COS: A Parallel Performance Model for Dynamic Variations in Processor Speed, Memory Speed, and Thread Concurrency

Bo Li
Virginia Tech
Blacksburg, Virginia
bxl4074@vt.edu

Edgar A. León
Lawrence Livermore National Laboratory
Livermore, California
leon@llnl.gov

Kirk W. Cameron
Virginia Tech
Blacksburg, Virginia
cameron@cs.vt.edu

ABSTRACT

Highly-parallel, high-performance scientific applications must maximize performance inside of a power envelope while maintaining scalability. Emergent parallel and distributed systems offer a growing number of operating modes that provide unprecedented control of processor speed, memory latency, and memory bandwidth. Optimizing these systems for performance and power requires an understanding of the combined effects of these modes and thread concurrency on execution time. In this paper, we describe how an analytical performance model that separates pure computation time (C) and pure stall time (S) from computation-memory overlap time (O) can accurately capture these combined effects. We apply the COS model to predict the performance of thread and power mode combinations to within 7% and 17% for parallel applications (e.g. LULESH) on Intel x86 and IBM BG/Q architectures, respectively. The key insight of the COS model is that the combined effects of processor and memory throttling and concurrency on overlap trend differently than the combined effects on pure computation and pure stall time. The COS model is novel in that it enables independent approximation of overlap which leads to capabilities and accuracies that are as good or better than the best available approaches.

ACM Reference format:
Bo Li, Edgar A. León, and Kirk W. Cameron. 2017. COS: A Parallel Performance Model for Dynamic Variations
in Processor Speed, Memory Speed, and Thread Concurrency. In *Proceedings of HPDC '17, Washington , DC, USA, June 26-30, 2017,* 12 pages.
DOI: http://dx.doi.org/10.1145/3078597.3078601

1 INTRODUCTION

Future high-performance, scientific applications will be highly parallel and designed to run in environments of enormous scale but limited power. Efficiency will be key to achieving the promise of exascale. Emergent systems will have large numbers of configurable operating modes that provide unprecedented control of processor speed and memory frequency and bandwidth. Unfortunately, very little is known about the combined effects of these operating modes and thread concurrency on execution time and efficiency.

The performance effects of various operating modes have been studied mostly in isolation. Dynamic voltage and frequency scaling (*DVFS*), the automated adjustment of processor power and speed settings, has been explored extensively [6, 17, 19, 34]. More recently, analogous research on the effects of dynamic memory voltage and frequency throttling (*DMT*), the automated adjustment of DRAM power and speed settings, has surfaced [10, 13, 29]. Other memory power modes such as dynamic bandwidth throttling [1] (*DBT*), where one or more idle clock cycles are inserted between memory accesses to lower peak bandwidth, are emergent. Dynamic concurrency throttling (*DCT*), the automated adjustment of thread concurrency, has also received widespread attention for some time [8].

While some have attempted to study the combined effects of two types of operating modes (e.g., CPU and memory scaling [10, 13], CPU scaling and concurrency throttling [8]), to the best of our knowledge, no one has accurately modeled the combined effects of CPU throttling, memory throttling, and concurrency throttling.

Modeling the combined effects of these three operating modes is incredibly challenging. Capturing the interactive performance effects of a highly configurable problem space could be intractable in highly-parallel, high-performance environments. Furthermore, the interactive effects of these modes are likely to be non-linear, complicating efforts to identify simple but useful analytical models of performance.

In this paper, we present the COS Model of parallel performance for dynamic variations in processor speed, memory speed, and thread concurrency. To the best of our knowledge, this is the first model to accurately capture the simultaneous, combined effects of these three operating modes.

The COS model is based on a simple observation. Past models of operating mode performance tend to combine the overlap of compute and memory performance into either compute time or memory stall time. However, we have observed that the behavior of overlap when these operating modes change is so complex that it must be modeled independently of these other times. This observation leads to the formulation of a Compute-Overlap-Stall (COS) Model where each term can be modeled independently to the others.

In addition to presenting the COS model, we demonstrate how to capture these important (and independent) parameters on both Intel servers and the IBM BG/Q system. We also show how the

HPDC '17, June 26-30, 2017, *Washington , DC, USA*

DOI: http://dx.doi.org/10.1145/3078597.3078601

[1] C.-H. R. Wu, "U.S. patent 7352641: Dynamic memory throttling for power and thermal limitations." Sun Microsystems, Inc., issued 2008.

COS model can be used to classify the best available models. We validate our modeling efforts on 19 HPC kernels and perform extensive sensitivity analyses to identify weaknesses. Our COS Model has more functionality than previously available and the accuracy is as good as or better than best available operating mode models with prediction errors as low as 7% on Intel systems and 17% on the IBM BG/Q system.

2 COMPUTE–OVERLAP–STALL MODEL

2.1 COS Model Parameters

The Compute–Overlap–Stall (COS) model estimates parallel execution time as the sum of pure compute time (T_c), overlap time (T_o), and pure stall time (T_s). More generally,

$$T = T_c + T_o + T_s, \tag{1}$$

where T is total time for a running application.

Figure 1 shows an example execution time profile for a simple, single-threaded application. A single core executes some computation that triggers two separate, non-blocking memory operations. As the code executes, portions of time are spent exclusively on on-chip, in-cache computations; exclusively on off-chip memory operations; and on some form of overlap between computation and memory accesses.

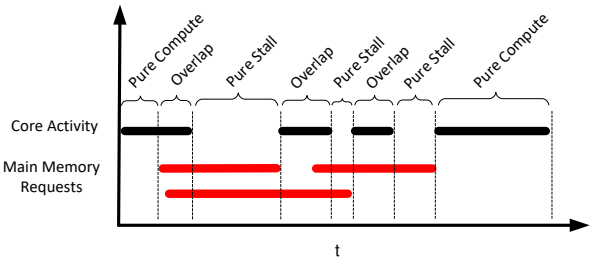


Figure 1: An example of a COS Trace for a simple, single-threaded application with hardware support for multiple, simultaneous memory accesses (e.g., multiple loads under misses).

Figure 1 provides context for defining terms of the COS model more precisely. T_c is the sum of the execution times of an application spent exclusively on computation[2] or the ***pure compute time***. In this example, T_c is the sum of the pure compute times identified at the start and end of the application's execution. T_o is the sum of the execution times of an application spent overlapping computation and memory operations or the ***overlap time***. In this example, T_o is the sum of the overlap times, there are three of these stage occurrences over the application's execution. T_s is the sum of the execution times of an application spent exclusively on memory stalls or the ***pure stall time***. In this example, T_s is the sum of the pure stall times, there are three of these stage occurrences over the application's execution.

2.2 The COS Trace

The ordered summation of the terms of the COS model constitutes a simplified trace of the application. We call this a ***COS Trace***. More precisely, the 8 stage occurrences for the example in Figure 1 are expressed in the following COS Trace:

$$\begin{aligned} T =& T_c(1) + T_o(1) + T_s(1) + T_o(2) + T_s(2) + \\ & T_o(3) + T_s(3) + T_c(2) \end{aligned} \tag{2}$$

Analogously, we propose a general COS Trace as follows:

$$T = \sum_{i=1}^{cP} T_c(i) + \sum_{j=1}^{oP} T_o(j) + \sum_{k=1}^{sP} T_s(k) \tag{3}$$

where cP, oP, sP are the number of stages corresponding to the three types of time in the COS trace: the pure compute time (T_c), the overlap time (T_o), and the pure stall time (T_s). For Figure 1, $cP = 2$, $oP = 3$, and $sP = 3$. Predicting parallel execution time using the COS model involves estimating the effect of a system or application change on the COS Trace[3].

2.3 COS Model Notations

In succeeding discussions we will use (f_c) and ($f_c^{'}$) to refer to a starting CPU frequency and the changed CPU frequency respectively. Moreover, Δf_c denotes the change from f_c to $f_c^{'}$. We can define Δf_m and Δt analogously. We use the shorthand $(f_c, f_m, t) \rightarrow (f_c^{'}, f_m^{'}, t^{'})$ to denote changes to DVFS, DMT/DBT, and thread count respectively. For example, $(f_c, f_m, t) \rightarrow (f_c^{'}, f_m, t)$ refers to an isolated change to CPU frequency while $(f_c, f_m, t) \rightarrow (f_c, f_m^{'}, t^{'})$ refers to simultaneous memory throttling and changes to thread counts.

