QPR.js: A Runtime Framework for QoS-Aware Power Optimization for Parallel JavaScript Programs

Wonjun Lee[†‡] Channoh Kim[‡] Houp Song[†] Jae W. Lee[‡]

[†]Samsung Electronics
Suwon, Korea
{wonjun44.lee, hu.song}@samsung.com

[‡]Sungkyunkwan University
Suwon, Korea
{wonjun.lee, channoh, jaewlee}@skku.edu

ABSTRACT

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 on the table 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 **Q**oS-aware **p**ower-optimizing **r**untime system for **J**ava**S**cript. Using the QPR.js API, the application developer can specify a QoS goal and provide a fitness function to quantify the current level of QoS. During execution the QPR.js runtime system uses this information to autonomously find an optimal operating point minimizing power consumption while satisfying the QoS goal. Our evaluation with five parallel JavaScript programs demonstrates an average of 35.2% power savings over the Linux Ondemand governor without degrading user experience.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Power Optimal Programming*; D.3.2 [**Programming Languages**]: Language Classifications—*JavaScript*

Keywords

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 these resources, multiple JavaScript parallelization frameworks have emerged, such as WebCL [3] and Web Workers [2]. Besides, modern multicore CPUs expose to software some knobs to improve power efficiency such as dynamic voltage/frequency/-core scaling (DVFCS) [9]. 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 [1] and Windows DVFS [5], 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 [6,7], or infer it from UI events (e.g., touches), which may erroneously interpret the user's intention and degrade use experience [8,10,11].

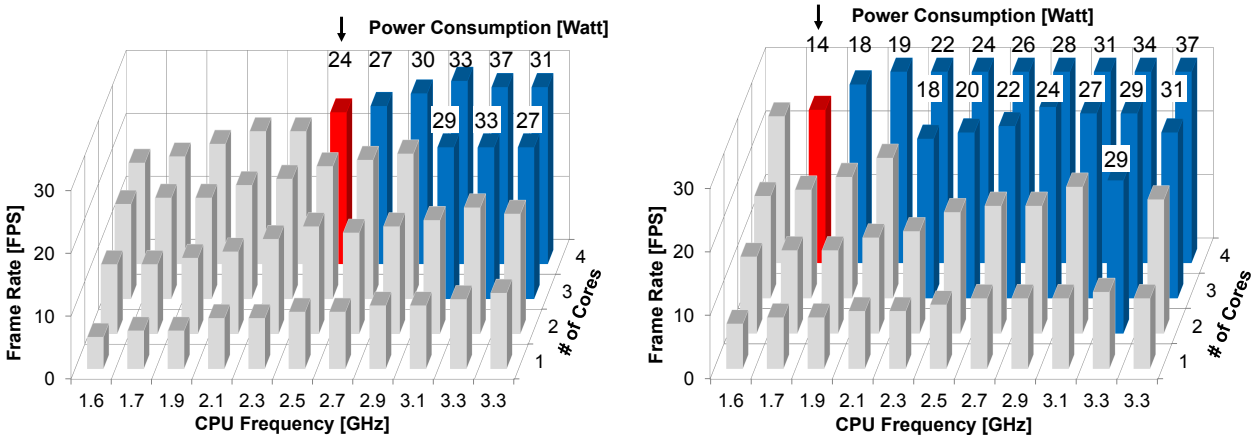
This paper proposes QPR.js, an API and runtime system that enables quality-of-service (QoS) aware DVFCS for parallel JavaScript programs. Using the QPR.js API, the user can specify a QoS goal and a fitness function that quantifies the current level of QoS. The QPR.js runtime system finds an optimal operating point that minimizes power consumption subject to satisfying the QoS goal (e.g., minimum frame rate). QPR.js is implemented on Intel's Sandy Bridge quad-core system running Linux and evaluated using 5 WebCL-based parallel JavaScript programs. Compared with the default Ondemand governor for DVFCS control, QPR.js reduces power consumption by 35.2% on average while satisfying the QoS goal specified by the user.

