Analysis of Event Processing Design Patterns and their Performance Dependency on I/O Notification Mechanisms

Ronald Strebelow[1] and Christian Prehofer[2]

[1] Institute of Computer Science, University of Augsburg, Augsburg, Germany
[2] Fraunhofer Institute for Communication Systems ESK, Munich, Germany

Abstract. Software design patterns reflect software engineering practices and experience by documenting approved design solutions. We consider here two widely used patterns, the Half-Sync/Half-Async and the Leader/Followers pattern, which aim for efficient processing of messages in multi-core environments.
We will analyze the performance differences not only between both design patterns but also between different implementation variants. These variants use different event notification mechanisms which are used to sense message arrivals on a set of connections. We will show that performance depends not simply on data sharing or lock contention but on the selected event notification primitives and their specific characteristics.
In more detail, we evaluated both patterns in terms of three different event notification mechanisms: `select`, level-triggered and edge-triggered `epoll`. The latter two are the operation modes of the `epoll` API. In particular, the right choice of the API can influence the performance by a factor of two. Secondly, the more recent `epoll` is overall faster, but in some aspects slower which strongly degrades the Half-Sync/Half-Async performance. Existing performance evaluations for these patterns do not analyze their multi-core performance. Furthermore, they do not include analysis of bottlenecks, data sharing, or operating system primitives.

1 Introduction

It is well known that designing efficient software is a challenging task. Software design patterns can help in this process. They are created through careful analysis of existing software and capture designs that have proven useful and have been used in many different systems. These design patterns then aid developers by providing extensive experience.

We consider the problem of handling messages from a large number of input streams on multiple cores of a single machine. There exist several design patterns that aim to solve this problem. For this paper we choose two of the most prominent ones, namely Half-Sync/Half-Async and Leader/Followers.

The Half-Sync/Half-Async pattern is widely used. It is used in the implementations of network stacks of most operating systems [9]. The work in [3]

found the pattern in 3 of 21 analyzed legacy software systems and their documentations. Both Half-Sync/Half-Async and Leader/Followers are also used in multi-threaded implementations of OMG's CORBA[2, 5, 7, 10].

Both patterns differ mainly in their distribution of tasks to threads. For instance, the Leader/Followers pattern uses a pool of identical threads which have to synchronize their access to the input streams. In contrast, Half-Sync/Half-Async has a pipeline-architecture. It uses a dedicated thread to retrieve incoming messages and forwards them to the remaining threads which process these messages.

The performance of both patterns does not solely depend on their architecture, i.e. usage of locks or sharing of data, but also on the availability of efficient API's to retrieve events about new messages from the operating system. In our paper, we analyze the common API's: `select` and the more recent `epoll`. Both are used to retrieve such events from a set of open connections through a single call. But the more recent `epoll` API was designed to handle high amounts of connections more efficiently compared to `select`. Both are widely used in many software systems but current software, e.g. Apache HTTP Server[3], Kamailio[4] (former OpenSER), or SQUID[5], use `epoll` if available.

Our aim is to analyze the performance of the aforementioned design patterns in conjunction with the different event notification mechanisms. We will show that both design patterns have significantly different performance characteristics. And we will show that performance depends not only on data sharing or lock contention but on the event notification primitives selected and their specific characteristics.

Both, the software design patterns and the event notification mechanisms have been subject of evaluations before. These evaluations consider very specific applications like web servers[1, 6] or CORBA [2, 5, 7, 10]. These include operations which are unrelated to patterns or notification mechanisms but affect performance. Also, these benchmarks do not consider scalability but performance for a particular set up. These set ups include up to quad-core CPU's. Here, we evaluate performance and scalability for up to 16 cores. Our set up uses two comparable implementations of the patterns, which focus on the core problem.

2 I/O and Event Notification Mechanisms

In this section, we present the different event notification mechanisms which we used to implement the Half-Sync/Half-Async and Leader/Followers patterns.

Retrieving information about message arrival on a single connection can be done by reading from that connection. That operation will block until a message arrives. For multiple connections this approach requires one thread per connection which is inefficient and often does not scale.

[3] httpd.apache.org
[4] www.kamailio.org
[5] www.squid-cache.org

Another approach is to group all connections into a so-called interest set. Instead of waiting for messages from a single connection, a specific call, the event notification mechanism, is used to wait for messages on any connection that is contained in the interest set.