2.4 The Importance of Isolating Overlap

Many existing models of parallel performance ignore overlap [3, 16, 22, 31, 33, 36]. When overlap *is* considered, the effects are either captured in the compute time (T_c) or memory stall time (T_s) parameters. If overlap is included in T_c, then the model assumes Δf_c effects apply equally to the overlap portion. If overlap is included in T_s, then the model assumes Δf_m effects apply equally to the overlap portion.

Figure 2 shows the stall time (y-axis) for a code region (R1) of the LULESH OpenMP application kernel [1]. The CPU voltage/frequency increases from left to right (x-axis). The figure shows the measured stall time and the predicted stall time for two best-available performance prediction approaches (stall- and leading-load-based [16, 22, 33]). Notice that both approaches consistently under-predict stall time. Furthermore, in another code region (R2) of the Lulesh OpenMP application (not shown), the same prediction techniques over-predict stall time.

[2]We include the performance impact of on-chip caches in pure compute time. This simplification significantly reduces the complexity of the COS model while enabling isolation of the performance effects of power-performance operating modes.

[3]The power-performance operating modes studied include CPU Dynamic Voltage and Frequency Scaling (DVFS) and DRAM Dynamic Memory Frequency Throttling (DMT) on Intel architectures; Dynamic Memory Bandwidth Throttling (DBT) on BG/Q architectures; and Dynamic Concurrency Throttling (DCT) on both architectures.

When stall time dominates, these mis-predictions lead to significant inaccuracies in execution time prediction. The effects are exacerbated by the complex computation and memory overlap scenarios that affect stall and compute time and are more common in mixed operating modes (DVFS, DMT, and DCT).

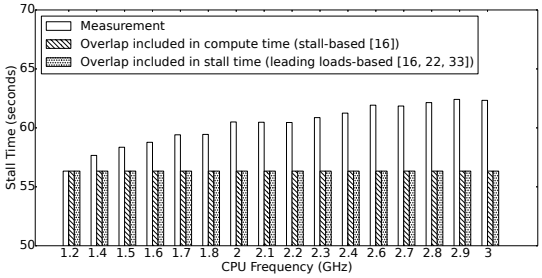


Figure 2: Stall time (y-axis) for varying CPU voltage/frequency settings (x-axis) for the LULESH benchmark on an x86 system. When stall time dominates, these mispredictions lead to significant inaccuracies in execution time prediction. The effects are exacerbated by the complex computation and memory overlap scenarios that affect stall and compute time and are more common in mixed operating modes (DVFS, DMT, and DCT).

2.5 The Challenge of Isolating Overlap

Figure 3 shows a simplified example for three CPU frequencies ($f_{c1} < f_{c2} < f_{c3}$) increasing from left to right. In each subfigure, core activity and memory activity are shown separately as a thread progresses in time (x-axis) from left to right. The COS trace is provided for each subfigure.

In the first subfigure, at the lowest CPU frequency f_{c1}, there are 4 distinct compute and memory overlap phases in the COS trace. This indicates regular memory accesses where the CPU is busy with work during the memory stall time. More precisely, the 9 stage occurrences for f_{c1} in Figure 3 are expressed in the following COS Trace:

$$T = T_c(1) + T_o(1) + T_c(2) + T_o(2) + T_c(3) + T_o(3) + T_c(4) + T_o(4) + T_c(5) \quad (4)$$

The change from $(f_c, f_m, t) \rightarrow (f_c^{'}, f_m, t)$ alters the COS Trace (f_{c2} in Figure 3) as follows:

$$T = T_c(1) + T_o(1) + T_c(2) \quad (5)$$

This reflects a dependency between the resulting COS trace and CPU frequency. In this case, there is a change in the arrival rate of the memory requests due to the CPU frequency changes. In the new configuration, there are no pure compute gaps between memory references leading to a change in the number and length of overlap stages.

Increasing the frequency a second time in this example (f_{c3} in Figure 3) alters the COS trace again, resulting in:

$$T = T_c(1) + T_o(1) + T_s(1) + T_o(2) + T_c(2) \quad (6)$$

This demonstrates the creation of a pure stall stage that did not exist in the previous two COS traces (f_{c1} and f_{c2}) in Figure

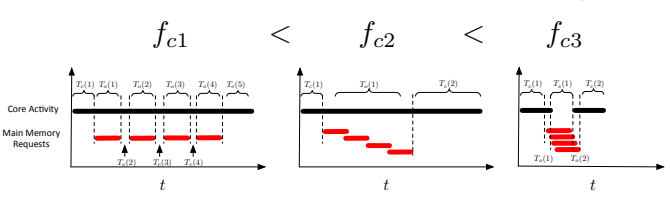


Figure 3: Overlap time and pure stall time are related to computation intensity.

3. We observe similar behaviors for memory throttling changes ($(f_c, f_m, t) \rightarrow (f_c, f_m^{'}, t)$) and for thread changes ($(f_c, f_m, t) \rightarrow (f_c, f_m, t^{'})$).

2.6 The Role of Computational Intensity

Estimating the COS terms for simultaneous changes in operating modes such as $(f_c, f_m, t) \rightarrow (f_c^{'}, f_m^{'}, t^{'})$ is even more challenging than the single CPU speed change described in Section 2.5. In theory, each term of the COS trace is affected by all operating mode changes (Δf_c, Δf_m, and Δt). In practice, it depends on the system and application design.

The initial focus of our work is on shared memory systems running multithreaded OpenMP applications where parallel threads are mostly homogeneous and synchronized and the programs use a bulk-synchronous programming model. This focus leads to some simplifying assumptions while still covering a large set of parallel applications of interest to a broad community of scientists [4, 5, 15, 21, 23, 26, 32].

Table 1 shows the application of these assumptions to reduce the set of interactions we need to consider for accurate predictions between system reconfigurations and model parameters. For example, to model $T_c^{'}$ for 7 rows of possible configurations, we need only consider changes to CPU frequency (Δf_c) and thread count (Δt). For $T_o^{'}$ and $T_s^{'}$, these assumptions simplify all predictions except when all operating modes change simultaneously $(f_c, f_m, t) \rightarrow (f_c^{'}, f_m^{'}, t^{'})$ for any Δf_c, Δf_m, and Δt. This explains why our prediction methods on real systems focus on separating overlap time $T_o^{'}$ from stall time $T_s^{'}$ (see Sections 2.4 and 2.5).

In addition to the system configuration changes (Δf_c, Δf_m, and Δt), Table 1 lists *CI* as a consideration for both overlap $T_o^{'}$ and pure stall $T_s^{'}$ times. *CI* here stands for *Computational Intensity*, or the percentage of memory stall time that is overlapped with useful work on the CPU. *CI* determines how much stall time is affected by CPU speed (Δf_c) and how much is affected by memory throttling (Δf_m).

We conducted statistical analyses to identify a correlation between stall time and measurable hardware counters available on most x86 architectures [2]. Through exhaustive experimentation for all available configurations $(f_c, f_m, t) \rightarrow (f_c^{'}, f_m^{'}, t^{'})$, we found that two widely available counters—last-level cache misses (LLCM) and time-per-instruction (TPI)— effectively captured the stall time effects of Δf_c, Δf_m, and Δt. This finding is key to the COS model's effectiveness since it enables us to use linear approximation methods to separate pure stall time from overlap stall time. For completeness, we studied the effects of *CI* on compute overlap but

Table 1: Effects on COS model parameters of any starting configuration (f_c, f_m, t) to any other operating mode configuration (each row) for changes in processor speed, memory throttling, and number of threads (Δf_c, Δf_m, and Δt). For some configurations, we have additionally identified CI (Computational Intensity) as having significant influence over COS model parameters.

Config	$T_c^{'}$	$T_o^{'}$	$T_s^{'}$
$f_c^{'}, f_m, t$	Δf_c	Δf_c CI	Δf_c CI
$f_c, f_m^{'}, t$		Δf_m CI	Δf_m CI
$f_c, f_m, t^{'}$	Δt	CI Δt	CI Δt
$f_c^{'}, f_m^{'}, t$	Δf_c	Δf_c Δf_m CI	Δf_c Δf_m CI
$f_c, f_m^{'}, t^{'}$	Δt	Δf_m CI Δt	Δf_m CI Δt
$f_c^{'}, f_m, t^{'}$	Δf_c Δt	Δf_c CI Δt	Δf_c CI Δt
$f_c^{'}, f_m^{'}, t^{'}$	Δf_c Δt	Δf_c Δf_m CI Δt	Δf_c Δf_m CI Δt

we found that compute time was dominated by effects from CPU speed and thread count and not affected significantly by CI.