2. QOS-POWER TRADE-OFFS

There is a class 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 me-

ISLPED'14, August 11–13, 2014, La Jolla, CA, USA.

http://dx.doi.org/10.1145/2627369.2627648.



(a) Barkley (b) Nbody

Figure 1: Trade-off between frame rate and power; the number above a bar shows power consumption at the operating point.

dia 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 quad-core desktop machine. Assuming the QoS constraint to be 24 FPS, those operating points that satisfy this constraint are colored in blue; the red 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 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 2.5 GHz, and `Nbody` with 4 cores at 1.7 GHz. However, conventional DVFCS governors, which control the voltage/frequency/core scaling daemon, 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 a 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 [4] running on Intel's OpenCL driver (Version 1.2). We use parallel JavaScript applications based on WebCL for performance evaluation in Section 4.

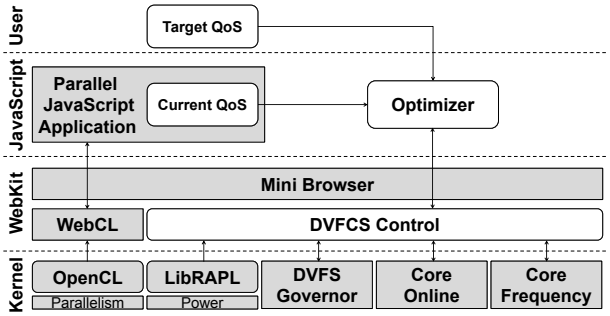


Figure 2: Overall structure of QPR.js runtime

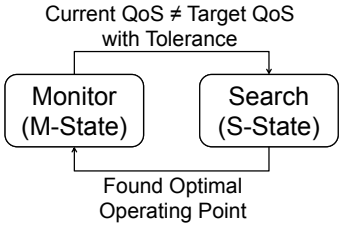


Figure 3: Optimizer two-state FSM

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 the Running Average Power Limit (RAPL) values in Model-Specific Registers (MSR) through LibRAPL on Intel's Sandy Bridge CPU. 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. 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.

3.2 Optimizer Algorithm

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 (provided by the user-provided fitness function) and compare 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 ten samples.

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 operating point yields the current QoS ($QoS_{current}$) equal to or lower than the target QoS (QoS_{target}). 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. If the search is finished, optimizer sets the optimal core and frequency. Once search is finished and an optimal operating point is found, the system enters M-State again.

Algorithm 1 presents the pseudo code of this algorithm. Note that this algorithm is invoked periodically, and that the search state ($freq$, $core$) is preserved across invocations. Since it is time consuming to search all operating points, we employ heuristics to reduce the size of search space using $\text{limit}_{\text{freq}}$; the main idea is that an operating point with fewer cores and the same frequency cannot perform better than the current operating point. Also, we stop searching with one fewer cores if there are relatively few frequencies that satisfy the QoS constraint with the current core count. By default this threshold is set to a half of the number of frequency steps on the platform.

Algorithm 1 Search Algorithm

Input: $QoS_{current}$, QoS_{target}
Output: $core_{opt}$, $freq_{opt}$
Initialize: $core \leftarrow core_{max}$, $freq \leftarrow freq_{max}$

if $QoS_{current} = QoS_{target}$ **then**
  if $freq > sizeof(freqsteps)/2$ **then**
    $core_{opt} \leftarrow core, freq_{opt} \leftarrow freq$
    return
  else if $core = core_{min}$ **then**
    $core_{opt} \leftarrow core, freq_{opt} \leftarrow freq$
    return
  else
    $limit_{freq} \leftarrow freq + 1$
    decrease $core, freq \leftarrow freq_{max}$
  end if
else if $QoS_{current} < QoS_{target}$ **then**
  if $freq = freq_{max}$ **then**
    $core_{opt} \leftarrow core, freq_{opt} \leftarrow freq$
    return
  else
    decrease $core, freq \leftarrow freq_{max}$
  end if
else
  if $freq \neq freq_{min}$ **and** $freq > limit_{freq}$ **then**
    decrease $freq$
  else if $freq > sizeof(freq)/2$ **then**
    $core_{opt} \leftarrow core, freq_{opt} \leftarrow freq$
    return
  else
    decrease $core, freq \leftarrow freq_{max}$
  end if
end if

