QoS-aware Dynamic Power Optimization for Data-Parallel JavaScript Programs

Young H. Oh, Sungkyunkwan University
Wonjun Lee, Samsung Electronics
Xianglan Piao, Sungkyunkwan University
Channoh Kim, Sungkyunkwan University
Jae W. Lee, Sungkyunkwan University

JavaScript has become a general-purpose programming environment that enables complex, media-rich web applications. An increasing number of JavaScript programs are parallelized to run efficiently on today's multicore CPUs, which are capable of dynamic core scaling (DCS) and voltage/frequency scaling (DVFS). However, significant power savings are still left unrealized since an operating point (in terms of the number of active cores and CPU voltage/frequency) is selected by monitoring CPU utilization or OS events, without considering the user's performance goal. To address this, we propose QPR.js, a QoS-aware power-optimizing runtime system for JavaScript. Using the QPR.js API, the application developer can specify an QoS constraint and provide a fitness function to quantify the current level of optimality. During execution the QPR.js runtime system uses this information to autonomously find an optimal operating point while satisfying this QoS constraint. Our evaluation demonstrates the effectiveness of QPR.js in finding optimal operating points while satisfying QoS constraints for three different optimization goals.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming—Power Optimal Programming; D.3.2 [Programming Languages]: Language Classifications—JavaScript

Additional Key Words and Phrases: Power optimization; DVFS; JavaScript; multi-core

1. INTRODUCTION

As more applications go online, there are strong demands for power-efficient performance of JavaScript. Many modern web browsers support HTML5, which enables sophisticated media-rich applications on the web, such as media players, 3D graphics and games. These applications are compute-intensive and consume a large amount of power, which is a serious concern, especially on mobile devices.

On the hardware front multicore CPUs have become commonplace in all scales of computing platforms to offer abundant execution resources. To efficiently utilize...
them, multiple frameworks for JavaScript parallelization have emerged, such as WebCL [WebKit-WebCL nd] and Web Workers [WebWorkers nd]. WebCL is a JavaScript binding to OpenCL to harness data parallelism in JavaScript; Web Workers allow multiple parallel contexts to execute concurrently, hence improving the responsiveness of the UI process while executing heavyweight background tasks.

Modern multicore CPUs also expose to software some knobs to improve power efficiency. Dynamic voltage/frequency scaling (DVFS) is a well-known example. Support for dynamically turning on and off cores, which is known as dynamic core scaling (DCS), is also popular. By applying DVFS and DCS, collectively noted as DVFCS, the system's power efficiency can be improved significantly. In a DVFCS-capable CPU, the number of active cores and their voltage/frequency settings constitute the CPU's operating point.

To maximize power efficiency of parallel JavaScript programs, it is required to find an optimal operating point for a given workload and performance goal. However, most of the popular DVFCS algorithms, such as Linux CPUFreq Governor [Brodowski nd] and Windows DVFS [Windows DVFS nd], only use system metrics, such as CPU utilization and event counts visible to OS, and do not take into user metrics (i.e., performance perceived by the user). This often leads to overly conservative voltage/frequency/core settings, to significantly increase the system's power and temperature with only marginal improvement, or even degradation, of user experience.

There are proposals to take user metrics into account to control DVFCS to expose additional opportunities for power savings. However, they either take a human-in-the-loop approach, requiring human intervention to quantify user satisfaction [Lin et al. 2009; Mallik et al. 2006], or infer it from UI events (e.g., touches), which may erroneously interpret the user's intention and degrade user experience [Shye et al. 2008; Woo et al. 2013; Yan et al. 2005].

This paper proposes QPR.js, an API and runtime system that enables quality-of-service (QoS) aware DVFCS for parallel JavaScript programs. Using the QPR.js API, the user can specify a QoS goal and a fitness function that quantifies the current level of QoS. The QPR.js runtime system finds an optimal operating point that maximizes the figure of merit (e.g., throughput) subject to satisfying the QoS constraint (e.g., minimum frame rate). QPR.js is implemented on Intel's Haswell-based mobile and Intel SandyBridge-based desktop platform running Linux. And we evaluated using 5 WebCL-based parallel JavaScript programs and 4 polyhedral computation application adopted from C-based languages. Compared with the default Ondemand governor for DVFCS control, QPR.js reduces power consumption by 59.0% on average while satisfying the QoS goal specified by the user.

