Efficient Execution of Bursty Applications

Milad Hashemi*, Debbie Marr†, Doug Carmean‡, Yale N. Patt*
*University of Texas at Austin, †Intel Corporation, ‡Microsoft
*{miladh, patt}@hps.utexas.edu, †debbie.marr@intel.com, ‡dcarmean@microsoft.com

***Abstract*—The performance of user-facing applications is critical to client platforms. Many of these applications are event-driven and exhibit "bursty" behavior: the application is generally idle but generates bursts of activity in response to human interaction. We study one example of a bursty application, web-browsers, and produce two important insights: (1) Activity bursts contain false parallelism, bringing many cores out of a deep sleep to inefficiently render a single webpage, and (2) these bursts are highly compute driven, and thus scale nearly linearly with frequency. We show average performance gains/energy reductions of 14%/17% respectively on real hardware by statically moving threads from multiple cores to a single core. We then propose dynamic hardware driven thread migration and scheduling enhancements that detect these bursts, leading to further benefits.**

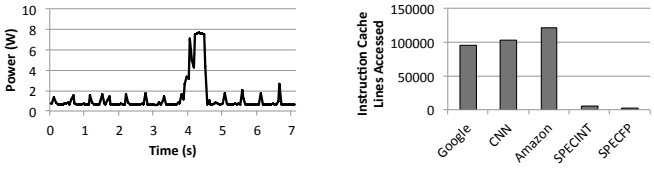
Index Terms: **Multicore, Energy, Performance, Webpages**

I. Introduction

Due to the hardware and software specialization of mobile devices and cloud platforms, computing systems have largely split into the client space and the server space. In client systems, many relevant applications are *bursty*, involving either a human or long latency request as a component of the experience and are therefore often event-driven in nature. Examples include web browsers, user interfaces, gesture processing, and navigation functions.

The power profile of an activity burst is shown in Figure 1a. The system is otherwise idle until a webpage is loaded in Firefox (a web-browser) at about four seconds. This load causes a short burst of processor activity that settles back to the idle baseline after about one second. As the responsiveness of the application greatly influences the user's experience on the client system, these bursts are highly latency sensitive and a key area for hardware optimization.

However, bursty applications do not exhibit behavior common to traditional computer architecture workloads and are harder for architects to evaluate as code-footprints are much larger than kernels and relevant execution timescales are on the order of seconds. This is illustrated in Figure 1b. We run all of the SPEC CPU2006 benchmarks with the reference input set and compare the number of different 64-byte instruction cache lines touched by each application to a web browser loading three different webpages: Google, Amazon, and CNN. Even loading a seemingly simple webpage like the Google homepage touches a code footprint that is 10x larger than the average footprint touched by a SPECint benchmark.



(a) (b)

Fig. 1: (a) Core power consumption burst when loading a webpage (Google homepage). (b) Total number of 64-byte instruction cache lines accessed by SPECint/SPECfp 2006 and three webpage loads.

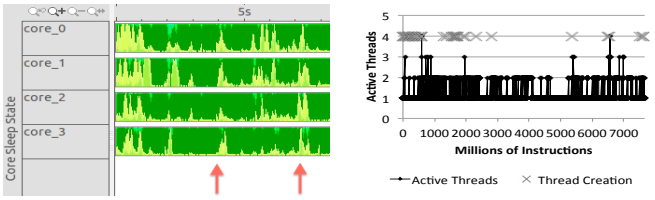
In this work, we seek to understand how these bursty workloads perform on current multicore processors, which have penetrated the mobile market. We make the following contributions in this paper:

- We study one example of a bursty application, Firefox, on both real hardware and with Pin [7] instrumentation. We make two observations. First, the bursts of activity that occur while loading a webpage contain "false-parallelism" that cause inefficiency by bringing all of the cores of a multi-core processor out of sleep for small periods of time. Second, these bursts are highly compute-driven, and do not appear to be memory-limited.
- Using these insights, we evaluate a simple static policy on real hardware that clusters all threads on a single core. This policy results in a 14% average reduction in webpage load time and a 17% reduction in energy consumption.
- We propose a dynamic hardware-driven thread clustering policy that detects the bursts of activity that we observe to be common to webpage loads. We evaluate this policy using Pin and show that it outperforms a static thread clustering policy.

