HAL
open science

An Analysis of Web Servers Architectures Performances on Commodity Multicores

Sylvain Genevès

To cite this version:

Sylvain Genevès. An Analysis of Web Servers Architectures Performances on Commodity Multicores. [Research Report] 2012. hal-00674475v2

HAL Id: hal-00674475

https://inria.hal.science/hal-00674475v2

Submitted on 26 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An Analysis of Web Servers Architectures Performances on Commodity Multicores

Sylvain Genevès
Grenoble University
LIG
Grenoble, France
sylvain.geneves@inria.fr

***Abstract*—We study the impact of concurrent programming models on multicore performances of Web servers. More precisely, we consider three implementations of servers, each being representative of a particular model: Knot (thread-based), μserver (event-driven), Watpipe (stage-based). Our experiments show that memory access costs increase with the number of cores. We also show that at 8 cores we reach a point where the memory is fully saturated, leading to all Web server implementations having the same performance. Using fine-grain profiling, we are able to pinpoint the cause of this issue as a hardware bottleneck: the saturation of the address bus. Further memory benchmarking on a 24-cores machine show that a memory-related scalability issue is still present beyond this point.**

***Keywords*-Multicore architectures, Web servers, performance, scalability, parallelism, memory**

I. Introduction

As traditional monocore architectures reach electrical and power limits, multicore processors arise as a solution to continue to gain performance. The current hardware trend is to add more and more cores on a chip. Multicore is now mainstream on general-purpose processors as well as on specialized hardware, thus we believe that every developer should feel concerned about parallelism.

In this context, this paper focuses on the increasingly important issue of data server scalability. As the Internet is growing wider and wider, popular websites must now handle an incredible load. For example, Wikipedia's website has to deal with periods of up to 90 000 requests/second for several hours, every day[1]. We focus our study on servers serving static content. This, however, does not limit the scope of this paper only to static Web sites. Indeed, high-traffic dynamic Web sites wide use of proxies and caches make studies of servers serving static content relevant. As an example, when at peak, Facebook serves over 1,000,000 images per second[1].

Unfortunately, it is well-known that Web server throughput does not scale well with the number of cores. With the spreading of multicores technologies, this is becoming an increasing economic concern for building efficient data centers.

[1]http://en.wikipedia.org/wiki/Wikipedia:Statistics

Our overall goal is to search for the most efficient ways to serve many concurrent client requests. There exist several concurrent programming models to develop software leveraging multicores, each of them being of very different nature. This paper aims at comparing the different programming models from the point of view of scalability. For this purpose we focus on the three mainly used models in the data server context: the thread-based model, the event-driven model and a stage-based hybrid model.

Contributions

The main contribution of this paper is a thorough comparative analysis of the scalability of the different programming models and a report on extensive experiments that provide clear – and sometimes surprising – insights about the identified bottlenecks. Those insights include the following:

- Unexpectedly, the same scalability issue is observed on all the tested Web servers once a certain threshold on the number of cores is reached. Profiling shows that the encountered bottleneck is memory-related, indicating that a solution residing in hardware memory management should be looked for, independently from any chosen programming model.
- Fine-grain profiling helped us to pinpoint the root of the encountered hardware issues. The cause lies in the cache coherence protocol implementation that saturates the address bus.
- A direct approach to reduce memory sharing between cores (NCOPY) shows an improvement of 13% at 4 cores for event-driven and stage-based implementations. This is however not enough to solve our scalability concern, as performances at 8 cores are not improved.
- Finally, our experiments show that the impact of system-wide tuning prior to Web server tuning is non-negligible, as Web servers cannot use 100% of each available core's computational power with standard system tunings.

Outline

The paper is organized as follows. We describe the programming models and the reference implementations we choose for our analysis in Section II. Then in Section III we

describe our experimental setup including preliminary work needed for a fair analysis. Section IV comments analysis results. We then discuss related work in Section V before concluding in Section VI.

II. Parallel programming models

In this section we describe the three commonly used models to program data server applications that we consider in our analysis: threads, events and stages. We also introduce the corresponding implementations that we use in our experiments, as well as the modifications we brought to them in order to bring each model to its best potential on multicores, and thus allow for a fair comparison.

A. Threads

The most direct way to program a parallel server is to use one thread per connection and blocking I/O. Throughout this paper we use the common definition of a thread, as seen in standards such as POSIX: a schedulable entity with an automated stack management. The thread-per-connection model usually uses a pool of threads, thus avoiding thread creation/destruction costs with each new connection. The widely used Apache[2] Web server uses this model. However, Apache supports a lot of features, such as virtual hosts, which could incur additional costs which are irrelevant in this study. For this reason, using this implementation for reprensenting the thread-per-connection model in our comparison would be unfair.

