Initial RAM Disk (initrd)* image is mounted as *root file system (rootfs)* [10][sec. 6.1]. This is a file system packed into a *cpio* archive and linked into the Linux kernel image. It is unpacked completely into the main memory during kernel boot process.

3.3 Deployment

The generated *bitstream* of the hardware system representing the SoC, needs to be programmed into the FPGA on every power-on. This is required, because the configuration of the FPGA being set by the *bitstream* is volatile. Programming the *bitstream* is straight forward and can be accomplished using the *Xilinx iMPACT* tool, which is part of the *Xilinx ISE Design Suite*.

When the Linux kernel build succeeded, the resulting *Executable and Linkable Format (ELF)* file can be found at `arch/microblaze/boot/simpleImage.<dts-name>`. This file needs to be loaded into the FPGA using *Xilinx Microprocessor Debug Engine (XMD)*, which is also part of the *Xilinx ISE Design Suite*.

Additionally the path to the standard libraries on the system needs to be set through the `--sysroot` parameter. These libraries are part of the tool chain for *MicroBlaze* systems provided by *Xilinx*.

By specifying the `--static` parameter all referenced libraries are build into the resulting binary. Therefore the binary has less dependencies for execution.

Combining it all together, the `--with-cc-opt` parameter needs to be set to the following value:

```
--with-cc-opt="-mxl-multiply-high -mno-xl-soft-mul -mno-xl-soft-div ↩
-mxl-barrel-shift -mxl-pattern-compare -mcpu=v8.30.a --static ↩
--sysroot=/home/peschuster/project/microblaze-unknown-linux-gnu/ ↩
microblaze-unknown-linux-gnu/sys-root"
```

5.3.5 Memory Leaks

One problem that arose already during very early tests were memory leaks. It turned out that *nginx* allocated about 4,053.2 bytes of memory for each request. Assuming available memory of about 200 MB, *nginx* crashed the complete system after approximately 50,500 requests in total. Obviously this is an unacceptable behavior for a (web) server system.

requests	master / KB	worker / KB
0	3064	3204
1000	3064	7296
2000	3064	11256
3000	3064	15348
4000	3064	19440
5000	3064	23404
6000	3064	27496
7000	3064	31588
8000	3064	35552
9000	3064	39644
10000	3064	43736

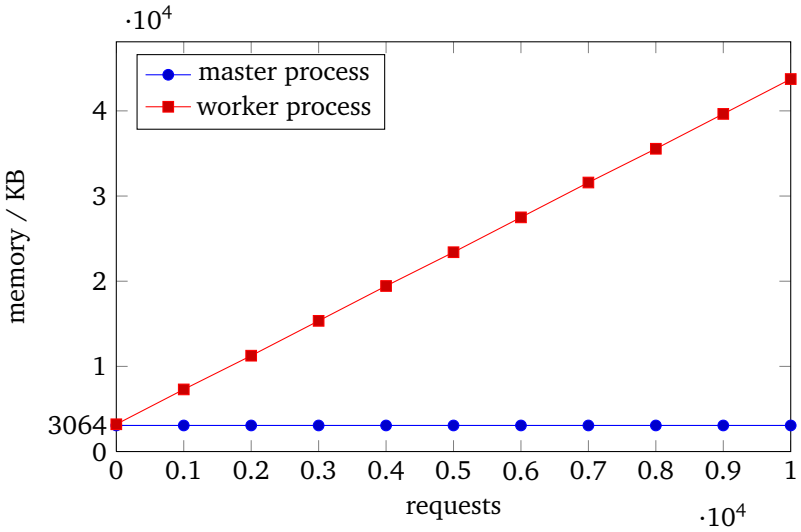

Figure 5.3.: Memory consumption of nginx processes over total requests.

Investigations

The described behavior of *nginx* could not be replicated on a standard x86 server system running *Ubuntu Linux 12.04*. That means *nginx* in general should work correct, using just as much memory as required and releasing unused memory to the OS. But this does not work properly on the *MicroBlaze* system.

With activated `debug` log. *nginx* writes some information about inner workings to the error log. The debug log can be activated with the following option in the *nginx* runtime configuration file:

```
error_log logs/error.log debug;
```

There must be only one `error_log` line in the *nginx* configuration file, but the option specifying the severity level is inclusive for all upper levels.

