Node.js Asynchronous Compute-Bound Multithreading

Nicholas Kelly

***Abstract*—We propose a system for asynchronous multithreading in Node.js. By using annotations and extending V8 and Node.js, common asynchronous paradigms can be accelerated past the single-threaded model of Node.js. Our system extends the ECMA 5.1 directive syntax to allow for compute-bound annotations. Additionally, V8 is modified to allow Node.js to run a function in a separate V8 instance (Isolate). Upon completion of the compute-bound function, a callback is executed within the main event loop of Node.js. In our initial results, we recorded at least a 3x improvement (6x maximum) in speedup over a grouping of V8 and custom compute-bound benchmarks on a 4-core Intel Core-i5 system.**

I. Introduction

The Internet has moved into a new era of responsive web applications. These are applications which require fast client and server-side response to user interactions. On the client-side, Javascript and HTML5 APIs facilitate the GUI transitions, graphics, and other features. However, nearly all of the interactions have server data associated with them, meaning that the server response must be quick, possibly over a large user-base. Since more Comet (e.g. AJAX) server interactions are being used, requests are often short, sending database data with little processing. In this case, backends such as Node.js are becoming popular. Node.js utilizes asynchronous I/O in an event-driven manner, allowing for a single-threaded web server to have better response times than common threadpool based solutions. Additionally, Node.js uses Googles V8 Javascript engine to achieve good server-side performance while allowing for an entire ecosystem of existing Javascript modules/applications.

However, data-mining and other compute-bound tasks (e.g. clustering, graph analysis, etc) are becoming widespread, especially in websites with a social component. While map-reduce solutions are widespread, many sites still use Node.js as their web backend. Since Node.js is only non-blocking for I/O-bound tasks, compute-bound tasks block other HTTP requests from being fulfilled (especially if they have a compute-bound component as well). Thus, in this paper we propose a method of extending event-driven model of Node.js to include compute-bound tasks. It utilizes a function callback similar to the built-in Node.js asynchronous I/O methods. Additionally, it extends the ECMAScript 5.1 standard [1] to include a "use cpu_bound" directive which is used to annotate which methods are compute-bound.

II. Background

A. Node.js

Node.js [2] supports Asynchronous I/O in javascript using an event-driven model. Essentially, Node.js is composed of three main parts: Node.js C++/Javascript APIs, the event loop, and V8. The Node.js C++ and Javascript APIs expose missing backend functionality to developers such as file I/O, networking, cryptography, and more. Many of these APIs are essentially Javascript bindings for C++ functions, allowing for high performance for some tasks such as cryptography. Node.js uses the high-performance V8 Javascript engine (also used in Google Chrome) for Javascript parsing, compilation, and binding. This effectively allows developers to use Javascript modules or libraries from client-side applications or NPM (Node Package Manager). For the event loop, the creators of Node.js (Joylet) have created a custom library (libuv) for asynchronous I/O. The event loop essentially keeps a queue of I/O tasks which are still "running". In the event loop, each task is monitored, waiting for new events. Upon the completion of a task, an event is fired which posts a callback function to be called within the event loop. Since the event loop is serial, if multiple events occur at the same time, or a callback from another task is still being processed, the events/tasks are handled serially (blocking). Thus, if a callback takes a long time to complete, it can hold up the whole event loop.

B. Libuv

As discussed, libuv [3] is the event loop and asynchronous I/O library used in Node.js (and other applications). It features asynchronous file I/O, TCP/UDP sockets, and DNS resolution. Even though the event loop is single-threaded, tasks or I/O is essentially run in a threadpool since most OSs do not handle asynchronous I/O well (e.g. a non-blocking read will still require a "spinning" thread). Thus, libuv additionally supports many threading features including threadpools, work queues, synchronization, and IPC. More importantly, libuv supports worker-threads which allows a function to be added to a work queue (event loop) and a callback to be associated with it. Using pthreads, libuv will create a new thread for the function and upon completion, similar to the asynchronous I/O tasks, a callback is posted to the event loop. This effectively allows for seamless introduction of additional (i.e. compute-bound) tasks into the libuv event loop.

