The CRISP Performance Model for Dynamic Voltage and Frequency Scaling in a GPGPU

Rajib Nath
University of California, San Diego
9500 Gilman Drive, La Jolla, CA

Dean Tullsen
University of California, San Diego
9500 Gilman Drive, La Jolla, CA

ABSTRACT

This paper presents CRISP, the first runtime analytical model of performance in the face of changing frequency in a GPGPU. It shows that prior models not targeted at a GPGPU fail to account for important characteristics of GPGPU execution, including the high degree of overlap between memory access and computation and the frequency of store-related stalls.

CRISP provides significantly greater accuracy than prior runtime performance models, being within 4% on average when scaling frequency by up to 7X. Using CRISP to drive a runtime energy efficiency controller yields a 10.7% improvement in energy-delay product, vs 6.2% attainable via the best prior performance model.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; C.4 [**Performance of Systems**]: Modeling techniques

Keywords

Critical Path, GPGPU, DVFS

1. INTRODUCTION

This paper describes the CRISP (CRItical Stalled Path) performance predictor for GPGPUs under varying core frequency. Existing analytical models that account for frequency change only target CPU performance, and do not effectively capture the execution characteristics of a GPU.

Dynamic voltage and frequency scaling (DVFS) [1] has shown the potential for significant power and energy savings in many system components including processor cores [2, 3], memory system [4, 5, 6], last level cache [7], and interconnect [8, 9, 10]. DVFS scales voltage and frequency for energy and power savings, but can also address other problems such as temperature [11], reliability [12], and variability [13]. However, in all cases it trades off performance for other gains, so properly setting DVFS to maximize particular goals can only be done with the assistance of an accurate estimate of the performance of the system at alternate frequency settings.

While extensive research has been done on DVFS performance modeling for CPU cores, there is little guidance in the literature for GPU and GPGPU settings, despite the fact that modern GPUs have extensive support for DVFS [14, 15, 16, 17]. However, the potential for DVFS on GPUs is high for at least two reasons. First, the maximum power consumption in a GPGPU is often higher than CPUs, e.g., 250 Watts for the NVIDIA GTX480. Second, while the range of DVFS settings in modern CPUs is shrinking, that is not yet the case for GPUs, which have ranges such as 0.5V-1.0V (NVIDIA Fermi [18]).

This work presents a performance model that closely tracks GPGPU performance on a wide variety of workloads, and significantly outperforms existing models which are tuned to the CPU. We show that the GPU presents key differences that are not handled by those models.

Most DVFS performance models are empirical and use statistical [19, 20, 21, 22, 23] or machine learning methods [24, 25], or assume a simple linear relationship [26, 27, 28, 29, 30, 31, 32] between performance and frequency. Abe et al., [14] target GPUs, but with a regression based offline statistical model that is neither targeted for, nor conducive to, runtime analysis.

Recent research presents new performance counter architectures [33, 34, 35] to model the impact of frequency scaling on CPU workloads. These analytical models (e.g., leading load [33] and critical path [34]) divide a CPU program into two disjoint segments representing the non-pipelined (T_{Memory}) and pipelined ($T_{Compute}$) portion of the computation. They make two assumptions: (a) the memory portion of the computation never scales with frequency, and (b) cores never stall for store operations. Though these assumptions are reasonably valid in CPUs, they start to fail as we move to massively parallel architecture like GPGPUs. No existing runtime analytical models for GPGPUs handle the effect of voltage frequency changes.

MICRO 2015 Waikiki, Hawaii USA

A program running on a GPGPU, typically referred to as a kernel, will have a high degree of thread level parallelism. The single instruction multiple thread (SIMT) parallelism in GPGPUs allows a significant portion of the computation to be overlapped with memory latency, which can make the memory portion of the computation elastic to core clock frequency and thus breaks the first assumption of the prior models. Though memory/computation overlap has been used for program optimization, modeling frequency scaling in the presence of abundant parallelism and high memory/computation overlap is new and a complex problem. Moreover, due to the wide single instruction multiple data (SIMD) vector units in GPGPU streaming multiprocessors (SMs) and homogeneity of computation in SIMT scheduling, GPGPU SMs often stall due to stores.

The CRISP predictor accounts for these differences, plus others. It provides accuracy within 4% when scaling from 700 MHz to 400 MHz, and 8% when scaling all the way to 100 MHz. It reduces the maximum error by a factor of 3.6 compared to prior models. Additionally, when used to direct GPU DVFS settings, it enables nearly double the gains of prior models (10.7% EDP gain vs 5.7% with critical path and 6.2% with leading load). We also show CRISP effectively used to reduce ED^2P, or to reduce energy while maintaining a performance constraint.

2. MOTIVATION AND RELATED WORK

Our model of GPGPU performance in the presence of DVFS builds on prior work on modeling CPU DVFS. Since there are no analytical DVFS models for GPGPUs, we will describe some of the prior work in the CPU domain and the shortcomings of those models when applied to GPUs. We also describe recent GPU-specific performance models at the end of this section.

2.1 CPU-based Performance Models for DVFS

Effective use of DVFS relies on some kind of prediction of the performance and power of the system at different voltage and frequency settings. An inaccurate model results in the selection of non-optimal settings and lost performance and/or energy.

Existing DVFS performance models are primarily aimed at the CPU. They each exploit some kind of linear model, based on the fact that CPU computation scales with frequency speed while memory latency does not. These models fall into four classes: (a) proportional, (b) sampling, (c) empirical, and (d) analytical. The proportional scaling model assumes a linear relation between the performance and the core clock frequency. The sampling model better accounts for memory effects by identifying a scaling factor (i.e., slope) of each application from two different voltage-frequency operating points. The empirical model approximates that slope by counting aggregate performance counter events, such as the number of memory accesses. This model cannot distinguish between applications with low or high memory level parallelism (MLP) [33].

An analytical model views a program as an alternating sequence of compute and memory phases. In a compute phase, the core executes useful instructions without generating any memory requests (e.g., instruction fetches, data loads) that reach main memory. A memory request which misses the last level cache might trigger a memory phase, but only when a resource is exhausted and the pipeline stalls. Once the instruction or data access returns, the core typically begins a new compute phase. In all the current analytical models, the execution time (T) of a program at a voltage-frequency setting with cycle time t is expressed as

$$T(t) = T_{Compute}(t) + T_{Memory} \quad (1)$$

They strictly assume that the pipeline portion of the execution time ($T_{Compute}$) scales with the frequency, while the non-pipeline portion (T_{Memory}) does not. The execution time at an unexplored voltage-frequency setting of v_2,f_2 can be estimated from the measurement in the current settings v_1,f_1 by

$$T(v_2, f_2) = T_{Compute}(v_1, f_1) \times \frac{f1}{f2} + T_{Memory} \quad (2)$$

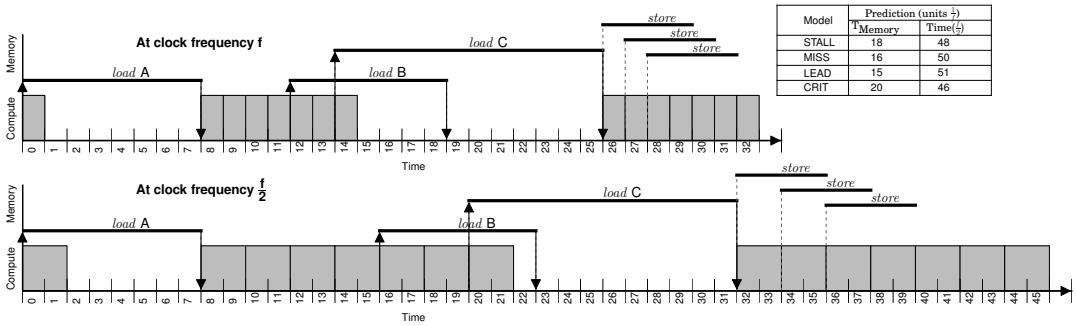
The accuracy of these models relies heavily on accurate classification of cycles into compute or memory phase cycles, and this is the primary way in which these models differ. We describe them next.

Stall Time: Stall time [35] estimates the non-pipeline phase (T_{Memory}) as the number of cycles the core is unable to retire any instructions due to outstanding last level cache misses. It ignores computation performed during the miss before the pipeline stall. Since the scaling of this portion of computation can be hidden under memory latency, stall time overpredicts execution time.

Miss Model: The Miss model [35] includes ROB fill time inside the memory phase. It identifies all the stand alone misses, but only the first miss in a group of overlapped misses as a contributor to T_{Memory}. In particular, it ignores misses that occur within the memory latency of an initial miss. The resulting miss count is then multiplied by a fixed memory latency to compute T_{Memory}. This approach loses accuracy due to its fixed latency assumption and also ignores stall cycles due to instruction cache misses.