The contributions of this paper are summarized as follows:

— We introduce QPR.js, the first runtime support for QoS-aware DVFCS in JavaScript.
— We devise an algorithm that efficiently finds an optimal operating point that minimizes power consumption subject to satisfying the user-specified QoS goal.
— We prototype the QPR.js runtime system and provide detailed evaluation on Intel's mobile/desktop quad-core platform to demonstrate the effectiveness of QPR.js.

2. BACKGROUND
2.1. DVFCS Algorithms

The operating point of a CPU significantly affects its performance and power consumption. Modern multicore CPUs support DVFCS capabilities to exploit performance-power trade-offs for a given workload and performance goal at runtime. In multimedia
applications with real-time constraints, the trade-offs translate to ones between the quality of service (e.g., frame rate) and power consumption.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static governors</strong></td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td>Executing at max clock frequency</td>
</tr>
<tr>
<td>Powersave</td>
<td>Executing at min clock frequency</td>
</tr>
<tr>
<td>Userspace</td>
<td>Executing at clock frequency specified by user</td>
</tr>
<tr>
<td><strong>Dynamic governors</strong></td>
<td></td>
</tr>
<tr>
<td>Ondemand</td>
<td>Executing at clock frequency set by core load; setting frequency to maximum when busy; decrementing frequency step by step when idle</td>
</tr>
<tr>
<td>Conservative</td>
<td>Executing at clock frequency set by core load; incrementing frequency step by step when busy; decrementing frequency step by step when idle</td>
</tr>
</tbody>
</table>

Table I: 5 DVFS governors provided by cpufrequtils

There are many DVFS governors, which control the voltage/frequency/core scaling daemon. Table I tabulates 5 governors provided by the popular cpufrequtils package on Linux [Brodowski nd]. Static governors run at a single fixed frequency; dynamic governors vary the operating point adaptively to the workload. The Ondemand governor sets the frequency immediately to the maximum when the core is busy but decrements it step by step when the core utilization is low. The Conservative governor uses the entire frequency range and adjusts the operating point incrementally for both scaling up and scaling down. However, the core runs at either maximum or minimum frequency most of the time even with the Conservative governor.

Recently new DVFS governors that consider factors other than the core load have emerged, many of which target mobile devices. Some governors adjust the operating point as triggered by user activities like touches [Woo et al. 2013]. Other governors aim to maximize energy efficiency by differentiating CPU-intensive and memory-intensive workloads to save energy for the latter case [Kim et al. 2012]. The Hotplug governor controls core scaling to complement DVFS governors [hotplug nd]. However, all of these governors adjust the operating point by monitoring core utilization and OS events and
miss opportunities for further power reduction by considering the user’s performance goals.

2.2. QoS-Power Trade-offs
There are classes of applications that require not only correctness but also performance to be useful. Many multimedia applications on the web fall in this category, such as media players and 3D games. For example, a video player has a minimum performance (QoS) constraint in terms of frames per second (FPS) to guarantee user satisfaction. Once this constraint is met, the improvement of user satisfaction by further increasing the FPS is only marginal. For this class of applications it is possible to achieve additional power savings by exploiting QoS-power trade-offs.

Figure 1 illustrates this QoS-power trade-off controlled by varying the operating point via DVFCs for two parallel JavaScript applications: Barkley and Nbody. The graphs show the achieved FPS for all operating points on a mobile quad-core platform. Assuming the QoS constraint to be 24 FPS, those operating points that satisfy this constraint are colored in dark gray; the black bar (indicated by an arrow) shows the optimal operating point, which has minimum power consumption while satisfying the constraint. The operating points that fail to satisfy the constraint are shown in light gray.

We can infer two points from the results: (1) The optimal operating point is determined by the QoS constraint. This motivates QoS-aware DVFCs control to reduce power consumption. (2) Even for the same QoS constraint (say, 24 FPS), the optimal operating point differs by applications. In the previous example, Barkley runs optimally with 4 cores at 1.7 GHz, and Nbody with 3 cores at 2.4 GHz. However, conventional DVFCs governors discussed in Section 2.1 do not take into account either QoS constraints or application characteristics, hence leaving potential power savings on the table. Therefore, it is highly desirable to have runtime support for communicating the application’s QoS constraint and its current level of QoS to the DVFCs governor.

