Node.js Performance Optimization

6/8/2016

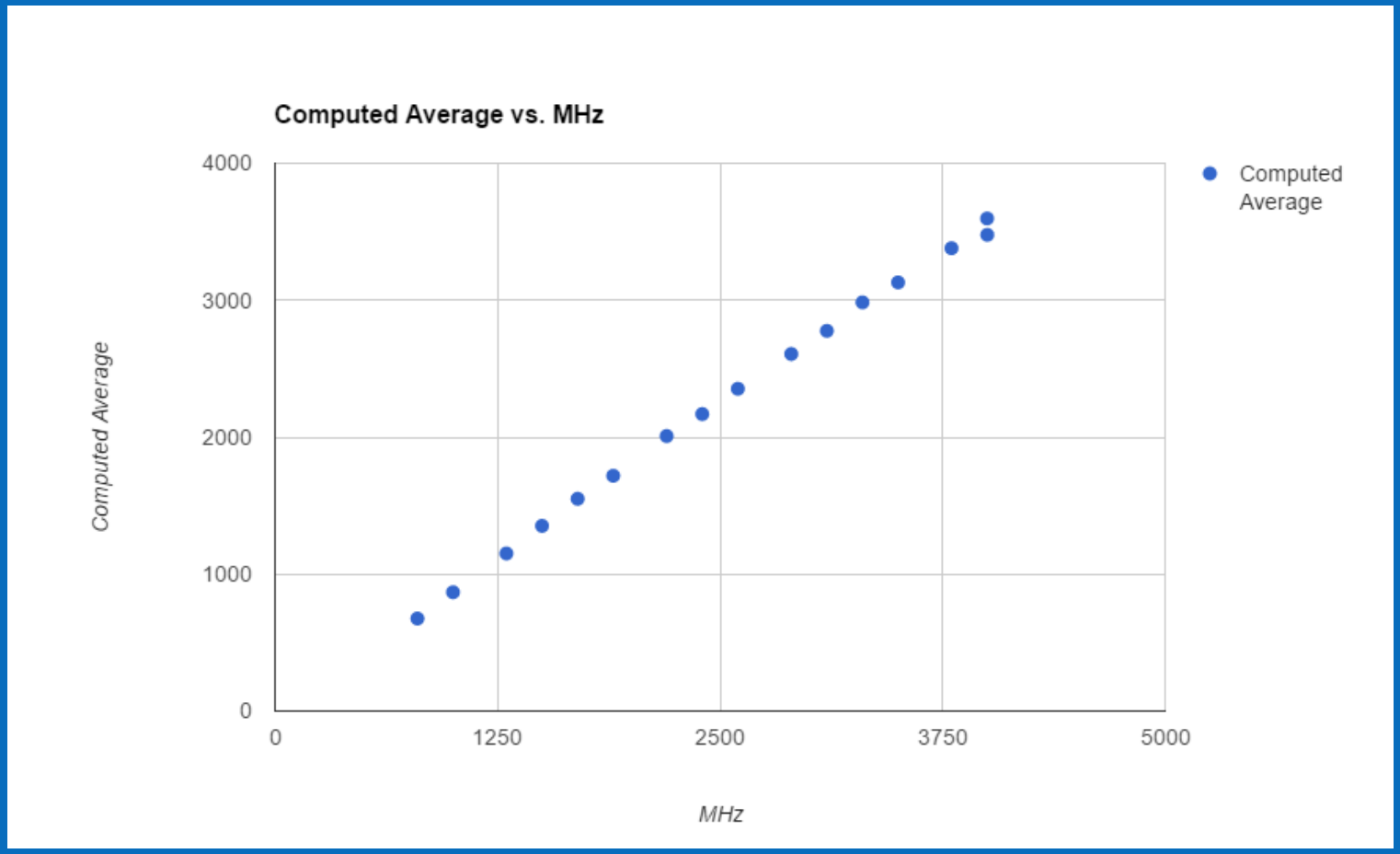
Sorin Baltateanu
Senior Software Engineer
Server Dynamic Languages Optimization Team
SSG / STO / DCST

(Intel Confidential)

Agenda

- Per frequency performance scaling
- Acmeair
  - Benchmark description and scoring
  - Reducing CoV (coefficient of variation)
  - Analysis
  - Learned lessons
- The V8 inliner
  - Learned lessons
- Other work
  - Industry case studies
  - Conferences
- Future work

Per frequency performance scaling (Acmeair)



