Eve: A parallel event-driven programming language

Alcides Fonseca[1], João Rafael[1], and Bruno Cabral[1]

University of Coimbra, Portugal
{amaf,bcabral}@dei.uc.pt, jprafael@student.dei.uc.pt

Abstract. We propose a model for event-oriented programming under shared memory based on access permissions with explicit parallelism. In order to obtain safe parallelism, programmers need to specify the variable permissions of functions. Blocking operations are non existent, and callback-based APIs are used instead, which can be called in parallel for different events as long as the access permissions are guaranteed. This model scales for both IO and CPU-bounded programs.
We have implemented this model in the Eve language, which includes a compiler that generates parallel tasks with synchronization on top of variables, and a work-stealing runtime that uses the epoll interface to manage the event loop.
We have also evaluated that model in micro-benchmarks in programs that are either CPU-intensive or IO-intensive with and without shared data. In CPU-intensive programs, it achieved results very close to multithreaded approaches. In the share-nothing IO-intensive benchmark it outperformed all other solutions. In shared-memory IO-intensive benchmark it outperformed other solutions with a more or equal value of writes than read operations.

Keywords: Event-oriented, Parallel Programming, IO performance

1 Introduction

The high-performance of IO applications has become more important in the last decades as the Internet applications are required to handle a large number of clients with a high throughput and low latency. More recently, event-loop based models have become popular for high-performance applications.

However, it is not possible to assert whether event-loop models are better or worse than shared-memory multithreaded models. Event-loops have become popular because several applications using that model have shown to have a lower memory consumption, better performance and better scalability than equivalent programs written in a threaded model [1] [2]. The event-based model is also considered simpler than using threads, since threading requires proper synchronization and it is more difficult to debug[3]. A counter-argument against events is that reasoning about the control flow is difficult and with careful reengineering, threaded approaches can achieve similar performance values[4].

The actor model has been a popular approach to unify both models, with each actor running on its own thread, each with its own event-loop [5]. This approach allows for an event-loop approach to scale across multiple processor cores with a composable interface. However, the actor model limits the memory access to the actor, which has less flexibility that threaded models for some applications.

The Eve language introduces a new model that supports the event-loop and task-based parallelism to allow for both IO and CPU bounded programs to achieve high concurrency. Unlike the actor model, the eve language is shared memory and any task can access data from any other task. But in order to reduce the complexity of handling all the synchronization necessary to avoid deadlocks and to guarantee data consistency, the language makes use of access permissions, which have to be specified in the program. From these access permissions, the program is automatically parallelized, and monitors are added when necessary to guarantee consistency. This works across the event loop, allowing for shared-memory task-based parallelism inside the event loop.

The main contributions of this paper are: A new model for shared-memory task parallelism within the event-loop; The definition of a language that supports that model; The implementation of a compiler for that language and a runtime library to support the execution; and an evaluation and comparison of that language against popular languages.

The rest of the paper is organized as follows: Section 2 explains the new programming model proposed; Section 3 details the implementation of the compiler and runtime; Section 5 compares our model to other state of the art approaches; finally Section 6 concludes the document and presents some future work.

2 Approach

We propose a model that combines the event loop and the shared-memory aspect of threaded programming. We will focus firstly on the programming model, and then on the execution aspect.

The three main differences between the programming model of Eve and those of mainstream object-oriented languages is the usage of tasks, permissions and event callbacks.

Eve allows programmers to execute methods and blocks of code as parallel tasks. This is expressed using the `@` symbol. Program 1 is a parallel implementation of the Fibonacci function. Since the `a` and `b` assignment statement is prefixed with the `@` symbol, they are executed in parallel. After the `@`, the programmer has to write the access permissions required to execute that block of code. As long as the access permissions are correct, tasks can be introduced in any part of the code.