Knot: We choose the Knot Web server to represent the threaded server implementations. Our choice is directed by a previous study of this kind by Pariag *et al.* [3]. Knot is a fast, efficient C implementation built to prove the efficiency of the cooperative threading Capriccio [4] library. As Capriccio uses strictly one core, we modified Knot to leverage multicore architectures. To achieve that we made it link against the standard Linux threading package, the NPTL [5]. In the NPTL implementation, each thread is mapped on a kernel thread. This gives the kernel the power to decide how to map threads on cores, thus leveraging multicore architectures. The scheduling policy is also different from Capriccio, as it is preemptive. Though this was originally the purpose of the design of the Capriccio library, the NPTL can also handle thousands of threads, while leveraging multicores. We also added support for the zero-copy `sendfile` syscall. Since the contents of the files are not copied in userland, only the kernel memory is used, thus allowing to fully leverage the filesystem cache.

B. Events

In event-driven programming, the application only reacts to (mostly external) events, such as network I/O. An event loop is in charge of polling for new events and calling the appropriate application handlers. The common model is knowm as SPED[2]. Event handlers should not block in any way, since there is only one execution thread involved for the whole application. Instead of using blocking calls, one can postpone a computation until it can be executed in a non-blocking fashion. This results in event handlers calling one another asynchronously. Taken as it is, the SPED model cannot leverage multicores since there is only one thread for the whole application. The most direct approach to do so is to use NCopy. NCopy consists in duplicating the process onto each core of the machine. It can then leverage multicores, even if the main known drawback of this approach is the absence of data sharing between cores. To overcome this, the SYMPED[3] architecture allows to share the listening socket among SPED processes.

Userver: μserver is an efficient, event-driven Web server implementation. It is written in C, and has support for the SYMPED architecture. Thus, there is no need for external load balancing like in traditional NCopy, where each copy must listen to a different port. We choose to use the `sendfile` and `epoll` primitives, as those lead to better performances than their alternatives in our setup.

C. Stages

The stage-based model has been introduced a decade ago [6], [7]. In this model the application is composed of stages, that look like event handlers at first sight. The idea behind this is to give a set of execution resources to a stage. Those resources can be threads for example, or anything that the runtime controls. Thus there is a difference with event handlers: a single stage can be executed simultaneously by multiple threads. This allows for the runtime to finely control execution of an application, by dynamically reallocating the number of threads on a stage, or ease the use of admission control.

Watpipe: Watpipe [8] is a C++ implementation of a stage-based Web server, derived from μserver's code. Watpipe's stage system does not dynamically tune the server according to the load. However, since our workload does not evolve over time, and since in addition we finely tune each server there is no need for dynamic auto-tuning in our setting.

D. Remark

As we saw in this section, changing the concurrency programming model in a Web server has a big impact on code paths. Thus comparing different Web server implementations makes sense, as long as they provides a similar set of features. To the best of our knowledge no Web server implementation provide more than one concurrency model. Even if Apache's httpd provides different Multi-Processing

[2]SPED: Single Process Event-Driven
[3]SYmmetric Multiple Process Event-Driven

Modules (MPM), in the end they all use the thread-per-connection model[4]. The difference lies in that the worker MPM uses a hierarchy of threads and processes, whereas the prefork MPM uses only process and no threads.

III. Experimental setup

Each of the Web server implementations we chose thus leverages multicores. More precisely, the load is evenly balanced among cores for each of the three implementations. Thus we expect some sort of performance scalability with the number of cores, at least until the reach of maximum network bandwidth.

In order to make a fair comparison from which we can extract meaningful results, several additional aspects of crucial importance must be taken into account. We present them below, while detailing the hardware setup and the workload we consider.

A. Important aspects for a fair comparison

Same set of features for each implementation: Implementations must be comparable in terms of kind and quantity of computations associated with each request. Features in Apache such as virtual hosts, security checks or other hooks for optional plugins would make a performance comparison with μserver unfair. By choosing Knot, μserver and Watpipe we ensure a very similar set of features. To be even closer, we added support in Knot for the sendfile primitive that μserver and Watpipe use. Comparing these implementations against the same workload thus makes sense.

Reaching the optimal performance for each model: To enforce fairness in our comparison we must make sure that each server works at full capacity. To do this, we finely and independently tune each server until it reaches its maximum throughput. We detail the tuning procedure that we applied in III-D, after presenting the hardware setup and considered workload.