II. Browser Compute Behavior

To understand the compute behavior of a web-browser on a multicore processor, we use Intel's VTune [5] to track core utilization when a web page is loaded. VTune uses performance counters to track how much time each core spends executing an application in different power-saving modes, or sleep states. This correlates to the application's compute utilization as an inactive or sleeping processor is unutilized. We use Firefox as a web browser in our experiments and suggest in Section VII that the observed behavior extends to other browsers.

Figure 2a shows the core-activity data for each of the four cores of an Intel Ivy-Bridge processor. The brightest yellow colors signify time spent in the C0 state, where the processor is operating at its full-frequency. Darker colors denote power-saving states where different parts of the processor are powered down, or the clock frequency is reduced. Every data point is a sorted bar representing the fraction of time during that interval that the core spent in each sleep state.



(a) (b)

Fig. 2: (a) Core sleep state data. Firefox is started at time 0 followed by two Google page loads at 4 and 7 seconds. Each tick mark on the x-axis denotes one second. (b) The number of active threads and thread spawn events for starting Firefox and loading the Google homepage twice. Every thread creation event is marked with an 'x'.

There are three events captured in Figure 2a. First, Firefox is started at time 0. Second, at about 4 seconds we load Google using a bookmarked link. Third, at about 7 seconds we reload Google by refreshing the webpage[1]. For each of these events we observe the bursty behavior that we expected: the system becomes very active for a short period of time and then settles back to idle. Moreover, we

Manuscript submitted: 02-Jun-2015. Manuscript accepted: 28-Jun-2015. Final manuscript received: 4-Jul-2015.

[1]We disable the browser's software cache for all experiments in this paper

consistently observe all of the four cores powering up from a deep sleep state (signified by a dark green color) to C0 (bright yellow). The operating system has scheduled threads for execution on each core, implying that Firefox is a very parallel program with many threads available for execution. The application is making use of all of the available compute resources during each burst.

Figure 2a shows that loading Google causes high compute utilization for a short period of time. However, Google is not the only website to display this trend. Based on our experiments, we observe this behavior for all of the websites we loaded, from simple micro-benchmarks to full webpages such as Google. In general, the bursts of activity caused by loading a website result in high compute utilization and appear to be driven by many threads. We further explore this effect by instrumenting Firefox using Pin [7]. This allows us to observe how many threads are active in a given interval as well as the total number of threads that are spawned while loading a webpage.

We experiment once again by loading the Google homepage. We use an interval length of one million instructions, and define any given thread to be active if it executes a substantial portion of the total number of instructions in that interval (100,000 instructions out of a 1,000,000 instruction interval, or 10%). As in Figure 2a, we start instrumentation when Firefox begins and then load the Google homepage twice. These results are shown in Figure 2b.

Figure 2b marks thread creation events with an 'x'. These events are concentrated around three different points. During the first billion instructions, when Firefox is starting, around 1.5 billion instructions, during the first page load, and at 5.0 billion instructions during the second page load. To accomplish the task of starting Firefox and loading two web pages, 41 threads are spawned. This is a large number of threads, far more than a modern multi-core chip is designed to run at once. With this level of apparent parallelism, it is not surprising that Figure 2a shows all four cores powering up out of deep sleep to execute all of the available threads.

However, Figure 2b also shows that for the vast majority of intervals a single thread dominates the total number of executed instructions. In fact, out of the 7600 total intervals there are four active threads for only four intervals, and three active threads for only 24 intervals. This implies that the 41 threads that were spawned tend to each do a very small amount of total work, accounting for under 100,000 instructions, and there are one or two main threads that are active during most intervals.