4. EVALUATION

Table 1 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` [4]. Note that, QPR.js is flexible enough to accommodate applications from other domains as well. We set 24 FPS (minimum FPS for TV) to be the target QoS level and adjust the iteration count of an inner loop, if necessary, for all applications to achieve at least 30 FPS at the maximum operating point.

Name	Descriptions
Core	Intel i5-2500 CPU (4 ea)
Frequency	1.6GHz-3.3GHz (11 steps)
Memory	8GB
GPU	Nvidia Geforce GT 530
OS	Ubuntu 12.04 64bits
Platform	WebKit-WebCL EFL Port [4]
Power	RAPL (Intel Sandy Bridge)
Parallel JavaScript	WebCL (Intel OpenCL v1.2 CPU)

Table 1: System specifications

Figure 4 compares the power consumption of QPR.js-enabled parallel programs against QoS-oblivious Linux DVFS governors. Compared to the default Ondemand governor, QPR.js achieves 35.2% of power savings while satisfying the QoS goal. Note that, both Performance and Ondemand governors run at the maximum operating point most of time to yield low power efficiency. The power consumption of the Powersave governor is minimal, but at the cost of violating the QoS constraint.

Figure 5 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. `PathIntegrals`, `VideoCube`, and `XY` all follow this pattern. However, the optimal operating point of `Nbody` is not stable, making the optimizer continuously switch between M-State and S-State. Nevertheless, the overall power consumption is still signif-

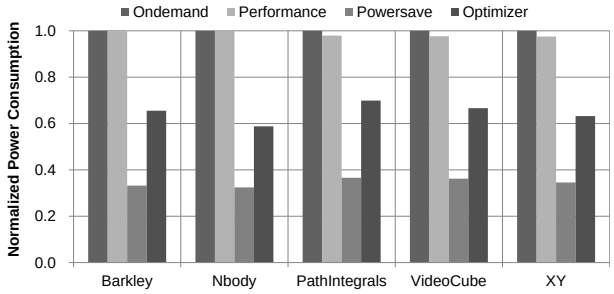
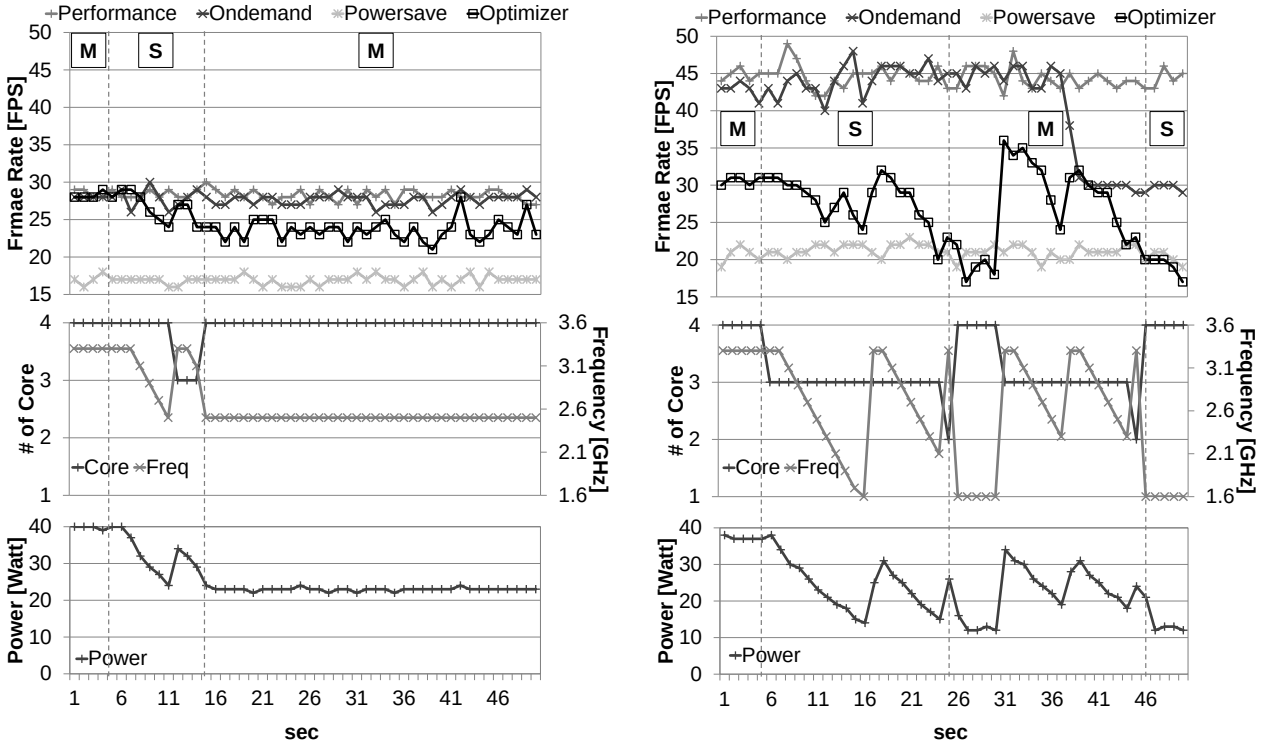