Reproducibility of results: For all the results presented in this paper, we make sure that the standard deviation is under 4% over at least 10 runs for Web server experiments, and 100 runs for memory benchmarks. All the discussions in this paper are based on performance differences significantly greater than observed standard deviation.

B. Hardware setup

We use the hardware from Grid5000 experimental platform. More precisely the cluster "chinqchint" from Lille's site is composed of 46 identical nodes. Each of those is a bi 4-cores Intel Xeon E5440 clocked at 2.83 GHz, totalling 8 cores per machine. Each core has a 32KB L1 instruction cache and a 32KB data cache. All L1 caches are private. A unified 6MB L2 cache is shared by two cores on the same package. Each node has 8GB of main memory. Each node is also equipped with 10Gb Myri-10G (10G-PCIE-8A-C) network card and two 1Gb Intel e1000, totalling 12Gb of network bandwith capacity. The network interrupts are affinitized to all the cores, as it is the most efficient way to balance the load among all the cores in our setup.

[4]there is an event MPM but it is still based on threads and does not uses the event-driven programming model.

Table I
System parameters used.

net.ipv4.tcp_tw_recycle	1
vm.max_map_count	786762
net.ipv4.tcp_fin_timeout	1
kernel.randomize_va_space	0
net.core.rmem_default	87380
fs.file-max	1048576
net.core.wmem_default	65536
kernel.pid_max	786762
net.core.rmem_max	4194304
net.ipv4.tcp_max_syn_backlog	200000
net.core.wmem_max	4194304
net.core.netdev_max_backlog	400000
net.ipv4.conf.all.arp_filter	1
net.core.somaxconn	200000
net.ipv4.conf.ethX.arp_filter	1
net.ipv4.ip_local_port_range	1024 65000
net.ipv4.tcp_syncookies	0
net.ipv4.tcp_rmem	4096 87380 4194304
net.ipv4.tcp_wmem	4096 65536 4194304

For all experiments, the machines run a 2.6.37 Linux kernel, with the 2.11.2 libc. For reproducibility purposes, Table I shows our main changes in the system configuration. We found those changes necessary to benchmark the system at full capacity, especially the ones related to the maximum number of connections, files and backlog queue size.

We use one of these nodes as a server, and 35 others as clients. All the nodes are connected via a 128-port 10Gb switch for their 10Gb interface, and gigabit switches for the other two network cards.

Each client machine runs 6 instances of the workload generator. In this way, we ensure that the clients are sufficiently provisioned not to constitute a bottleneck. We also check that a client is never saturated, would it be in terms of available file descriptors, network bandwidth, CPU or memory resources.

C. Workload

We use the same HTTP workload as in [3]. This workload is based on the static part of SpecWeb99 [9]. We note that static content is still a large part of each workload in the recent SpecWeb2009 [10]. Besides, using dynamic workloads would impose the use of external processes accessed by the server through CGI. The evaluation of the impact of those additional processes with the different programming models and their scheduling is beyond the scope of this paper. Our file distribution consists of 24480 files, totalling 3.3GB of data. For all experiments, this distribution is prefetched in RAM using the filesystem cache. This way we avoid limitations due to disk I/O. The access pattern to the files

recreate a Zipf distribution, we can note that this is also the case for the static part of SpecWeb2009.

As in [3], we use `httperf` [11] to replay HTTP access log files. This addresses two problems of the SpecWeb99 benchmark suite. First, SpecWeb99 uses a closed-loop injection scheme which leads to the injection being driven by the server at some point [12]. The second problem noticed in SpecWeb99 is the lack of think times. We thus use the partially-open loop injection scheme suggested in [12] to address the first problem, and the same HTTP trace files as in [3] to solve the second one, since they contain think times. Two types of think times are used: inactive off-period and active off-period. Inactive off-periods model the time taken by the user between two consecutive requests, taking into account actions such as reading the page, as well as other delays occurring between user requests. Active off-periods model the time needed by the browser for parsing an initial page, initiating requests to fetch objects embedded in this page, and waiting for the responses. For the sake of allowing comparisons with the study described in [3], we choose the same parameters: 3.0 seconds for inactive off-periods and 0.343 for active off-periods.

D. Tuning

In this study we carefully tune each server independently to reach its peak performance. As previously shown by [3], server tuning is crucial prior to any performance comparison. In the present work, we further observed that, even prior to server tuning, system tuning is critical in order to bring all the software stack at its full potential. Indeed, the Linux kernel is shipped with standard tunings that do not meet the requirements for Web server benchmarking. Our first experiments with standard backlog queues tunings show that the server cannot use all CPU resources when the load increases. The most dramatic case appears with Knot, where at 8 cores the average CPU usage is stuck at 33%, the throughput remaining flat as the load increases from 5000 connections/sec to 20000 connections/sec. We observe similar behaviour with μserver and Watpipe.