C. V8 Javascript Engine

V8 [4] is the Javascript engine created by Google for their popular web-browser, Chrome. Since it is a C++ application, it allows for developers to embed V8 into their current project, as Node.js did. Unlike many Just-in-time (JIT) compilers, V8 does not use interpretation. Instead, it has multiple levels of code generation. The first (FullCodeGen) quickly generates native code without any optimizations. When a function becomes hot enough, V8s optimizing compiler (either Crankshaft or TurboFan) is used. Profiling comes from inline caches (ICs) and profiling ticks. Inline caches are used to help support the dynamic and prototype-based nature of Javascript. Since type information can change at anytime, V8 needs to be responsive to changes but still be able to optimize code sections for the common-case. Thus, ICs keep track of variable type consistency, as well as other information such as hidden classes (optimized classes). Additionally, profiling information is gathered in the granularity of ticks, effectively keeping track of relative function execution. Crankshaft uses IC and profiling data along with heuristics to make decisions on what functions to optimize. Functions which are executed often and have unchanging variable types are prime candidates for optimization.

III. Related Work

There are some examples of previous work which implement multithreading in an asynchronous manner. One example is the Node.js module Threads gogo [5] which implements asynchronous threading in Node.js. Similar to our solution, it utilizes separate Isolates for each thread and compiles the passed function with a registered callback. However, their solution does not utilize the existing libuv event loop and threadpool. Instead, they use pthreads directly and define their own threadpool. Additionally, the module must be included with require, the module initialized, and the threaded function explicitly defined in eval syntax (as a string). Thus, our method allows for quick addition into existing codebase (due to V8 support) as well as the possibility for profiling to determine which functions are compute-bound.

Additionally, this work is similar to the native Node.js cluster module. The module is similar to a load balancer where multiple Node.js processes are spawned for a single server. A master process accepts each HTTP request and forwards it to a certain Node.js cluster process (depending on load balancing policies). However, each Node.js cluster process has its own event loop since it is essentially a separate Node.js instance. While the cluster module could possibly have a faster response time than our initial implementation, it has higher memory usage and less potential for IPC. A similar solution is nginx [6] which is a web-server as well as a top performing load balancer. It additional has the possibility to be the master process for the Node.js cluster module.

In general there are other automatic multithreading implementations, HELIX [7] is an example. Campanoni, et al. implemented a system for automatically utilizing the inherent parallelism in programs. For example, HELIX parallelizes each iteration of a loop, handling dependencies through communication and helper threads to prefetch values. Of course, not every thread can be adequately parallelized; thus, HELIX defines a set of heuristics which are used to dynamically select loops to parallelize. Additionally, Auslander, et al. [8] introduce a dynamic compilation method which uses annotations and templates. This is effectively similar to our method of using annotation to flag compute-bound functions. Furthermore, their templates could be similar to the asynchronous (callback) function paradigm in Javascript which is utilized in our method.