Our studies attribute this thread behavior to the event-driven, asynchronous nature of web-browsers and JavaScript. Analysis of the work that each thread is completing is generally obscured by indirect jumps in the main event-handler loop in Firefox. The main thread that handles the event loop is usually active, while many other threads are generated as work/events occur.

Yet, as each core is 2-way SMT, this threading behavior poses a question. Why are all four cores powering out of deep sleep to complete this work when the majority of the time, only a small number of threads are active? The operating systems treats all threads equally whereas these threads exhibit very different compute behavior. There is a significant static-power overhead to turn on a whole new core if it is only going to be active for a very short period of time, as caches require time to warm-up after power-gating. If the running thread only runs for a short amount of time, this data migration cost cannot be amortized. We explore this observation in Section III by instrumenting real hardware to measure power and performance during website loads.

III. Single-Core vs. Multi-Core

In Section II we identified that while there appears to be significant parallelism when loading a webpage, there are predominantly one or two threads that dominate execution time. With this little effective parallelism, there may be energy and performance benefits to running all of the threads on a single core instead of across all four-cores of a multi-core processor. This minimizes data movement and sharing between cores, thereby increasing compute efficiency and reducing overall static power overhead.

However, as Figure 2b demonstrates, a full webpage load requires executing billions of instructions, making traditional cycle-accurate simulation methods intractable. Therefore, we use a special tablet-like, mobile machine instrumented to capture power consumption to explore the energy effects of loading a webpage on a single-core vs a multi-core processor. The machine is based on Windows 8 and a quad-core Intel Haswell processor.

The proprietary power instrumentation hardware samples at 20 Hz and breaks out and records the major sources of power consumption in a non-intrusive fashion including: core (inclusive of the last level cache), DRAM, wireless, display, and SSD power consumption. The system is otherwise idle until a webpage is loaded, the load causes a burst of core activity that registers on the power monitoring hardware.

We measure the load time of a website by the length of time between the rising edge and falling edge of the burst (Figure 1a). The energy consumed by the load is the area underneath this portion of the power curve. Each webpage load is repeated five times, and we take the average of these numbers. All of the energy results we report are for the power consumption of the package, inclusive of all four cores and the shared cache.

We vary two parameters in this test, the number of active cores and the frequency of each of the active cores. If a core is disabled, it is turned off and the operating system is unable to schedule tasks to run on that core. Turbo is disabled to ensure deterministic test parameters, so once a frequency is set for a core, it will not increase. We use five commonly accessed webpages with a variety of different content and complexity to conduct our test: the Google homepage, a sports news website (ESPN), a shopping website (Amazon), and two news sites (Google News and CNN).

Overall, we find that the single-core runs have a consistent performance/energy advantage over the quad-core runs. Our experiments also show that performance scales nearly linearly with frequency from 1.2 GHz to 2.4 GHz for all of the websites, implying that loading a webpage is not memory-bound. As 2.4 GHz is the highest performing frequency for all webpage loads, Figure 3 shows the average performance/energy difference between the single-core runs and the quad-core baseline at 2.4 GHz.

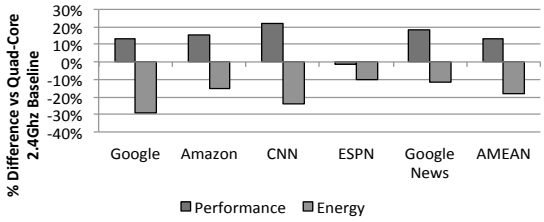


Fig. 3: Average performance/energy difference between a single-core and a quad-core baseline at 2.4GHz.

Figure 3 indicates a 14% performance advantage to using one core and a 17% energy reduction. This demonstrates that the low-amount of thread level parallelism shown in Figure 2b translates to a performance and efficiency advantage for requiring the application to run on a single SMT core. The additional compute resources of the multi-core processor are unnecessary.