Once the system is properly tuned, the CPU and network load on the server increase along with the clients inputs, until we reach CPU saturation of the server. We are then able to finely tune each server independently, as done in [3]. Since our workload is partially-open loop, it consists in finding the appropriate number of concurrent requests for a server to reach its maximum throughput, while avoiding to trash the system. This tuning phase is done for 1, 2, 4 and 8 cores. In Knot, the number of concurrent connections is directly connected to the number of threads in the pool. For μserver, this corresponds to the size of the event queue. The corresponding Watpipe's parameter is the number of threads associated to read and write stages.

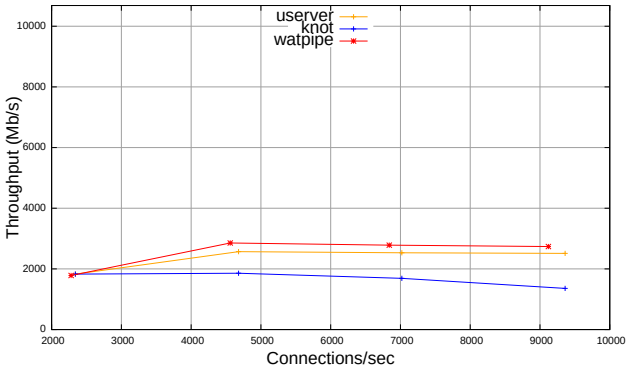

Figure 1. Performance comparison of the three servers at 1 core.

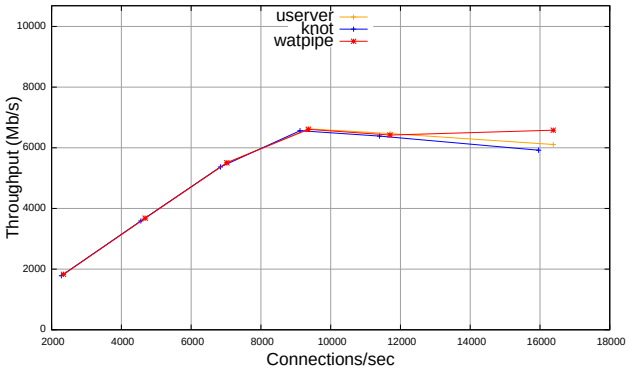

Figure 2. Performance comparison of the three servers at 8 cores.

IV. Results

This section comments our analysis results. For the sake of comparison fairness, only results pertaining to the best throughput of each server are discussed.

A. Performance comparison of programming models

Figure 1 shows our results using only 1 core on the server, reproducing Pariag *et al.* [3] work. Instead of the 18% maximum throughput difference observed in their setup, we observe a 35% maximum difference between Knot and Watpipe. The throughput observed here is globally much higher than the one observed in [3]. For example, when using a similar configuration (one core server), we obtain throughputs up to 2.8Gb/s instead of the barely reached 1.5 Gb/s in [3]. Nonetheless, if the numbers change because of the setup, our conclusions remain globally similar as those drawn by [3]. That is, Watpipe and μserver clearly outperform Knot's performances. The fact that we use the standard NPTL instead of the Capriccio library when running Knot is worth noticing because it results in a pure threaded execution.

Figure 2 presents the Web servers throughput when using 8 cores on the server machine. Only the best configurations for 8 cores of each server are considered. It is worth

noticing that the peak throughput point is simultaneously reached by all the three implementations, and at this point the throughputs coincide (6.6Gb/s). After this point, all the servers are saturated, and throughput decreases. The network is not the bottleneck here, as we can achieve throughputs up to 11Gb/s using network benchmarking tools such as netperf or iperf. This is surprising since it corresponds to an improvement of only 2.3 times of the performances observed using 1 core. As all of the three models fully leverage multicores architectures, one could expect a better scalability, at least until the network becomes a bottleneck. However, our experimental data indicate that the full network capacity is not reached and the servers expose poor scaling. At this stage we perform a further scalability analysis in order to better understand the performances of the three servers.