3. QPR.JS RUNTIME SYSTEM

3.1. Overall Structure
Figure 2 illustrates the block diagram of the QPR.js runtime system. The main component is the optimizer module in JavaScript, which builds on lower-level hooks for
DVCFS control and power monitoring. Our prototype system is based on WebKit-WebCL [WebKit-WebCL nd] running on Intel’s OpenCL driver (Version 1.2). We use parallel JavaScript applications based on WebCL for performance evaluation in Section 6.

To control DVCFS we add a JavaScript binding to the kernel-level DVFS and Hotplug governors to QPR.js using Web Interface Description Language (WebIDL). The DVCFS control module monitors and changes the governor policy, the number of active cores, and voltage/frequency settings by reading and writing to DVFCS node files.

To monitor instantaneous power we exploit energy counters provided by modern multicore CPUs. For our prototype we provide a JavaScript binding to read energy parameters in Model-Specific Registers (MSR) on Intel’s Haswell CPU through Running Average Power Limit (RAPL) interface using LibRAPL. By reading RAPL values we can easily measure the power consumption of each hardware component, such as CPU, GPU, package, and caches.

The optimizer exposes an API for the programmer to specify a QoS goal and a fitness function to calculate the current level of QoS. The QoS goal is typically represented by a floating-point number, which is compared against the return value of the fitness function to adjust power allocation. By combining this information with the capabilities of power monitoring and DVFCS control, the optimizer searches for an optimal operating point while a parallel JavaScript application is running.

Figure 3 illustrates the two-state finite state machine (FSM) implemented by the optimizer. Initially, the system starts with the monitoring state (M-State). In M-State the optimizer monitors the current QoS level (calculated by the user-provided fitness function) and compares it with the target QoS level. If the difference is greater than the threshold, the optimizer enters the search state (S-State). To improve the stability of the algorithm, the current QoS takes a running average of the last $N$ samples, where $N$ is 10 by default.

In S-State the optimizer sets the initial operating point by turning on all cores and scaling up to the maximum frequency. The search algorithm first scales down the frequency step by step until the current QoS ($QoS_{current}$) becomes equal to the target QoS ($QoS_{target}$) within a tolerance ($\beta$). Then the algorithm turns off one core and repeat the process except that it does not go below the frequency limit ($limit_{freq}$) found with one more core turned on. Note that, a pair of ($QoS_{current}$, $Power_{current}$) are logged into the configuration table by calling $insertConfig()$. Once the search is finished, the optimal point will be retrieved by calling $getBestConfig()$, and the system will enter M-State again.

4. OPTIMIZER ALGORITHM

In this section we describe algorithms of three goals: (4.1) minimize power satisfying minimum FPS, (4.2) maximize throughput within power budget, (4.3) minimize EDP (Energy Delay Product).

4.1. Goal 1: Minimize Power Satisfying Minimum FPS

Algorithm 1 presents the pseudo code of goal 1 which satisfies minimum FPS by minimizing power. As we described, there are two states in this optimizer algorithm: the search state and the monitor state. In search state, optimizer searches two-dimensional table to find all FPS that is larger than Target FPS and save corresponding core, frequency, and power consumption. Among the power consumption, the optimizer find the lowest power consumption with corresponding core and frequency set. Then in the monitor state, the optimizer set the core and frequency from the search state to guarantee the lowest power and target FPS.

To illustrate algorithm 1, \(QoS_{current}\) means current FPS, and \(QoS_{target}\) means target FPS that user want to reach. If \(QoS_{current}\) is equal and greater than \(QoS_{target}\), the optimizer save the current \(core, freq, QoS_{current}\), and \(Power_{current}\). Then, the optimizer keep decrementing frequency if the frequency is not minimum frequency. For the meeting the minimum frequency, the optimizer will turn off a core, set maximum frequency, and keeping searching. If \(QoS_{current}\) is smaller than \(QoS_{target}\), the optimizer will turn off a core and set maximum frequency. If there's no more core can be turned off, then the optimizer will go into monitor state with minimum power setting of the \(core\) and \(freq\) configuration.