Acmeair – Benchmark description

- Node.js benchmarking group
- Implementation of the "Acme Air" airline
  - Create users, search for flights, make bookings, etc.
  - Node.js server; MongoDB, Cloudant or Cassandra database
  - Mode for micro-service (default is monolithic)
- Relevance as a Node.js workload
  - REST-full
  - http request/response stack intensive usage
  - express and mongodb
  - caching (ttl-lru-cache module)
  - I/O intensive
  - long callback chains

Acmeair – Measuring performance

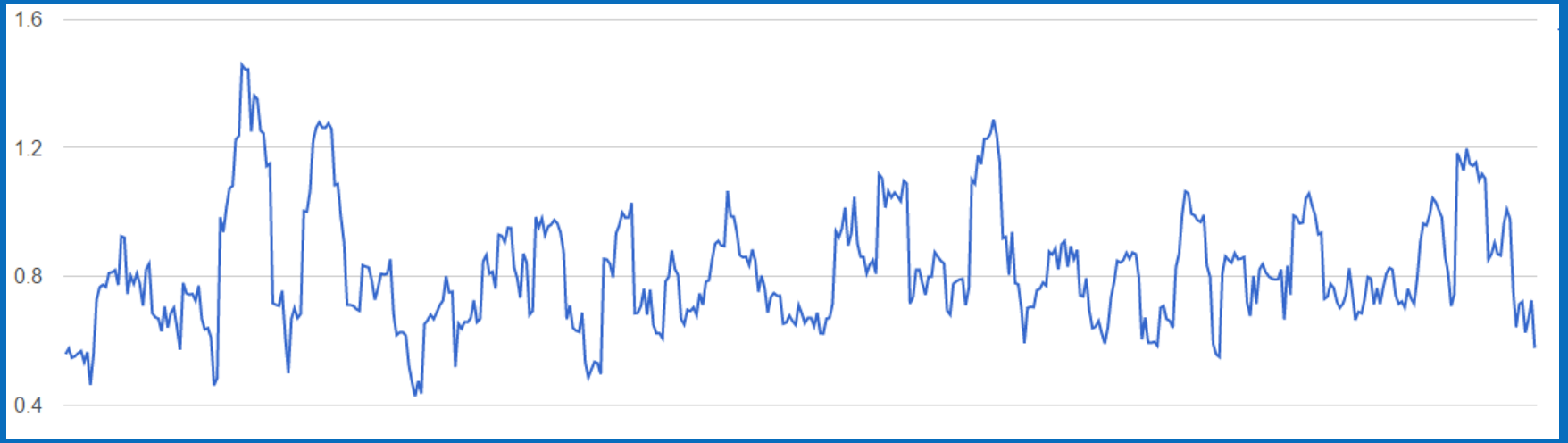
- Scoring
  - Start the Node.js http server
  - Use jmeter client to send http requests
  - Use the reported RPS (requests per second)
  - Run 10 times and average the reported number of RPS
- Jmeter configuration: the duration of the test, the number of threads, the ramp-up time, the number of iterations, the request sequence, etc.
- Jmeter client sequence
  - login
  - 25% get user info
  - 25% of time set user info
  - 5 flight queries
  - make booking if flight is available
  - 25% of time list bookings
  - cancel booking if customer has more than 2 bookings
  - logout

Acmeair – Coefficient of Variation

- Common problem: Variation between runs
- Start server and run the jmeter client multiple times. Report the average. Two types of variations:
  - Between the averaged values (run to run - R2R)
  - Between the averages (day to day – D2D)
- CoV history:
  - Initial: R2R up to 3% / D2D up to 5%
  - Intermediary: R2R up to 1% / D2D up to 1.5% (system wide configurations, event loop pinning, increase number of clients)
  - Final: R2R up to **0.3%** / D2D up to **0.3%**