The following table shows all memory related debug messages as found in the error log for a single request to a static file on the *MicroBlaze* system running an *nginx* instance, which is configured as described in the previous section (sec. 5.3):

Log entry	ptr	allocated	freed
*1 malloc: 10080890:644	10080890	644	
*1 malloc: 10080B18:1024	10080B18	1024	
*1 posix_memalign: 1007B5B0:4096 @16	1007B5B0	4096	
*1 free: 1007B5B0, unused: 2079	1007B5B0		4096
*1 free: 10080890	10080890		644
*1 free: 10080B18	10080B18		1024

Table 5.1.: Memory management related debug messages of a single request.

All pointers to allocated memory for one request are passed to the `free(..)` function of the *C Standard Library* (*libc*) and therefore should be released to the system. But performance and load tests on the system proved a different behavior: The *nginx* worker process consumes more memory for each request with a linear relation to the number of handled requests (see figure and table 5.3).

Therefore it can be assumed that the error causing this misbehavior is not located in *nginx* itself, but in the underlying system layers. This implies that memory management of the Linux kernel or the *libc* port to the *MicroBlaze* architecture are broken in the described respect. While being a strong allegation, especially because fully analyzing the problem on this wide dimensions was beyond the scope of this bachelor thesis, there are a few points promoting this theory:

Xilinx support answer `AR #12421` states that memory management (especially the function `free(..)`) is "very system-specific" and not fully supported and implemented for *MicroBlaze* processors [12]. The answer was published in September 2010 and its validity is explicitly limited to versions of the *MicroBlaze* processors without a hardware memory management unit. Therefore it should not apply to the used version of the *MicroBlaze* processor and *C Standard Library*, but it shows that these memory management functions were added to the toolchain and supporting environment only recently and might not be as stable as other, more major parts.

A possible explanation why the described faulty behavior is only visible when using *nginx*, is the unusual way *nginx* deals with memory. Memory management is done by *nginx*'s pool allocator. This could be seen as an abstraction to the memory allocation mechanisms provided by the OS. When another part of *nginx* requires dynamically allocated memory, it requests it from the pool allocator, which itself requests a larger chunk of memory (called "pool") from the OS and distributes small blocks of the pool on requests from other parts of *nginx*. When free blocks of a pool are exhausted, but none are used anymore, the complete pool is returned to the OS. *nginx* uses this design to minimize system calls and reduce expensive requests for memory allocation by the hardware memory management unit [2]. One consequence of this design is that *nginx* does not reuse once allocated memory, but just allocates new memory blocks when required and releases them to the system upon finished operations. This is by design and beneficial for speed and efficiency, but comes in unfavorable, when memory management of the system (OS and processor) might not work properly.

A First Workaround

It was not possible to fix the root cause of the memory leaks during this bachelor thesis. To circumvent the described arising problems, another solution needed to be found. Otherwise practical usage of the system and extensive performance benchmarks would not be possible.

A workaround to circumvent complete system crashes during tests due to exhausted memory is to restart the *nginx* worker process on low remaining system memory.

Taking this usage pattern of websites into consideration, tests were performed using static files with two sizes: one file having exactly 10 KB (10.240 bytes) from now on referred to as "10K.html" and one very minimal HTML page having just 354 bytes ("index.html"). The major difference between these two files from a "network perspective" is that the small file can be transmitted in two TCP packets, one is used for the HTTP header and one for the actual data (see section 5.4), whereas the 10 KB file needs to be split up in 9 TCP packets due to the default MTU size of 1500 bytes. A third test-run was performed requesting a file that does not exist. This results in a HTTP response with code "404 – Not Found". It is a special case, because the response is transmitted in a single TCP packet and no file needs to be read from the file system.

Figure 6.3 shows request and response rates for all three test cases with an increasing connection rate (20 conn./s to 120 conn./s, step size: 10). On each connection three HTTP requests were made to the respective file.

On every connection rate level 3,000 connections were opened to obtain reliable results. Therefore each test run included 33,000 connections and up to 99,000 requests.