We used two common event notification mechanisms: Firstly, the `select` system call which is available on most operating systems. And secondly, the more recent `epoll`, a Linux specific API which provides two modes: level-triggered and edge-triggered. All three mechanisms are summarized in the following:

select The `select` API maintains the interest set, actually a bit set, in user space. On each invocation that set is copied into kernel-space. `select` does not return occurred events for a connection but the state of the same. Thus, it does not report incoming messages but that data is available for reading.

epoll The `epoll` API maintains the interest set in kernel-space using a red-black tree. Unlike `select`, an invocation returns a sub-set of the interest set, such that idle connections are omitted. This makes `epoll` suitable for big sets with a high number of idle connections.

- **level-triggered** This mode behaves much like `select` since it returns the state of a connection instead of events. Multi-threaded software has to avoid that a connection is falsely reported several times to have data available. For this purpose it is possible to deactivate a connection automatically. Once a connection is deactivated it has to be reactivated explicitly using another system call.
- **edge-triggered** In this mode, `epoll` does indeed return occurred events and the software is required to manage connection state which hinders programming. Since only new events are reported, it is not necessary to de- and reactivate connections. This saves a system call per event and may provide for better performance.

3 Design Patterns

In this section we briefly introduce the Half-Sync/Half-Async and the Leader/Followers pattern. For a detailed description including both patterns we refer to [8].

3.1 Half-Sync/Half-Async

The Half-Sync/Half-Async pattern provides parallelism by executing different services in different threads. These services may communicate with each other by utilizing a central queue and are divided into two groups:

Asynchronous services are triggered by events. These services can also process data read from the queue.

Synchronous services get their data from the queue exclusively. If no data is available these services wait for it to arrive.

Our implementation of Half-Sync/Half-Async, shown in Fig. 1, uses two kinds of services. The first one, an asynchronous service, multiplexes events from a set of event sources, indicating incoming messages. Each new event is inserted into the queue for further processing. This service utilizes one thread. That decision is motivated by the Leader/Followers pattern which has to synchronize its access to the set of connections. We will explain this in detail in the next section. To remain comparable only a single thread is allowed to multiplex these events. Apart from this, a multi-threaded asynchronous service is possible. The second service, a synchronous one, demultiplexes events from the queue to multiple threads which process them, one at a time. Processing essentially consists of reading the message, sending the response and, in case of level-triggered `epoll` and `select`, reactivation of the connection. Pseudo code for both services is shown in Fig. 2 and 3.

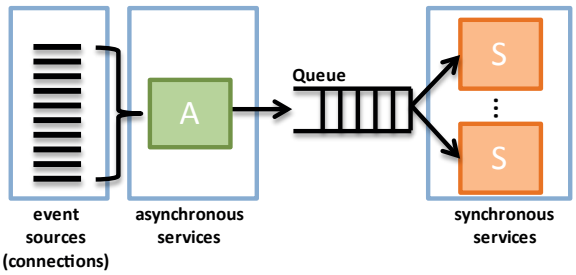

Fig. 1. Components of the Half-Sync/Half-Async pattern

```
loop forever
do
  poll system for events

  for each reported event
  do
    push event into queue
  done
done
```

Fig. 2. Pseudo code of the asynchronous service

```
loop forever
do
  pop event from queue

  read message
  process message
  send response
  reactivate connection
done
```

Fig. 3. Pseudo code of the synchronous service; the code is executed by each of the multiple threads

The access to the event sources is more subtle than shown in Fig. 1. The asynchronous service must access these to report any new events. The synchronous service on the other hand must access these when sending or receiving on the associated connection.

The pattern as shown above has three potential bottlenecks: the asynchronous service, the queue, and the synchronous service. The asynchronous service may

bottleneck because of his single thread. If the thread reaches full utilization, the throughput reaches its maximum. The queue is a central resource protected by a mutex. If the number of threads is too high, lock contention is going to decrease overall performance. The threads of the synchronous service, especially in case of level-triggered `epoll`, have to perform connection reactivation. It includes a lock (in the kernel) which may be heavily contended.