---

Algorithm 1 Goal 1: Minimize power satisfying minimum FPS

**Input:** \(QoS_{current}, QoS_{target}\)

**Output:** \(core, freq\)

**Initialize:** \(core ← core_{max}, freq ← freq_{max}\)

1. if \(QoS_{current} ≥ QoS_{target}\) then
   2. \(saveCurrentConfig(core, freq, QoS_{current}, Power_{current})\)
   3. if \(freq ≠ freq_{min}\) then
      4. decrement \(freq\)
   5. else
      6. decrement \(core\)
      7. if \(core ≥ core_{min}\) then
         8. \(freq ← freq_{max}\)
      9. else
         10. \((core, freq) ← getBestConfig()\)
            \(EnterM – State\)
   11. end if
12. end if
13. else
14. decrement \(core\)
15. if \(core ≥ core_{min}\) then
16. \(freq ← freq_{max}\)
17. else
18. \((core, freq) ← getBestConfig()\)
19. \(EnterM – State\)
20. end if
21. end if
22. end if
4.2. Goal 2: Maximize Throughput Within Power Budget

Algorithm 2 gives the pseudo code of goal 2 which satisfies maximum throughput with minimum power budget. Very similar to Goal 1, there are also two states in this optimizer algorithm: the search state and the monitor state. In search state, optimizer searches two-dimensional table to find all power that is smaller than Target power and save corresponding core, frequency, power, and throughput. Among the power consumption, the optimizer find the highest throughput with corresponding core and frequency. Then in the monitor state, the optimizer set the core and frequency from the search state to guarantee the lowest power and highest throughput.

To illustrate algorithm 2, \( QoS_{current} \) means current power, and \( QoS_{target} \) means target power. If \( QoS_{current} \) is equal and smaller than \( QoS_{target} \), the optimizer save the current core, freq, \( QoS_{current} \), and Thruput\(_{current} \). Also, the optimizer will turn off a core since the current power satisfying the target power budget and throughput below the current frequency is low even if there’s lower power consumption. But if \( QoS_{current} \) is greater than \( QoS_{target} \), the optimizer keep searching by turning off a core, set maximum frequency, and decrementing frequency until there’s no more core can be turned off and no more frequency can be scales down, then the optimizer will go into monitor state with minimum power with highest throughput setting of the core and freq configuration.

**Algorithm 2** Goal 2: Maximize throughput within power budget

**Input:** \( QoS_{current}, QoS_{target} \)

**Output:** core, freq

**Initialize:** core \( \leftarrow \) core\(_{max} \), freq \( \leftarrow \) freq\(_{max} \)

1: if \( QoS_{current} \leq QoS_{target} \) then
2: \( \text{saveCurrentConfig}(\text{core, freq, } QoS_{current}, \text{Thruput}_{current}) \)
3: \( \text{decrement core} \)
4: if \( \text{core} \geq \text{core}_{min} \) then
5: \( \text{freq} \leftarrow \text{freq}_{max} \)
6: else
7: \( (\text{core, freq}) \leftarrow \text{getBestConfig}() \)
8: \( \text{Enter M - State} \)
9: end if
10: else
11: if \( \text{freq} \neq \text{freq}_{min} \) then
12: \( \text{decrement freq} \)
13: else
14: \( \text{decrement core} \)
15: if \( \text{core} \geq \text{core}_{min} \) then
16: \( \text{freq} \leftarrow \text{freq}_{max} \)
17: else
18: \( (\text{core, freq}) \leftarrow \text{getBestConfig}() \)
19: \( \text{Enter M - State} \)
20: end if
21: end if
22: end if

4.3. Goal 3: Minimize EDP

Algorithm 3 presents the pseudo code of goal 3 which find minimum EDP(energy Delay Product). Similar to previous goals, there are two states in this optimizer algorithm: the search state and the monitor state. In search state, optimizer searches two-dimensional table to find smallest EDP and save corresponding core and frequency. Then in the monitor state, the optimizer set the core and frequency from the search state to guarantee the minimum EDP. Different from goal 1 and goal 2, there’s no target EDP. We just want to find minimum EDP in all core and freq configuration.

