# Scheduling for HPC Systems with Process Variation Heterogeneity

Ehsan Totoni, Akhil Langer, Josep Torrellas, Laxmikant V. Kale

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
E-mail: {totoni2, alanger, torrella, kale}@illinois.edu

***Abstract*—Variation in the CMOS manufacturing processes cause the transistors on each chip to differ, which results in many-core chips being inherently heterogeneous. For example, frequency and power consumption profiles of cores can span a wide range. This makes optimal scheduling of applications under a power budget computationally difficult, because of the combinatorially large number of choices available. To facilitate this, we model the performance and power consumption of HPC applications on such heterogeneous chips. Based on our models, we propose a scheduling framework using integer linear programming (ILP), which enables efficient scheduling with various power consumption and performance constraints. Using this framework, an HPC runtime system can decide how many and which cores of a chip to use depending on the application, the properties of the chip, and the imposed constraints. Our results show that our framework finds configurations that are up to 2.5 times faster than the ones obtained from simple heuristics. We also propose various research directions for this problem based on our framework.**

## I. Introduction

Process variation is the deviation of transistor parameters from their design (nominal) values, which is caused by systematic effects (e.g., lithographic inconsistencies) and random effects (e.g., varying dopant concentrations) [1]. Affected parameters include effective channel length, channel width, and threshold voltage. Therefore, transistor characteristics such as switching speed and current leakage can vary widely across the chip. Incorporating very small feature sizes in succeeding CMOS technology generations and lowering the supply voltage, which is necessary for power efficiency [2], exacerbate the process variation problem.

At the architectural level, process variation results in cores and on-chip memories having different frequencies and static power consumption profiles. The reason is that a core's frequency is determined by the switching speed of the transistors on its critical path, which depends on the characteristics of those transistors. In addition, static power has an exponential relationship with the threshold voltage of transistors. Many designers tackle this issue by leaving design margins, but this solution is deemed too wasteful, especially for future generation technologies and many-core architectures. A recent study estimated the within-die frequency variation of many-core chips in 11nm technology generation to be 2.3x for conventional high voltage operation (known as Super-Threshold Voltage Computing) and 3.7x for very low voltage operation (known as Near-Threshold Voltage Computing) [3].

Process variation leads to high heterogeneity in processor chips, which has direct consequences to High Performance Computing (HPC) environments. For example, frequency heterogeneity can slow down most multithreaded applications written in various parallel programming paradigms such as MPI [4] and Pthreads [5]. This is because parallel programmers usually assume different cores/processors have the same speed and assign the computational load uniformly. In addition, the execution of different processes/threads is synchronized in most HPC applications. Therefore, the slowest core determines the execution time, unless variation-aware load balancing is performed (as we evaluate in Section IV).

Furthermore, heterogeneity makes power and energy management harder, since different parts of the chip can have widely different performance and power characteristics. The challenge is to utilize future HPC machines efficiently while staying within the performance, power, and energy constraints. Previous studies have developed scheduling heuristics for multiprogrammed environments [6], [7], [3], but HPC environments are different because usually only one parallel application runs on the whole chip. We strive to solve the performance, energy, and power problem for heterogeneous chips by developing a novel scheduling framework, which can be implemented in intelligent HPC runtime systems. The only requirement is being able to migrate work units and balance the load according to the configuration chosen by our framework. Our solution does not change the programming paradigms and existing codes and has negligible execution time overheads.

For an application running on a processor chip, the runtime has to choose a configuration among potentially billions of options. Each configuration instructs how many and which cores will execute the parallel program, and leaves other cores off. For a chip with $n$ frequency domains, there are $2^n - 1$ configurations[1]. This translates to 67 billion configurations for a chip proposed by previous work with $n = 36$ [3]. The number of frequency domains is expected to be even larger for future Exascale many-core chips, resulting in an enormous number of possible configurations. Testing all of these configurations is infeasible for the runtime system. Since Exascale architectures are also very likely to be over-provisioned [8], [9], variation-aware power management is essential for them to stay within their power budget. We develop a scheduling framework with accurate performance and power models that explores the combinatorial search space efficiently and finds a close to optimum configuration quickly. It only needs a few samples of application execution to build the required models.

Performance modeling efforts for parallel applications have

[1] 'all cores off' is not useful

usually assumed that different processor cores have the same speed [10], [11], or the system is heterogeneous but there are a few processor types (e.g. GPU and other accelerators) [12], [13]. However, process variation causes a new form of heterogeneity that potentially makes all of the cores/processors of the system different. We develop and study the accuracy of four different performance models and apply the most accurate one for making scheduling decisions. Studying performance models also gives us insight into the impacts of heterogeneity on performance and power consumption.

For a large chip with many frequency domains, evaluating even simple models for all the configurations is infeasible. Therefore, we use integer linear programming (ILP) to explore the search space efficiently. Our results show that our ILP-based scheduling provides configurations that perform 25% better on average for a compute-bound application and 16% better for a memory-bound application as compared to scheduling algorithms based on heuristics. In some cases, our framework finds configurations that are up to 2.5 times faster than the baseline heuristic. Furthermore, we demonstrate how different performance and power scheduling constraints can be expressed as linear models to be used by our ILP scheduling framework. Since ILP provides optimality guarantees (assuming that the models are accurate), our framework can be used to evaluate simpler scheduling heuristics as well.