Permissions only apply to objects that are shared among different parts of the code. Local objects do not require access permissions since they are guaranteed to execute in the same thread without the need for synchronization.

```
fib: (n: int) int:
    if n < 2:
        return n

    a, b : int@
    finish:
        @ [+a, =n]: a = fib(n - 1)
        @ [+b, =n]: b = fib(n - 2)

    return a + b
```

Program 1: Parallel computation of the n^{th} Fibonacci number using the `finish` block for synchronization. `int@` defines a shared object of base type `int`.

Objects can have three different permissions on a method of code block: If no special annotation is added, the default permission is *Read permission*, which allows function to access a certain object, but an attempt to modify it will result in a compile-time error. If the programmer wants the shared object to be modified, then a *Full permission* is required and the variable should be prepended with a `+` sign. Finally, if only the reference is needed, without any read or write, a *Null permission* can be annotated by using the `-` sign. The main usage for this permission is to bind a reference to the object to the local context.

Using the same syntax as C++ a variable can be captured by copy or by reference using the `=` and `&` prefixes respectively.

Since it is frequent for sub-tasks to require access to the same objects, those operations must be executed inside a special *finish* block. When the execution reaches the *finish* special block, it releases all shared object, so that they can be used by the subtasks. This approach allows for a consistent view of the objects. Inside a finish block tasks can only require a subset of the parent's permission set, which prevents deadlocks between parent and child tasks.

Tasks have further restrictions in order to guarantee the corrected of concurrent programs: Tasks may not have infinite loops or blocking operations, as this could lead to live-locks. Instead, eve programs use a event-based non-blocking asynchronous API to interact with the Operating System.

The event-based callback system is another of the core features in eve. Any type in the language can enumerate the set of events if can trigger. Events are named types and they can contain objects of any type. Objects that can emit events can be used with the *on* construct to define a event callback. Program 2 shows an implementation of a simple socket-based chat showing the use of the *on* keyword to define callbacks and the @ keyword for parallel execution of tasks. When the socket object receives data, the `on client data` callback is executed, for instance. While the buffer reading is done in the current task, the writes to each client buffer is done in parallel tasks.

The execution model is based on the same task-oriented work-stealing scheduler present in Cilk[6] and many other frameworks. This approach has been proved to support several CPU-bounded operations with a good occupation of

```
import io.socket.*
import util.timeout

main: () void:
    clients: set<socket@>@

    tcp_socket.listen(8080):
        connection = [+clients] (c: connection&) bool:
            client: socket@ = c()
            clients.insert(client)

            on client data [clients]:
                message : vector<char>@ = client.read_buffer()
                @ for (c: socket@ in clients) [message, +c]:
                    c.write_buffer(message)

            on client close [+clients]:
                clients.remove(client)

            return true

        error = [+stderr] (e: error&) bool:
            stderr.write("Failed to start server: %s", error)
            return true
```

Program 2: A TCP broadcast server that accepts connections on port 8080.

multiple processors. A fixed number of POSIX threads are created, each with its own queue. Worker threads process the tasks in their queue and, when the queue is empty, they steal tasks from other queues.

Whenever a new task is being scheduled, the required permissions are verified that they are available. If they are not, the task is moved to the end of the queue, for a later execution. Since this adds an unwanted overhead, tasks should require as few permissions as possible.

New tasks can also be scheduled by the kernel, when a new kernel event is generated. These tasks will have to be executed in the right order and they cannot conflict on the shared objects that they required. In order to execute in the right order, the first callback should execute completely. If there are some operations pending because of other IO operations (such as writes), the remaining callbacks for the first event will only be called when the callback for this new event is completed.

3 Implementation

The implementation of the Eve language is divided in two main components: the compiler and the runtime. The compiler follows a traditional approach, with the code-generator phase emitting C++ code instead of machine code, thus being, in fact, a transpiler. Further compilation with GCC is performed to obtain binary files. The generated code makes heavy use of the Eve runtime to obtain parallelism and to enforce proper synchronization of data.

The Eve runtime architecture (in Figure 1) has a task management core, which is responsible for task handling. This includes creating and managing the worker POSIX threads, the management of tasks, load balancing of tasks using a work-stealing approach and also to guarantee proper access to shared objects. Additionally, the core of the runtime is also responsible for wrapping the *epoll* system calls, enabling the transition from kernel callbacks to events in Eve. The Libraries package exposes common system tasks, such as socket operations, using event handling for callback registration.