B. Performance scaling analysis

Figure 4 presents the scalability of the three server implementations. This plot is obtained using the highest throughput configuration for each server on each core configuration. The most obvious behaviour is a scalability issue arising at 4 cores. This issue is so important that all the implementations expose the same highest performance at 8 cores. We use the profiling tool OProfile [13] to pinpoint the possible causes of this issue. A good metric to start with is cycles per instructions, also known as CPI. Table II summarizes the CPI of each Web server for all core configurations. We observe that the number of cycles needed to execute one instruction doubles when switching from 1 to 8 cores. We eliminate the possibility of lock contention as a cause of the issue encountered at 8 cores, as the use of standard locks would put the competing process to sleep and prevent a full usage of the CPU (at least for the event-driven case, since there is only one process per core). As well, contention on spinlocks would cause a CPI decrease, as the number of lock acquiring instructions would dramatically increase. An increase of CPI usually corresponds to stalls. Using more advanced hardware performances counters[5] we can break down the increase of CPI. Figure 3 shows a decomposition of CPI for μserver in five categories. The cycles presented here are classified as:

- ”Branch”: which corresponds to stalls cycles due to Branch misprediction
- ”ROB”: which represents cycles where the pipeline blocked any further execution due to the Re-Order Buffer being full
- ”Load/Store”: which counts cycles Load/Store buffers are all being used
- ”RS”: which counts cycles the Reservation Station[6] was full

[5]The counters used here are named RESOURCE_STALLS.* for Intel Core2 Microarchitecture.

[6]Reservation stations permit the CPU to fetch and re-use a data value as soon as it has been computed, rather than waiting for it to be stored in a register and re-read. From http://en.wikipedia.org/wiki/Reservation_stations.

nb cores	CPI μserver	CPI Knot	CPI Watpipe
1	2.1	2.0	1.8
2	2.5	2.1	2.2
4	3.4	3.0	3.3
8	3.9	3.4	4.1

Table II
CPI EVOLUTION WITH THE NUMBER OF CORES.

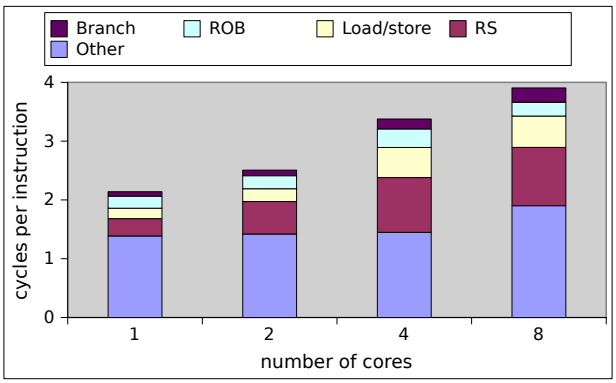

Figure 3. Cycles Per Instruction breakdown of μserver, by number of cores.

- ”Other”: which is the remaining cycles. It corresponds both to cycles spent executing usefull instructions, and possibly other stalls not counted here.

We observe that stalls due to Loads&Stores instructions and full Reservation Station put together represent 40% of the total CPI at 8 cores.

The full Reservation Station event indicates long latency operations in the pipeline, most likely memory accesses, and other instructions that depend on these cannot execute until the former instructions complete their execution. In this situation new instructions can not enter the pipeline and start execution. This means that 40% of the cycles at 8 cores are spent accessing memory. From these numbers we deduce that the main causes of non-scalability of Web servers are related to high memory latencies. Those can be explained by further fine-grain profiling.

Note that, at 4 cores, Knot is performing better than μserver and Watpipe. The fact that Knot's CPI is smaller than the other two proves that it suffers less from stalls. Scalability of a memory-bound application can benefit from high context switches rate, but only if it decreases the number of memory stalls. This corresponds to the behaviour observed here for Knot.

C. Memory scalability

To analyze the origin of the high memory latencies, we conducted profiling experiments similar to the ones found in Veal *et al.* [14]. We did not notice any irregularity in caches and TLB behaviours, thus we analyzed the front side bus usage, as shown in Figure 5. In our experiments the data bus

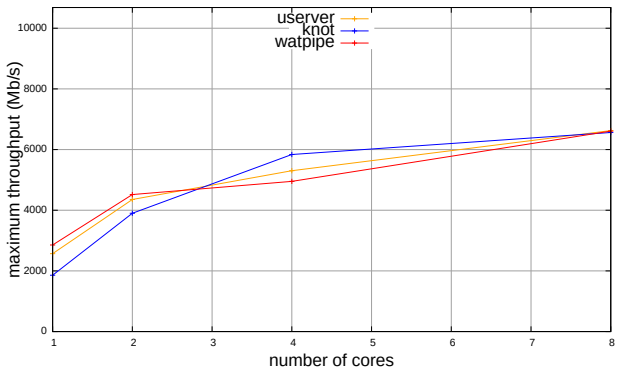

Figure 4. Scalability of the 3 Web servers.