To illustrate algorithm 3, QoS current means current EDP, and QoS min means minimum EDP. If QoS current is equal and smaller than QoS min, the optimizer saves the current core, freq, QoS current, and QoS min. Then, the optimizer keep decrementing frequency if the frequency is not minimum frequency. For the meeting the minimum frequency, the optimizer will turn off a core, set maximum frequency, and keeping searching. If QoS current is greater than QoS min, the optimizer will turn off a core, set maximum frequency, and keep searching by scaling down the frequency. Otherwise, the optimizer will go into monitor state with minimum EDP setting of the corresponding core and freq configuration.

**Algorithm 3 Goal 3: Minimize EDP**

**Input:** QoS current, QoS min, \( \beta \) (tolerance)

**Output:** core, freq

**Initialize:** core \( \leftarrow \) core max, freq \( \leftarrow \) freq max

1. \( QoS_{min} = \text{Infinity} \)
2. if QoS current \( \leq \) QoS min + \( \beta \) then
3. \( QoS_{min} = \text{QoS current} \)
4. saveCurrentConfig(core, freq, QoS current)
5. if freq \( \neq \) freq min then
6. decrement freq
7. else
8. decrement core
9. if core \( \geq \) core min then
10. freq \( \leftarrow \) freq max
11. else
12. (core, freq) \( \leftarrow \) getBestConfig()
13. Enter M – State
14. end if
15. end if
16. else
17. \( QoS_{min} = \text{Infinity} \)
18. decrement core
19. if core \( \geq \) core min then
20. freq \( \leftarrow \) freq max
21. else
22. (core, freq) \( \leftarrow \) getBestConfig()
23. Enter M – State
24. end if
25. end if
Table II: QPR.js API

5. QPR.js API

Table II summarizes the API exposed by QPR.js. The API functions are classified into user and optimizer API functions. Users only need to use the User API for optimizing their own program with an appropriate optimizer. In the user API updateCqos is called periodically to maintain the current QoS level up-to-date so that the optimizer algorithm works effectively. This is the only function that must be implemented by the user. setCqos sets the value of a global variable that represents the current QoS level and is typically called within the updateCqos function. setTqos sets the target QoS level with lower and upper tolerance limits. setOptimizer sets an optimizer function what the user wants to be adapted. The optimizer function is implemented by the optimizer. startIter and endIter represents the start and end of target loop iteration, respectively.

Anyone can create their own optimizer with the optimizer API. In the optimizer API getCqos returns the most recent QoS value that is setted by setCqos. getTqos returns the target QoS value that is setted by setTqos. getFreq returns the current frequency of CPU, because the frequencies of all cores are changed at the same time in our platform. getOnline returns the number of active cores. getPower and getDelay returns amount of power consumption and spent time between startIter and endIter, respectively. To measure power consumption, getPower reads a CPU counter by Intel like RAPL. decreaseFreq decreases the CPU frequency to the next level. If the frequency level is the lowest, then turnOffCore is called. turnOffCore turns off a core and sets the frequency maximum. If either decreaseFreq or turnOffCore cannot turn off a core, then QPR.js stops the searching and selects a configuration based on values that are returned by the optimizer function.

Figure 4 illustrates an example of applying the QPR.js API to transform a JavaScript program for QoS-aware power optimization at runtime. The original program in Figure 4(a) decodes one video frame at a time within a loop. The optimizer code of Algorithm 1 is shown in Figure 4(b). The optimizer function returns the negative of power consumption, so that QPR.js selects the configuration where power consumption is minimum with satisfying QoS constraint. Note that, QPR.js selects the configuration where the returned value from the optimizer function is maximum. The transformed...
A function main loop 

```javascript
function main() {
  while (true) {
    var frame = decodeFrame();
    draw(frame);
  }
}
```

(a) Original code

```javascript
function powerOpt() {
  if (QPR.getCqos() >= QPR.getTqos()) {
    QPR.decreaseFreq();
    return -QPR.getPower();
  } else {
    QPR.turnOffCore();
    return -Infinity;
  }
}
```

(b) Optimizer code

```javascript
var n_frames, t0; // global variables for updateCqos
function updateCqos() {
  var t1 = new Date().getTime();
  var fps = n_frames / (t1 - t0);
  QPR.setCqos(fps);
  n_frames = 0;
  t0 = t1;
}
```