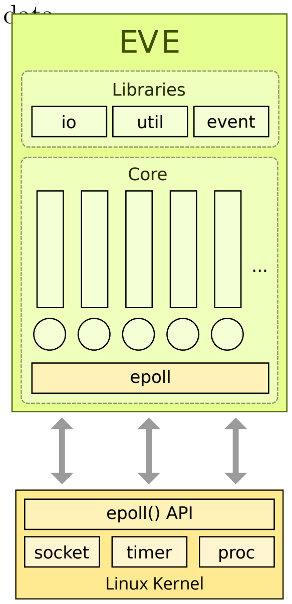

Fig. 1. Architecture of the Eve runtime, and its connection with the Linux kernel using the `epoll` interface.

The work-stealing approach was heavily based on the THE algorithm[7] from Cilk[6], with the suspend-steal method[8] for avoiding overheads of double stealing. Since the runtime integrates tightly with epool, workers call epoll_wait() instead of sleeping when it has no available tasks for running.

While events in Eve are instances of any class, event emitters have to extend the `emitter<T>` class, indicating that it can emit events of type T. Each instance of the class stores callbacks for this event on this object. This allows for a distributed callback table, effectively avoiding unnecessary contention with global table locks.

4 Evaluation

In this section we compare the performance of the Eve platform with existing popular frameworks for high-performance IO and parallel programming. The evaluation focused on two programs: Echo Server, representative of IO-intensive applications; and Atomic Counter, representative of concurrent programs with synchronization.

In terms of Lines of Code, one heuristic frequently used to compare complexity of programming expression, programs written in Eve are smaller than other low-level frameworks such as libev, TBB or Fork/Join. It also performs fairly well against higher-level frameworks such as gevent and REV despite achieving much better performance.

Each execution was repeated 30 times from which the average values and respective standard deviations are shown. Additionally, a first execution was performed before the 30 repetitions to avoid interference of the JIT compilation

	Ingrid	Astrid
Motherboard	Dell Inc. 0CRH6C	SuperMicro X9DAi
Processor	2x Intel(R) Xeon(R) X5660 2.80GHz, 24 hardware threads	2x Intel(R) Xeon(R) E5-2650 2.00GHz, 32 hardware threads
Memory	24 GB DDR3 1333 MHz	32 GB DDR3 1600 MHz
Connectivity	Broadcom Corporation NetXtreme BCM5761 Gigabit Ethernet PCIe	Intel Corporation I350 Gigabit Network Connection

Table 1. Hardware specification of benchmark hosts.

and caching mechanisms. The information of the two machines used is presented in Table 1. Single host benchmarks were executed on *Astrid*. For communication benchmarks, *Astrid* was used for the server while *Ingrid* was used for the client. The two hosts were directly connected using a ethernet cable, to avoid external interference.

The following versions were used: GCC 4.7.2; Erlang R15B01 (-S16); Go 1.0.2; GHC 7.4.2; node.js 0.6.19; Ruby 1.9.3p194; Python 2.7.3; Java OpenJDK 23.7-b01; libev 1.4-2; Intel TBB 4.0+r233-1; gevent 0.13.7; REV 0.3.2.

4.1 Echo Server

Facilitating the developing high-performance web applications is one of the goals of Eve. This benchmark compares Eve to other languages and frameworks used for this purpose. The test consists of creating a server that accepts TCP connections and re-emits the received data until the socket is closed. Although very simple, this test enables the comparison of key features of web servers. The first measured attribute is the request throughput. This indicates the number of requests per second the server can handle. The second measured attribute is latency. Low latency times are critical for soft real-time applications. Additionally, even for other applications, latency higher than 100ms is noticeable and has been linked to lower user dissatisfaction, higher bounce rates and overall lower revenue [9].