2.7 Practical Estimation of COS Parameters

We can use the COS trace of Equation 3 to predict the parallel execution time (T') of another system configuration for any combination of Δf_c, Δf_m and Δt. Since the variables may not be directly measurable, the challenge is to collect accurate approximations without requiring system design changes or reverting to simulation. In this section we describe one method for predicting T' using direct measurements readily available on most x86 systems.

Several of the parameters of Equation 3 are directly measurable. Both total time T and the *pure stall time* T_s are directly measurable using the CPU hardware counters available on most modern platforms [2].

We've also observed in our experimental work that overlap consists of a portion affected by CPU speed changes (related to compute time and denoted as T_{oc}) and another portion affected by memory throttling (related to stall time and denoted as T_{os}). The portions vary according to the computational intensity CI of the application (see Section 2.6).

Under these measurements and observations, the operation mode change $(f_c, f_m, t) \rightarrow (f_c^{'}, f_m^{'}, t^{'})$ resulting in predicted time $T^{'}$ becomes:

$$T^{'} = [T_c^{'} + T_{oc}^{'}] + [T_{os}^{'} + T_s^{'}] \quad (7)$$

where $[T_c^{'} + T_{oc}^{'}]$ can be approximated as $[T - T_s] \times f_c/f_c^{'}$. Multiplying by the ratio of the CPU speed f_c and the new CPU speed $f_c^{'}$ follows the dependencies $(\Delta f_c, \Delta t)$ listed in Table 1 for the $(f_c, f_m, t) \rightarrow (f_c^{'}, f_m^{'}, t^{'})$ configuration for $T_c^{'}$. We will discuss how thread changes affect predictions in Section 2.8.

Approximating $[T_{os}^{'} + T_s^{'}]$ is more difficult. Table 1 shows that time for the $(f_c, f_m, t) \rightarrow (f_c^{'}, f_m^{'}, t^{'})$ configuration for $T_s^{'}$ is affected by a combination of Δf_c, Δf_m, Δt, and IC. Ignoring the impact of multi-threading again for now (see Section 2.8), we propose a linear combination of direct measurements for $LLCM$ and TPI with direct observations of changes to CPU Speed and memory throttling ($f_c^{'}$ and $f_m^{'}$). Recall the $LLCM$ and TPI terms capture the Computational Intensity CI effects on the COS trace. This gives the following approximation for the remaining portion of Equation 7:

$$[T_{os}^{'} + T_s^{'}] = \alpha_1 \times LLCM + \alpha_2 \times TPI + \alpha_3 \times f_c^{'} + \alpha_4 \times f_m^{'} \quad (8)$$

Combining our approximations for both sets of terms in Equation 7, our approximation of $T^{'}$ for a operating mode configuration change $(f_c, f_m, t) \rightarrow (f_c^{'}, f_m^{'}, t^{'})$ is:

$$T^{'} = [T - T_s] \times \frac{f_c}{f_c^{'}} + \alpha_1 \times LLCM + \alpha_2 \times TPI + \alpha_3 \times f_c^{'} + \alpha_4 \times f_m^{'} \quad (9)$$

In the next section, we describe how we use training sets and linear regression to identify the alpha parameters in this equation to develop a general model for each application in our set of 19.

2.8 Offline Training and Online Prediction

We use a training set measured offline to predict online a larger set of Δf_c, Δf_m, and Δt configurations. Figure 4 illustrates this two step process.

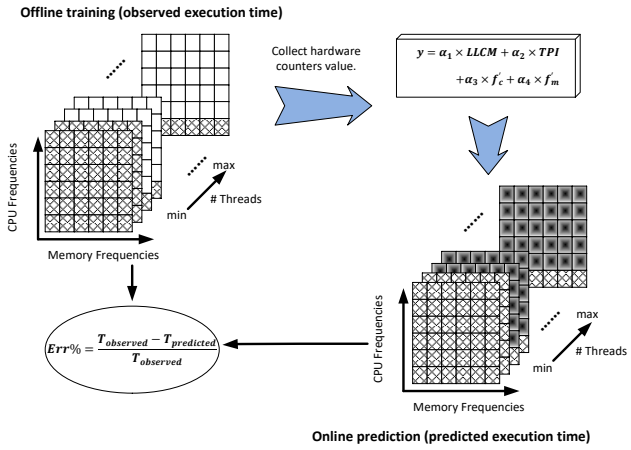


Figure 4: Offline training and online prediction.

The astute reader will notice that Equation 9 contains no term for the number of threads despite our claim to predict for dynamic concurrency changes. The impact of threads is captured in a set of linear approximations for Equation 9 applied to our training sets. What follows is an explanation of the algorithm we use to predict the simultaneous effects of Δf_c, Δf_m, and Δt configurations.

Figure 4 and Algorithm 1 describe our sampling techniques in detail. Basically we gather a set of data for a given application and take samples at various configurations for Δf_c, Δf_m, and Δt. We use this data to conduct linear regression on Equation 9 to determine the values of the four α parameters. For each measurement, we simultaneously gather execution time (T), stall time (T_s), $LLCM$ values, and TPI values.

We designed Algorithm 1 to formally describe the process illustrated by Figure 4. We define f_c^{min}, f_m^{min}, and t^{min} as the minimum speed setting for CPU, minimum throttling setting for memory, and the smallest number of threads respectively for a training set.

Algorithm 1 Train the COS Model for any application

1: **for all** $f_m^{'} \neq f_m^{min}$ **do**
2: Measure $T, T_s, LLCM, TPI$ for $(f_c^{min}, f_m^{min}, t^{min}) \rightarrow (f_c^{'}, f_m^{'}, t^{'}) \; \forall \; f_c^{'} \neq f_c^{min}$ and $t^{'} = 4, 6$
3: Measure $T, T_s, LLCM, TPI$ for $(f_c^{min}, f_m^{min}, t^{min}) \rightarrow (f_c^{min}, f_m^{'}, t^{'}) \; \forall \; t^{'} \neq t^{min}$
4: **end for**
5: Use measured data and linear regression to find α coefficients for Equation 9
6: Use Equation 9 to predict any $(f_c, f_m, t) \rightarrow (f_c^{'}, f_m^{'}, t^{'}) \; \forall \; f_c, f_m, t$ and $\forall \; f_c^{'}, f_m^{'}, t^{'}$ for this application

In Algorithm 1, thread behavior is captured by the training set. Basically, by reapplying Equation 9 to different thread configurations (steps 2 and 3 in Algorithm 1) we are able to capture the effects of threads on the COS model parameters using a combination of direct measurements and linear regression. These effects are incorporated in both $[T - T_s]$ and the *LLCM* and *TPI* terms of Equation 9. Thread effects are implicitly captured in the algorithmic application of Equation 9 and thus not explicitly in the formulation.

For a memory modes, b CPU modes, and c thread settings, we require $a \times b \times 2$ measurements for step 2 in Algorithm 1 and $a \times c$ measurements for step 3 in Algorithm 1. These measurements are captured visually by the hashed squares on the left side of Figure 4. This is compared to our ability to predict $a \times b \times c$ combinations using a single training set (see the darker squares on the right side of Figure 4). We have also determined that of the 19 applications studied, only 4-6 models are needed to accurately predict $T^{'}$ for all 19 applications. In future work, we are attempting to reduce the training sets further for online usage. In the remainder of this paper, we compare our predictions with direct measurements and use the resulting COS model for analysis for 19 applications on Intel x86 and IBM BG/Q systems.

3 EMPIRICAL MODEL VALIDATION

In this section, we validate the COS model on a multi-core machine using several application benchmarks with different computational characteristics. We measure the accuracy of the model by comparing the model's prediction versus observed values measured on real hardware.

3.1 Machine Characteristics

We validate the COS model on a cluster comprised of Dell PowerEdge R430 servers. Each node has two Intel Xeon E5-2623 v3 (Haswell) processors and 32 GB of DDR4 memory. Each processor has four cores and each core supports two hardware threads. The Haswell processor supports 16 CPU frequencies ranging from 1.2 to 3.0 GHz. The memory system supports three bus frequencies: 1.333, 1.600, and 1.866 GHz.