(c) User code

```
function main() {
  QPR.setTqos(24, 0, Infinity); // set target fps to 24 with tolerance
  QPR.setOptimizer(powerOpt); // set optimizer
  n_frames = 0;
  t0 = new Date().getTime();
  setInterval(updateCqos, 500); // calculate fps every 500 ms

  while (true) {
    QPR.startIter();
    var frame = decodeFrame();
    draw(frame);
    QPR.endIter();
    n_frames++;
  }
}
```

Fig. 4: Example code using QPR.js API

code is shown in Figure 4(c). Note that, the QoS constraint is set to 24 FPS. Line 2 registers updateCqos to be called every 500 ms via timer interrupt in JavaScript. Variables n_frames, t0, and t1 are introduced to calculate the instantaneous frame rate, which specifies the current QoS level. Function updateCqos actually performs the calculation of the frame rate and communicate it to the optimizer by calling setCqos (Line 5). Note that, although we use FPS as QoS metric, the QPR.js API is flexible enough to support any combination of user-defined metrics.

6. EVALUATION

6.1. Methodology

Table III summarizes the experimental setup. Five rendering JavaScript programs are selected since it is easy to define the QoS goal for them: Barkley, Nbody, PathIntegrals, VideoCube, and XY [WebKit-WebCL nd]. Also, four linear algebraic applications are chosen: gemm, syrk, syr2k and gesummv [?]. Because, the generally-used JavaScript benchmarks take too short execution times to apply DVFS techniques.
On the other hands, they represents computation-intensive JavaScript applications long enough to optimize various metrics. Originally, those are ported from C-based polyhedral benchmark suite [7]. Note that, QPR.js is flexible enough to accommodate applications from other domains as well. To clarify the practicality of QPR.js, we tested 3 example scenarios with each goals specified in Section ???. Thus, goal 1 is applied to [WebKit-WebCL nd] and goal 2 and 3 is applied to [?]. Each examples are implemented with the public API of QPR.js to dynamically regulate customly defined QoS metrics.

To debate the fairness of QPR.js, static and dynamic governors are also tested for all results. Especially, Intel turbo boost mode are turned off when QPR.js is run. Because, if once turbo boost mode is turned on, they tend to hold the previous frequency settings even though the runtime changes frequency. Therefore, to correctly manipulate DVFS and DCS, we decided to limit those frequency settings. Other governors are set to turn on the turbo boost mode which is original settings of linux. Furthermore, to clearly see the effect of DVFS, we turn off SMT supports for notebook platform – Our desktop platform doesn’t support SMT.

The power measurement is done by reading energy and time counters provided by Intel architecture and WebKit respectively. For the measurement of energy, Kicherer [Librapl nd] implements the libRAPL library, which reads the RAPL values in MSR registers. To avoid sampling, we integrates libRAPL libraries into WebKit and provides bindings so that the user can access energy counters within JavaScript environment. And if the user defines where to measure it, QPR.js internally calculates the power by using those bindings and time-measuring functions which WebKit already provides.

<table>
<thead>
<tr>
<th>Name</th>
<th>Mobile platform</th>
<th>Desktop platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel i5.2700HQ mobile quad-core</td>
<td>Intel i5.2500 desktop quad-core</td>
</tr>
<tr>
<td>SMT</td>
<td>Turned off</td>
<td>Not supported</td>
</tr>
<tr>
<td>Frequency</td>
<td>0.8GHz-2.4GHz (11 steps)</td>
<td>1.6GHz-3.3GHz (10 steps)</td>
</tr>
<tr>
<td>Memory</td>
<td>8GB</td>
<td>8GB</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia Geforce GT 750M</td>
<td>Nvidia Geforce GTX 650</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu 12.04 64 bits</td>
<td></td>
</tr>
<tr>
<td>SW Platform</td>
<td>WebKit-WebCL EFL Port [WebKit-WebCL nd]</td>
<td></td>
</tr>
<tr>
<td>Energy Counters</td>
<td>RAPL</td>
<td></td>
</tr>
<tr>
<td>Parallel JavaScript</td>
<td>WebCL (Intel OpenCL v1.2 CPU)</td>
<td></td>
</tr>
</tbody>
</table>

Table III: System specifications

### 6.2. Results

Figure 5 shows the target QoS constraint and the average QoS compared among the default governors and QPR.js. The values are taken from the monitoring state of each profiling results. (a), (b) shows the results for goal 1 with each platform and target QoS(FPS) is set to 24. For a stable behavior, monitoring state have threshold (-1, +6). Also, (c), (d) shows the results for goal 2 and target QoS(Power) for desktop platform is 15 watt and those for notebook platform is 8 watt. The monitoring threshold for goal 2 is set to (-Infinity, 1). Goal 3 has no QoS constraint, thus, the monitoring threshold isn’t needed.