For this benchmark, the following solutions were tested: `eve`, `erlang`, `haskell`, `go` and `Node.js` are implementations of an echo server using the respective languages, `rev` is an implementation using the Ruby Event Machine platform, `gevent` and `libev` make use of the homonymous libraries (for python and C++ respectively), and finally `cluster` is a Node.js application that uses the cluster library for parallelism. The source code for each application was selected from an existing benchmark, publicly available at `https://github.com/methane/echoserver`. However, this benchmark suite does contain the cluster implementation. Additionally, the client software used the thread-per-connection model which delivered low performance. A new implementation based on this code was created using the Eve runtime. For each test, 150 concurrent connections were created, each sending 10000 sixteen byte messages.

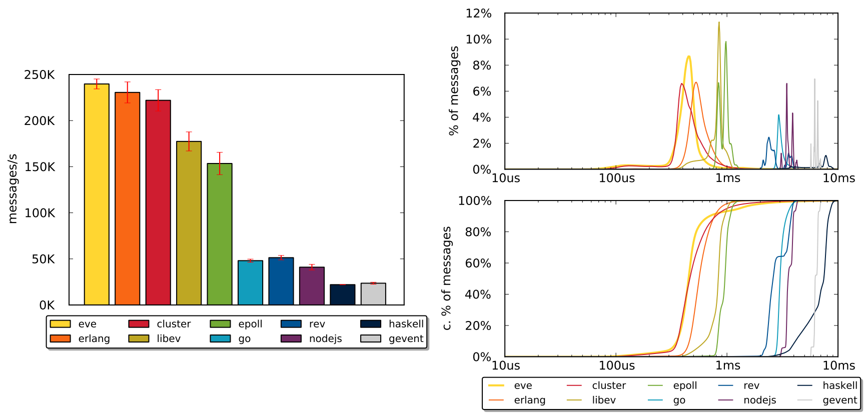

Fig. 2. Request throughput of the echo servers on the left, and reply latency on the right.

Figure 2 indicate the performance obtained using these solutions. The `gevent` implementation is the slowest of all alternatives. This is most likely because the entire framework is executed by the python interpreter which is inherently slower than a native implementation. This is also true for `Node.js` and `rev`, although not to the same extent (the event loop is implemented in native code). Additionally, `gevent` uses the *libevent* library while `Node.js` uses *libuv*and `rev` uses a custom implementation. According to [10], *libev* outperforms *libevent*/*libevent2* which also reinforces the poor performance of `gevent`. The `epoll` implementation makes direct access to the `epoll()` system call using C++ while `libev` uses the wrapper library around `epoll()` on UNIX systems. As expected, their performance is much better than the already mentioned solutions. In fact, `libev` alone is 3.88 times faster than `Node.js`. However, in our case, the `epoll` implementation is slightly worse than `libev`. This is because the benchmarked `epoll` code is poorly optimized, making use of unnecessary memory allocations that are not present in `libev`.

All the remaining solutions make use of multi-threaded runtime environments and were expected to outperform the single-threaded implementations. This is not true for `haskell` and `go`. Regarding the first case, the `haskell` runtime has known IO scalability issues. According to [11] this will be fixed in GHC version 7.8.1, which has not yet been released. The reason behind `go`'s poor performance is more obscure since documentation of its runtime architecture is not available. Both the `erlang` runtime and `cluster` implementation show good performance. Nonetheless, the `eve` framework surpasses both with a 35.5% increase in throughput on localhost, and a more modest 3% increase compared to `erlang` and 7% increase compared to `cluster` on different hosts. One interpretation of these values is that the Eve runtime is more optimized and/or requires less operations. In fact, the `erlang` language was designed for real-time systems and each actor is scheduled using a preemptive fair algorithm. Even if no preemption occurs, this algorithm is more expensive than the execution of eve tasks. Regardless, even though the Eve runtime provides less guarantees on the response time, the obtained latency and jitter are comparable if not better than `erlang`'s.