Leading Load: Leading load [33, 36] recognizes both front-end stalls due to instruction cache misses and back-end stalls due to load misses. Their model counts the cycles when an instruction cache miss occurs in the last level cache. It adopts a similar approach to the miss model to address memory level parallelism for data loads. However, to account for variable memory latency, it counts the actual number of cycles a contributor load is outstanding.

Critical Path: Critical path [34] computes T_{Memory} as the longest serial chain of memory requests that stalls the processor. The proposed counter architecture maintains a global critical path counter (CRIT) and a critical path time stamp register (τ_i) for each outstanding load i. When a miss occurs in the last level cache (instruction or data), the snapshot of the global critical path counter is copied into the load's time stamp register.



Model	Prediction (units $\frac{1}{f}$)	
	T_{Memory}	Time($\frac{1}{f}$)
STALL	18	48
MISS	16	50
LEAD	15	51
CRIT	20	46

Figure 1: T_{Memory} computation for a CPU program running at frequency f with different linear performance models

Once the load i returns from memory, the global critical path counter is updated with the maximum of CRIT and $\tau_i + L_i$ where L_i is the memory latency for load i.

Example: Figure 1 provides an illustrating example of a CPU program that has one stand alone load miss (A at cycle 0) and two overlapped load misses (B at cycle 12 & C at cycle 14). At clock frequency f, the core stalls between cycles 1-7 and 15-25 for the loads to complete. The total execution time of the program at clock frequency f is 33 units (18 units stalls + 15 units computation). As we scale down the frequency to $\frac{f}{2}$, the compute cycles (0, 8-14, 26-32) scale by a factor of 2. However, the expansion of compute cycles at 0 and 14 are hidden by the load A and C respectively. Thus the execution time at frequency $\frac{f}{2}$ becomes 46 units.

For each performance model, we identify the cycles classified as the non-pipelined (does not scale with the frequency of the pipeline) portion of computation. Leading load (LEAD) computes T_{Memory} as the sum of A and B's latencies (7+8=15). Miss model (MISS) identifies two contributor misses (A & B) and reports T_{Memory} to be 16 (8×2). Among two dependent load paths (A+B with length 8+7=15, A+C with length 8+12=20), critical path (CRIT) selects A+C as the longest chain of dependent memory requests and computes the critical path as 20. Stall time (STALL) counts the 18 cycles of load related stalls.

We plug in the value of T_{Memory} in equation 2 and predict the execution time at clock frequency $\frac{f}{2}$. The predicted execution time by the different models is shown in Figure 1. Note that the models also differ in their counting of $T_{Compute}$. Critical path (CRIT) outperforms the other models in this example. Although these models show reasonable accuracy in CPU, optimization techniques like software and hardware prefetching may introduce significant overlap between computation and memory load latency, and thus invalidate the non-elasticity assumption of the memory component. In that case, the CRISP model should also provide significant improvement for CPU performance prediction.

2.2 DVFS in GPGPUs

To explore the applicability of DVFS in GPGPUs, we simulate an NVIDIA Fermi architecture using GPGPUSim [37] and run nine kernels from the Rodinia [38] benchmark suite at different core clock frequencies –

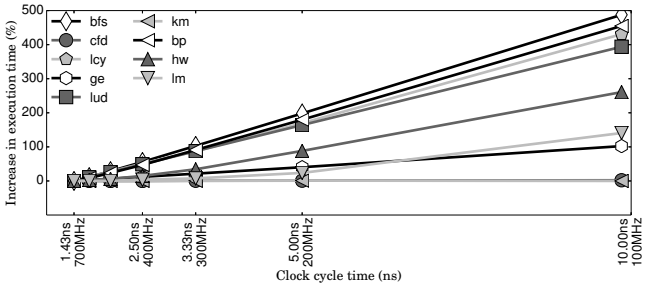


Figure 2: Performance impact of core clock frequency scaling in a subset of GPGPU kernels from rodinia benchmark suite

from 700MHz (baseline) to 100MHz (minimum). The performance curve in Figure 2 shows the increase in runtime as we scale down the core frequency i.e. increase clock cycle time (x-axis). We make two major observations from this graph. First, like CPUs, there is significant opportunity for DVFS in GPUs – we see this because two of the kernels (cfd and km) are memory bound and insensitive to frequency changes (they have a flat slope in this graph), while three others (*ge*, *lm*, and *hw*) are only partially sensitive. For the memory bound kernels, it is possible to reduce clock frequency and thus save energy without any performance overhead.

Our second observation is that the linear assumption of all prior models does not strictly hold, as the partially sensitive kernels (e.g., *lm* and *hw*) seem to follow two different slopes, depending on the frequency range. At high frequency, they follow a flat slope, as if they were memory bound (e.g., between 700MHz and 300MHz in *lm*), but at lower frequencies, they scale like compute bound applications (e.g., below 300MHz in *lm*).

So while several of the principles that govern DVFS in CPUs also apply to GPUs, we identify several differences which make the existing models inadequate for GPUs. All of these differences stem from the high thread level parallelism in GPU cores, resulting from the SIMT parallelism enabled through warp scheduling. The first difference is the much higher incidence of overlapped computation and memory access. The second is the homogeneity of computation in SIMT scheduling, resulting in the easy saturation of resources like the store queue. The third is the difficulty of classifying stalls (e.g., an idle cycle may occur when 5 warps are stalled for memory and three are stalled for float-

ing point RAW hazards). We will discuss each of these differences, as well as a fourth that is more of an architectural detail and easily handled.

Memory/Computation Overlap: Existing models treat computation that happens in series with memory access the same as computation that happens in parallel. When the latter is infrequent, those models are fairly accurate, but in a GPGPU with high thread level parallelism, it is common to find significant computation that overlaps with memory.

This overlapped computation has more complex behavior than the non-overlapped. In the top timeline in Figure 1 the computation at cycle 0 is overlapped with memory, so even though it is elongated it still does not impact execution time – thus, that computation looks more like memory than computation. However, if we extend the clock enough (i.e., more than a factor of 8 in this example), that computation completely covers the memory latency and then begins to impact total execution time – now looking like computation again. This phenomenon exactly explains the two-slope behavior exhibited by *hw* and *lm* in Figure 2.

Store Stalls: A core (CPU or GPU) would stall for a store only when the store queue fills, forcing it to stall waiting for one of the stores to complete. While this is a rare occurrence in CPUs, in a GPU with SIMT parallelism, it is common for all of the threads to execute stores at once, easily flooding the store queue, resulting in memory stalls that do not scale with frequency.

Complex Stall Classification: Identifying the cause of a stall cycle is difficult in the presence of high thread level parallelism. For example if several threads are stalled waiting for memory, but a few others are stalled waiting for floating point operations to complete, is this a computation or memory phase? Prior works (which start counting memory cycles only after the pipeline stalls) would categorize this as computation because the pipeline is not yet fully stalled for memory, and the ongoing floating point operations will scale with frequency.

Consider what happens when, as described earlier, the overlapped computation is scaled to the point where it completely covers the memory latency and now starts to interfere with other computation. The originally idle cycles (e.g. between dependent floating point ops) will not contribute to extending execution time, but rather will now be interleaved with the other computation. Only the cycles where the pipeline was actually issuing instructions will impact execution time. Thus, unlike other models, we generally treat mixed-cause idle cycles (both memory and CPU RAW hazards) as memory cycles rather than computation cycles.

L1 Cache Miss: A substantial portion of memory latency in a typical GPGPU is due to the interconnect that bridges the streaming cores to L2 cache and GPU global memory. The leading load and critical path models start counting cycles as the load request misses the last level cache, which works for CPUs where all on-chip caches are clocked together; however, in our GPU the interconnect and last level cache are in an independent clock domain.

Since this is not a shortcoming in the models, just a change in the assumed architecture, we adapt these models (as well as our own) to begin counting memory stalls as soon as they miss the L1 cache. This results in significantly improved accuracy for those models on GPUs. Thus, all results shown in this paper for leading load and critical path incorporate that improvement.

2.3 Models for GPGPUs

A handful of power and energy models have recently been proposed for GPGPUs [18]. However, very few of these models involve performance predictions. Dao et al. [39] proposes linear and machine learning based models to estimate the runtime of kernels based on workload distribution. The integrated GPU power and performance (IPP) [40] model predicts the optimum number of cores for a GPGPU kernel to reduce power. However, they do not address the effect of clock frequency on GPU performance. Moreover, they ignore cache misses [41], which is critical for a DVFS model. Song et al. [42] contribute a similar model for a distributed system with GPGPU accelerators. Chen et al. [43] tracks NBTI-induced threshold voltage shift in GPGPU cores and determines the optimum number of cores by extending IPP. Equalizer [44] maintains performance counters to classify kernels as compute-intensive, memory-intensive, or cache sensitive, using those classifications to change thread counts or P-states. However, because they have no actual model to predict performance, it still amounts to a sampling-based search for the right DVFS state.