3.2 Application Benchmarks

We employ a set of benchmarks and kernels that represent diverse computational characteristics appearing in high-performance, parallel, scientific applications. The application benchmarks include the following codes:

- LULESH (CORAL benchmark suite[4], 5 code regions)
- AMGmk (CORAL benchmark suite, 3 kernels)
- Rodinia benchmark suite (6 applications)
- pF3D from LLNL (5 kernels)

LULESH is an explicit hydrodynamics proxy application that contains data access patterns and computational characteristics of larger hydrodynamics codes at LLNL [1]. We use five code regions within an OpenMP version of LULESH that represent different phases of the application and consume over 90% of the runtime [27]. These five code regions (R1 to R5) were selected in collaboration with domain scientists to isolate the code regions with a diverse set of computational intensity characteristics.

AMGmk includes three compute intensive kernels from AMG, an algebraic multigrid benchmark application derived directly from the BoomerAMG solver in the Hypre linear solvers library [18]. This code is used broadly in a number of applications [26] of interest to the multi-physics community. The default Laplace-type problem is built from an unstructured grid with various jumps and anisotropy in one part. We label these kernels K1 to K3.

Rodinia is a benchmark suite for heterogeneous computing [7]. We use six OpenMP codes from the domains of data mining, graph algorithms, physics simulation, molecular dynamics, and linear algebra: Kmeans, k-Nearest Neighbors (kNN), Breadth-First Search (BFS), HotSpot, LavaMD, and LU Decomposition (LUD). Components of this application suite such as HotSpot are of high interest to domain scientists for use in structured grid applications [4, 5, 21]. There is also high demand for optimized linear algebra solvers [32, 37] such as kNN, Kmeans, and LUD that are used regularly in many high-performance applications and systems.

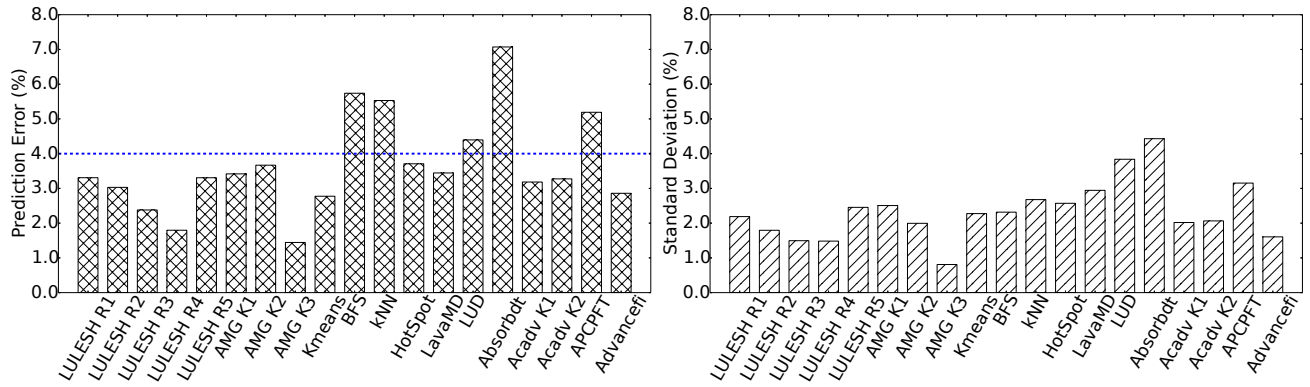
pF3D is a massively parallel application that simulates laser-plasma interactions at the National Ignition Facility at LLNL [24]. This simulator aids scientists in tuning plasma and laser beam experiments crucial to experimental physics [23]. The pF3D kernels derive from the functions that consume the most time during a typical pF3D run and are written in OpenMP. We use the following kernels: Absorbdt, Acadv K1, Acadv K2, APCPFT, and Advancefi.

In total we used 5 + 3 + 6 + 5 = 19 code regions and application kernels to evaluate the proposed model. For simplicity, we refer to these as *codes* or *applications* although they are *application benchmarks*.

3.3 Performance Prediction Accuracy

We compare the execution time predicted using modeling with the execution time observed by running the codes. First, for each code, we train its model offline (see Section 2.8) using a sample of Δf_c, Δf_m, and Δt as shown in Table 2. With these configurations we derive the model coefficients. At this point, we can use the model to predict the execution time of any given configuration. Second, we run the code under the configurations not in the training set

[4] See https://asc.llnl.gov/CORAL-benchmarks



(a) Average prediction error. **(b) Standard deviation of the prediction error.**

Figure 5: Model prediction accuracy for a wide-range of codes.

for a total of 225 configurations. Each of these is run 20 times to smooth out system noise effects and the average execution time is calculated. Third, we do this for all 19 codes.

Table 2: We use 4- and 6-thread configurations to predict 8, 10, 12, 14, and 16 thread configurations where Δf_c=16 modes, Δf_m=3 modes, Δt=7 thread configurations.

Total	Δf_c (GHz)	Δf_m	Δt (num. threads)
Training configurations			
$16 \times 3 \times 2$	All	All	4, 6
$1 \times 3 \times 5$	1.2	All	8, 10, 12, 14, 16
Configuration space			
$16 \times 3 \times 7$	All	All	4, 6, 8, 10, 12, 14, 16

Figure 5 shows the model prediction accuracy for all of the codes. Figure 5a shows the average prediction error of each code across the entire configuration space not in the training set. The prediction error is calculated as follows (also shown in Figure 4):

$$Err\% = \frac{|T_{measure} - T_{predict}|}{T_{measure}}$$

Figure 5 shows that the average prediction error per code is significantly low: varying from 1.4% to no more than 7%. Most of the codes though have an error lower than 4%. This demonstrates the proposed model is highly accurate for a broad range of applications. We also measure the standard deviation of the prediction error as shown in Figure 5b. The standard deviations for all the codes is within 4.5%. Our proposed model is significantly accurate for the three dimensional configuration space for all 19 applications.

To verify that the tested codes include a wide range of different computational characteristics, we measured the *sensitivity* of a subset of our codes to certain parameters such as processor speed. To capture an application's sensitivity, we focus on *pressure to the memory system* measured as last level cache misses per second. We expect, for example, low memory pressure for compute-intense applications (see Section 2.6) and high pressure for memory bandwidth-intense applications.

Figure 6 shows last-level cache misses (LLCM) per second as a function of different processor and memory speeds and thread concurrency. We employ two processor speeds (1.2 and 3.0 GHz), two memory speeds (1.333 and 1.866 GHz), and two thread counts (4 and 16). Each configuration is represented as a tuple of the following form:

(C: *cpu_frequency*, M: *memory_frequency*, T: *num_threads*)

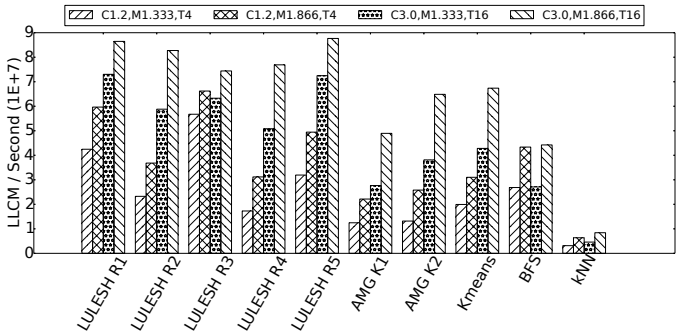
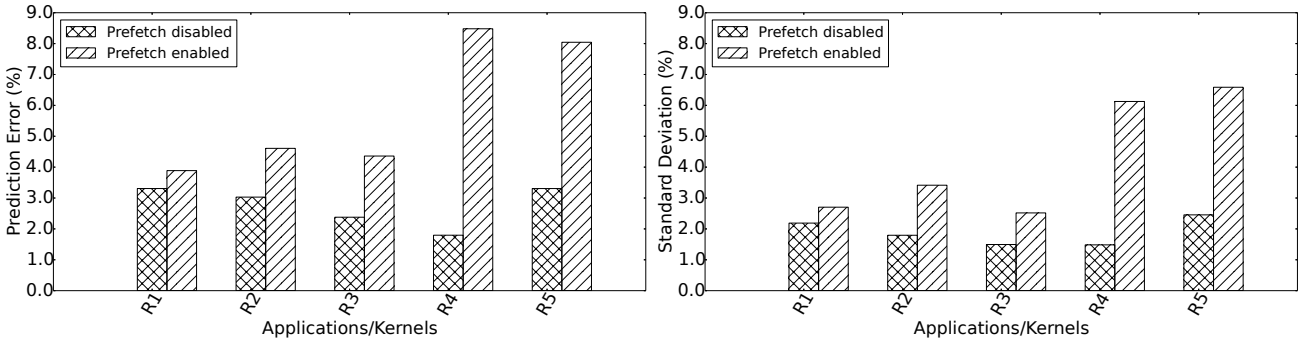


Figure 6: Impact of configuration on memory pressure.