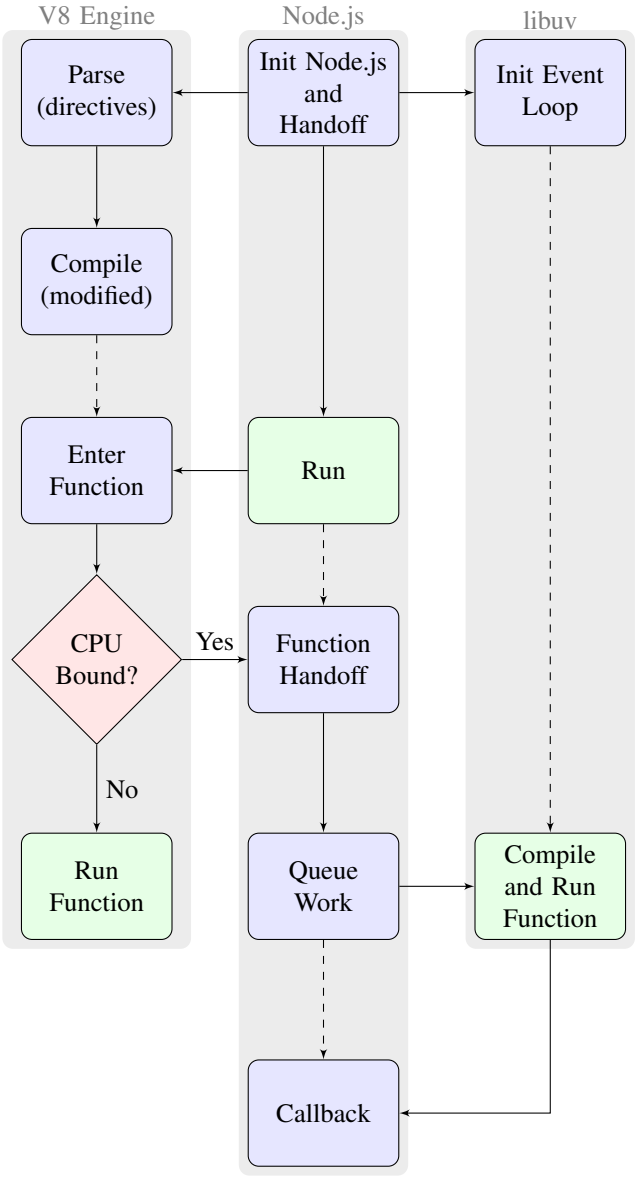

Fig. 1: Asychronous Multithreading Process

IV. Methods

A. Overall System

The overall system consists of modifications and usage of Node.js, V8, and libuv. Modifications to V8 were done in order to support directive parsing (annotations) and "function-handoff." Many of the modifications and additions were done in Node.js to add the ability to launch libuv worker threads in

the main event loop. As seen in Figure 1, Node.js facilitates where functions/scripts are run (main thread or threadpool). V8 is used for parsing, compiling, and running Javascript functions and scripts. Additionally, libuv is used to schedule worker threads in a threadpool and post callbacks to Node.js.

B. Annotations

In order to multithread compute-bound functions, the system must first know which functions are compute-bound. We decided to initially support an annotation-based method, since developers have a good understanding of the run times of functions in their application. Additionally, we could have had a profiling-based approach, where each function (or flagged function) is profiled during runtime. This approach is discussed in greater detail in future work. Since EMCAScript 5.1 introduced the support for directives (namely "use strict"), we decided to use the same syntax (as asm.js did). Thus, we proposed to use "use cpu_bound" which the developer puts in the function-scope that they intend to allow multithreading. This directive is parsed in V8 and propagates to all major function objects, allowing for usage in multiple areas post-parse time.

C. Function Interruption and Handoff

In order to support asynchronous multithreading, a function needs to be "handed off" or redirected to Node.js. We take this approach due to the limited multithreading capabilities of V8. While V8 supports multithreading, it only supports one thread in one Isolate at a time. In general, V8 contains:

- **Isolates:** an isolated instance of the V8 engine, including garbage collection, code, c ntext, and other states
- **Contexts:** an execution context for objects and functions associated with an Isolate
- **Handles:** an object reference which is associated with an Isolate or Context
- **Scripts:** compiled scripts, often associated with a Context
- **Locker/Unlocker:** used to lock and unlock Isolates for multithreading purposes

Additionally, there seems to be some initial support for queuing functions to be run in an Isolate and supporting a callback, but it is experimental. Thus, due to the above issues, the threading is not done within V8 internally, but instead facilitated by Node.js and libuv. However, this means that V8 must halt upon reaching a compute-bound function and redirect the execution to Node.js.

V8 as well as many other JIT compilers use a method of code discovery known as lazy-parsing and lazy-compiling (unless disabled). Effectively, this means that V8 only parses and compiles functions when they are needed (i.e. upon calling). Thus, a function call in the global scope might not be known to be compute-bound when it is initially parsed. Only when it is called and its function is parsed/compiled is the annotation known. Because of this limitation, function handoff must occur at the beginning of the function. The advantage to this method is there is less code expansion, since handoff code only needs to be inserted once for each function. Essentially,

```
function:
  ; Function headers
  ; Push stack frame
  jz not_cpu_bound
  call FunctionHandoff
  jmp return
not_cpu_bound:
  ; Function implementation
return:
  ret
  ; Function codestub
```

Listing 1: Function macro-assembly

```
function function_name(<args...>, callback_function)
{
  "use cpu_bound";
  // Code
  return callback_function(<args...>);
}
```

Listing 2: Asynchronous function format

the handoff contains some modifications to the V8 macro-assembler. These modifications are outlined in Listing 1. If the function is compute-bound it jumps into the runtime, hands off the function information to Node.js, and then skips the function by skipping to the end. As outlined in the next section, the asynchronous function format does return any data, but instead pass it on to the callback function. Upon inspection, it seems that V8 only creates one return statement per function. If there are multiple statements, the control flow simply jumps to the postamble. Thus, we can skip a function be simply skipping to the return statement.

The runtime function is used to pass the function data from V8 to Node.js through a specified handoff function associated with the Isolate. Function data, including arguments, function code, the name of the function, and its binding, is gathered from the stack frame and passed to Node.js. After, Node.js adds the function to libuv queue, the runtime function returns and the main execution continues (skipping over the function). Conversely, if the function was not compute-bound, it would simply run normally in the main execution context.