Figure 4: Power comparison against Linux DVFS governors



(a) Barkley (b) Nbody

Figure 5: Runtime behavior of QPR.js-enabled parallel execution

icantly lower than the QoS-oblivious DVFS governors as shown in Figure 4.

5. 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 35.2% 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.

6. ACKNOWLEDGMENTS

This work was supported in part by the Korean Evaluation Institute of Industrial Technology funded by the Ministry of Science, ICT & Future Planning (KEIT-10047038) and the IT R&D program of MKE/KEIT [KI001810041244, Smart TV 2.0 Software Platform].

7. REFERENCES

[1] CPU frequency and voltage scaling code in Linux(TM) kernel. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[2] Web Worker. http://dev.w3.org/html5/workers/.

[3] WebCL. http://www.khronos.org/webcl/.

[4] WebCL for WebKit. https://github.com/SRA-SiliconValley/webkit-webcl.

[5] Windows Power Management and ACPI - Architecture and Driver support.

[6] B. Lin, A. Mallik, P. Dinda, G. Memik, and R. Dick. User-and process-driven dynamic voltage and frequency scaling. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 11–22. IEEE, 2009.

[7] A. Mallik, B. Lin, G. Memik, P. Dinda, and R. P. Dick. User-driven frequency scaling. *Computer Architecture Letters*, 5(2):16–16, 2006.

[8] A. Shye, Y. Pan, B. Scholbrock, J. S. Miller, G. Memik, P. A. Dinda, and R. P. Dick. Power to the people: Leveraging human physiological traits to control microprocessor frequency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 188–199, Washington, DC, USA, 2008. IEEE Computer Society.

[9] H. Wang, V. Sathish, R. Singh, M. J. Schulte, and N. S. Kim. Workload and power budget partitioning for single-chip heterogeneous processors. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 401–410, New York, NY, USA, 2012. ACM.

[10] S. Woo, W. Seo, C. Kim, and J. Huh. User input based power reduction technique for smartphone. In *Proceedings of the Korean Institute of Information Scientists and Engineers*, 2013.

[11] L. Yan, L. Zhong, and N. K. Jha. User-perceived latency driven voltage scaling for interactive applications. In *Proceedings of the 42Nd Annual Design Automation Conference*, DAC '05, pages 624–627, New York, NY, USA, 2005. ACM.