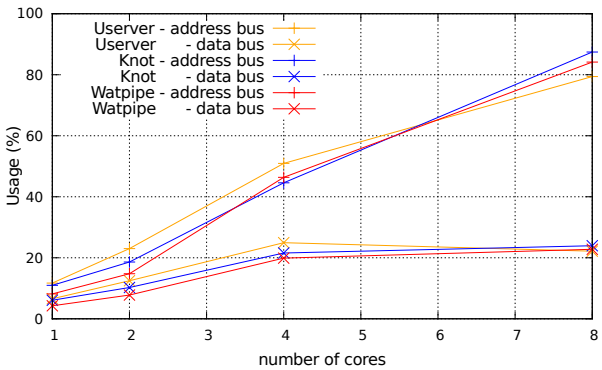

Figure 5. Data and Address bus usage.

seems to stall around 20% at 4 and 8 cores, not increasing anymore as it does from 1 to 4 cores. When analyzing the address bus, we find an abnormal high usage for all servers: around 50% at 4 cores and over 80% at 8 cores. We believe this situation corresponds to address bus contention, leading to data bus not being able to deliver more throughput. This saturation happens because all cache coherence protocol messages (also known as cache snoops) are sent on the address bus. The cache coherence protocol used in our case is the MESI protocol. In this implementation, many of the messages are broadcasted to all the cores, especially on every load from main memory. This is why the saturation happens so fast.

The case of NUMA architectures: The hardware used in this study is UMA: Uniform Memory Access. In NUMA (NonUniform Memory Access) hardware, there is no more data and address buses, instead high-bandwidth nodes interconnect, and the memory is distributed among NUMA nodes. We use Corey's memory benchmark [15] to investigate what can be expected as a worst case scenario concerning Web servers scalability. In this benchmark, each core allocates a 1GB memory area on its local node, and accesses it sequentially, no data sharing is involved. We run this benchmark on a 24-cores NUMA machine (2x12 cores), and measure the memory throughput in terms of bytes accessed per CPU cycle. Our experiments show that the worst case scenario (using optimized MMX instructions) can only achieve a 4.8 speedup (12.1 speedup with standard instructions) using 24 cores. These preliminary experiments tend to show that the scalability issue due to memory access costs also occurs with NUMA architectures. Extending our investigations to NUMA architectures, which is beyond the scope of the present paper, nevertheless constitutes an interesting direction for future work.

D. Using NCOPY to reduce sharing

Since the memory coherence protocol is the cause of the scalability problem here, we are interested in reducing data sharing among cores. To reduce sharing in our experiments, we benchmark NCOPY solutions for each model. This allows us to evaluate the impact of sharing a single socket for accepting connections. Indeed, in NCOPY, each copy is a completely independent process, and thus listen to a different port. Note that contrary to previous experiments, an external load-balancer is needed in this case. We benchmark each model with all possible NCOPY partitionnings.

The table III presents the results of our NCOPY experiments along with previous results. 1 copy results corresponds to our previous plots. Numbers of interest are red. Interestingly, at 4 cores, when original configurations of Watpipe and μserver cannot keep up with Knot's throughput, the NCOPY versions expose the similar performances as Knot. For μserver and Watpipe, this is a 13% improvement over their respective original configurations. This means that the sharing cost is reduced by using NCOPY. More precisely here, accepting connections on more than one socket reduces memory contention enough to provide better performances, except for Knot which reduces contention by having a inherently higher cost of context switching.

We also note that no configuration is always the best. NCOPY is better at 4 cores to reduce sharing of event-driven and stage-based implementations. At 2 cores, original SYMPED and stage-based configurations have higher throughput than their NCOPY configurations, or any of Knot configuration. It is because the duplication of data have a cost, and is not always worthy, as Web server performances also depend on their memory footprint [8]. Besides, the best 2 cores configurations in our experiments always happen with 2 cores sharing a L2 cache.