Note that over the all benchmarks and goals, QPR.js finds the optimal point of figure of merits(FoM) within points that satisfies QoS constraint. And the optimal QoS point generally forms at the right near of a defined QoS constraint. Goal 1 behaves to find right over the target QoS(FPS), whereas goal 2 searches the point right below the target QoS(Power).
Figure 6 represents figure of merits (FoM) over all benchmarks. Considering QoS constraint, performance and ondemand governor show the significant amount of power is wasted for goal 1. On the other hand, powersave shows two cases. At the first case (a) - Barkley, XY, (b) - All), powersave violates the lower QoS limit as expected, thus QPR.js finds the faster configuration to satisfy the target QoS. However, the result shows that even the powersave governor could over-use the resources. (a) - Nbody, PathIntegrals, VideoCube) In this case, QPR.js could find the more QoS-aware optimal point. It’s because the default governor doesn’t utilize DCS.

For the goal 2((c), (d)), FoM is set to throughput, thus maximize it within upper limit of QoS (power budget). With desktop platform, syr2k, syrk and gemm shows lower throughput than powersave, which means the power budget limits throughput. In the other case with mobile platform, powersave degrades too much throughput when considering QoS constraint.

(e), (d) shows the average FoM of goal 3 which is normalized to ondemand governor. Because, goal 3 doesn’t have the QoS constraint, QPR.js always finds the lowest EDP point.

6.2.1. **Goal 1: Minimize Power Satisfying Minimum FPS.**

Figure 7 illustrates the runtime behavior of the two applications: Barkley and Nbody. Barkley is an example with stable optimal operating point, where the optimal operating point does not fluctuate once entered. And, generally, notebook platform shows more stable behavior than desktop’s. Also, they tend to be unstable at low operating point (low freq, core). Among benchmarks, PathIntegrals, VideoCube, and XY all
follow the stable pattern. However, the optimal operating point of \texttt{Nbody} is not stable, as you can see at (d), FPS isn’t stable during monitoring state. Therefore, to avoid repeating transition to each state, a user can adjust the threshold of monitoring state. Nevertheless, the overall power consumption is still significantly lower than the QoS-oblivious DVFS governors as shown in Figure 6-(a),(b).

\textbf{6.2.2. Goal 2: Maximize Throughput Within Power Budget.}

Figure 8 also shows the runtime results for the goal 2 with two applications: \texttt{gemm} and \texttt{gesummv}. Note that profiling term is changed from time to iterations compared to goal 1. In this experiments, throughput and power graphs are more stable than goal 1. It’s
Fig. 7: Runtime behavior of QPR.js-enabled parallel execution - Optimization Goal 1

because it just has calculation without visual effects that interrupt profiling. For both platforms, \texttt{gemm} is stabilized at higher operating point (high freq, core) than \texttt{gesummv}. Because \texttt{gemm} is more compute-intensive, thus it's more sensitive to DVFS and DCS scaling. \texttt{gesummv} takes a large memory compared to its computation amount. So, the
Fig. 8: Runtime behavior of QPR.js-enabled parallel execution - Optimization Goal 2

application tend not to be sensitive to scaling. For both benchmarks and platforms, as you can see, QPR.js easily finds the optimal point with exactly satisfying QoS constraint (Power budget). Other benchmarks, syrk and syr2k follows the trend of gemm.
6.2.3. **Goal 3: Minimize EDP.**
Finally, Figure 9 shows the runtime results for the goal 3. As we denoted at Section 4.3, QPR.js finds the optimal point using kind of gradient-descent methods to minimize EDP. Note that EDP results of gemm at both platform ((a), (c)) goes into monitoring state with different searching latency. With notebook platforms, the scaling easily affects to delay. Therefore, minimal point generally is formed at higher frequency. On the other hands, with desktop platforms, still EDP is delay-sensitive, its amount of effect is lower than notebook’s. Theroere, they tends to find more frequency spaces.