First, we focus on one configuration: (C1.2, M1.333, T4). Codes including kNN, AMG K1 and K2, and LULESH R4 show low memory bandwidth presssure. This matches our expectation since AMG K1 and K2 are compute-intense kernels as is LULESH R4 [28]. While LULESH R1 and R3 are among the ones with the highest usage, Kmeans, BFS, and LULESH R2 exercise higher memory bandwidth utilization. These last two have been shown to be memory bandwidth intensive [27].

Second, we observe that some codes are significantly affected by different parameters such as memory speed and processor speed. LULESH R1 for example shows increased memory pressure with increases in processor speed and also with increases in memory speed. R1 has a high number of instructions per cycle (IPC) that benefit from the increased processor speed shifting the pressure to the memory system. Except for R3, other regions of LULESH show



(a) Average absolute prediction error. **(b) Standard deviation of the absolute prediction error.**

Figure 7: Impact of prefetching on prediction accuracy running LULESH.

a similar pattern but at different scales. Kmeans, AMG K1, and AMG K2 show significant sensitivity to processor speed because of their linear algebra computations.

Third, there are codes that show low sensitivity to different configurations. kNN is a clear example of this. BFS is not affected by increases in processor performance since there is little computation during the graph traversals but does show sensitivity to memory performance as a result of the operations fetching undiscovered graph nodes from memory. LULESH R3 is an interesting case since there are small changes with either processor or memory speed. This is the result of the code being almost exclusively memory bandwidth bound.

Thus, Figure 6 shows that the codes studied in this work capture a diverse set of computational characteristics. Furthermore, the resources in the critical path for some of these codes can change significantly with different configurations. For example, AMG K2 run with a 3.0 GHz processor becomes significantly dependent on the memory system when increasing memory frequency from 1.333 to 1.866 GHz. The low prediction error of the proposed COS model shows that we can capture the effect of these complex interactions accurately.

3.4 COS on Intel Sensitivity Analysis

3.4.1 Memory Prefetching. During development, we noticed the COS model accuracy was sensitive to prefetch settings. The effects of hardware and software prefetching on performance are captured in changes to the COS Trace described by Equation 3. For example, a successful prefetch could increase overlap by preemptively importing data from main memory to cache. An incorrect prefetch however causes cache pollution and could lead to more overlap stages and stall stages.

To better understand these effects, we ran LULESH with hardware prefetching enabled and hardware prefetching disabled and analyzed the results using COS. Figure 7 shows these results using 4 Intel prefetchers: DCU streamer prefetcher (load data to L1 data cache triggered by an ascending access of recently loaded data), DCU IP prefetcher (load data to L1 data cache based on load instruction and its detected regular stride), adjacent line prefetcher (fetch cache line to L2 and last level cache with the pair line), and hardware (streamer) prefetcher (fetch cache lines to L2 and last level cache based on detection of forward or backward stream of requests from L1).

After enabling all the hardware prefetchers, the accuracy of our predictor worsens as expected. For LULESH R1, the change in average prediction error (Figure 7a) and standard deviation (Figure 7b) are both very minimal. In contrast, LULESH R4 has the largest differential (4x) in accuracy when prefetching is enabled. This is likely due to a large increase in overlap when prefetching is enabled since the R4 region is dominated by compute when overlap is disabled.

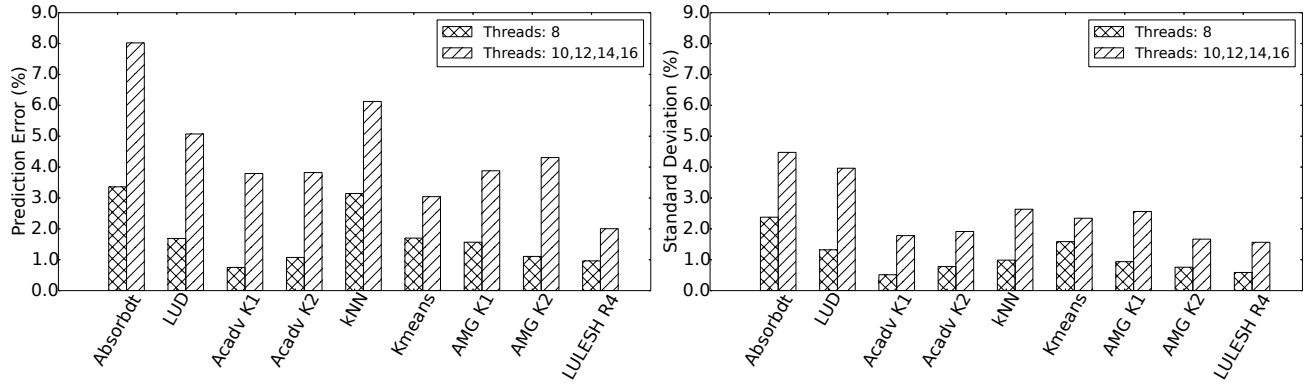
When prefetching is disabled, we get excellent accuracy using an extrapolation technique to predict configurations not observed directly in the training set. To improve the accuracy of COS for prefetching, we switched to an interpolation technique using 4- and 16-thread configurations to predict 6-, 8-, 10-, and 12-thread configurations.

The results validate that the COS model based predictor can successfully capture the impact of DVFS, DMT, and DCT simultaneously with prefetching but at the expense of predictor flexibility. The COS approximation techniques implemented in the Intel systems could be extended to better capture the effects of prefetching overlap on performance using approaches similar to those used for power-performance modes.

3.4.2 ROB and MSHR. The sizes of the reorder buffer (ROB) and miss status holding register (MSHR) increase with each generation in CPU design. The ROB reorders instructions to increase instruction-level parallelism and the MSHR increases the number of loads that can be handled under a previous miss. These techniques have the potential to increase overlap and can impact the accuracy of the COS predictor on Intel Systems

We picked nine applications to ascertain the sensitivity of COS to the ROB and MSHR. The Intel hyperthreading design enables us to indirectly control the size of the ROB and MSHR. For a single thread per core, the ROB and MSHR are fixed in size. However, if we overload a core with multiple threads, the ROB and MSHR resources are divided among the threads. We exploit this indirect control in our experiments.

In our experimental setup, we identify two basic configurations: 1) 4-, 6-, and 8-threads where at most one OpenMP thread is mapped to a core, and 2) 10-, 12-, 14-, and 16-threads where at least two



(a) Average absolute prediction error. **(b) Standard deviation of the absolute prediction error.**

Figure 8: Impact of ROB and MSHR on prediction accuracy.

cores run two OpenMP threads. As mentioned, these Intel machines have 8 cores each with two hardware threads per core.

For these experiments we disable prefetching and use our extrapolation approach with 4- and 6-threads for training. Figure 8 shows that both average error and the standard deviation for 8 threads are much better than all the other configurations of threads.

There appears to be a correlation between the least accurate of our earlier experiments, ABSORBDT (Figure 5), and the ROB and MSHR results. Though further experimentation is needed, it is likely that ABSORBDT is sensitive to ROB and MSHR sizes and we could consider improvements to our approximations of the COS model that incorporate these characteristics.

4 CROSS-ARCHITECTURE VALIDATION USING IBM'S BLUE GENE/Q

To demonstrate the portability and scalability of our model we validate COS on IBM's Blue Gene/Q (BG/Q) architecture. BG/Q is a scalable, energy efficient, high-performance system. The BG/Q architecture is capable of dynamic memory bandwidth throttling (DBT), where memory bandwidth is dynamically controlled through insertion of a configurable number of memory idle cycles between each DDR memory request.

BG/Q's DBT is different from the dynamic memory frequency throttling (DMT) common to Intel systems. While memory frequency throttling changes the latency of each main memory access, bandwidth throttling reduces the effective bandwidth through inserting idle cycles (or no-ops or bubbles) in the instruction pipeline. The number of memory idle cycles inserted is called the *throttling threshold* and ranges between 0 and 126. Studies have shown this parameter can affect the performance of applications as well as their power consumption [29].