None of these models address the effect of core clock frequency on kernel performance. CRISP is unique in that it is the first runtime analytical model, *either for CPUs or GPGPUs*, to model the non linear effect of frequency on performance – this effect can be seen in either system, but is just more prevalent on GPGPUs.

The only GPU analytical model that accounts for frequency is an empirical model [45] that builds a unique non linear performance model for each workload – this is intended to be used offline and is not practical for an online implementation. Abe, et al. [14, 15] also present an offline solution. They use a multiple linear regression based approach with higher average prediction error (40%). Their maximum error is more than 100%.

3. GPGPU DVFS PERFORMANCE MODEL

This section describes the **c**ritical **s**talled **p**ath (CRISP) performance predictor, a novel analytical model for predicting the performance impact of DVFS in modern many-core architectures like GPGPUs.

3.1 Critical Stalled Path

Our model interprets computation in a GPGPU as a collection of three distinct phases: load outstanding, pure compute, and store stall (Figure 3). The core remains in a pure compute phase in the absence of an outstanding load or any store related stall. Once a fetch or load miss occurs in one of the level 1 caches (con-

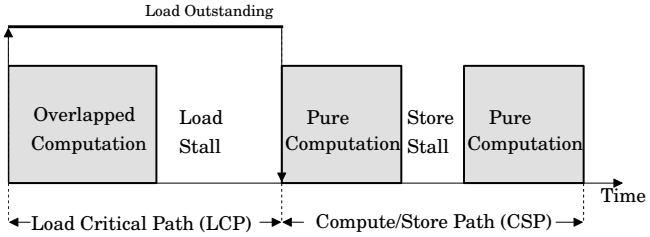


Figure 3: Critical Stalled Path - Abstract View

stant, data, instruction, texture), the core enters a load outstanding phase and stays there as long as a miss is outstanding. During this phase, the core may schedule instructions or stall due to control, data, or structural hazards. The store stall phase represents the idle cycles due to store operations, in the explicit case where the pipeline is stalled due to store queue saturation.

We split the total computation time into two disjoint segments: a load critical path (LCP) and a compute/store path (CSP).

$$T(t) = T_{LCP}(t) + T_{CSP}(t) \tag{3}$$

This division allows us to separately analyze (1) loads and the computation that competes with loads to become performance-critical, versus (2) computation that never overlaps loads and is exposed regardless of the behavior of the loads. Store stalls fall into the latter category because any cycle that stalls in the presence of both loads and stores is counted as a load stall.

The load critical path is the length of the longest sequence of dependent memory loads possibly overlapped with computations from parallel warps. Thus, it does not necessarily include all loads, enabling it to account for memory level parallelism by not counting loads that occur in parallel – similar to [34, 33]. Meanwhile, we form the compute/store path as a summation of non-overlapped compute phases, store stall phases, and computation overlapped only by loads not deemed part of the critical path. Since the two segments in Equation 3 scale differently with core clock frequency, we develop separate models for each.

3.2 Load Critical Path Portion

Streaming multiprocessors (SMs) have high thread level parallelism. The maximum number of concurrently running warps per SM varies from 24 to 64 depending upon the compute capability of the device (e.g., 48 in NVIDIA Fermi architecture). This enables the SM to hide significant amounts of memory latency, resulting in heavy overlap between load latency and useful computation. However, it often does not hide all of it, due to lack of parallelism (either caused by low inherent parallelism, or restricted parallelism due to resource constraints) or poor locality.

We represent the load critical path as a combination of memory related overlapped stalls ($T_{LCP}^{\overline{Stall}}$) and overlapped computations ($T_{LCP}^{\overline{Comp}}$). Prior CPU-based models' assumption of T_{Memory}'s inelasticity creates a larger inaccuracy in GPGPUs than it does for CPUs because of the magnitude of $T_{LCP}^{\overline{Comp}}$, which scales linearly with

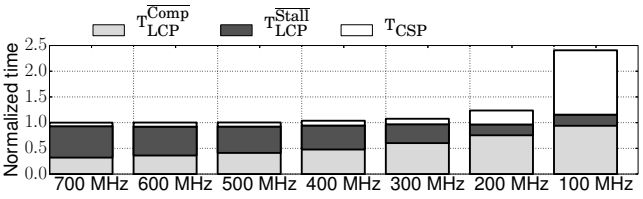


Figure 4: Runtime of the *lm* kernel at different frequencies

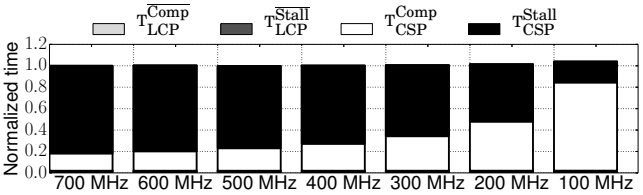


Figure 5: Runtime of the *cfd* kernel at different frequencies

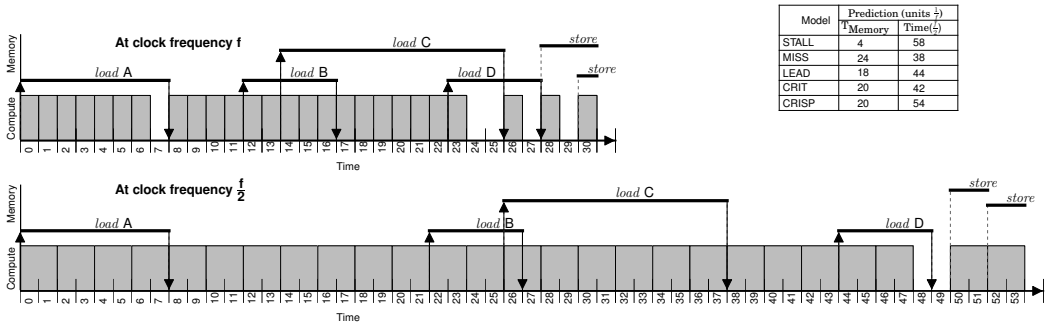
frequency. The expansion of overlapped computation stays hidden under T_{Memory} as long as the frequency scaling factor is less than the ratio between T_{Memory} and the overlapped computation. As the frequency is scaled further, the load critical path can be dominated by the length of the scaled overlapped computation. Thus, we define the LCP portion of the model as

$$T_{LCP}(t) = max(T_{Memory}, \frac{f_1}{f_2} \times T_{LCP}^{\overline{Comp}}) \tag{4}$$

We describe details on parameterizing this equation from hardware events in section 3.5 and section 3.6. Figure 4 shows the normalized execution time of the *lm* kernel for 7 different frequencies. As expected, we see the load stalls decreasing with lower frequency, but without increasing total execution time for small changes in frequency. As the frequency is decreased further, the overlapped computation increases, with much of it turning into non-overlapped computation (as some of the original overlapped computation now extends past the duration of the load) resulting in an increase in total execution time. Prior models assume that overlapped computation always remains overlapped, and miss this effect.

3.3 Compute/Store Path Portion

The compute/store critical path includes computation (T_{CSP}^{Comp}) that is not overlapped with memory operations and simply scales with frequency, plus store stalls (T_{CSP}^{Stall}). Streaming multiprocessors (SMs) have a wide SIMD unit (e.g., 32 in Fermi). A SIMD store instruction may result in 32 individual scalar memory store operations if uncoalesced (e.g., A[tid * 64] = 0). As a result, the load-store queue (LSQ) may overflow very quickly during a store dominant phase in the running kernel. Eventually, the level 1 cache runs out of miss status holding register (MSHR) entries and the instruction scheduler stalls due to the scarcity of free entries in the LSQ or MSHR. We observe this phenomenon in several of our application kernels, as well as microbenchmarks we used to fine-tune our models. Again, this is



Model	Prediction (units $\frac{1}{f}$)	
	T_{Memory}	Time($\frac{1}{f}$)
STALL	4	58
MISS	24	38
LEAD	18	44
CRIT	20	42
CRISP	20	54

Figure 6: T_{Memory} computation for a GPU program running at frequency f with different linear performance models

a rare case for CPUs, but easily generated on a SIMD core. Although the store stalls are memory delays, they do not remain static as frequency changes. As frequency slows, the rate (as seen by the memory hierarchy) of the store generation slows, making it easier for the LSQ and the memory hierarchy to keep up. A simple model (meaning one that requires minimal hardware to track) that works extremely well empirically assumes that the frequency of stores decreases in response to the slowdown of non-overlapped computation (recall, the overlapped computation may also generate stores, but those stores do not contribute to the stall being considered here). Thus we express the compute/store path by the following equation in the presence of DVFS.

$$T_{CSP}(t) = max(\mathrm{T}_{\mathrm{CSP}}^{\mathrm{Comp}} + \mathrm{T}_{\mathrm{CSP}}^{\mathrm{Stall}}, \frac{f_1}{f_2} \times \mathrm{T}_{\mathrm{CSP}}^{\mathrm{Comp}}) \qquad (5)$$

We see this phenomenon, but only slightly, in Figure 4, but much more clearly in Figure 5, which shows one particular store-bound kernel in *cfd*. As frequency decreases, computation expands but does not actually increase total execution time, it only decreases the contribution of store stalls.