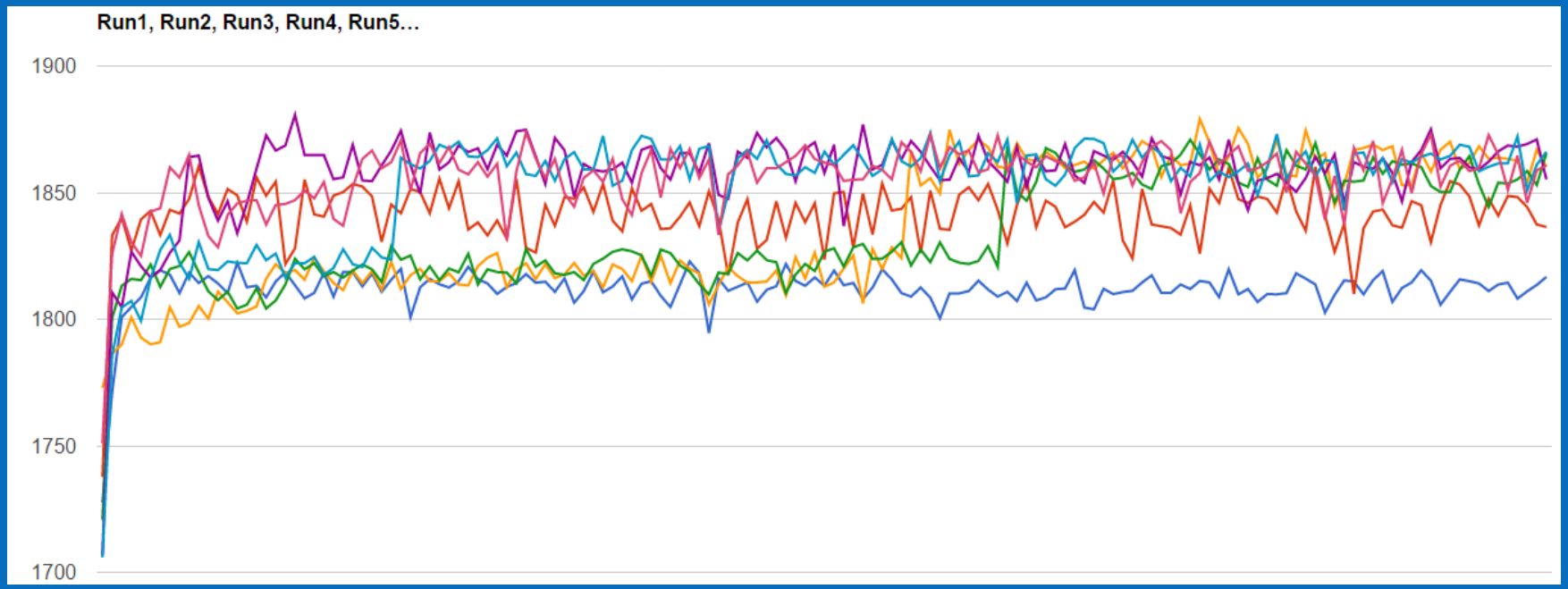
Acmeair – R2R variation

- Start server and run the jmeter client multiple times
- Use moving window (size 10)