While statically pinning all threads to a single core is one answer to the problem, browsers render many diverse workloads, and a static solution may work well for webpage loads, but not for other

scenarios. For example, we observe that loading ESPN (or watching a video in a browser) generates stable, long lived threads with independent working sets. This is a good scenario for the baseline round-robin policy, and these threads benefit from running on an independent core. This diversity requires a dynamic solution, not a simple static solution.

We therefore implement a simple policy that maximizes the thread to data locality that we observe in this application. Our measurements in Figure 3 show that it is generally advantageous to run all threads on as few cores as possible as spawned threads are short-lived and include a large amount of inter-thread data communication. This obseravtion is supported by further Pin cache studies, which show that on average, 39% of L1 cache misses for transient threads is for data that is resident in a different L1 cache on-chip.

We propose that the hardware detects the short lived threads generated during a computation burst. This is advantageous as the hardware requires very little overhead to do so and it can react much faster than the software - a key point during a short burst. Instead of spreading the threads across all cores, the hardware initially clusters threads on the core that the main thread is running on. Once it is established that the spawned thread is independent, it is then migrated to an available core. We describe this policy in Section IV and evaluate it in Section VI.

IV. Activity-Based Migration

We use several insights from our analysis to improve the performance of the static thread clustering policy with a dynamic policy. This dynamic policy (called activity-based migration) initially clusters all spawned threads on the core that their parent thread is running on. If the execution of a thread exceeds a threshold number of instructions (100,000 - as in Figure 2b), we determine the thread to be independent and power on a new core for the thread to execute on, if one is available. The rationale for this policy is that it is inefficient to spawn a short-lived thread on a new core, due to data-migration costs. However, if a thread has executed for a long period of time, it can efficiently utilize the increased resources of a core.

To improve the cost of the migration we track all of the data that each thread has touched in the first level cache (by adding a bit per cache line for each SMT context). Before the thread is migrated to the new core for execution, all of the working set of the thread is transfered to the data cache of the new core. This allows the thread to quickly resume execution and minimizes the cost of the migration. This optimization is denoted as activity-based migration + warmup.

Our evaluation compares activity-based migration to two different thread-to-core mapping policies. The baseline policy assigns threads to each of the four cores in a round-robin manner, starting with core 0. The second policy (static clustering) maps all threads to the same core that their parent thread is running on, if possible. This is effectively what was evaluated on the platform in Section II.

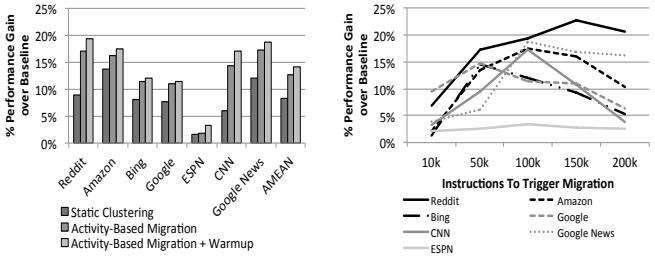
Although thread scheduling is generally handled by the operating system, these policies operate at an instruction granularity that is much finer than the operating system scheduler generally runs. The default for the Linux scheduler is to run on the order of 10's-100 milliseconds – this is between 300 and 3000 of our migration thresholds assuming a 3GHz clock rate and 1 IPC. While we currently set static policy parameters based on our observations (number of threads/instruction count), in future systems feedback from the hardware could allow the operating system to set scheduling parameters that it deems appropriate. These three policies (round-robin, clustering, and activity-based migration), are evaluated in Section VI.