We assume that the stretched overlapped computation and the pure computation will serialize at a lower frequency. This is a simplification, and a portion of CRISP's prediction error stems from this assumption. This was a conscious choice to trade off accuracy for complexity, as tracking those dependencies between instructions is another level of complexity and tracking.

In summary, then, CRISP divides all cycles into four distinct categories: (1) *pure computation* which is completely elastic with frequency, (2) *load stall* which as part of the load critical path is inelastic with frequency, (3) *overlapped computation* which is elastic, but potentially hidden by the load critical path, and (4) *store stall*, which tends to disappear as the pure computation stretches.

3.4 Example

We illustrate the mechanism of our model in Figure 6 where loads and computations are from all the concurrently running warps in a single SM. The total execution time of the program at clock frequency f is 31 units (5 units stall + 26 units computation). As we scale down the frequency to $\frac{f}{2}$, all the compute cycles scale by a factor of two. Due to the scaling of overlapped computation under load A (cycles 0-6) at clock frequency $\frac{f}{2}$, the compute cycle 8 cannot begin before time unit 14. Similarly, the instruction at compute cycle 28 cannot be issued until time unit 50 because the expansion of overlapped computation under load C (cycles 14-23) pushes it further. Meanwhile, as the compute cycle 28 expands due to frequency scaling, they overlap with the subsequent store stalls.

For each of the prior performance models, we identify the cycles classified as the non-pipelined portion of computation (T_{Memory}). Leading load (LEAD) computes $\mathrm{T_{Memory}}$ as the sum of A, B, and D's latencies (8+5+5=18). Miss model (MISS) identifies three contributor misses (A, B, & D) and reports $\mathrm{T_{Memory}}$ to be 24 (8 × 3). Among two dependent load paths (A+B+D with length 8+5+5=18, A+C with length 8+12=20), CRISP computes the load critical path (LCP) as 20 (for the longest path A+C). The stall cycles computed by STALL is 4. We plug in the value of $\mathrm{T_{Memory}}$ in equation 2 and predict the execution time at clock frequency $\frac{f}{2}$. As expected, the presence of overlapped computation under $\mathrm{T_{Memory}}$ and store related stalls introduces error in the prediction of the prior models. STALL model overpredicts while the rest of the models underpredict.

Our model observes two load outstanding phases (cycles 0-7 and 12-27), three pure compute phases (cycles 8-11, 28, 30), and one store stall phase (cycle 29). CRISP computes the load critical path (LCP) as 20 (8+12 for load A-C). The rest of the cycles (11=31-20) are assigned to compute/store path (CSP). The overlapped computation under LCP is 17 (cycles 0-6, 14-23). The non overlapped computation in CSP is 10 (cycles 8-13, 26-28,30). Note that cycle 29 is the only store stall cycle. CRISP computes the scaled LCP as max (17 × 2, 20) = 34 and CSP as max (10 × 2, 11) = 20. Finally, the predicted execution time by CRISP at clock frequency $\frac{f}{2}$ is 54 (34 + 20) which matches the actual execution time (54).

3.5 Parameterizing the Model

To utilize our model, the core needs to partition execution cycles along some fairly subtle distinctions, separating two types of computation cycles and two types of memory stalls, while accounting for the fact that in

a heavily multithreaded core there are often a plethora of competing causes for each idle cycle.

Despite these issues, we are able to measure all of these parameters with some simple rules and a very small amount of hardware. First, we categorize each cycle as either one of two types of stalls (load stalls within the load critical path vs store stalls within the compute/store path) or computation. Next we split computation into overlapped computation (within the load critical path) or non-overlapped.

Identifying Stalls: Memory hierarchy stalls can originate from instruction cache fetch misses, load misses, or store misses. The first two we categorize as load stalls and the last only manifests in pipeline stalls when we run out of LSQ entries or MSHRs. We initially separate these two types of stalls from computation cycles, then break down the computation cycles in the next section.

Identifying the cause of stall cycles in a heavily multithreaded core is less than straightforward. We assume the following information is available in the pipeline or L1 cache interface – the existence of outstanding load misses, the existence of outstanding store misses, the existence of a fetch stall in the front end pipeline, MSHR or LSQ full condition, and the existence of RAW hazard on a memory operation, a RAW hazard on an arithmetic operation causing an instruction to stall, or an arithmetic structural hazard. Most of those are readily available in any pipeline, but the last three might each require an extra bit and some minimal logic added to the GPU scoreboard.

Because our GPU can issue two instructions (two arithmetic or one arithmetic and one load/store), characterizing cycles as idle or busy is already a bit fuzzy. Two-issue cycles are busy (computation), zero-issue cycles are considered idle, while one-issue cycles will be counted as busy if there is an arithmetic RAW or structural hazard, otherwise it will be characterized as idle since there is a lack of sufficient arithmetic instructions to fully utilize the issue bandwidth.

Idle cycles will be characterized as load stalls, store stalls, or computation, according to the following algorithm, the order of which determines the characterization of stalls in the presence of multiple potential causes:

(A) If there is a control hazard (branch mispredict), cache/shared memory bank conflict, or no outstanding load or store misses or fetch stall, this cycle is counted as computation. Branch mispredicts are resolved at the speed of the pipeline, as are L1 cache and shared memory bank conflicts (L1 cache and shared memory run on the same clock as the pipeline).

(B) Else, if there is an outstanding load miss, then if there is a RAW hazard on the load/store unit, this cycle is a load stall. Otherwise, if there is an arithmetic RAW or structural hazard, then this cycle is computation. If neither is the case and the MSHR or LSQ are full, then this is a load stall. Finally, if there is a fetch stall in the front end, then it is a load stall.

(C) Else (no outstanding load miss), then if the MSHR or LSQ are full, this indicates a store stall.

(D) All other stalls must be solely due to arithmetic dependencies, and are classified as computation.

Classifying computation: After identifying the total cycles of computation, we divide those into two groups – the pure computation that is part of the compute/store path and the overlapped computation that is overlapped by the load-critical path. The former includes both computation that does not overlap loads and computation that overlaps loads that are not part of the critical path. The reason for dividing computation into these sets is that the compute/store computation is always exposed (execution time is sensitive to their latency) while the latter only become exposed when they exceed the load stalls. We describe the mechanism to compute the computation under the load critical path here, but revisit the need to precisely track the critical path in Section 3.4.

In order to track the computation that is overlapped by the load-critical path, we need to track the critical path using a mechanism similar to the critical path model [34]. This is done using a counter to track the global longest critical path (the current T_{Memory} up to this point), as well as counters for each outstanding load that record the critical path length when they started, so that we can decide when they finish if they extend the global longest critical path. However, we instead measure something we call the *adjusted load critical path*, which adds load stalls as 1-cycle additions to the running global longest critical path. In this way, when the longest critical path is calculated, its length will include both the critical-path load latencies and the stalls that lie between the critical loads – the non-critical load stalls. That adjusted load critical path is the $T_{LCP}(t)$ from Equation 3, which is also the "load critical path" from figure 3 and includes all load stalls (either as part of the critical load latencies or the non-critical stalls) as well as all overlapped computation. As a result, we can solve for the overlapped computation by simply subtracting the total load stalls from the adjusted load critical path; this is possible because we also record the total load stalls in a counter.

In the top timeline in Figure 6, the original critical path computation would count 8 (latency of A) + 12 (latency of C) = 20 cycles as the load critical path, but our adjusted load critical path will also pick up the stall at cycle 27, so would calculate 21 as our adjusted load critical path. At the same time, we would count 4 total load stall cycles. Subtracting 21 - 4 gives us 17 cycles of load-critical overlapped computation, which is exactly the number of computation cycles under the two critical path loads (A and C).