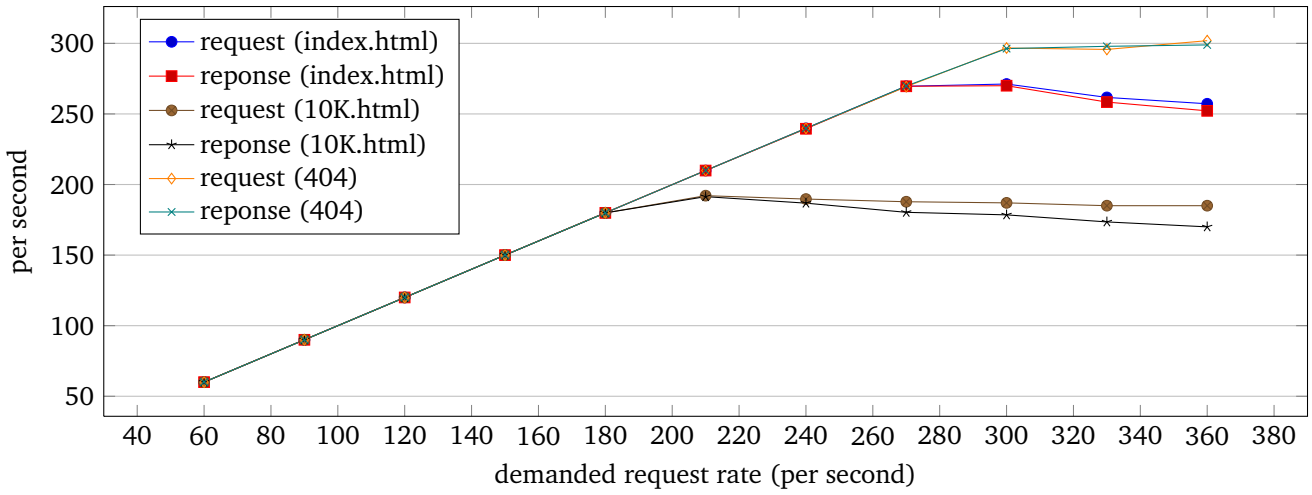

Figure 6.3.: Request and response rates for the different test runs.

The test result shows that at a certain request rate, the response rates levels off. For the two test runs requesting files, response rates even decrease slightly. At this points the server becomes saturated and the number of requests oversteps its actual capacity. These points are namely at about 300 req./s for the very small response, at about 270 req./s for the "index.html" file and at about 190 req./s for the "10K.html" file.

The difference between request and response rate origins in failed requests. The following figure shows the corresponding error rate. It increases with over-saturating the system.

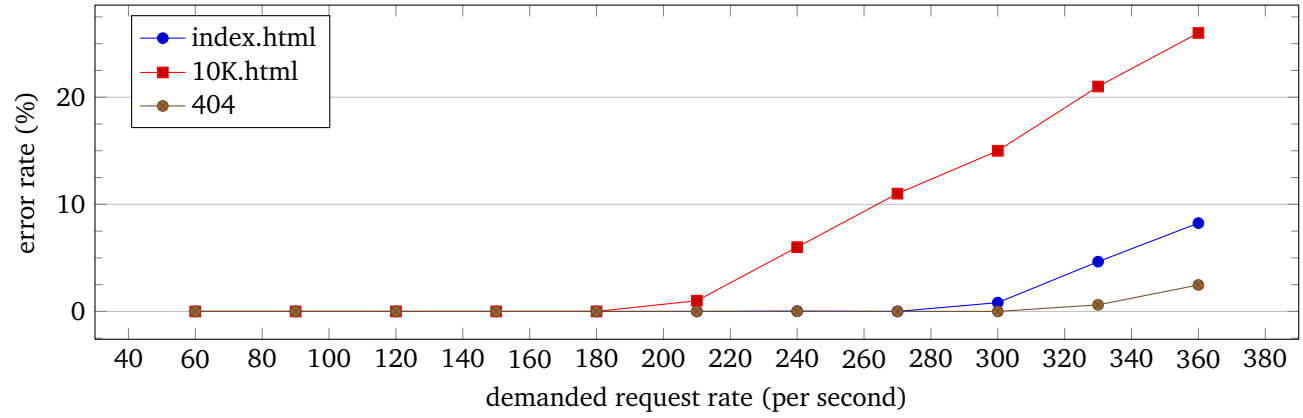

Figure 6.4.: Error rate (failed requests) for the different test runs.