3.2 Leader/Followers

The Leader/Followers pattern organizes its threads in a thread pool. All threads try to access the set of event sources and therefore must synchronize. To obtain an event one of the synchronous notification mechanisms are used and, if necessary, the connection is deactivated. Once a new event is obtained, it is processed by the same thread. Several threads may process distinct events in parallel. As was the the case for Half-Sync/Half-Async, processing consists of reading the message, sending the response and, in case of level-triggered `epoll` and `select`, reactivation of the connection. After processing has been finished the thread becomes idle and tries again to get access to the set of event sources. Figure 4 shows this cycle and Fig. 5 shows the pseudo code.

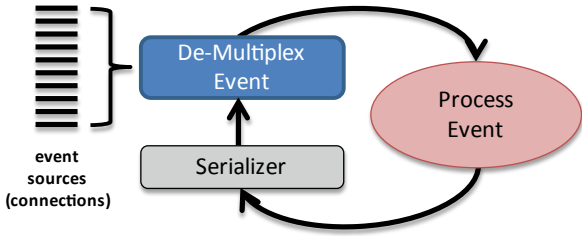

Fig. 4. Execution cycle each thread in the Leader/Followers pattern performs

```
loop forever
do
  enter mutual exclusive section
    poll system for one new event
  leave mutual exclusive section

  read message that caused event
  process message
  send response
      reactivate connection
done
```

Fig. 5. Pseudo code of the work that each thread in the Leader/Followers pattern performs

The performance of the Leader/Followers pattern is governed by a serial section in which the event notification mechanism is invoked. In our implementation serial access is enforced by a mutex.

Since all threads execute the same tasks they also have to share all data. In particular these are the state of the event sources and the data needed for any specific event notification mechanism.

4 Evaluation Set-Up

For the evaluation, presented in Sect. 5, we use two measurement metrics to support our findings: throughput and the time required for certain operations. These metrics were not measured during a complete run. Instead each run consists of 3 phases. The initialization phase, the measurement phase and the shutdown phase. The first and the last did not participate in measurements.

The steady-state throughput is measured in messages per second. For more detailed analysis we measured the service times of certain operations. Each particular measurement was run 5 times over a period of 5 minutes each. We present the mean value of these 5 runs. For the measurement of the service times an operation was surrounded by primitives returning the time difference in nano-second granularity.

For the evaluation we use a request-response micro-benchmark utilizing 512 TCP/IP connections lasting over an entire run. The client component sends for each connection at most one request at a time. Only when the reply was received a new request is send. To support high throughput the client maintains as many threads as CPU cores are assigned to it.

The evaluation was performed on an AMD Opteron 6134 system with 4 CPU's supporting 8 cores each, and clocked at 2.3GHz. As operating system we are using Linux with kernel-version 2.6.35. Client and server component are assigned to 2 CPUs each, hence each use up to 16 cores. The distribution was chosen so that CPU resources, e.g. caches or the memory bus, are not shared between both components.

For the evaluation the Leader/Followers pattern was run with 1 to 16 threads and Half-Sync/Half-Async with 2 to 16 threads, since that pattern requires at least two threads.

Message processing consists of reading the message, performing a simulated workload through busy waiting a defined duration, and sending a response. Throughout Sect. 5 we assume that this workload takes 0μs.

5 Evaluation Results

In the following, we discuss the main results of our evaluation, comparing the two patterns and APIs in detail.

Figure 6 shows the steady-state throughput of both patterns and all event notification mechanisms described in Sect. 2. In the best case, i.e. using edge-triggered `epoll`, the Half-Sync/Half-Async pattern outperforms the Leader/

Followers pattern. Figure 7 also shows that this is achieved with 10 instead of 13 threads. However, beyond 11 threads Leader/Followers shows better performance, as the Half-Sync/Half-Async has significant performance loss if too many cores are used.

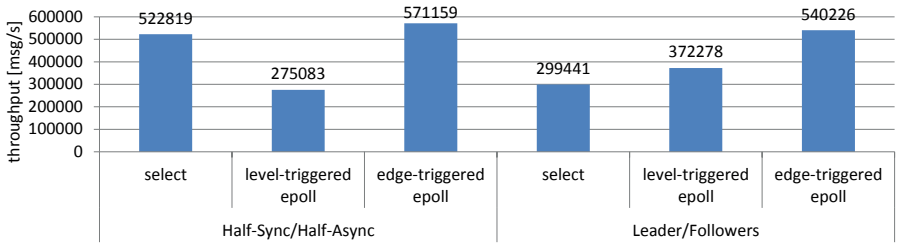

Fig. 6. Steady-state throughput of the Half-Sync/Half-Async and Leader/Followers patterns; both patterns are shown with their three implementation variants: select, level-triggered epoll, and edge-triggered epoll