D. Asynchronous Compute Functions

When function data is handed off to Node.js, a new "ComputeThread" is created for the worker which includes cloning some of the data as well as creating a new Isolate and Context for the function to be run in. Unfortunately, due to the way handles and data are associated with Isolates, some raw data must be copied directly so it can be used at runtime. In this case, the function code is copied to the new isolate and a function call string is constructed from the arguments and function name. The format in Listing 2 is what is assumed for an asynchronous compute-bound function.

Thus, it is assumed that the last argument which is a function is a callback. After all data is stored, the run function is queued in the libuv worker queue (event loop). When a free thread is available, the run function is called which initializes the thread for execution. The Isolate and Context

```
var __return__;
(function() {
  function __callback__() {
    __return__ = { recv: this, args: arguments };
  }
  function_name(<args...>, __callback__);
  return __return__;
})();
```

Listing 3: Function wrapper

are initialized and locked to the thread, and the function script(s) are compiled and run. In order to simplify running and gathering results, the following container function outlined in Listing 3 is used.

Instead of the original callback function being used, it is substituted for "__callback__" which captures the scope/binding of the callback and its arguments. Thus, when the function is executed in the V8 Isolate, the data is returned without V8 modifications. Upon completion, libuv allows a callback/after function to be called, which in this case will execute the actual Javascript callback function with the returned data.

E. Callbacks

When the callback/after function for the libuv worker has been called, it has effectively joined the main-thread of execution in the event loop. At this point, existing callback functions in Node.js are used in order to execute the Javascript callback function on the main Isolate with the received data from the worker. After the callback is complete, all the worker data, including the Isolate, Context, and cloned data, is deleted and collected.

V. Results

In order to gather performance results, several V8 benchmarks (rsa_en, delta, richards, and ray) were wrapped in the asynchronous function format expected by the system (with the "use cpu_bound" annotation inserted). Additionally, two custom benchmarks were created (fib and mapreduce) which are essentially compute-bound (fibonacci and mapreduce formulas, respectively). Each workload essentially calls the corresponding benchmark 16 times serially in with callbacks given for each. These benchmarks were run on a 4-core, Core-i5 (Sandy Bridge) system with varying amounts of threads for the libuv threadpool. The results are shown in Figure 2. As expected, for asynchronous (parallelizable) workloads perform very well on the modified system with asynchronous multithreading for compute-bound tasks. All workloads received at least 3x performance improvement when compared to the baseline (unmodified Node.js). Some workloads even see up to 6x speedup. However, since the system under test is a 4-core system (without hyperthreading), greater than 4 threads yields no additional improvement. However, it would be expected to yield significant gains on systems with more cores.

Additionally, the same benchmarks were wrapped into a simple HTTP web-server in order to test the performance (response time) of the system in a more typical Node.js use case. These tests use the built-in HTTP functionality