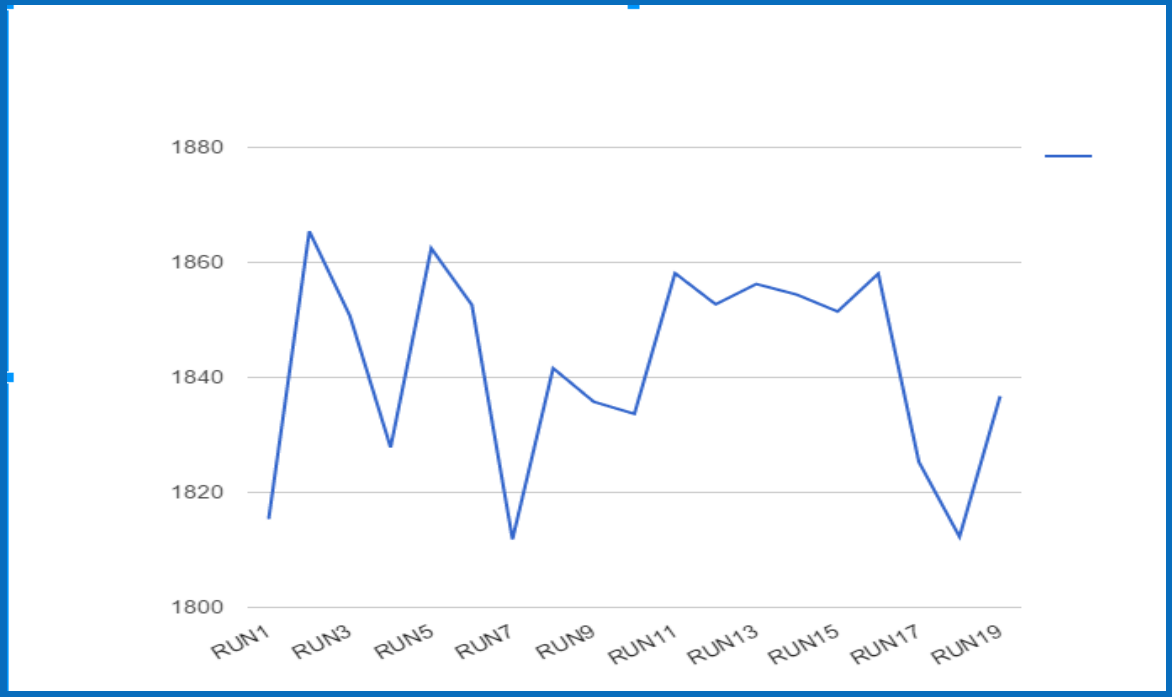


Acmeair – D2D variation

- Start Node.js server; Run jmeter multiple times; Average the RPS; Obtain a per server score; Close the Node.js server; This is what is reported in the email
- Repeat this multiple times



Acmeair – D2D variation (2)



Acmeair – Increase stability of results

- System wide configurations
- Jmeter:
  - Increase number of threads (100% CPU utilization)
  - Run for fixed duration
  - Increase duration of one jmeter run (reduce the number of runs)
  - Discard ramp-up time
  - Increase JVM heap memory
- Database:
  - Disable periodic administration tasks; run them manually before starting the benchmark
  - (also tried Mongodb Enterprise with in memory storage engine)
- Scheduling
  - Event loop, libuv, mongodb thread pinning (more on wiki)
  - Move jmeter to second socket

Acmeair – Increase stability of results(2)

- Node.js server
  - Use 10 libuv threads
  - Warm up V8 and libuv thread pool
  - Disable Acmeair logging (+10% performance)
  - Increase nice and ionice values
- End result: Both CoV (R2R and D2D) below **0.3%**

Acmeair – Lessons learned

- Reducing the CoV is specific to each workload
- Reducing the CoV needs a multi-layer approach
- Some variation is still there:
  - ttl-lru-cache has 1.6% CoV (throw statement disables optimizations)
  - The Node.js runtime generates variation (the runtime profiler is the current suspect)

Acmeair – Execution time breakup

- Nodejs 83.38%
  - event loop: 99.13%
  - libuv threads: 0.73%
  - v8 threads: 0.14%
- Mongodb 16.62 %
  - mongo maintenace threads: 1.85%
  - mongo worker threads: 98.15%

Acmeair analysis – Nodejs execution time breakup

Execution time breakup	Context switches			Percent	Execution time CoV	Context Switches CoV
356835.6735	409202		event loop	99.13710453	N/A	N/A
122.028596	141		v8 thread	0.1362832499	0.7235412068	2.647496458
121.791235	134		libuv thread	0.7266122244	0.1939635722	0.002333744064
123.69975	135					
123.02052	133					
262.084706	40652					
261.511826	40652					
262.391682	40650					
261.52499	40650					
261.498422	40650					
261.893497	40651					
261.66352	40650					
260.868309	40650					
261.134349	40652					
260.808316	40650					

Acmeair analysis – Compiler improvements

- 0day: Performance improvement of **21.66%**
- Identified V8 port from 4.9.385 to 5.0.71.32
- Nodejs 77.66% / 83.38%
  - event loop: 98.26 % / 99.13%
  - libuv threads: 1.63% / 0.73%
  - v8 threads: 0.11% / 0.14%
- Mongodb 22.34 % / 16.62 %
  - mongo maintenace threads: 1.85% / 1.85%
  - mongo worker threads: 98.15% / 98.15%

Acmeair analysis – Lessons learned

- Acmeair:
  - Acmeair is event loop bound
  - Improving the compiler is worthwhile
  - Spending more time compiling for server side workloads is worthwhile
- Libuv
  - Small opportunity to improve
  - RR balancer for libuv I/O requests. Consider “weight” based balancing of I/O operations
- MongoDB
  - Mongodb scales well. Will not be a bottleneck

Acmeair – Vtune analysis

- No hotspot identified at the moment
- Backend bound
- DTLB overhead
- Most of the time spent in
  - Writing and reading I/O buffers provided by libuv
  - Http parsing and stream management
  - Others
    - V8: string/array/buffer management, GC (including write barriers)
    - Event loop management
    - Node modules (e.g. mongo)
- Synchronization issue (1%)

Acmeair – Vtune analysis (2)

Clockticks:	250,754,494,974
Instructions Retired:	144,625,652,296
CPI Rate:	1.734

The CPI may be too high. This could be caused by issues such as memory hardware-related metrics to identify what is causing high CPI.