All of this suggests that paying attention to the underlying hardware setup can help predicting the best configuration to achieve good performances. For example, we know that if we choose 2 cores sharing a cache then choosing a NCOPY configuration is paradoxal, regarding memory sharing. The same goes at 4 cores, as we know that in our setup the last level of cache cannot be shared by all the chosen cores,

nb cores	number of copies	μserver	Watpipe	Knot
1	1	2567	2853	1856
2	1	4355	4516	4090
	2	4362	4027	4090
4	1	5302	4950	5835
	2	5968	5584	5848
	4	5806	4447	4399
8	1	6626	6607	6563
	2	6569	6552	6399
	4	6564	6561	5598
	8	6566	5634	3327

Table III
PEAK THROUGHPUT FOR EACH WEB SERVER (MB/S)

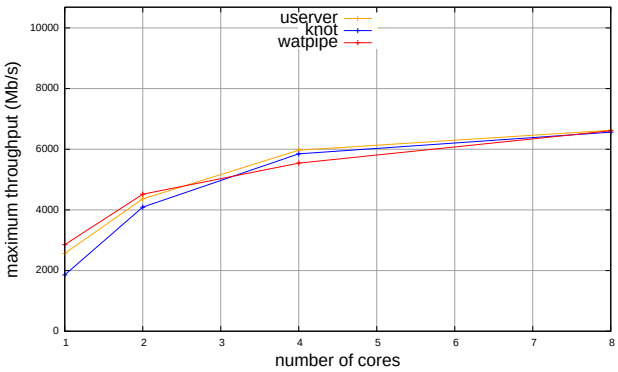

Figure 6. Scalability of the 3 Web servers including NCOPY configurations.

then NCOPY is a good lead. More precisely, if we choose two pair of cores sharing a L2 cache, then we will choose to have two copies, each working on such a pair of cores. Our experiments show that having four copies in such a case decrease performance. We want to emphasize here the impact of hardware setup, and especially memory hierarchy, on Web server performances.

The figure 6 presents the best peak throughput achieved for each server among all its possible configurations (including NCOPY). We can observe a better performance scaling than without NCOPY at 4 cores for μserver and Watpipe. This is however not enough to provide better performance at 8 cores for any of the servers studied.

V. RELATED WORK

The three programming models studied here have been extensively studied in a monocore context by Pariag *et al.* [3]. The main – and crucial – difference is that our comparative analysis deals with the multicore setting, and can thus be seen as a generalization of their work. Our methodology is inspired by the one found in [3]: in particular, we use the same workload and similar Web server implementations. A notable difference is that we changed the underlying thread library used for Knot. There are two reasons for this change: first, the legacy Capriccio [4] library is event-based and we want Knot to be representative of the thread-based model. Second, Capriccio cannot leverage multicore architectures and is thus inappropriate in our case.

A similar comparative study has recently been conducted by Harji *et al.* [16] on small multicore hardware (4 cores). In this paper though, the authors have removed the thread-per-connection model from their comparison. Our study differs in that we use larger hardware, which allows us to observe a new behaviour at 8 cores, not present with smaller hardware, and we include the thread-per-connection model in our comparison.

Multicore Web servers have been studied in the past. Choi *et al.* [17] simulate simple hardware to compare different server designs with workloads built on real-world traces. We believe their workloads are not appropriate in the data server case that we consider due to their high I/O rate. In comparison, our approach considers a real hardware setup with recent hardware and in-memory data fileset. Though we do not deny the interests of simulation, we also believe that experiments on real hardware can lead to significantly different results.

Veal *et al.* [14] have also studied the scalability of the Apache Web server on very similar 8-cores machines. Using the SpecWeb2005 workload, they are able to show that Apache does not scale with the number of cores. They conclude that their address bus is the primary cause of this performance issue. The major difference with the work presented here is that we consider three programming models in a comparative analysis (through 3 representative implementations), whereas their study is limited to the Apache Web server (relying on the thread-based model). We can therefore draw more general conclusions that span over the three considered models.

Other works such as Corey [15] and Barrelfish [18] have made multicore Web servers experiments with ad-hoc implementations. PK [19] and DProf [20] have also benchmarked Apache in a multicore context. All these experiments were mostly aimed at stressing the kernel or the software stack, whereas we use a much more sophisticated workload (based on SpecWeb99) that simulates access patterns to a large file distribution, which are much closer to those encountered in the real-world. Furthermore, instead of using a modified kernel such as the ones found in [15], [18], [19], we use a standard, unchanged kernel running on a commodity Linux operating system.

Finally, Voras *et al.* [21] compare some known multi-threading models for high-performance I/O network servers in a small multicore environment. Though they compare event and stage architectures within their database, they omit to compare them with the thread-per-connection model. They also notice biais in their methodology as the client and the server are run on the same machine, leading to an

execution without any actual network I/O, and impacting the scheduling of the server. Such results are difficult to apply to Web servers in our context.

VI. Conclusions

We carried out a thorough comparative analysis with extensive experiments that bring new insights on current limitations concerning the scalability of the three main models used for implementing Web servers on multicores.