3.6 Hardware Mechanism and Overhead

We now describe the hardware mechanisms that enable CRISP. It maintains three counters: adjusted load critical path (T_{Memory}), load stall ($T_{LCP}^{\overline{Stall}}$), and store stall ($T_{CSP}^{Stall}$). The rest of the terms in equation 3 and 4 can be computed from these three counters and total time by simple subtraction. Like the original critical path (CRIT) model, it also maintains a critical path

CRISP Component	Count	Overhead (Bytes)	
		Per Element	Total
Adjusted LCP	1	4	4
Time stamp registers	164	4	656
Overlapped stall	1	4	4
Non overlapped stall	1	4	4
Total Storage Overhead(Bytes)		CRISP	668
		CRIT	660
		LEAD	18
		STALL	4

Table 1: Hardware storage overhead per SM of CRISP

timestamp register (ts_i) for each outstanding miss (i) in the L1 caches (one per MSHR). Table 1 shows the total hardware overhead for these counters, 99% of which is inherited from CRIT. We discuss the software overhead in Section 4.

At the beginning of each kernel execution, the counters are set to zero. Once a load request i (instruction, data, constant, or texture) misses in the level 1 cache, CRISP copies $\mathrm{T_{Memory}}$ into the pending load's timestamp register (ts_i). When the load i's data is serviced from the L2 cache or memory after L_i cycles, $\mathrm{T_{Memory}}$ is updated with the maximum of $\mathrm{T_{Memory}}$ and $ts_i + L_i$. In the occurrence of a load stall, CRISP increments $\mathrm{T_{Memory}}$ and $\mathrm{T_{LCP}^{\overline{Stall}}}$. For each non-overlapped (store) stall, CRISP increments $\mathrm{T_{CSP}^{Stall}}$.

An unoptimized implementation of the logic that detects load and store stall needs just 12 logic gates and is never on the critical path. Most of the signals used in the cycle classification logic are already available in current pipelines. To detect different kinds of hazards, we alter the GPU scoreboard, which requires 1 bit per warp in the GPU hardware scheduler and minimal logic.

4. METHODOLOGY

CRISP, like other existing online analytical models, requires specific changes in hardware. As a result, we rely on simulation (specifically, the GPGPUSim [37] simulator) to evaluate the benefit of our novel performance model, as in prior work (CRIT [34], LEAD [33, 36]). The GPGPU configuration used in our experiment is very similar to the NVIDIA GTX 480 Fermi architecture (in Section 5.4, we also evaluate a Tesla C 2050). To evaluate the energy savings realized by CRISP, we update the original GPGPUSim with online DVFS capability. All the 15 SMs in our configured architecture share a single voltage domain with a nominal clock frequency of 700 MHz. We define six other performance states (P-State) for the SMs – ranging from 100 MHz to 600 MHz with a step size of 100 MHz [18]. Similar to GPUWattch [18], we select the voltage for each P-State from 0.55V to 1.0V. We rely on a fast-responding on-chip global DVFS regulator that can switch between P-States in 100ns [46, 18, 47, 48]. We account for this switching overhead for each P-State transition in our energy savings result.

We extend the GPUWattch [18] power simulator in GPGPUSim to estimate accurate power consumption in the presence of online DVFS. For each power calculation interval of GPUWattch, we collect the dynamic power at the nominal voltage-frequency (700MHz, 1V) setting and scale it for the current voltage-frequency ($\mathrm{P_{Dynamic}} \propto v^2 f$). We use GPUSimpow [49] to estimate static power of individual clock domains at nominal frequency, which is between 23%-63% for the Rodinia benchmark suite. The static power in other P-States are estimated using voltage scaling ($\mathrm{P_{Static}} \propto v$) [50].

We use the Rodinia [38] benchmark suite in our experiments. To initially assess the accuracy of different models, we run the first important kernel of each Rodinia benchmark for 4.8 million instructions – we exclude *nw* because the runtime of each kernel is smaller than a single reasonable DVFS interval. The 4.8 million instruction execution of each kernel at nominal frequency translates into at least 10μs execution time (ranging from 10μs to 600μs). We find that 10μs is the minimum DVFS interval for which the switching overhead (100ns) of the on chip regulator is negligible ($\leq$ 1%). Meanwhile, to evaluate the energy savings, we run the benchmarks for at least 1 billions instructions (except for *myc*, which we ran long enough to get 4286 full kernel executions). For some benchmarks, we ran for up to 9 billion instructions if that allowed us to run to completion. Eight of the benchmarks finish execution within that allotment. For presentation purposes, we divide the benchmarks into two groups. The benchmarks (bp, hw, hs, and patf) with at least 75% computation cycles are classified as compute bound. The rest of the benchmarks are memory bound.

In Section 5.2, we demonstrate an application of our performance model in an online energy saving framework. In order to make decisions about DVFS settings to optimize EDP or similar metrics, we need both a performance model and an energy model. The energy model we use assumes that the phase of the next interval will be computationally similar to the current interval [34], which is typically a safe assumption for GPUs.

At each decision interval (10μs), the DVFS controller uses the power model to find the static and dynamic power of the SMs at the current settings (v_c,f_c). It also collects static and dynamic power consumed by the rest of the components including memory, interconnect, and level 2 cache. For each of the possible voltage-frequency settings (v,f), it predicts (a) static power of the SMs using voltage scaling, (b) dynamic power of the SMs using voltage and frequency scaling, (c) execution time $\mathrm{T}(f)$ using the underlying DVFS performance model (leading load, stall time, critical path, or critical stalled path). The static and dynamic power of the rest of the system are assumed to remain unchanged since we do not apply DVFS to those domains.

The controller computes the total static energy consumption at each frequency f as a product of total static power $\mathrm{P_{Static}}(v, f)$ and $\mathrm{T}(f)$. Meanwhile, the total dynamic energy consumption at frequency f is computed as $\mathrm{P_{Dynamic}}(f) \times \mathrm{T}(f_c)$. The static and dynamic energy components are added together to estimate the predicted energy $\mathrm{E}(f)$ at P-State (v,f). Finally, the

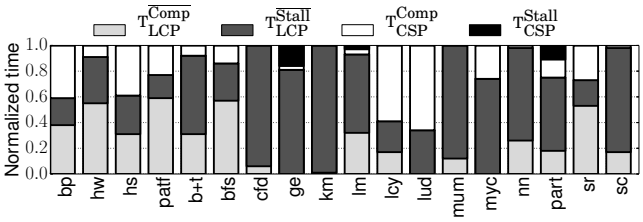


Figure 7: Breakdown of execution time

controller computes $ED^2P(v, f)$ as $E(f) \times T(f)^2$ and makes a transition to the P-State with the predicted minimum ED^2P. For the EDP operating setting, the controller orchestrates DVFS in similar fashion, with minimum EDP as the goal.

We measure the online software overhead at 109 cycles per decision interval by running the actual code on a CPU. Note that the same software could also run on an SM. This overhead is included in our modeled performance overhead, in addition to the 1% P-state switching overhead. It should be noted that we do not have any overhead for measurement since we implement it in the hardware using performance counters, except for reading the counters which is part of our software overhead.

In our results, we average performance counter statistics across all SMs in the power domain to drive our performance model. However, our experiments (not shown here) demonstrate that we could instead sample a single core with no significant loss in effectiveness.

5. RESULTS

This section examines the quality of the CRISP predictor in two ways. First, it evaluates the accuracy of the model in predicting kernel runtimes at different frequencies, compared to the prior state of the art. Second, it uses the predictor in a runtime algorithm to predict the optimal DVFS settings and measures the potential gains in energy efficiency. In each case we compare against leading load (LEAD), critical path (CRIT) and stall time (STALL). We also explore the performance of a computationally simpler version of CRISP. Last, we examine the generality of CRISP.

5.1 Execution Time Prediction

We simulate the first instance of a key kernel of each Rodinia [38] benchmark at all seven P-states and collect the execution time information at each frequency. We also save the performance model counter data at the nominal frequency (700 MHz) to drive the various performance models. We show the breakdown of the classifications computed by CRISP in Figure 7 – this raw data on each benchmark is useful in understanding some of the sources of prediction error. This data shows several interesting facts – (1) the benchmarks are diverse, (2) while the load critical path often covers the majority of execution time, the overlapped portion is often quite large, and (3) three of the kernels have noticeable non-overlapped store stalls (T_{CSP}^{Stall}).