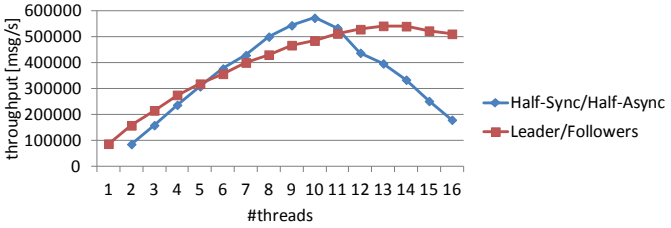

Fig. 7. Scaling of Half-Sync/Half-Async and Leader/Followers (both edge-triggered epoll) over increasing numbers of threads

5.1 Half-Sync/Half-Async

The Half-Sync/Half-Async pattern consists of a two-staged pipeline: The first stage is the asynchronous service and the second stage is the synchronous one. Thus, best performance is achieved if both services are fully utilized and perform equally fast.

In case of edge-triggered `epoll` and `select` both services are fully utilized if the synchronous service uses 9 threads (Fig. 8), i.e. 10 threads are used overall. Using fewer threads causes an under-utilization of the asynchronous service. Figure 9 shows the time which is required to obtain a new event and the time it takes to put it into the queue. For small numbers of threads the asynchronous service spends its time waiting on new events, i.e. incoming messages. Adding threads to the synchronous service increases processing rate and in turn increases the number of overall messages in our benchmark system. This increased rate is

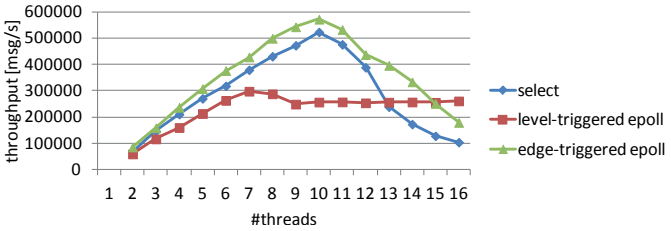

Fig. 8. Half-Sync/Half-Async: Throughput comparison of all three notification mechanisms

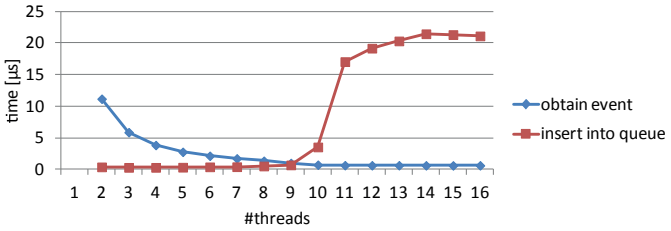

Fig. 9. Half-Sync/Half-Async (edge-triggered epoll): Average required time to obtain one event and to insert it into the queue

observable through the reduced time spent for waiting. This rate is decreasing with the same rate as threads are added to the synchronous service.

The gap between the peak performance of the `select`- and the edge-triggered `epoll`-based implementation is caused by differences in the API's. Since `select` maintains its interest set in the user space, that set is required as parameter to the call. The same parameter is used to return the desired information and is therefore altered in place. This has to effects: On the one hand, a copy of the interest set is required instead of the original one. One the other hand, since the interest set is a bit set, the returned set has to be scanned for the desired information. In the worst case, the entire set contains only one event indication. We evaluated the number of events returned in average. We found that, in case of Half-Sync/Half-Async, no more than 8 such events are reported. In contrast, `epoll` omits the copy and also requires no scan since all returned elements indicate events.

The case of level-triggered `epoll` requires a more detailed analysis: Figure 10 shows the time it takes to process a single message and breaks that time down into the three operations reception, connection re-activation, and transmission of the response.