Considering the `libev` implementation as a baseline for a single-threaded runtime, one would expect the performance of parallel implementations to achieve better speedups. Two reasons were found that can explain this fact. The first is the very nature of the problem. Unlike CPU intensive tests, the echo server test is IO intensive. In particular, `read/write()` operations require large memory bandwidth, which unfortunately does not scale with added worker threads. To mitigate this bottleneck zero-copy operations could be implemented [12]. The second has to deal with normal parallel slowdown causes. Problems such as cache misses aggravate the memory bandwidth bottleneck and are more common in parallel architectures due to inter-process invalidation [13]. Additionally, synchronization is required to maintain a coherent application state. This synchronization is employed by the Eve runtime (using spinlocks, monitors and atomic operations), but also by the Linux kernel since spinlocks and mutexes are used in epoll functions to prevent race-conditions. Even in the absence of concurrent accesses, these primitives incur in additional overhead that is not present in single threaded architectures. Additionally, this overhead may increase when used simultaneously by more threads.

4.2 Atomic Counter

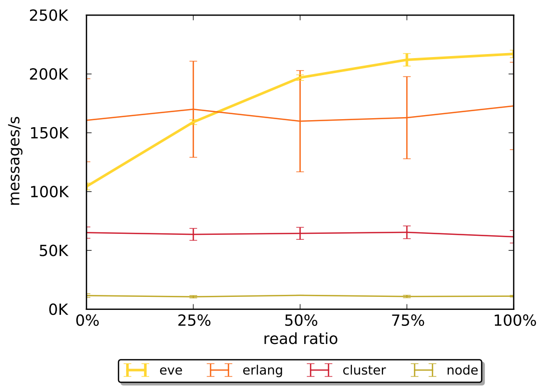

Fig. 3. Throughput of the atomic counter servers subject to the percentage of read operations.

The echo test described in the previous section exemplifies an embarrassingly parallel problem. There is no shared state between clients, which allows them to be handled separately without synchronization. The atomic counter test is a modification to this example, where shared state is maintained. In particular, a single variable `counter` is accessed by all clients. Two types of operations are permitted: *read* which allows each client to retrieve the value stored in `counter` and *increment* which atomically reads and increases stored value by one and returns the new value. These operations are transmitted through the network using a single byte `00` and `01` respectively. Because both operations require a response packet, this application has similar IO patterns to the echo server, allowing the previous results to estimate an upper limit on performance.

Ideally, `read` operations can execute in parallel but each `increment` operation must be executed in mutual exclusion. The Eve implementation makes use of the language's access permissions to achieve this semantics. Erlang however, does not have this feature. For this reason, the counter is maintained by a single actor, using message passing for synchronization. This solution sequentializes all accesses, including `read` operations. The remaining actions are still executed in parallel (e.g.: IO, parsing). With Node.js `cluster` library, each worker executes in a new process. For this reason, shared memory solutions are not possible. For testing purposes we decided to approach this limitation with a commonly used alternative: in-memory databases. In particular we selected mongodb which suports the required atomic operations. Figure 3 shows the throughput obtained for each server. The `erlang` implementation suffers from performance loss, averaging at 74%. The large standard deviation observed for this test is very high, ranging from 20.99% to 27.04%.

The proportion of `read` operations are key to the performance of the Eve runtime. On one end, with 100% `read` operations the `counter` value is constant and complete parallelization is possible. The performance obtained for this case is around 90% of the expected value, indicating that the overhead of additional synchronization is low. For this ratio, Eve outperforms the erlang by 25%. On the other end, with 0% `read` operations, each action must wait for its predecessor to relinquish access to the shared variable. In this case, the performance drops to 43.5%, being slower than `erlang` by 35%. The other implementations do not suffer significantly from this ratio: `erlang`'s implementation sequentializes every operation and mongodb uses atomic operations instead.