Based on the measured runtime and performance counter

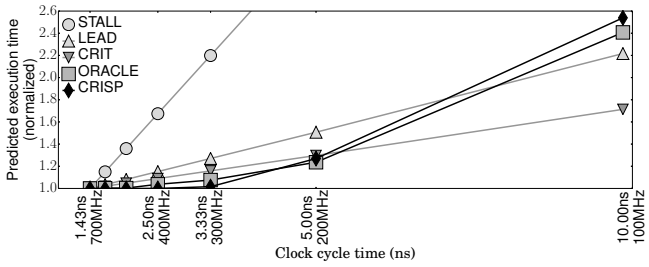


Figure 10: Execution time prediction for *lm*

data at 700 MHz frequency, the performance models predict the execution time at six other frequencies (100 MHz–600 MHz). Figure 8 shows the prediction error of each of the models for three target frequencies: 100 MHz, 400 MHz, and 600 MHz. We also highlight the mean absolute percentage error (MAPE) at these target frequencies in the rightmost graph of Figure 8. The summarized average absolute percentage errors of the models across all six target frequencies for each benchmark are reported in Figure 9.

STALL always overpredicts the execution time, conservatively treating all stalls as completely inelastic. The inaccuracy of STALL can be as high as 288%. Both LEAD and CRIT suffer from underprediction (65% or more for *bfs*) and overprediction (109% or more for *ge*). The accuracy of CRISP also varies, but the maximum prediction error (30%) is reduced by a factor of 3.6 compared to the maximum error (109%) with the best of the existing models.

We see that both LEAD and CRIT are more prone to underpredicting the memory bound kernels (those on the right side of the graph); this comes from their inability to detect when the kernels transition to computation-bound and become more sensitive to frequency. The major exception is *ge*, where they both fail to account for the store related stalls and heavily over-predict execution time.

We also note from these results that as we try to predict the performance with a high scaling factor (from 700 MHz to 100 MHz), the error of the prior techniques becomes superlinear – that is, they become proportionally more inaccurate the further you depart from the base frequency. The leading load paper acknowledges this. A big factor in this superlinear error is that the greater the change in frequency, the more likely it is that the load stalls are completely covered and the kernel transitions to a compute-bound scaling factor. This explains why the CRISP results do not appear to share the superlinear error, in general. To illustrate this, we show the predicted execution time for the *lm* kernel as we scale the frequency from 700 MHz to 100 MHz (Figure 10). CRIT measures T_{Memory} as 0.88 while LEAD estimates it as 0.80. We measure the load critical path length of *lm* to be 0.93 of which 34% are overlapped computations; thus we expect the load critical path to be completely covered at about 233 MHz. We see that above this frequency CRIT is fairly accurate, but diverges widely after this point. Beyond this point, CRISP treats the kernel as completely com-

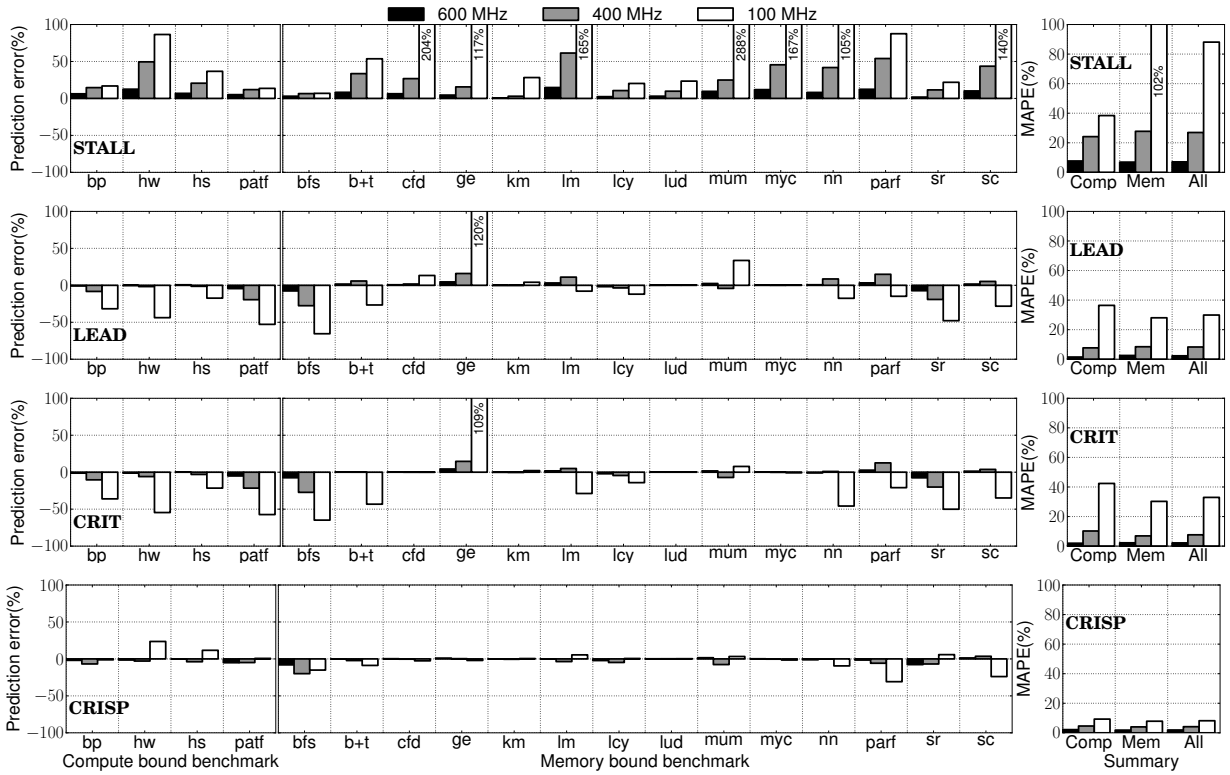


Figure 8: Execution time prediction error for different target frequencies with baseline frequency of 700 MHz

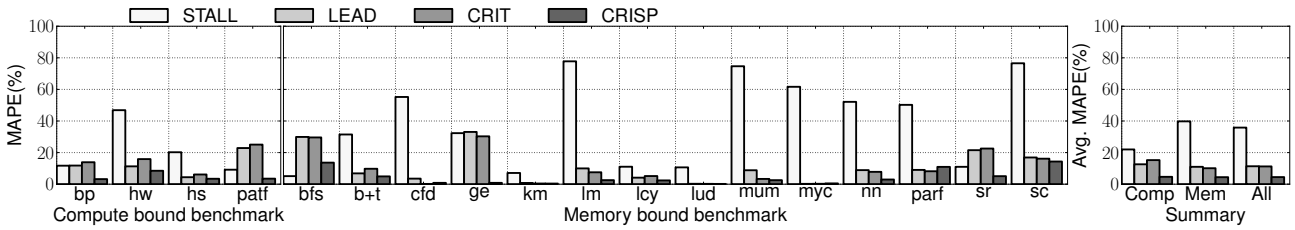


Figure 9: Mean absolute percentage error averaged across all target frequencies with baseline frequency of 700 MHz

pute bound, while at 100 MHz CRIT is still assuming 52% of the execution time is spent waiting for memory. In contrast, CRISP closely follows the actual runtimes (ORACLE) both at memory-bound frequencies and at compute-bound frequencies.

We observe similar phenomenon in the kernels *hw*, *hs*, and *bfs*. In each of these cases, CRISP successfully captures the piecewise linear behavior of performance under frequency scaling. As summarized in Figure 9, the average prediction error of CRISP across all the benchmarks is 4% vs. 11% with LEAD, 11% with CRIT, and 35% with STALL.

5.2 Energy Savings

An online performance model, and the underlying architecture to compute it, is a tool. An expected application of such a tool would be to evaluate multiple DVFS settings to optimize for particular performance and energy goals. In this section, we will evaluate the utility of CRISP in minimizing two widely used metrics: energy delay product (EDP) and energy delay squared product (ED^2P).

We compute the savings relative to the default mode which always runs at maximum SM clock frequency (700 MHz, 1V). We run two similar experiments with our simple DVFS controller (Section 4) targeting (a) minimum EDP, and (b) the minimum ED^2P operating point. During both experiments, the controller uses the program behavior during the current interval to predict the next interval. We do the same for each of the other studied predictors – all use the same power predictor.