6.2.4. **Multi-program Workload.**
Also, we launch a background program after Barkley enters into an optimal operating point to demonstrate the performance robustness of QPR.js. The result is shown in Figure 10. When the background program is launched, the difference between target QoS and current QoS gets greater than the threshold, thus making the optimizer enter the Search state (S-State) again. Since the background program shares resources, the optimizer finds a new optimal operating point with an additional core to enter M-State with four cores at 2.4 GHz.
QoS-aware Dynamic Power Optimization for Data-Parallel JavaScript Programs

7. RELATED WORK

DVFCS is one of the most popular techniques to save power. Conventional DVFCS schemes (e.g., the default Ondemand governor on Linux [Pallipadi and Starikovskiy 2006]) vary voltage/frequency/cores of the CPU by monitoring system metrics such as CPU utilization and OS events. There have been proposals to achieve additional power savings over them.

User-driven DVFCS: User-driven frequency scaling (UDFS) reduces power consumption by taking the user's feedback [Mallik et al. 2006]. Likewise, Mallik et al. extend this to devise more accurate metrics for the satisfaction of user-perceived performance in display applications: Average Pixel Change (APC) and Rate of Average Pixel Change (APR) [Mallik et al. 2008]. However, they all require the user's direct intervention, which limits their applicability.

Yan et al. propose power optimization with computer responsiveness for interactive applications [Yan et al. 2005]. It is similar to the touch in mobile devices and thus its quality highly depends on the user's behavior. Woo et al. propose to consider the information of user's touch behavior together with the core load in the mobile device [Woo et al. 2013]. Shye shows the biometric devices such as eye tracker, galvanic skin response (GSR), and force sensor to obtain more information from user [Shye et al. 2008]. Inferring user satisfaction from UI events, however, may erroneously interpret the user's intention and degrade use experience.

Application-specific DVFCS: Baek et al. propose to skip some of the frames by utilizing the characteristic of MPEG decoder [Baek et al. 2010]. Kim et al. introduce energy-centric DVFS (eDVFS) for applications requiring energy efficiency more than power efficiency [Kim et al. 2012]. Hwang et al. implement adaptive dynamic voltage scaling (DVS) algorithms based on either feedback or buffering according to the bit rate of a movie [Hwang et al. 2008]. Choi et al. propose to predict the frame-dependent decoding time precisely and thus exploit DVFS to match the exact decoding time [Choi et al. 2005]. In addition, to prevent user's degradation, when the prediction is failed, it is compensated in the frame-independent decoding time until the deadline. How-
ever, it is generally difficult to measure the exact deadline and also predict the frame-dependent decoding time. In addition, as the result of the prediction, the power may be wasted. Gu et al. focus on the interactive game and thus analyze the characteristics of workload of the game [Gu et al. 2006]. All of these proposals target specific applications or application domains, hence they are not as applicable as QPR.js presented in this thesis.

**JavaScript APIs to control hardware:** Recently, the web browser has become a general-purpose programming platform, and there are proposals to provide APIs to control the hardware. Firefox OS [Mozilla Developers nd] and Tizen [Tizen Developers nd] are such examples. Although they provide an API to access hardware such as light sensor, battery, screen orientation, camera, and system information, they do not provide an API to control CPU frequency or DVFCS. To the best of our knowledge, QPR.js is the first JavaScript API to support QoS-aware DVFCS.

8. CONCLUSION

The demands for power-efficient JavaScript performance are higher than ever with widespread adoption of web applications. As web applications become more complex and compute-intensive, the demands will continue to grow. In this paper we present QPR.js, the first JavaScript API that enables QoS-aware power reduction while satisfying user-specified QoS constraints. Our evaluation with five WebCL-based parallel JavaScript applications shows promising results; QPR.js achieves an average of 59.0\% power savings compared to the default Ondemand Linux governor. This benefit is realized with relative simple modifications to the original program. We plan to extend this work to accommodate applications with more complex QoS constraints and improve the efficiency of the optimizer algorithm.

REFERENCES