The response **rate** decreases only slowly with growing saturation, but the average response **time** increases from 5 ms to about 500-700 ms:

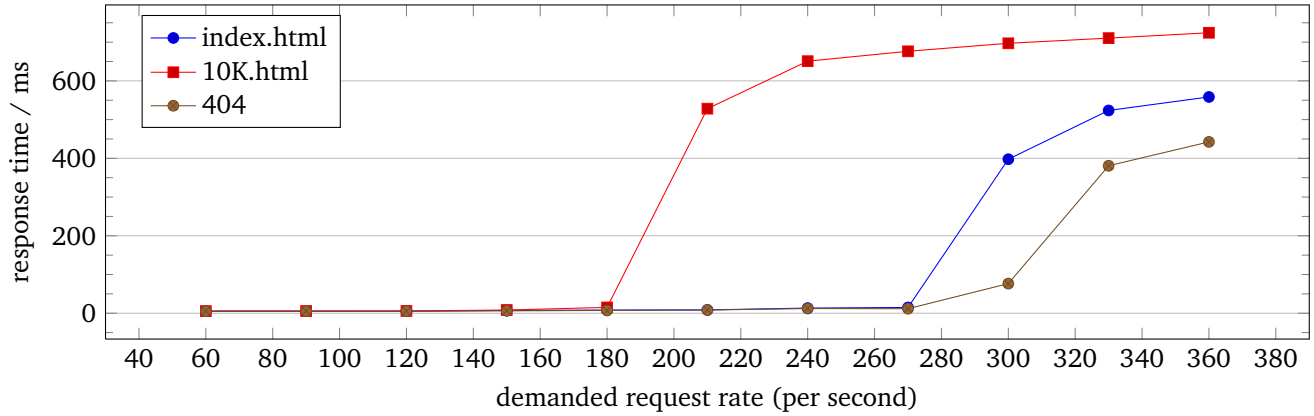

Figure 6.5.: Response time for the different test runs.

The reason for decreasing slopes in response times might be a clogged backlog for incoming connections, resulting in an early, and therefore cheap rejection of further incoming requests.

The difference in request and response rates the system can handle is also visible in **CPU utilization**. Main responsibility of *nginx* for serving requests is to handle the HTTP part of the process. This includes parsing request headers, generating responses and delivering static files. The OS on the other hand does the "translation" to and from TCP and handles all work on lower network layers (see also figure 6.2). An increased payload size of single responses entails therefore a shift in processing time from *nginx* ("user space") towards the operations inside the Linux kernel. Figure 6.6 shows exactly this correlation.

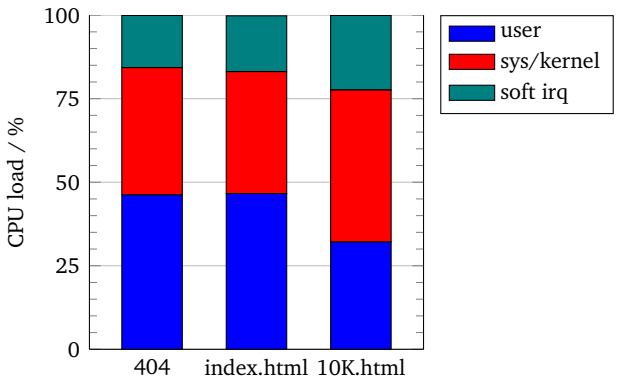

Figure 6.6.: CPU load distribution for different request types.

Network **throughput** of the tests follows naturally the trend of response rates. For tests requesting the "index.html" file and those resulting in "HTTP 404 – Not Found" responses, the shown throughput is very low, because *httperf* calculates throughput using the formula $\frac{[\text{size}] \cdot [\text{total repsonses}]}{[\text{test time}]}$. Therefore very small response sizes yield a low average throughput.

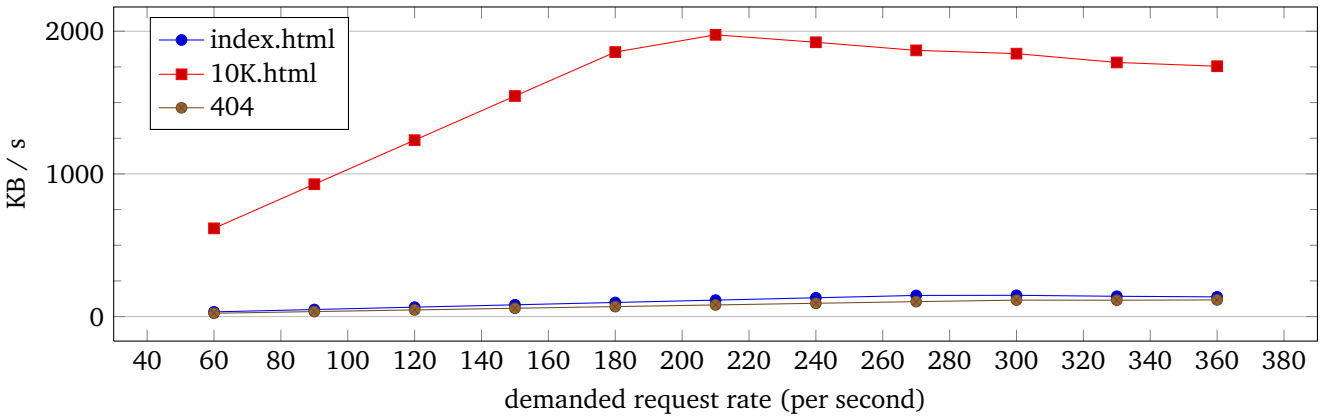

Figure 6.7.: Throughput for the different test runs.