Using CRISP for this purpose does raise one concern. All of the models are asymmetric (i.e., will predict differently for lower frequency A->B than for higher frequency B->A) due to different numbers of stall cycles, different loads on the critical path, etc. However, the asymmetry is perhaps more clear with CRISP. While predicting for lower frequencies we quite accurately pre-

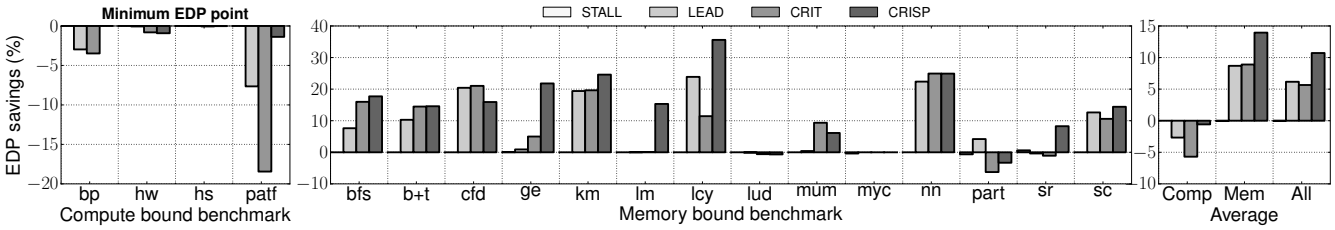


Figure 11: EDP reduction while optimized for EDP

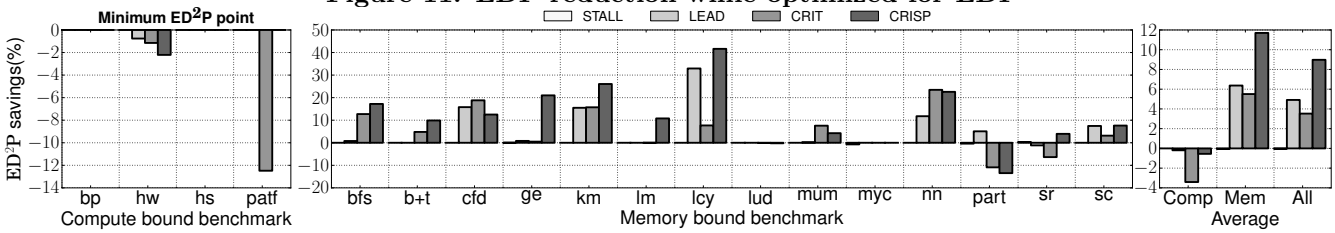


Figure 12: ED^{2}P reduction while optimized for ED^{2}P

dict when overlapped computation will be converted into pure computation, it is more difficult to predict which pure computation cycles will become overlapped computation at a higher frequency. We thus modify our predictor slightly when predicting for a higher frequency. We calculate the new execution time as $T_{Memory} + T_{CSP}^{Stall} + T_{CSP}^{Comp} \times \frac{f_c}{f}$. This ignores the possible expansion of store stalls, and does not account for conversion of pure computation to overlapped. While this model will be less accurate, in a running system with DVFS (as in the next section) if it mistakenly forces a change to a higher frequency, the next interval will correct it with the accurate model. While this could cause ping-ponging which we could solve with some hysteresis, we found this unnecessary.

Optimizing for EDP: Figure 11 shows EDP savings achieved by the evaluated performance models when the controller is seeking the minimum EDP point for each kernel. We included all the runtime overhead – software model computation and voltage switching overhead – in this result. For the 14 memory bound benchmarks, CRISP reduces EDP by 13.9% compared to 8.9% with CRIT, 8.7% with LEAD, and 0% with STALL. CRISP achieves high savings in both *lm* (15.3%) and *ge* (21.8%). The other three models fail to realize any energy savings opportunity in *lm* and achieve very little savings (5% with CRIT and 1% with LEAD) in *ge*. As we saw previously in Figure 10, CRISP accurately models *lm* at all frequency points. Thus, it typically chooses to run this kernel at 300 MHz, resulting in 1.25% performance degradation but 16.4% reduced energy. CRIT and LEAD both over-estimate the performance cost at this frequency and instead choose to run at 700 MHz.

For the compute bound benchmarks, CRISP marginally reduces EDP by 0.5%, while LEAD and CRIT increase EDP by 2.7% and 5.7% respectively. Due to the under prediction of execution time (e.g., *patf*), CRIT aggressively reduces the clock frequency and experiences 13.3% performance loss on average (33% in *patf*) for the compute bound benchmarks.

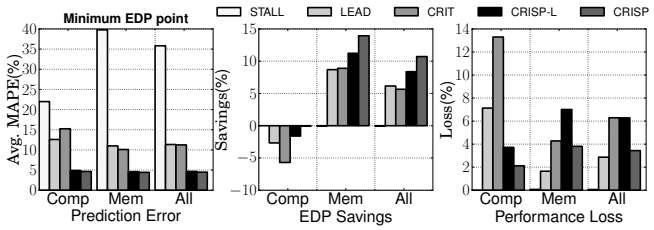


Figure 13: Accuracy, benefit, and drawback of CRISP-L

In summary, the average EDP savings for the Rodinia benchmark suite with CRISP is 10.7% vs. 5.7% with CRIT, 6.2% with LEAD, and 0% with STALL. The selected EDP operating points of CRISP translate into 12.9% energy savings and 15.5% reduction in the average power with an average performance loss of 3.4%. CRIT saves 10.6% energy and LEAD saves 8.1%. The performance loss is 6.3% with CRIT and 2.9% with LEAD.

Optimizing for ED^{2}P: When attempting to optimize for ED^{2}P, CRISP again outperforms the other models by reducing ED^{2}P 9.0% on average. The corresponding average savings realized by CRIT, LEAD, and STALL are only 3.5%, 4.9%, and 0%. Figure 12 shows that the savings with CRISP is much higher (11.7%) for memory bound benchmarks compared to CRIT (5.5%), LEAD (6.4%), and STALL (0%).

CRISP achieves its highest ED^{2}P savings of 42% for *lcy*, by running the primary kernel at 500 MHz most of the time. CRIT and LEAD underpredict the energy saving opportunity and keep the frequency at 700 MHz for 50% and 41% of the duration of this kernel, respectively. As we analyze the execution trace, we find that 67% of cycles inside the load critical path for that kernel are computation, and are thus not handled accurately by the other models.

5.3 Simplified CRISP Model

CRISP can be simplified to estimate the load critical path (LCP) length as the total number of load outstanding cycles, without regard for which stall cycles

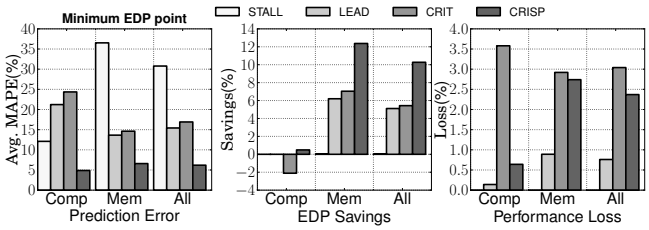


Figure 14: Effectiveness of CRISP on NVIDIA Tesla C2050

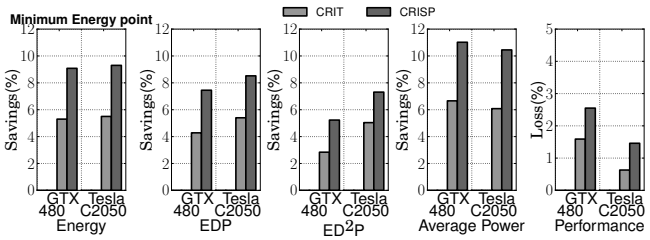


Figure 15: Effectiveness of CRISP while optimizing for energy

and in particular which computation is under the critical path. We simply treat all computation cycles that are overlapped by a load as if they were on the critical path. This approximated version of CRISP (CRISP-L) requires three counters, and thus removes 98% of the storage overhead of CRISP (all the timestamp registers).

CRISP-L demonstrates competitive prediction accuracy (4.66% on average in Figure 13) as CRISP (4.48% on average) does for the experiment in Section 5.1. However, when used in an EDP controller, CRISP-L suffers relative to CRISP, but still realizes far better (8.38%) EDP savings than CRIT(5.65%) and LEAD (6.16%). The inclusion of more compute cycles inside LCP tends to create underpredictions of execution time, resulting in more aggressive frequency scaling. CRISP-L experiences more (6.28%) performance loss than CRISP (3.44%).

5.4 Generality of CRISP

All of the results so far were for an NVIDIA GTX 480 GPU. To demonstrate that the CRISP model is more general, we replicated the same experiments for a modeled Tesla C 2050 GPU. The Tesla differs in available clock frequency settings, number of cores, DRAM queue size, and in the topology of the interconnect. Results for predictor accuracy and effectiveness were consistent with the prior results. In this section, we show in particular the result for the EDP-optimized controller. Figure 14 shows that the mean absolute percentage prediction error of CRISP (6.2%) is much lower than LEAD (16.9%) and CRIT (15.4%). Meanwhile, CRISP realizes 10.3% EDP savings, which is 5% more than the best of the prior models.

We also experimented with another DVFS controller goal – this one seeks to minimize energy while keeping the performance overhead under 1%. Figure 15 shows the energy, EDP, ED^2P, average power savings, and the performance loss for GTX480 and Tesla C2050. In Tesla C2050, CRISP achieves 9.3% energy savings, which is 4% more than CRIT.