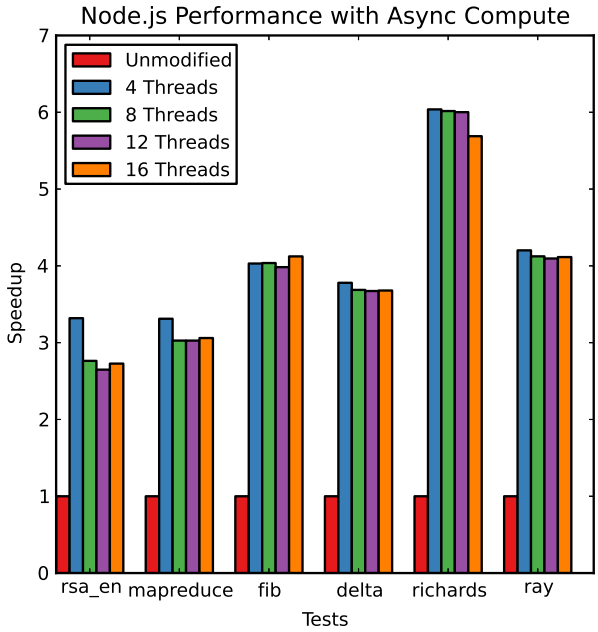

Fig. 2: V8 and Custom Benchmarks

of Node.js which is asynchronous in terms of data transfer, but not data generation. Since most websites online are not static (e.g. static HTML file) but instead dynamic (e.g. PHP/Python/Javascript), this is a very important test to identify practicality. In order to stress the HTTP servers, Apache Bench was used to make HTTP (GET) requests. Figure 3 shows the overall results of the HTTP tests in terms of the response time speedup of 90% of requests (nearly worst case). In each test, 1000 total requests are made in increments of 1, 25, 50, 75, and 100 concurrent requests. Concurrent requests would represent multiple users accessing the website at the same time (which can be in the millions for large sites). While not indicated in the initial graphs, there is some inherent overhead in the current implementation. Additional time and memory is expended in order to copy over data, compile the asynchronous function, and transfer results. Induced overhead can be seen for rsa_en, mapreduce, and fib for 1 concurrent request. Since the this effectively forces serialization, it reveals the overhead in the multithreading strategy. Interestingly, there is still performance improvement for the other workloads, which may be due to the Node.js HTTP library or more architectural reasons (e.g. local L1 caches). Otherwise, for multiple concurrent requests, the performance improvement is similar to the previous results, yielding at least 3x performance improvement. As expected, little improvement occurs over 25 concurrent requests since it is over 4 threads (default for all tests).

In terms of memory usage, it is expected that the asynchronous multithreading improvement will increase memory usage. Primarily, this effect is due to creating multiple Isolates each with their own code and context. Thus, the peak memory for the process depends on the execution rate the asynchronous functions and how long each Isolate is resident in memory.

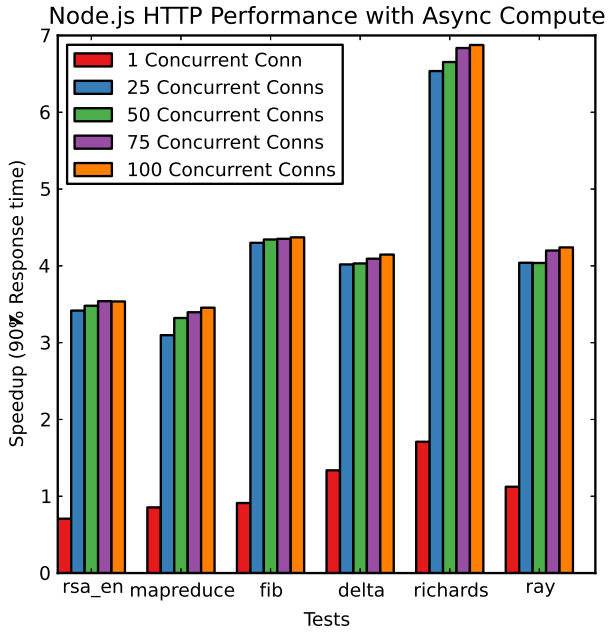

Fig. 3: HTTP-Based Benchmarks (Response speedup)

Figure 4 shows the relative peak memory usage for each workload over multiple threads. As expected, there is additional memory usage over the unmodified Node.js. However, the memory does not scale linearly for the amount of threads used which is probably due to the time each thread is resident.

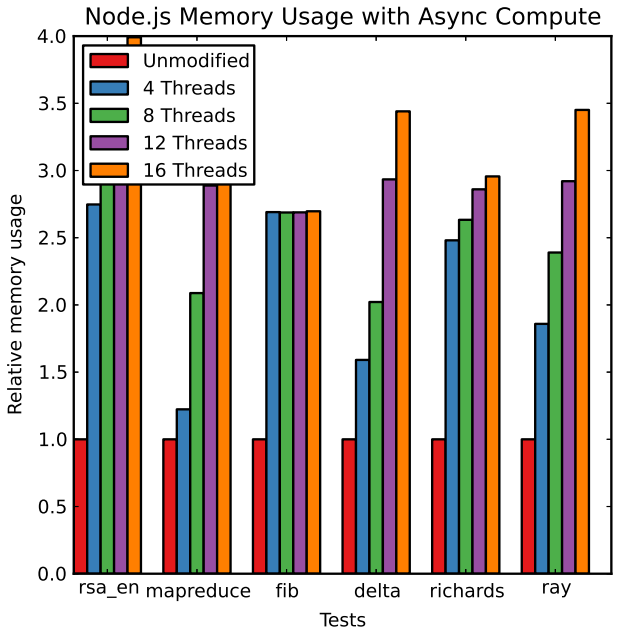

Fig. 4: Memory Usage

VI. Future Work

Currently, there are some limitations with the initial implementation. Firstly, only code within the function bounds can be called. Helper functions included with "require()" or at higher scope cannot be called since only the function code is compiled in a new Isolate. In order to solve this, a pool of Isolates could be created which contains compiled/parsed code from the main program. Unfortunately, due to lazy-compilation, the code needs to be run as well in order to propagate into the Context. Additionally, issues would arise when Isolates get out of synchronization, meaning discovered code is different than the main Isolate or other Isolates. If lazy compilation can be disabled at the Isolate-level or script-level it could fix code discovery issues.