Connection deactivation is needed for `select` and level-triggered `epoll` to avoid that new messages are reported several times. Deactivation is performed automatically without performance loss and we do not consider it here. Reactivation after reading data is needed to receive the next data on one socket. In case of `select` it is performed by atomically setting a bit in the interest set. In case of level-triggered `epoll` the operation is much more time consuming. It requires a switch into kernel space, where the `epoll` API then searches the

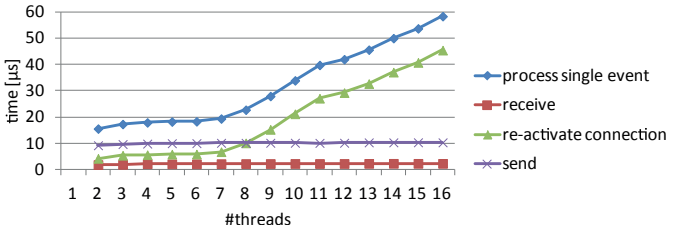

Fig. 10. Half-Sync/Half-Async (level-triggered epoll): Time required to process a single event, split into its basic operations receive, re-activate (connection), and send (a response)

connection within a red-black tree and makes it available for subsequent calls. Afterwards it checks if events are already available and sets a specific flag in the control structure. Both operations are part of a serial section which is guarded by a lock. As Fig. 10 shows, the costs of these operations increase with the number of threads which inhibits a further increase of overall throughput.

5.2 Leader/Followers

The performance of the Leader/Followers pattern is limited by the serial section in which the set of event sources is accessed. Thus, there is a sweet spot at which the time required to process a message divided by the number of threads equals the time required to perform the serial section. Adding more threads does not increase performance.

Figure 11 shows the throughput achieved with each notification mechanism. Striking is the performance difference between the edge-triggered `epoll`- and the `select`-based implementation. Also the graph of level-trigger `epoll` is peculiar. We will explain both peculiarities in the following:

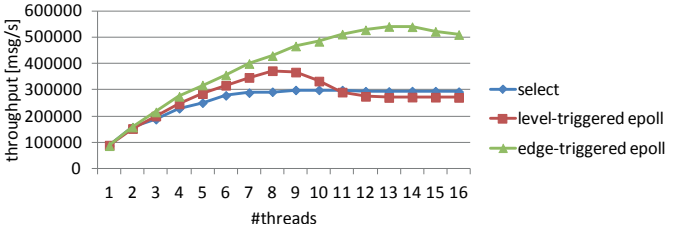

Fig. 11. Leader/Followers: Throughput comparison of all three notification mechanisms

Compared to Half-Sync/Half-Async the Leader/Followers pattern is more sensible for the efficiency of `select`. The Half-Sync/Half-Async pattern is not limited by its asynchronous service and hence, not limited by the efficiency of `select`. On the contrary, in case of Leader/Followers the lower performance of `select` takes effect (Fig. 12). Thus, on average a call to `select` needs almost 80% longer to complete than one to edge-triggered `epoll`. Hence, the serial section takes longer which causes a lower performance limit.

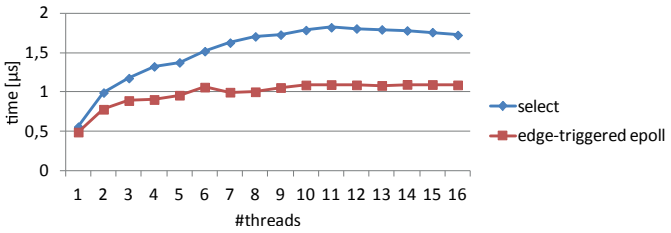

Fig. 12. Leader/Followers: Average time it takes to sense a single event using select or edge-triggered epoll

The case of level-triggered `epoll` is similar to that of the Half-Sync/Half-Async pattern. As Fig. 13 shows the time required for processing a message increases as well. As before, this is again caused by connection re-activation. In consequence, increasing the number of threads compensates for the increased processing time but does not increase performance.

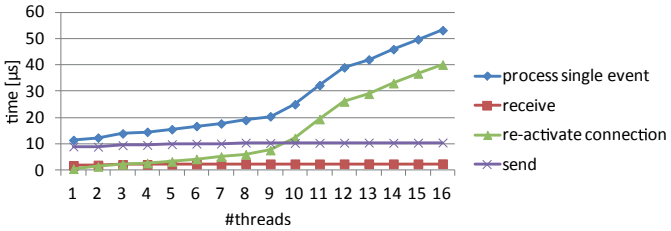

Fig. 13. Leader/Followers (level-triggered epoll): Time required to process a single event split into its basic operations receive, re-activate (connection), and send (a response)

6 Related Work

In the existing literature, patterns and multi-threading strategies related to event handling are often implemented and evaluated in the context of OMG's CORBA[2, 5, 7, 10] and web servers[4, 1, 6]. In both cases Half-Sync/Half-Async and Leader/Followers are used to parallelize request processing.