5 Related Work

In this section we will focus on comparing Eve with other approaches that combine the event oriented aspect with shared memory multithreading. As previously mentioned, the Actor implementation in Scala[5]. Scala actors have two possible behaviors for processing an incoming message, one with threading semantics and other with the same semantics of event-based programming. The second approach is based on continuations and allows for parts of the message processing to be scheduled for a later time, without having to suspend the thread. While this approach has good performance results on the actor model, Eve allows for more flexible and complex programs, given that memory is fully shared, and not partitioned by actor.

Capriccio[14] has a threading implementation that takes advantage of asynchronous IO. Capriccio is implemented using user-level threads on top of coroutines. In terms of performance, Capriccio is always slower than epoll, something that does not occur in our micro-benchmarks. Eve is a full programming language, while Capriccio is a library that implements the POSIX threading API, which allows for usage in existing applications without much work. On the other hand, the language features of Eve allow for more information regarding shared objects, automatically synchronizing accesses, something not possible in Capriccio.

Events and Threads have also been combined in GHC[15]. The main difference to Capriccio is that there was an explicit asynchronous IO API with event handlers, like in Eve. However, the threading API, including synchronization using mutexes, is explicit unlike in Eve. Furthermore, the GHC makes use of Software Transactional Memory, while Eve does not. Another work on GHC[16] has also improved the performance of asynchronous IO on multithreaded environments by improving the data structures on which the event handlers are stored, but in multicore environments, each event source is attached to a single thread.

Libasync-smp[17] allows event handlers to be executed in parallel, as long as they do not share any mutable state. This is done by assigning a tag (”color”) to events according to the shared state they use in their computations. Thus, events of different colors can be executed in parallel without any extra synchronization. This approach is more similar to Eve, but less expressive as Eve requires information regarding the variables and automatically detects the events that can execute in parallel. In Libasync-smp programmers must express that using colors, and fine-grained synchronization using shared variables is not supported. Instead all events that shared memory, even if only in a small part of the handler, execute serially.

Mely[18] uses the same API as Libasync-smp, but uses workstealing to lower scheduling overheads to improve performance with short-running events. The workers steal colors instead of tasks, in order to maintain the serial execution inside each color. Although this is close to the implementation of Eve, the same drawbacks of using colored events instead of annotating variables applies to Mely.

Finally, Eve can be compared to Æminium[19] in the sense that Æminium also uses access permissions on variables to automatically manage synchronization between different running tasks. While Æminium automatically parallelizes the whole code based on the access permissions, Eve uses programmer annotations to mark parallelization points in the code. However Æminium is only concerned with CPU-bounded parallelism, without any event-oriented API.

Node.Scala[20] also shares a similar approach to Eve. Programmers write Scala applications using a single-threaded event-loop approach, with the same API as in Node.js. Event handlers can then be executed in parallel whether or not they are marked as exclusive or not. Compared to Node.Scala, Eve can parallelize more than just event callbacks, featuring a full work-stealing scheduler, more suitable for CPU-intensive tasks, while Node.Scala is optimized only for IO processing.

6 Conclusions and Future Work

Event-driven architectures have been proved to work well for network-based applications, but it has been hard to integrate asynchronous IO APIs in shared-memory multithreaded programming. This difficulty is two-folded. The expressiveness of using multithreaded programming is not directly compatible with the traditional callback-based single-threaded event-loop approach. Performance is other field in which combining these two different programming styles is not trivial, as event-loops are mostly bound to a single thread and require extra synchronization, which adds an overhead.

We propose Eve as a parallel event-oriented language, in which programmers use a event-oriented programming style and special syntax for creating new parallel tasks, and for access permissions on variables. This small extra annotations on the code allow for parallel execution of different parts of the code, as well as a guarantee of a safe parallel event callbacks execution.

Our benchmarks have shown Eve to have a similar performance as Intel TBB and Java ForkJoin frameworks in CPU-bounded programs. Additional, Eve outperformed other languages in IO-bounded programs by making a more efficient use of threads in event-based programming. A Localhost share-nothing application had a 35.5% improvement over the second best solution, and server-only execution had a 7% increase. Another IO application with some 50% of the requests requiring synchronization was 23% faster than the next best solution.