The throttling threshold affects those memory accesses that occur within the threshold window. For instance, if the time between two dependent memory requests at the memory controller is larger than the throttling threshold, the latency of these memory requests is not affected. When the time between two memory requests is smaller than the threshold, the latency of the second memory request would increase by the configurable number of memory idle cycles.

Unlike the Intel system, BG/Q is not capable of CPU frequency scaling and thus we limit our validation of COS to variations in memory bandwidth and thread concurrency. BG/Q has only two levels of cache, L1 and L2, compared to 3 levels of cache on our x86 experimental system.

4.1 Approximating the COS Trace

Assume we change memory speed from f_m to $f_m^{'}$ (Δf_m) using DBT. To illustrate the effect on performance, Figure 9 shows the execution time in cycles (y-axis) for different throttling thresholds represented by number of memory idle cycles (x-axis) for all regions of the LULESH application. For each region (R1, R2, R3, R4, R5) of LULESH, two different phases can be distinguished: 1) a nearly flat or constant segment in the function at low thresholds and 2) a linearly increasing function at a threshold that appears to be different for each region. This forms a hockey stick shaped function for each region with a different inflection point. In a way, this is an example of Amdahl's law applied to an architectural enhancement. A portion of the code (phase 1 in this example) *is not affected* by the enhancement (e.g., insertion of memory idle cycles) while a portion of the code (phase 2 in this example) *is affected* by the enhancement. While this is an oversimplification in some ways, it implies that we can potentially use a piece-wise function to approximate the performance for these codes if we can identify the inflection point (i.e., the number of memory idle cycles) where performance loss begins.

Following a series of experiments, we determined the inflection points correlate to characteristics of a region's memory access behavior. We approximate the COS Trace expressed by Equation 3 using a piecewise function of performance:

$$T = \begin{cases} t_0 & \text{if } f_m \leq a \\ b f_m + t_0 & \text{if } f_m > a \end{cases} \tag{10}$$

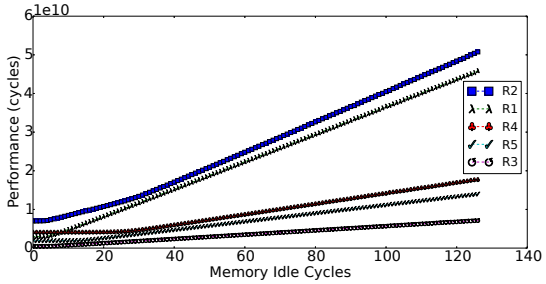


Figure 9: Impact of memory bandwidth throttling on LULESH.

where t_0 is the performance with no memory throttling, a is the threshold of the inflection point, and b is the slope of the linear function.

For the constant function ($T = t_0$), memory throttling has little impact on the COS Trace: performance does not change by inserting memory idle cycles. This can be explained with the following two cases. First, the gap between most of the application memory accesses is larger than the throttling threshold. The number of inserted memory idle cycles is too small to cause delays in memory accesses (T_s is not changing) and thus total execution time. In this case, inserting idle cycles does not change T_c, T_o, and T_s. Second, the gap between memory accesses is smaller than the throttling threshold, but the memory accesses overlap with processor computation. Inserting idle cycles can delay issuing new memory requests but does not change the length of any of the three stages, T_c, T_o, and T_s.

The inflection point a in Equation 10 depends on the memory access patterns of applications. A correlation analysis among some critical compute/memory related hardware events (e.g. floating point operations, L2 cache misses per second, etc.) shows that its value is highly related to memory intensity: *L2 cache misses (L2M) per instruction (INST)*. By applying linear regression, we can approximate the value of a with the following:

$$a = \alpha \times \frac{L2M}{INST} + c_1$$

where α is a coefficient and c_1 is a constant and both will be determined using linear regression.

The impact of memory throttling on the second segment is linear ($T = bf_m + t_0$). This can be explained with the COS Trace as follows. For a sufficiently large number of idle cycles, application memory accesses cannot overlap with computation. In this case, the length of the pure compute stages would not change with memory throttling; the length of the overlap stages would be zero; and the length of the pure stall stages would change linearly with the number of memory idle cycles inserted. Approximating the number of memory accesses with L2 misses, the impact on the COS Trace can be expressed as follows:

$$\Delta T_c = 0$$
$$\Delta T_o = 0$$
$$\Delta T_s = \beta \times L2M \times \Delta f_m$$

where ΔT_c, ΔT_o, and ΔT_s are the resulting change to execution time for each respective phase. Thus, we can approximate the value of b as follows:

$$b = \beta \times L2M + c_2$$

where β is a coefficient and c_2 is a constant and both will be determined using linear regression.

Based on the equations above, we can predict performance using Equation 10 as follows:

$$T = \begin{cases} t_0 & \text{if } f_m \le \alpha \times L2M/INST + c_1 \\ (\beta \times L2M + c_2) \times f_m + t_0 & \text{if } f_m > \alpha \times L2M/INST + c_1 \end{cases} \quad (11)$$

4.2 Offline Training and Online Prediction

We apply linear regression to approximate the model coefficients of Equation 11. The configuration space includes two parameters: the throttling threshold (Δf_m) and the number of threads (Δt). The threshold ranges from 0 to 126 idle cycles and the number of threads from 4 to 64 with an interval of 4. The details of the training configurations and the overall configuration space is given in Table 3. We use the five code regions of LULESH to train the model. Each region has its own trained coefficients.

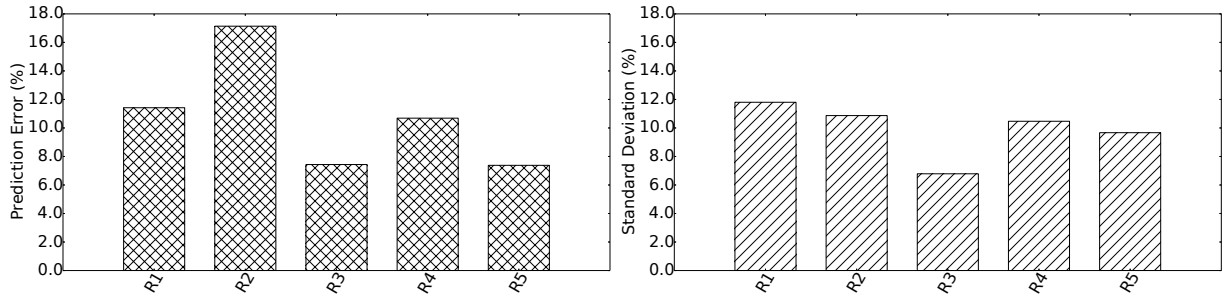
Table 3: The training configurations and the overall configuration space on BG/Q. The *Total* column shows the number of configurations.

Total	Δf_m	Δt (num. threads)
Training inflection point a		
127×16	0 - 126 cycles	4, 8, 12, ..., 64
Training slope b for Region 1		
27×14	100 - 126 cycles	12, 16, ..., 64
Training slope b for Region 2		
16×10	100 - 115 cycles	28, 32, ..., 64
Training slope b for Region 3		
11×12	35 - 45 cycles	20, 24, ..., 64
Training slope b for Region 4		
16×9	100 - 115 cycles	32, 36 , ..., 64
Training slope b for Region 5		
11×11	65 - 75 cycles	24, 28, ..., 64
Configuration space		
127×16	0 - 126 cycles	4, 8, 12, ..., 64

We use the model to predict the performance of the five code regions of LULESH for those configurations in the configuration space that are not in the training set. To measure the accuracy of the model, we compare these predicted values with the performance measured by running the same configurations on the machine.

Figure 10 shows the average error and standard deviation of our model. Four of the five code regions show a reasonable average error, around 10% or less. Region 2, however, shows a large average error of 17%.

There are several factors that affect the model accuracy of the BG/Q implementation of the COS model. First, our experiments



(a) Average prediction error. **(b) Standard deviation of the prediction error.**

Figure 10: Multivariate linear regression of LULESH on IBM BG/Q system.

on BG/Q included prefetching, which makes the overlap time T_o more complex as observed on the Intel system. On BG/Q we used L2 cache misses to represent memory pressure similar to the Intel system. We could relax this requirement and use the number of load and store operations on BG/Q for our approximations. This may improve accuracy.