MUX Reliability:	0.570
Front-End Bound:	**14.7%**
Bad Speculation:	**5.2%**
Back-End Bound:	**72.1%**

Identify slots where no uOps are delivered due to a lack of required reso of the pipeline where the out-of-order scheduler dispatches ready uOps program order. Stalls due to data-cache misses or stalls due to the overlo

Memory Bound:	**57.7%**

The metric value is high. This can indicate that the significant fracti Amplifier XE Memory Access analysis to have the metric breakdown b

L1 Bound:	**0.061**

This metric shows how often machine was stalled without missin blocked on older stores, a load might suffer a high latency even t or Cycles of 1 Port Utilized issues.

DTLB Overhead:	0.162

V8 inliner - Why

- Expose server-side optimization opportunity
- Raytrace (google benchmark)
- Intensive loop (not relevant for typical Node.js application)
- Opportunity to learn about V8
- 23% performance opportunity:
  - Inline functions by hand
  - Object reuse exposed after inlining
- Result: Obtained **30%** improvement by reconfiguring the inliner

Understanding the inliner

- How it started
  - Evaluate the performance opportunity: inlining vs object reuse
  - Track GC for both the original and modified version. Nothing
  - Add flags to inline different sets of functions
  - Identified case when manually inlining just a function brought a performance drop of 10%
- How it continued
  - Trace the optimization decisions and the generated assembly code
  - Trace inlining decisions and found that some functions were not inlined anymore
- How it ended
  - Found culprit. Tune runtime parameters: 30% improvement
  - Dive into V8 and check other flags and reasons that disable inlining

V8 inliner flags

Flags that influence the inlining level:

- --max_inlined_nodes_cumulative: limits the total number of AST nodes that can be inlined
- --max_inlined_nodes: limits the number of AST nodes to be inlined at once
- --max_inlining_levels: limits the maximum depth of the inlining
- --max_inlined_source_size: limits the size of the function that can be inlined
- --inline_arguments: affects functions that use the arguments object

V8 inliner

Other functions that can’t be inlined:

- Recursive functions
- Constructors
- Functions that use rest parameters
- Functions that use unsupported syntax (e.g. try/catch)
- Builtin functions
- Functions with context-allocated variables
- Other functions (more on wiki)

Octane suite with flags

- Tune the max_inlined_nodes_cumulative, max_inlined_nodes, max_inlining_levels, max_inlined_source_size, inline_arguments flags
  - Crypto: No benefit. Small opportunity for inlining. The most expensive functions contain loops that don't have function calls inside them.
  - Delta Blue: No benefit.
  - Earley Boyer : No benefit. Most of the functions are recursive and can't be inlined.
  - NavierStokes: **+22%**. Target text too big prevented inlining.
  - Richards: **+25%**. Target text too big or cummulative ast node limit reached prevented inlining.
  - Raytrace: **+30%**. Cummulative AST node limit reached or target text too big prevented inlining.
  - Splay: No benefit. Most of the time spent in the splay_function that has no inlining opportunity.
- Note: Inlining exposes other optimizations, especially if the function call is inside a loop.

Octane Emon Data

	Raytrace	Richards	Navystokes
RPS	**30.85**	**25.47**	**22.50**
metric_CPU utilization% in kernel mode	**12.50**	**1.46**	**9.64**
metric_CPI	9.68	5.17	-14.29
metric_branch mispredict ratio	**28.20**	**22.45**	**-4.45**
ITLB, DTLB (per request)	<0.1%	<0.1%	<0.1%
INST_RETIRED.ANY (per request)	**-30.70**	**-23.97**	**-4.52**
CPU_CLK_UNHALTED.THREAD (per request)	-24.00	-20.04	-18.16
BR_INST_RETIRED.ALL_BRANCHES (per request)	-32.40	-21.30	-0.95
BR_MISP_RETIRED.ALL_BRANCHES (per request)	-13.34	-3.63	-5.36

V8 inliner

- Nodejs micro-benchmarks
  - No benefit
  - Compact code
  - Functions are called a couple of times
- Acmeair
  - No benefit
  - Most of the function calls are I/O bound and the native functions can't be inlined
  - Most of the functionality is broken up in callbacks, so there are less opportunities for inlining

V8 Inliner – Lessons learned

- Is tuned for client side workloads
- Using non-default flag values may improve performance, but this is workload specific
- Depending on the profiler, the optimization order may vary. This can generate performance variation
- How to track the generated assembly code, the optimization order, the inlining decisions and the GC