Moving objects across Isolates is currently unsupported in V8. While it seems that Handles associated with another Isolate can be used in the main Isolate, the other Isolate cannot be running in a thread while the variable are accessed (which conflicts with the above solution). Additionally, some objects may be Isolate-specific. For example, a function object will contain different pointers, and possibly even slightly different data than its equivalent on another Isolate. Thus, some objects cannot be passed (without synchronization or locking) or copied, making it incredibly difficult to support a wide range of objects for both input and output. Currently, input arguments are converted to string equivalents (limited on type and depth), and output arguments are sourced from the thread Isolate. Another important object which is not currently included is the binding of the function (i.e. "this"). Unfortunately it suffers from the same issues as above, especially since can potentially be a large object. Additionally, since it potentially points to function definitions it cannot easily be copied to another Isolate. However, it is potentially very useful for fixing the first issue since it would contain function definitions. Unfortunately, possibly the only existing method for transferring objects to other Isolates is through serialization (e.g. JSON). Although, it becomes cumbersome with large objects and potentially impossible.

Future work could also include using profiling to flag functions for asynchronous multithreading instead of annotations. In this case, a counter would need to be associated with each functions shared info. Upon the run of a function, the completion time would be recorded using a lightweight instruction (i.e. rdtsc in x86) or a simple runtime function. Then based upon certain heuristics, a function would be flagged as compute-bound and run in a new thread. In addition to the current annotation syntax, options to allow threading ("use try_cpu_bound") and forbid threading ("use no_cpu_bound") could be added. Furthermore, multiple levels of compute-bounding could be added in the form of separate pools or queues each with a different amount of dedicated threads. Since Node.js is based upon one event loop, and therefore one threadpool, additional pools or event loops would possibly need to be added.

VII. Conclusion

In this work, we proposed a system for asynchronous multithreading in Node.js. It uses ECMA 5.1 directives ("use cpu_bound") in order flag compute-bound functions. Then utilizing V8 and libuv, flagged functions are run in a separate

thread within the main libuv threadpool. In order to make the transition, the generated assembly was modified to trap into the VM and handoff the function to Node.js. Libuv, the main Node.js event loop, is used to create a work queue (threadpool) for each compute-bound function. After execution, a callback is executed within the Node.js event loop with the results (arguments) from the multithreaded function. In our initial results, all V8 and custom benchmarks saw at least a 3x speedup improvement in traditional and HTTP workloads with a maximum of 6x improvement. While there are some limitations in terms of input and output (arguments) with the current system, there is the possibility of improvement. In particular, arguments could be serialized and functions could be automatically flagged based upon dynamic profiling data. Additionally, since the asynchronous-function paradigm is widespread in Javascript (especially in Node.js), our solution proves to be practical for emerging, compute-bound web-services.

REFERENCES

[1] "ECMAScript Language Specification - Standard ECMA-262 5.1 Edition." Internet: http://www.ecma-international.org/ecma-262/5.1/, June 2011 [Dec. 12, 2014].

[2] "node.js" Internet: http://nodejs.org/, 2014 [Dec. 12, 2014].

[3] "libuv" Internet: https://github.com/libuv/libuv, 2014 [Dec. 12, 2014].

[4] "V8 Javascript Engine" Internet: https://code.google.com/p/v8/, 2014 [Dec. 12, 2014].

[5] "Threads gogo" Internet: https://github.com/xk/node-threads-a-gogo/, 2011 [Dec. 12, 2014].

[6] "nginx" Internet: http://nginx.org, 2014 [Dec. 12, 2014].

[7] S. Campanoni, T. Jones, G. Holloway, V.J. Reddi, G.Y. Wei, and D. Brooks. "HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing." *In Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO 2012)*, pp 8493, 2012.

[8] Auslander, J., Philipose, M., Chambers, C., Eggers, S.J., and Bershad, B.N. "Fast, effective dynamic compilation." *In Proceedings of the SIGPLAN96 Conference on Programming Language Design and Implementation (PLDI96).*, 1996.