6. CONCLUSION

We introduce the CRISP (CRItical Stalled Path) performance model to enable effective DVFS control in GPGPUs. Existing DVFS models targeting CPUs do not capture key GPGPU execution characteristics, causing them to miss energy savings opportunity or cause unintended performance loss. CRISP effectively handles the impact of highly overlapped memory and computation in a system with high thread level parallelism, and also captures the effect of frequent memory store based stalls.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive comments and suggestions that helped us to improve the paper. This work was supported by NSF grants CCF-1219059 and CCF-1302682.

8. REFERENCES

[1] P. Macken, M. Degrauwe, M. V. Paemel, and H. Oguey, "A voltage reduction technique for digital systems," in *IEEE Int. Solid-State Circuits Conf.*, 1990.

[2] D. Snowdon, S. Ruocco, and G. Heiser, "Power management and dynamic voltage scaling: Myths and facts," in *Power Aware Real-time Computing*, 2005.

[3] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *ISLPED*, 2007.

[4] H. David, C. Fallin, E. Gorbatov, U. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proceedings of the 8th International Conference on Autonomic Computing (ICAC)*, 2011.

[5] Q. Deng, D. Meisner, A. Bhattacharjee, T. Wenisch, and R. Bianchini, "Multiscale: Memory system dvfs with multiple memory controllers," in *International Symposium on Low power electronics and design (ISLPED)*, 2012.

[6] Q. Deng, D. Meisner, L. Ramos, T. Wenisch, and R. Bianchini, "Memscale: active low-power modes for main memory," in *ACM SIGPLAN*, 2011.

[7] X. Chen, Z. Xu, H. Kim, P. Gratz, J. Hu, M. Kishinevskyy, and U. Ograsy, "In-network monitoring and control policy for dvfs of cmp networks-on-chip and last level caches," in *NOCS*, 2012.

[8] X. Chen, Z. Xu, H. Kim, P. V. Gratz, J. Hu, M. Kishinevsky, U. Ogras, and R. Ayoub, "Dynamic voltage and frequency scaling for shared resources in multicore processor designs," in *DAC*, 2013.

[9] G. Liang and A. Jantsch, "Adaptive power management for the on-chip communication network," in *Euromicro Conference on Digital System Design*, 2006.

[10] A. Mishra, R. Das, S. Eachempati, R. Iyer, N. Vijaykrishnan, and C. Das, "A case for dynamic frequency tuning in on-chip networks," in *International Symposium on Microarchitecture (MICRO)*, 2009.

[11] D. Brooks and M. Martonosi, "Dynamic thermal management for highperformance microprocessors," in *High-Performance Computer Architecture (HPCA)*, 2001.

[12] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip

multiprocessors," in *Proc. 35th Ann. Int'l Symp. Computer Architecture (ISCA)*, 2008.

[13] D. Ernst, N. Kim, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, , and K. Flautner, "Razor: A low-power pipeline based on circuit-level timing speculation," in *International Symposium on Microarchitecture (MICRO)*, 2003.

[14] Y. Abe, H. Sasaki, S. Kato, K. Inoue, M. Edahiro, and M. Peres, "Power and performance characterization and modeling of gpu-accelerated systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[15] Y. Abe, H. Sasaki, S. Kato, K. Inoue, M. Edahiro, and M. Peres, "Power and performance analysis of gpu-accelerated systems," in *Proceedings of the USENIX conference on Power-Aware Computing and Systems (HotPower)*, 2012.

[16] X. Mei, L. S. Yung, K. Zhao, and X. Chu, "A measurement study of gpu dvfs on energy conservation," in *Proceedings of the Workshop on Power-Aware Computing and Systems (HotPower)*, 2013.

[17] "Nvidia gpu boost,"

[18] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *ISCA*, 2013.

[19] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times," in *DATE*, 2004.

[20] G. Contreras and M. Martonosi, "Power prediction for intel xscale processors using performance monitoring unit events," in *ISLPED*, 2005.

[21] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online power-performance adaptation of multithreaded programs using hardware event-based prediction," in *Int. Conf. Supercomputing*, 2006.

[22] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction models for multi-dimensional power-performance optimization on many cores," in *PACT*, 2008.

[23] J. Carter, H. Hanson, K. Rajamani, F. Rawson, T. Rosedahl, and M. Ware, "Dynamic voltage and frequency scaling (dvfs) control for simultaneous multi-threading (smt) processors," 2011.

[24] G. Dhiman and T. S. Rosing, "Dynamic voltage frequency scaling for multi-tasking systems using online learning," in *ISLPED*, 2007.

[25] M. Moeng and R. Melhem, "Applying statistical machine learning to multicore voltage and frequency scaling," in *Int. Conf. Computing Frontiers*, 2010.

[26] C. Hughes, J. Srinivasan, and S. Adve, "Saving energy with architectural and frequency adaptations for multimedia applications," in *MICRO*, 2001.

[27] C. Isci, A. Buyuktosunoglu, C. Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *MICRO*, 2006.

[28] F. Xie, M. Martonosi, and S. Malik, "Efficient behavior-driven runtime dynamic voltage scaling policies," in *CODES+ISSS*, 2005.

[29] D. Snowdon, S. Petters, and G. Heiser, "Accurate on-line prediction of processor and memory energy usage under voltage scaling," in *2007*, EMSOFT.

[30] C. Isci and A. B. andM. Martonosi, "Long-term workload phases: Duration predictions and applications to dvfs," in *IEEE Micro*, 2005.

[31] A. Weissel and F. Bellosa, "Process cruise control: Event-driven clock scaling for dynamic power management," in *IntâĂŹl Conf. Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2002.

[32] C. Poellabauer, L. Singleton, and K. Schwan, "Feedback-based dynamic voltage and frequency scaling for memory-bound real-time applications," in *IEEE Real-Time Embedded Technology and Applications Symp. (RTAS)*, 2005.

[33] S. Eyerman and L. Eeckhout, "A counter architecture for online dvfs profitability estimation," in *IEEE Trans. Comput.*, 2010.

[34] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt, "Predicting performance impact of dvfs for realistic memory systems," in *International Symposium on Microarchitecture (Micro)*, 2012.

[35] G. Keramidas, V. Spiliopoulos, and S. Kaxiras, "Interval-based models for run-time dvfs orchestration in superscalar processors," in *ACM Int. Conf. Computing Frontiers (CFâĂŹ10)*, 2010.

[36] B. Rountree, D. K. Lowenthal, M. Schulz, and B. R. de Supinski, "Practical performance prediction under dynamic voltage frequency scaling," in *Int. Green Computing Conf. and Workshops*, 2011.

[37] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *IEEE ISPASS*, 2009.

[38] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proceedings of the IEEE International Symposium onWorkload Characterization*, 2010.

[39] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee, "A performance model for gpus with caches," in *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[40] S. Hong and H. Kim, "An integrated gpu power and performance model," in *Proc. 37th Ann. Int'l Symp. Computer Architecture (ISCA)*, 2010.

[41] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *ISCA*, 2009.

[42] S. Song, C. Su, B. Rountree, and K. Cameron, "A simplified and accurate model of power-performance efficiency on emergent gpu architectures," in *IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013.

[43] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang, "Run-time technique for simultaneous aging and power optimization in gpgpus," in *Proceedings of the 51st Annual Design Automation Conference*, 2014.

[44] A. Sethia and S. Mahlke, "Equalizer: Dynamic tuning of gpu resources for efficient execution," in *MICRO*, 2014.

[45] J. Issa and S. Figueira, "A performance estimation model for gpu-based systems," in *2nd International Conference on Advances in Computational Tools for Engineering Applications (ACTEA)*, 2012.

[46] W. Kim, M. Gupta, G. Y. Wei, and D. Brooks, "System level analysis of fast, per-core dvfs using on-chip switching regulators," in *High-Performance Computer Architecture (HPCA)*, 2008.

[47] L. Clark, E. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K. Velarde, and M. Yarch, "An embedded 32-b microprocessor core for low-power and high-performance applications," in *IEEE Journal Of Solid-State Circuits*, 2001.

[48] T. Fischer, J. Desai, B. Doyle, S. Naffziger, and B. Patella, "A 90-nm variable frequency clock system for a power-managed itanium architecture processor," in *IEEE Journal Of Solid-State Circuits*, 2006.

[49] J. Lucas, S. Lal, M. Andersch, M. A. Mesa, and B. Juurlink, "How a single chip causes massive power bills gpusimpow: A gpgpu power simulator," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

[50] J. A. Butts and G. S. Sohi, "A static power model for architects," in *International Symposium on Microarchitecture (MICRO)*, 2000.