Future directions – V8 Opportunities

- V8 is tuned for client side workloads
  - Compilation is done at runtime. Code is seen for the first time
  - Latency is very important. Generate code fast
  - Do not spend a lot of time profiling or compiling. Bad decisions do not hunt you
  - Targeted to Mobile and PC. Does not use all server specific instructions

Industry case studies - Bitdefender

- About the company:
  - Security solutions world-wide
  - Uses Amazon but has also private cloud
- Node.js is used by different teams
  - Used mainly inside their cloud infrastructure for routing messages to the processor nodes. The main entry point is a Nginx server that routes the messages to dispatcher nodes, which distribute them to the registered processor nodes. The processor nodes either log the messages to a mongo database or process them using native applications
  - Used in another solution as a http server. Computing intensive tasks are performed by native applications by spawning a new process
  - Another solution that uses a MEAN stack for data processing
- Common scenario in the industry: use Node.js to route messages. Log messages in database or use native/existing applications at node level

Industry case studies - 1&1

- About the company:
  - One of the world's leading Web hosting providers
  - Cloud solutions powered by Intel. Dedicated servers
  - Bitnami client
  - More than 70,000 servers. Hosting of about 19 million domains
- Node.js powers a lot of their infrastructure
  - The first use is for the control panel they offer to their clients
    - Open sourced (rainjs)
    - Will be replaced with a new technology
  - The second use is a tool that monitors their cloud infrastructure. Node.Js is used for routing messages, filtering them and triggering alarms
  - Use custom Solaris distribution which comes with integrated tracing support for V8

Conferences

- JsCamp participation
  - Tracking HTTP requests as transactions by using AsyncWrap
  - Nodejs use as a bridge between IoT devices that use different communication protocols
  - Nodejs use in P2P system
- Conferences submissions
  - NodeJs Interactive
  - SWPC

Node.js challenges

- Node.js event-loop implications
  - Do not hog the event-loop
    - Discourages long running code (e.g loops): bad cache behaviour, disables compiler optimizations opportunities
    - Most functionalty gets broken in small chunks that are executed interleaved: bad cache behaviour, disables compiler optimization opportunities
    - Each callback needs some VM code to be executed
    - Encourages the execution of computing intensive code in native process
  - Asynchronous
    - Current context needed later: memory pressure, disables compiler optimization opportunities (e.g. Inlining)

Nodejs challenges (2)

- Javascript specific challenges
  - Dynamic code:
    - Lots of checks (branches): Out-of-order execution engine pressure
    - Polimorfic objects: hard to predict branches and hard to specialize assembly code
    - Large code footprint: instruction cache and TLB pressure
  - Language features (e.g. Number representation)
  - GC:
    - DTLB and DCache pressure (e.g. marking, compacting)
    - Overhead (e.g. references, barriers)
- Programming model challenges
  - Single-threaded
    - event loop, asynchronous programming model
    - javascript programming model
    - V8: e.g. Compilers, GC

Node.js challenges (3)

- Industry and community usages
  - I/O bound
  - Rest applications: stateless
- „Microarchitectural Implications of Event-driven Server-side Web Applications”

Future directions

- Restart Acmeair analysis
- More research on workloads relevant to the industry
  - Give feedback to NodeELS
  - Identify functionality that favors IA (e.g. Regex filtering, hashing, etc.)
- Evaluate possibility and benefits of using precompiled JS code
- Evaluate possibility to automatically detect the best configuration at runtime (e.g. V8 flags, GC generations sizes)
- Make list of backend optimizations done by V8. Implement more complex optimizations that can produce better code, but take more time to be performed (take into account competing processors)
- Dive into the profiler code and check if we can improve it. More resources can be spent for profiling if we generate better code
- Make list of popular modules. Check optimization opportunities

Future directions

- Make list of libraries that are optimized for IA (e.g. math library). Check opportunity to expose this libraries as node modules. When it comes to packages it is important who is the first, not who is the best
- Check open-source tools that do binary/assembly level analysis/optimization. Automatically identify suboptimal assembly generated code (e.g. peephole optimizations)
- Trace callbacks (e.g. using AsyncWrap). Get a feeling on how the callbacks are interleaved. Evaluate potential for callback scheduling (e.g. group callbacks to improve instruction locality) or using prefetching
- Smart code placement
- Evaluate potential of using data and instruction prefetching
- Evaluate potential for smart cache placement policies for callbacks code

Links

- https://wiki.ith.intel.com/display/dcstroautomation/Romania+DCST+Nova+Wiki