6.3.1 Dedicated Network Throughput Tests

To test the maximal network throughput the server can provide, a test needs to be designed focusing on just this parameter. This approach prevents other parts of the system from getting into the way. A constant high exchange of TCP packets can be achieved by transferring very few, but large files over the network. Therefore a large binary file having a size of 33.4 MB was used for throughput tests.

Figure 6.8 shows the network throughput for requests to this file with an increasing connection rate and two HTTP requests on each connection. The system under test goes into saturation at about 61 Megabits/sec. This is equivalent to 6.1 % of the available network bandwidth, providing a maximum of 1 Gigabit/sec. It is important to note that the number of 61 Megabits/sec. is not equivalent to the number of actually transferred bits over the network, but only includes the received "usable" data. Transferred connection setup and close, acknowledgment and TCP, IP and Ethernet header messages are not included in this number.

The figure also shows the total CPU utilization of the tested system.

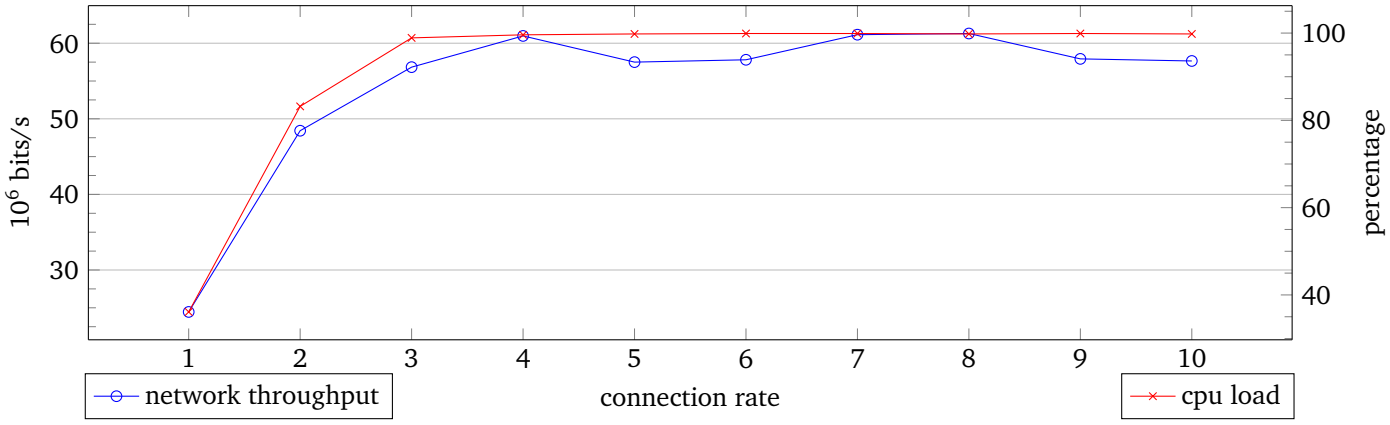

Figure 6.8.: Network throughput and CPU load.

Figure 6.9 shows the CPU utilization by time spent in the different types of the system. It shows that most time is spent in kernel space and handling interrupt requests by the network driver and sub system (soft irq). nginx (user space) takes only very low CPU utilization and is not the limiting factor in this test. This conforms with test results of the previous tests on smaller files.