Second, in some cases the value of a may not be strictly constant but a linear function with a small slope value. The number of idle cycles that can be inserted (integers) does not provide fine-enough granularity to estimate a more accurately.

Third, we used a small number of sample configurations for training a because we limited our configuration space to only a subset of the available number of threads. We expect that a larger space using all 64 (from 1 to 64 threads) configurations along the Δt dimension would have resulted in better accuracy.

5 LIMITATIONS AND DISCUSSION

We have demonstrated the use and accuracy of the COS model for predicting the performance of a set of DVFS, DMT/DBT, and DCT configurations. The model can be used in a software or hardware implementation to allocate or deallocate resources to the working threads in a parallel application. This is an advantage to a parallel scheduler or runtime system.

As mentioned the key concept of the COS model is the isolation of overlap. While in an abstract sense this is straight forward, we show in Sections 2.4 – 2.6 that empirically isolating overlap is wrought with challenges. We resolved a number of these challenges using the assumption of regular parallel applications where threads are mostly homogeneous and computation proceeds in a bulk synchronous way with no other dependencies among threads. This leaves a number of limitations to the model that must be addressed to consider irregular parallel codes (e.g., heterogeneous threads, asynchronous, cross-thread dependencies).

Overlap types In our earlier discussions, we simplified the definition of overlap into computation overlap and memory overlap. In general, overlap can also occur between multiple threads on a single core/CPU or across multiple cores/CPUs accessing the same memory. Under our assumptions, these don't affect the COS Trace much, but these must be considered for irregular parallel codes.

Computational intensity Computational intensity (*CI*) has impact on the overlap as discussed earlier. A key insight gained from this work is that *CI* can be used to predict the impact of simultaneous configuration changes in CPU, memory, and threads on overlap. More overlap types, combined with irregular codes, are likely to make accurate prediction more challenging. There could also be non–*CI* effects that we've not accounted for in parallel irregular applications.

Role of Co-design These challenges could be alleviated somewhat by improvements in our ability to directly measure the overlap of parallel codes. This could be accomplished in software, but would be most effective when co-designed with hardware. Our work indicates that there are meaningful representative hardware counters that give insight to overlap and computational intensity, but they are indirect at best. Furthermore, this data is usually limited to an individual thread with no context for other concurrent threads on the same core or CPU. Mechanisms for tracking this type of information could vastly improve our understanding of overlap as well as our ability to optimize parallel applications and systems.

6 RELATED WORK

To the best of our knowledge, this work is the first to propose an analytical performance model that captures the simultaneous effects of DVFS, DCT, and DMT/DBT on the performance of multithreaded applications on real systems.

Table 4 provides a synopsis of work most closely related to ours. There has been extensive work focused on modeling the effects of CPU DVFS on performance using stall-based approaches [22]; leading loads [16, 22, 33]; CRIT-BW, a leading load derivative [31]; and DEP-BURST, a CRIT-BW derivative [3]. While all of these consider the effects of out-of-order execution and non-blocking caches, only DEP-BURST considers multithreading using a critical path analysis to determine which core to boost.

Table 4 shows how the resulting CPU DVFS performance models capture the characteristics of the COS model: T_c, T_o, T_s from Equation 1. Stall-based approaches assume the CPU DVFS affects overlap time in the same way it affects pure compute time and thus combine T_c and T_o. They also purport that pure stall time (T_s) is constant with changes in CPU frequency – this is in direct contrast to our findings that stall time is affected by CPU frequency (see Figure 2).

Table 4: Summary of existing performance models and the proposed COS model. *OOO Exec* and *NB Cache* refer to processor out-of-order execution and non-blocking cache, respectively.

Model	Predicts	Under			Considering			On	Captures		
		DVFS	DMT	DCT	Multithread	OOO Exec	NB Cache		T_c	T_o	T_s
Stall-based	Runtime	✔				✔	✔	Simulation			
Leading loads	Runtime	✔				✔	✔	Simulation			
CRIT-BW	Runtime	✔				✔	✔	Simulation			
DEP-BURST	Runtime	✔			✔	✔	✔	Simulation			
MemScale	CPI		✔					Simulation			
CoScale	CPI	✔	✔					Simulation			
Joint	# micro-ops	✔	✔		✔	✔		Real system			
COS	Runtime	✔	✔	✔	✔	✔	✔	Real system			

The leading load model and its derivatives combine overlap time (T_o) with pure stall time (T_s) and assume the combined value is constant while T_c is proportional to CPU frequency. This assumption leads to inaccuracies since the impact of CPU frequency on T_o can be quite different from T_c and T_s – as we discussed in Sections 2.4 – 2.7.

Su et. al. [35] is the only work we know of that implements the leading load model on real systems. This is the most accurate model available for a real system but it only models DVFS on AMD architectures. The COS Model implementations in this paper are as accurate or better than this and model the combined effects of DVFS, DMT/DBT, and DCT across multiple architectures. Su et. al. also showed that the leading load approach is less accurate for memory intensive applications and that the accuracy of the leading load model is highly dependent on the level of memory boundedness – these match our findings as well.

Table 4 also shows a comparison with memory power performance modeling tools. Deng et. al. [12, 13] presented a performance model for memory frequency scaling (MemScale and MultiScale) of single threaded applications on in-order processors. They made similar assumptions as those in the CPU DVFS models that the overlap time (T_o) is combined with pure compute time (T_c). Deng et. al. [11] created CoScale to extend MemScale to consider DVFS. The accuracy is very good for single threaded applications on in-order processors. But the limiting combination of T_o and T_c remains.

Sundriyal and Sosonkina [36] proposed the "Joint" performance model that considers the simultaneous effects of CPU DVFS and DMT. However, the model estimates T_o as a constant for all applications on a single system. This contradicts our findings that overlap is affected by CPU frequency (see Figure 2).

Less directly related work relevant to our discussions include: David et al. [10] investigated the impact of memory frequency scaling on power and performance and proposed a model for real systems; Li et al. studied the throttling interface on IBM BG/Q systems and demonstrated its ability to optimize system efficiency [29]; Ercan et. al. [14] presented a heuristic runtime solution for coordinating CPU and memory frequencies to improve energy efficiency; Curtis-Maury et al. created heuristic models that manage DVFS and DCT simultaneously for multi-threaded applications [8, 9, 20, 25, 30].

7 CONCLUSIONS AND FUTURE WORK

In this paper, we propose the COS Model of parallel performance to accurately capture the combine effects of DVFS, DMT/DBT, and thread concurrency on real systems. We applied the COS model to both Intel and IBM architectures within 7% and 17% accuracy for a set of 19 important applications. The key insight to the COS model is the separation of memory and compute overlap from pure compute and pure memory stalls. This separation enables more accurate approximations and a straightforward methodology that is capable of modeling the complexity introduced with concurrency. A key limitation of the model is the focus on structured parallel codes that while representative of many important applications precludes accurate use on irregular parallel codes for now. Despite the limitations, we provide strong evidence that the fundamental focus on overlap in the COS model will be key to steering future high-performance systems and applications to maximize their efficiencies. In future work, we plan to explore extending the COS model to irregular parallel applications in both OpenMP and MPI. We also plan to adapt the techniques described for use in runtime systems.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1565314 and 1422788. Lawrence Livermore National Laboratory is operated by Lawrence Livermore National Security, LLC, for the U.S. Department of Energy, National Nuclear Security Administration under Contract DE-AC52-07NA27344. LLNL-CONF-728263.

REFERENCES

[1] *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory.* Technical Report LLNL-TR-490254. 1–17 pages.
[2] *Intel 64 and IA-32 Architectures Developer's Manual, Intel Corporation.* Technical Report Volume 3B, Part 2.

[3] Shoaib Akram, Jennifer B Sartor, and Lieven Eeckhout. 2016. DVFS Performance Prediction for Managed Multithreaded Applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 12–23.

[4] Sadaf R. Alam and Jeffrey S. Vetter. 2006. An Analysis of System Balance Requirements for Scientific Applications. In *International Conference on Parallel Processing (ICPP'06)*. IEEE, Columbus, OH.

[5] Teresa Bailey, W. Daryl Hawkins, Marvin L. Adams, Peter N. Brown, Adam J. Kunen, Michael P. Adams, Timmie Smith, Nancy Amato, and Lawrence Rauchwerger. 2014. Validation of Full-Domain Massively Parallel Transport Sweep Algorithms. In *American Nuclear Society Winter Meeting and Nuclear Technology Expo*. Anaheim, CA.