Unlike the monocore case where the thread-based, event-driven and stage-based implementations are known to exhibit very different performances, surprisingly, we show that the situation is significantly different in the multicore setting. First, at 4 cores, the thread-per-connection implementation naturally outperforms original event-driven and stage-based ones in our case. Careful use of NCOPY configurations can improve the performance of event-driven and stage-based implementations, thus balancing the performances of all implementations. Second, When the number of cores increases, a point can be reached from which the different programming model's implementations tend to show very similar performances. Our experiments show that this point is already reached at 8 cores. Specifically, we show that at this threshold, all implementations of a programming model run into the same problematic memory limitation: a saturation of the address bus caused by the cache coherency protocol.

To the best of our knowledge, the fact that the different Web servers implementations, which reflect the main three programming models for data servers, tend to exhibit the same performance limits when increasing the number of cores is a new result. Furthermore, our experiments shed light on the nature of this bottleneck originating from the cache coherence protocol. Our experiments confirm the importance of tuning the operating system prior to tuning the Web server in order to bring the performances to their full potential, which we believe would be true for every I/O intensive application. Finally, we show that deployment configuration of a Web server should be done regarding the underlying hardware, and more precisely the memory architecture.

We believe that these contributions provide important insights for the further development and deployment of scalable data servers. Some of the possible solutions for better scalability of data servers could take place in hardware, by modifying the memory behaviour. More precisely, one could change the cache coherency protocol. On manycore architectures, another approach could consist in having only a local per-NUMA-node memory consistency and no hardware global memory consistency. This behaviour has been adopted with the new Intel's SCC prototype [22]. This will obviously result in huge changes in software development, if this solution becomes mainstream.

An interesting direction for future work consists in conducting similar investigations for NUMA architectures, since our preliminary experiments in this direction show that the cache coherency protocol also greatly affects scalability on NUMA 24-cores machines.

References

[1] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: facebook's photo storage," in *OSDI'10*, 2010.

[2] Apache foundation, "The apache http server project," 2007, http://httpd.apache.org.

[3] D. Pariag, T. Brecht, A. Harji, P. Buhr, and A. Shukla, "Comparing the Performance of Web Server Architectures," in *EuroSys*, 2007.

[4] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *SOSP*, October 2003.

[5] U. Drepper and I. Molnar, *The Native POSIX Thread Library for Linux*, feb 2005. [Online]. Available: http://www.akkadia.org/drepper/nptl-design.pdf

[6] M. Welsh, D. Culler, and E. Brewer, "SEDA: An architecture for well-conditioned scalable internet services," in *SOSP*, 2001.

[7] J. R. Larus and M. Parkes, "Using Cohort Scheduling to Enhance Server Performance," in *USENIX ATC*, 2002.

[8] A. Harji, "Performance comparison of uniprocessor and multiprocessor web server architectures," Ph.D. dissertation, 2010.

[9] SPEC, (Standard Performance Evaluation Corporation). SPECweb99. http://www.spec.org/web99/.

[10] ——, (Standard Performance Evaluation Corporation). SPECweb2009. http://www.spec.org/web2009/.

[11] D. Mosberger and T. Jin, "Httperf, a tool for measuring web server performance," *First Workshop on Internet Server Performance*, 1998.

[12] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open Versus Closed: a Cautionary Tale," in *NSDI*, 2006.

[13] Maynard Johnson et al., "OProfile homepage," http://oprofile.sourceforge.net.

[14] B. Veal and A. Foong, "Performance Scalability of a Multi-Core Web Server," in *Proceedings of ANCS '07*. Orlando, FL, USA: ACM Press, December 2007.

[15] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An Operating System for Many Cores," in *OSDI*, 2008.

[16] P. B. Ashif Harji and T. Brecht, "Comparing high-performance multi-core web-server architectures," in *5th Annual International Systems and Storage Conference (SYSTOR)*, Haifa, Israel, Jun. 2012, to appear.

[17] G. S. Choi, J.-H. Kim, D. Ersoz, and C. R. Das, "A Multi-Threaded PIPELINED Web Server Architecture for SMP/SoC Machines," in *WWW*, 2005.

[18] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schuepbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *SOSP*, 2009.

[19] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An Analysis of Linux Scalability to Many Cores," in *OSDI*, 2010.

[20] A. Pesterev, N. Zeldovich, and R. T. Morris, "Locating Cache Performance Bottlenecks Using Data Profiling," in *EuroSys*, 2010.

[21] I. Voras and M. Zagar, "Characteristics of multithreading models for high-performance io driven network applications," *AFRICON 2009*, 2009.

[22] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on intel's single-chip cloud computer processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 73–83, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/1945023.1945033