For future work, it would be important to improve the performance of epoll in a multithread environment. The epoll set is currently shared by all workers, causing synchronization to happen at the kernel level. It would be interesting to have a epoll set per worker, in order to minimize contention, with an extra global set for load-balancing.

Acknowledgments

This work was partially supported by the Portuguese Research Agency FCT, through CISUC (R&D Unit 326/97), the `CMU|Portugal` program (R&D Project Aeminium CMU-PT/SE/0038/2008), the iCIS project (CENTRO-07-ST24-FEDER-002003), co-financed by QREN, in the scope of the Mais Centro Program and European Union's FEDER and by the COST framework, under Actions IC0804 and IC0906. The first author was also supported by the Portuguese National Foundation for Science and Technology (FCT) through a Doctoral Grant (SFRH/BD/84448/2012).

References

1. Dabek, F., Zeldovich, N., Kaashoek, F., Mazieres, D., Morris, R.: Event-driven programming for robust software. In: Proceedings of the 10th workshop on ACM SIGOPS European workshop, ACM (2002) 186–189
2. Krohn, M.N., Kohler, E., Kaashoek, M.F.: Events can make sense. In: USENIX Annual Technical Conference. (2007) 87–100
3. Ousterhout, J.: Why threads are a bad idea (for most purposes). In: Presentation given at the 1996 Usenix Annual Technical Conference. Volume 5. (1996)

4. von Behren, J.R., Condit, J., Brewer, E.A.: Why events are a bad idea (for high-concurrency servers). In: HotOS. (2003) 19–24
5. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science **410**(2) (2009) 202–220
6. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Volume 30. ACM (1995)
7. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. SIGPLAN Not. **33**(5) (May 1998) 212–223
8. Kumar, V., Frampton, D., Blackburn, S.M., Grove, D., Tardieu, O.: Work-stealing without the baggage. SIGPLAN Not. **47**(10) (October 2012) 297–314
9. Hamilton, J.: The cost of latency. URL: http://perspectives. mvdirona. com/2009/10/31/TheCostOfLatency. asp (2009)
10. Lehmann, M.A.: Benchmarking libevent against libev. `http://libev.schmorp.de/bench.html/` (2011) [Online; accessed 31-Aug-2013].
11. Voellmy, A., Wang, J., Hudak, P., Yamamoto, K.: Mio: A high-performance multicore io manager for ghc
12. Thadani, M.N., Khalidi, Y.A.: An efficient zero-copy I/O framework for UNIX. Citeseer (1995)
13. Eggers, S.J., Katz, R.H.: The effect of sharing on the cache and bus performance of parallel programs. SIGARCH Comput. Archit. News **17**(2) (April 1989) 257–270
14. Von Behren, R., Condit, J., Zhou, F., Necula, G.C., Brewer, E.: Capriccio: scalable threads for internet services. ACM SIGOPS Operating Systems Review **37**(5) (2003) 268–281
15. Li, P., Zdancewic, S.: Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. ACM SIGPLAN Notices **42**(6) (2007) 189–199
16. O'Sullivan, B., Tibell, J.: Scalable i/o event handling for ghc. In: ACM Sigplan Notices. Volume 45., ACM (2010) 103–108
17. Zeldovich, N., Yip, A., Dabek, F., Morris, R., Mazieres, D., Kaashoek, M.F.: Multiprocessor support for event-driven programs. In: USENIX Annual Technical Conference, General Track. (2003) 239–252
18. Gaud, F., Geneves, S., Lachaize, R., Lepers, B., Mottet, F., Muller, G., Quéma, V.: Efficient workstealing for multicore event-driven systems. In: Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on, IEEE (2010) 516–525
19. Stork, S., Marques, P., Aldrich, J.: Concurrency by default: using permissions to express dataflow in stateful programs. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, ACM (2009) 933–940
20. Bonetta, D., Ansaloni, D., Peternier, A., Pautasso, C., Binder, W.: Node. scala: implicit parallel programming for high-performance web services. In: Euro-Par 2012 Parallel Processing. Springer (2012) 626–637