V. Methodology

To evaluate the mechanism described in Section IV we simulate a quad-core system using Pin [7] on Ubuntu 12.04.3 LTS. We model the CPI of each instruction as 1 plus caching latencies, as we do not notice high branch misprediction rates or other events that require detailed core evaluation in our analysis on real hardware. We model a two level cache hierarchy. Each core has a private 32KB data and instruction cache and the four cores share a 1MB last-level cache with a 20-cycle hit latency. A last-level-cache miss results in a 200 cycle memory access. A standard invalidation-based cache coherence protocol is modeled. Each of the four simulated cores is four-way SMT (supports the execution of four threads at once). As we can not force a thread to wait in Pin, if the SMT contexts of Core 0 are currently filled during our clustering policies, threads are mapped to core 1 until all of its SMT contexts are full. This continues as necessary for all four cores.

VI. Simulation Results

Figure 4a displays our simulated performance results, normalized to the round-robin baseline.



(a) (b)

Fig. 4: (a) Simulated performance of clustering and activity-based migration policy relative to a round-robin baseline. (b) Sensitivity to the length of time that a thread runs before it is migrated to a new core.

By enabling the hardware to dynamically make fine-grained thread and data migration decisions, our activity-based migration policy outperforms the clustering policy in Figure 4a, increasing the average performance gain from 8% to 12.5%/14% without/with cache warmup vs. the static clustering policy. We show a performance gain across all of the different webpage loads.

Activity-based migration allows the hardware to migrate a thread to a new core after a threshold number of instructions have been executed. The main parameter to the policy is the number of instructions that a thread executes before migration is triggered. We find 100,000 instructions to be an optimal period before migration across the webpages in our evaluation. Figure 4b shows sensitivity to this parameter for three of the webpages that we simulated. Overall, we notice that a 10k instruction threshold appears to be too short, and we see a performance penalty when compared to the clustering policy. Many of the websites see a performance drop off 150k to 200k instructions, implying that we are reaching the maximum length of time a thread runs for when loading the webpage. Overall there is a general plateau from 50k to 150k instructions, and we choose 100k instructions as the base value to trigger thread/data migration.

Activity-based migration + warmup pre-loads the cache of the new core with the working set of the current thread. Once that thread is scheduled for migration the hardware transfers the working set of the thread to the new data-cache before the thread is set to execute on the new core and its architectural state is migrated. On average, we find that 2.7KB of data is transferred per thread migration and that there are an average of 42 migrations per webpage load.

TABLE I: Number of active threads for each browser running on Windows 8.1 (Mac OS 10.9.4).

Browser	Reddit	Amazon	Bing	Google	ESPN	CNN	Google News
Internet Explorer 11	50	58	65	59	99	102	84
Safari 7	(66)	(58)	(48)	(43)	(58)	(65)	(57)
Chrome 37	56 (68)	79 (64)	67 (73)	65 (77)	67 (84)	80 (85)	64 (73)
Firefox 32	52 (52)	52 (54)	53 (53)	52 (53)	58 (59)	57 (52)	52 (53)

VII. Applicability to other Browsers

Throughout this paper, we study the thread behavior of one application, Firefox. In our analysis, we use data from both hardware (Section III), and dynamic instrumentation (Section VI). This analysis is completed on two different operating systems, a Windows 8 based machine as well as a Ubuntu machine, illustrating that the problem is not just limited to a single operating system.

However, security mechanisms (such as sandboxing and multiple processes) prevent us from using dynamic instrumentation (such as Pin) on browsers besides Firefox. To show that the problem we observe is not limited to just Firefox, we use the activity monitor of two different operating systems (Windows 8, OS X) to corroborate the key observation from Section II: that browsers employ a very large number of active threads. This data is shown in Table I.

To collect this data we disabled each browser's cache and then loaded each of the seven websites in sequence in a single tab. The active threads are monitored on Windows using the task manager and on OS X using the activity monitor. For browsers that spawn multiple processes (Chrome, IE, Safari) we sum the maximum observed thread counts across all active processes. The results shown are the average of three experiments. Overall, all active thread counts are above, and generally much higher than, the 41 threads observed in Section II. The websites with very high thread counts (ESPN, CNN) include streaming video content. With this data, we suggest that our observations are not simply a result of the architecture of Firefox, but rather a trend present in web browsers in general.