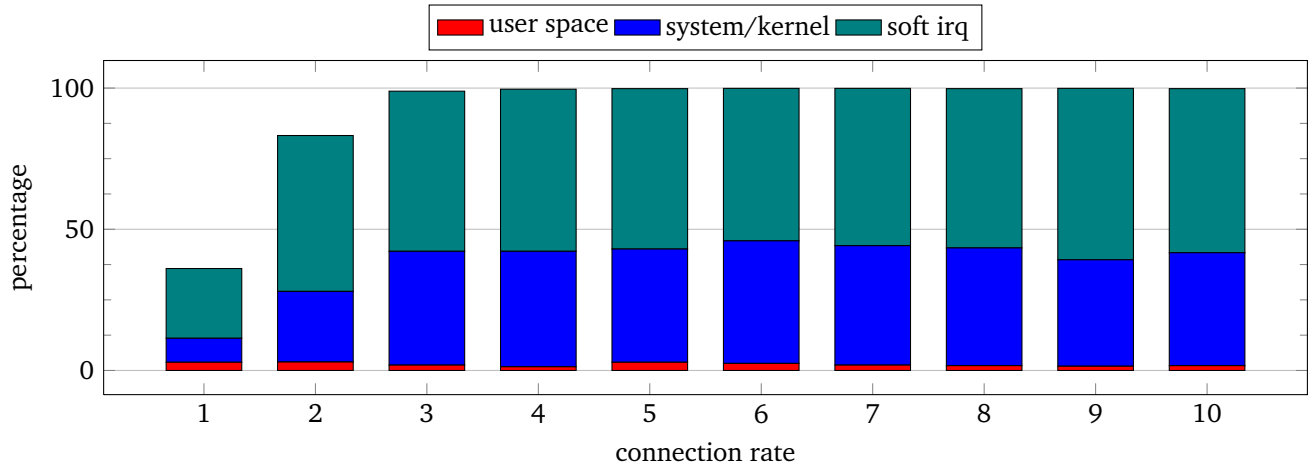

Figure 6.9.: CPU utilization by category.

To proof that obtained performance limits were not limits of the client executing the tests or the network infrastructure, the same test was also executed against a "reference system" running a *Microsoft IIS 7.5* web server. Figure 6.10 shows the results. Measured throughput (averaged per request) had a maximum at about 515.7 Megabits/sec which outplays the results of the *MicroBlaze* system by more than a factor of eight.

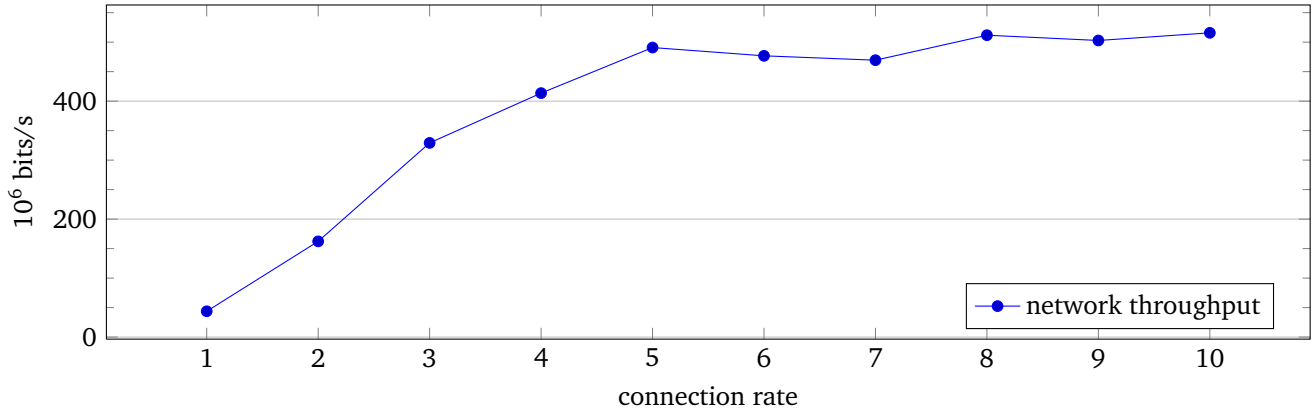

Figure 6.10.: Network throughput of an AMD Phenom II X4 processor at 3.40 GHz running Microsoft IIS 7.5 on Windows 7.

6.4 Discussion of Results

Obtained performance results show that the designed System on Chip with custom configured and compiled Linux kernel is capable of running a web server serving up to about 300 requests per second and 100 concurrent connections. Compared to the low performance indicator of 48.94 BogoMIPS[8], which is less than a $\frac{1}{130}$ of the value Linux kernel reports for a virtualized *AMD Phenom II X4* processor at 3.40 GHz running on two cores[9], this is not to bad. But the capacity of the server would not be sufficient for running it as a public web server connected to

[8] "The number of million times per second a processor can do absolutely nothing." (untraceable source: Eric S Raymond, Geoff Mackenzie – `http://www.tldp.org/HOWTO/html_single/BogoMips/`)

[9] 6414.98 BogoMips