Evaluation of performance is present in most of these papers but is performed on quad-core processors at best. Also benchmarks are affected by application-specific infrastructure. Only [7] compares Half-Sync/Half-Async and Leader/Followers but does not provide any detailed analysis about the results. The authors found that Leader/Followers generally outperforms Half-Sync/Half-Async. Although not stated explicitly, the description of Half-Sync/Half-Async suggests that asynchronous operations are used for notification about incoming requests while `select` is used for Leader/Followers. In an earlier paper[4] the authors found that those asynchronous operations are efficient only for big messages of at least 50kByte.

In [1] `select` and `epoll` are evaluated in the context of userver[6] using dual-core Xeon system. In case of many idle connections, the authors found `epoll` to perform better than `select`. On the contrary, in case of few or no idle connections, both can perform equally well. Similar findings are presented in [6]. This paper uses a quad-core Xeon system but does not evaluate scalability. Both papers evaluated level-triggered `epoll` and used connection de- and reactivation. They consistently found that connection reactivation slowed down their benchmarks.

7 Conclusions

In this paper we evaluated the software design patterns Half-Sync/Half-Async and Leader/Followers. We implemented both patterns in terms of three different event notification mechanisms: `select`, level-triggered and edge-triggered `epoll`. The latter two are the operation modes of the `epoll` API.

In summary, we found that the performance of both patterns highly depends on the efficiency of the utilized API. We observed speed-ups up to 2 just by switching from `select` to edge-triggered `epoll`. Hence, the developer of such patterns currently has to know the multi-core or parallel performance impact of the specific aspects of the patterns and APIs in order to obtain optimal performance.

The Leader/Followers pattern has lower peak performance but does not degrade as strongly as Half-Sync/Half-Async if too many cores are used. In practice, this means that the patterns should adapt the number of cores according to load.

In more detail: Although `epoll` is generally faster, connection reactivation is slower than the corresponding `select` implementation. I.e. the operation of reactivation is slower and in particular does not perform well for multi-core environments with many threads. In our case, this has led to the case that Half-Sync/Half-Async has performed better with `select` than with level-triggered `epoll`. The reason is that the reactivation is performed within the critical pipeline stage of this pattern. Hence, it causes performance loss.

On the other hand, if the event notification mechanism is called within the critical path, as is the case for the Leader/Followers pattern, `select` is outperformed by `epoll`.

References

1. L. Gammo, T. Brecht, A. Shukla, and D. Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of the Ottawa Linux Symposium*, 2004.
2. M. Harkema, B. Gijsen, R. van der Mei, and Y. Hoekstra. Middleware performance: A quantitative modeling approach, 2004.

[6] http://www.hpl.hp.com/research/linux/userver/

3. N. Harrison and P. Avgeriou. Analysis of architecture pattern usage in legacy system architecture documentation. In *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*, pages 147–156, 2008.
4. J. Hu, I. Pyarali, and D. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Global Telecommunications Conference, 1997. GLOBECOM '97., IEEE*, volume 3, pages 1924–1931 vol.3, Nov. 1997.
5. B. Natarajan, A. Gokhale, S. Yajnik, and D. Schmidt. Applying patterns to improve the performance of fault tolerant corba. In M. Valero, V. Prasanna, and S. Vajapeyam, editors, *High Performance Computing - HiPC 2000*, volume 1970 of *Lecture Notes in Computer Science*, pages 107–120. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-44467-X_10.
6. B. O'Sullivan and J. Tibell. Scalable i/o event handling for ghc. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 103–108, New York, NY, USA, 2010. ACM.
7. I. Pyarali, M. Spivak, R. Cytron, and D. C. Schmidt. Evaluating and optimizing thread pool strategies for real-time corba. In *Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*, OM '01, pages 214–222, New York, NY, USA, 2001. ACM.
8. D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.
9. P. Willmann, S. Rixner, and A. L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 8–8, Berkeley, CA, USA, 2006. USENIX Association.
10. Y. Zhang, C. Gill, and C. Lu. Real-time performance and middleware for multiprocessor and multicore linux platforms. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pages 437–446, 2009.