VIII. Related Work

To our knowledge, we are the first to suggest clustering threads on cores to optimize short-term data locality. There has been prior work that propose clustering threads from a scheduler perspective. Tam et al. [11] concentrate on optimizing cross-package communication by requiring the system to deliver a summary vector of the memory regions accessed by the application during each scheduler quantum, and then migrate threads based on locality. Vega et al. [12] study PARSEC and also optimize long term data locality. Both of these prior scheduling papers run at operating system quantum intervals.

We are not the first to point out the predominantly single-threaded behavior for many desktop applications. Blake et al. [1] do a wide study (including two web-browsers: Firefox and Safari) and find that many applications show very low effective TLP, while pointing out the potential benefit of SMT in these scenarios. To our knowledge, we are the first to observe that the low TLP is not due to a lack of threading, but rather an inefficient use of threads. Hauser et al. [4] explore why this could be the case and found that programmers generally use threads to structure code and not to increase performance.

Prior work has also characterized mobile applications [10], [8], [6], [3], observing many of the microarchitectural event trends that we also note, including the instruction cache behavior differences between SPEC and web workloads. Rantanaworabhan et al. [9] also point out this behavior, and find that many mobile JavaScript benchmarks are also not representative of real websites. Chadha et al. propose instruction prefetching based on event-signatures [2]. Zhu et al. discuss QoS-driven scheduling techniques for event-based applications [13] and hardware and observe that web-page loading is not generally network limited and can benefit from heavyweight compute hardware [14].

IX. Conclusion

In this paper, we studied the runtime behavior of Firefox as an example of an event-driven, bursty application. We observe that bursty applications have different run-time properties when compared to applications that exhibit data-parallel threading or loopy, steady-state behavior. By taking these differences into account, architects can make modifications to create more specialized, efficient processors.

X. Acknowledgment

We thank Intel Corporation and the Cockrell Foundation for their continued generous financial support of the HPS Research Group.

References

[1] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," in *Proc. of the 37th Annual Intl. Symp. on Computer Architecture (ISCA)*, 2010.

[2] G. Chadha, S. Mahlke, and S. Narayanasamy, "Efetch: Optimizing instruction fetch for event-driven webapplications," in *Proc. of the 23rd Intl. Conference on Parallel Architectures and Compilation (PACT)*, 2014.

[3] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2011.

[4] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser, "Using threads in interactive systems: A case study," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, 1994, pp. 94–105.

[5] "Intel VTune Amplifier XE 2013," https://software.intel.com/en-us/intel-vtune-amplifier-xe, 2013.

[6] C. Lee, E. Kim, and H. Kim, "The AM-Bench: An android multimedia benchmark suite," Tech. Rep. GIT-CERCS-12-04, 2012.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Programming Language Design and Implementation (PLDI)*, 2005.

[8] D. Pandiyan, S.-Y. Lee, and C.-J. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite-mobilebench," in *IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2013.

[9] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn, "Jsmeter: Characterizing real-world behavior of javascript programs," *Microsoft Research Technical Report*, 2009.

[10] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. C. Paver, "A structured approach to the simulation, analysis and characterization of smartphone applications," in *IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2013.

[11] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," in *ACM SIGOPS Operating Systems Review*, 2007.

[12] A. Vega, A. Buyuktosunoglu, and P. Bose, "SMT-centric power-aware thread placement in chip multiprocessors," in *22nd Intl. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[13] Y. Zhu, M. Halpern, and V. Reddi, "Event-based scheduling for energy-efficient qos (eqos) in mobile web applications," in *IEEE 21st Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2015.

[14] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *IEEE 19th Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2013.