[6] Aaron Carroll and Gernot Heiser. 2010. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1855840.1855861

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite For Heterogeneous Computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. 44–54. DOI: http://dx.doi.org/10.1109/IISWC.2009.5306797

[8] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2006. Online Power-performance Adaptation of Multithreaded Programs Using Hardware Event-based Prediction. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*. ACM, New York, NY, USA, 157–166. DOI: http://dx.doi.org/10.1145/1183401.1183426

[9] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. 2008. Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 250–259. DOI: http://dx.doi.org/10.1145/1454115.1454151

[10] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. 2011. Memory Power Management via Dynamic Voltage/Frequency Scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11)*. ACM, New York, NY, USA, 31–40. DOI: http://dx.doi.org/10.1145/1998582.1998590

[11] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. 2012. CoScale: Coordinating CPU and Memory System DVFS in Server Systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 143–154. DOI: http://dx.doi.org/10.1109/MICRO.2012.22

[12] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. 2012. MultiScale: Memory System DVFS with Multiple Memory Controllers. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '12)*. ACM, New York, NY, USA, 297–302. DOI: http://dx.doi.org/10.1145/2333660.2333727

[13] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. 2011. MemScale: Active Low-power Modes for Main Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 225–238. DOI: http://dx.doi.org/10.1145/1950365.1950392

[14] Furkan Ercan, Neven Abou Gazala, and Howard David. 2012. An Integrated Approach to System-level CPU and Memory Energy Efficiency on Computing Systems. In *Energy Aware Computing, 2012 International Conference on*. 1–6. DOI: http://dx.doi.org/10.1109/ICEAC.2012.6471018

[15] Constantinos Evangelinos, Robert Walkup, Vipin Sachdeva, Kirk E. Jordan, Hormozd Gahvari, I-Hsin Chung, Michael P. Perrone, Ligang Lu, Lurng-Kuo Liu, and Karen A. Magerlein. 2013. Determination of Performance Characteristics of Scientific Applications on IBM Blue Gene/Q. *IBM Journal of Research and Development* 57, 1/2 (2013), 9. DOI: http://dx.doi.org/10.1147/JRD.2012.2229901

[16] S. Eyerman and L. Eeckhout. 2010. A Counter Architecture for Online DVFS Profitability Estimation. *Computers, IEEE Transactions on* 59, 11 (Nov 2010), 1576–1583. DOI: http://dx.doi.org/10.1109/TC.2010.65

[17] R. Ge, X. Feng, W. c. Feng, and K. W. Cameron. 2007. CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters. In *2007 International Conference on Parallel Processing (ICPP 2007)*. 18–18. DOI: http://dx.doi.org/10.1109/ICPP.2007.29

[18] Van Emden Henson and Ulrike Meier Yang. 2000. BoomerAMG: a Parallel Algebraic Multigrid Solver and Preconditioner. *Applied Numerical Mathematics* 41 (2000), 155–177.

[19] Shadi Ibrahim, Tien-Dat Phan, Alexandra Carpen-Amarie, Houssem-Eddine Chihoub, Diana Moise, and Gabriel Antoniu. 2016. Governing Energy Consumption in Hadoop Through CPU Frequency Scaling: An Analysis. *Future Generation Computer Systems* 54 (2016), 219 – 232. DOI: http://dx.doi.org/10.1016/j.future.2015.01.005

[20] Changhee Jung, Daeseob Lim, Jaejin Lee, and SangYong Han. 2005. Adaptive Execution Techniques for SMT Multiprocessor Architectures. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. ACM, New York, NY, USA, 236–246. DOI: http://dx.doi.org/10.1145/1065944.1065976

[21] Ian Karlin and Mike Collette. 2014. Strong Scaling Bottleneck Identification and Mitigation in Ares. In *Nuclear Explosives Code Development Conference (NECDC'14)*. Los Alamos, NM.

[22] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. 2010. Interval-based Models for Run-time DVFS Orchestration in Superscalar Processors. In *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF '10)*. ACM, New York, NY, USA, 287–296. DOI: http://dx.doi.org/10.1145/1787275.1787338

[23] Steven H. Langer, Abhinav Bhatele, and Charles H. Still. 2014. pF3D Simulations of Laser-Plasma Interactions in National Ignition Facility Experiments. *Computing in Science & Engineering* 16, 6 (Nov 2014), 42–50.

[24] Steve H. Langer, A. Bhatele, and C. H. Still. 2014. pF3D Simulations of Laser-Plasma Interactions in National Ignition Facility Experiments. *Computing in Science Engineering* 16, 6 (Nov 2014), 42–50. DOI: http://dx.doi.org/10.1109/MCSE.2014.79

[25] Jaejin Lee, Jung-Ho Park, Honggyu Kim, Changhee Jung, Daeseob Lim, and SangYong Han. 2010. Adaptive execution techniques of parallel programs for multiprocessors. *J. Parallel and Distrib. Comput.* 70, 5 (2010), 467 – 480. DOI: http://dx.doi.org/10.1016/j.jpdc.2009.10.008

[26] Edgar A. Leon, Ian Karlin, Abhinav Bhatele, Steven H. Langer, Chris Chambreau, Louis H. Howell, Trent D'Hooge, and Matthew L. Leininger. 2016. Characterizing Parallel Scientific Applications on Commodity Clusters: An Empirical Study of a Tapered Fat-Tree. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*. IEEE/ACM, Salt Lake City, UT.

[27] Edgar A. León, Ian Karlin, and Ryan E. Grant. 2015. Optimizing Explicit Hydrodynamics for Power, Energy, and Performance. In *International Conference on Cluster Computing (Cluster'15)*. IEEE, Chicago, IL.

[28] Edgar A. León, Ian Karlin, Ryan E. Grant, and Matthew Dosanjh. 2016. Program Optimizations: The interplay Between Power, Performance, and Energy. *Parallel Comput.* 58 (Oct. 2016), 56–75. http://dx.doi.org/10.1016/j.parco.2016.05.004

[29] Bo Li and Edgar A. León. 2014. Memory Throttling on BG/Q: A Case Study with Explicit Hydrodynamics. In *6th Workshop on Power-Aware Computing and Systems (HotPower 14)*. USENIX Association, Broomfield, CO. https://www.usenix.org/conference/hotpower14/workshop-program/presentation/li

[30] Dong Li, B.R. de Supinski, M. Schulz, K. Cameron, and D.S. Nikolopoulos. 2010. Hybrid MPI/OpenMP Power-Aware Computing. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. 1–12. DOI: http://dx.doi.org/10.1109/IPDPS.2010.5470463

[31] R. Miftakhutdinov, E. Ebrahimi, and Y.N. Patt. 2012. Predicting Performance Impact of DVFS for Realistic Memory Systems. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*. 155–165. DOI: http://dx.doi.org/10.1109/MICRO.2012.23

[32] Pier Giorgio Raponi, Fabrizio Petrini, Robert Walkup, and Fabio Checconi. 2011. Characterization of the Communication Patterns of Scientific Applications on Blue Gene/P. In *International Workshop on System Management Techniques, Processes, and Services (SMTPS'11)*. Anchorage, AK.

[33] B. Rountree, D.K. Lowenthal, M. Schulz, and B.R. de Supinski. 2011. Practical Performance Prediction under Dynamic Voltage Frequency Scaling. In *Green Computing Conference and Workshops (IGCC), 2011 International*. 1–8. DOI: http://dx.doi.org/10.1109/IGCC.2011.6008553

[34] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. 2009. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 460–469. DOI: http://dx.doi.org/10.1145/1542275.1542340

[35] Bo Su, Joseph L. Greathouse, Junli Gu, Michael Boyer, Li Shen, and Zhiying Wang. 2014. Implementing a Leading Loads Performance Predictor on Commodity Processors. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA. https://www.usenix.org/conference/atc14/technical-sessions/presentation/su

[36] Vaibhav Sundriyal and Masha Sosonkina. 2016. Joint Frequency Scaling of Processor and DRAM. *The Journal of Supercomputing* 72, 4 (2016), 1549–1569. DOI: http://dx.doi.org/10.1007/s11227-016-1680-4

[37] Jeffrey S. Vetter and Frank Mueller. 2002. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. In *International Parallel and Distributed Processing Symposium (IPDPS'02)*. IEEE, Fort Lauderdale, FL.