The rest of the paper is organized as follows. Section II presents background on process variation. Section III describes our evaluation setup. Section IV discusses the requirements of programming systems, and evaluates the load imbalance caused by process variation heterogeneity. In Section V, we design and evaluate different performance models. We use these models in Section VI to define an ILP-based scheduling framework. We evaluate this framework versus simple heuristics in Section VII. We discuss the related work in Section VIII, and conclude in Section IX.

## II. Background on Process Variation

Ideally, all transistors of a die should be identical and have the same parameters as designed, but this is hard to achieve in manufacturing. Therefore, there are static, spatial fluctuations of parameters around the nominal values. The variation of transistors across different dies is called *die-to-die variation*, while the difference of transistors on the same die is called *within-die variation*.

Variation affects two critical parameters of transistors: threshold voltage ($V_{th}$) and effective channel length ($L_{eff}$). These parameters determine switching speed and leakage of the transistors. At the architectural level, process variation causes some processor cores to run faster or slower than the intended design. This is determined by the speed of the transistors on the critical path of each core. Figure 1 illustrates the frequency variability of a hypothetical chip. In addition, the static power consumption of different cores and on-chip memory units is determined by the leakage of their transistor, which varies.

To display a uniform view of the system to the user, the designers usually include margins to cover variations. However, due to growing variations in successive processor generations, researchers believe that this will become too costly [14]. For example, by using all the cores at a very low frequency, one pays the static power cost of all the cores but achieves limited performance. On the other hand, to alleviate power limitations, low voltage operation seems to be required [2], but it will exacerbate the variation issues. Hence, process variation needs to considered for future processor chips.

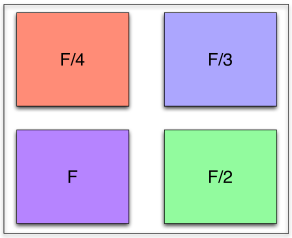



Fig. 1. An example of core frequency variation on the same chip.

## III. Evaluation Setup

For evaluation of our approach, we model heterogeneous chips using the Sniper simulator [15]. We use Sniper's default core model, which is similar to the Intel's Gainestown microarchitecture and has been validated [15].

To model process variation at the micro-architecture level, we use VariusNTV [16]. It models systematic variation by dividing the die into a grid, and assigning $V_{th}$ and $L_{eff}$ to each point by sampling from a multivariate Gaussian distribution. It models both spatially correlated variation and random variation, and its results have been validated against chip measurements [16].

We simulate 12-core or 36-core chips with each core in a different frequency domain, but one voltage for the whole chip. In the following sections, we refer to frequency domains simply as cores for convenience. Some previous works propose multiple small cores (a cluster) in each frequency domain. However, we use a simpler architecture to be able to evaluate our models more accurately and to simulate the possible configurations exhaustively for some cases. We believe that our results extend to other architectures as well. We have verified our simulation setting by comparing the simulation results of a corresponding homogeneous architecture against a 12-core Ivy Bridge machine. Table I presents the parameters of the modeled system in this paper.

TABLE I. Simulated processor's parameters

| Sniper parameters | |
|---|---|
| Chip | 12 or 36 Core CMP |
| Core | x86, 4-wide issue out-of-order |
| Instruction L1 (L1I) | 32 KB, 4 way |
| Data L1 (L1D) | 32 KB, 8 way, private. |
| L2 | 256 KB, 8 way, private. |
| Memory latency (no contention) | 75ns |
| **VariusNTV parameters** | |
| Technology | 11 nm |
| Average frequency | 2.6 GHz |
| $V_{dd}$ | 0.765 |
| Correlation range $\Phi$ | 0.1 |
| Total $(\sigma/\mu)$ for $V_{th}$ | 15% |

We use MiniMD and Jacobi3D to represent typical HPC

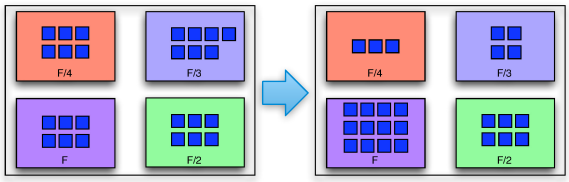



Fig. 2. An example of load balancing across cores with different frequencies.

workloads. MiniMD represents molecular dynamics workloads, which are compute-intensive. Jacobi3D represents stencil computations, which are typically memory-bound. Other HPC applications are typically between these two in terms of being compute-intensive or memory-bound. We use instructions per cycle (IPC) as a proxy for application's performance. In each simulation, after initialization, and warm up for two seconds, we run the application for about six seconds of simulated time, and take the average of the several IPC samples collected at the intervals of milliseconds over the period of six seconds. Since the configuration is heterogeneous and frequencies are not the same, we normalize the IPC statistics of different cores by multiplying them with the frequencies of the corresponding cores and dividing by the frequency of the slowest core on the chip. We use McPAT [17] to evaluate dynamic power consumption and VariusNTV to evaluate static power consumption.

## IV. Programming Systems

Some parallel programming systems provide adaptive features such as automatic load balancing. Essentially, a scheduler decides how much work is assigned to each core. For example, an OpenMP runtime system can assign loop iterations to idle cores dynamically. Furthermore, the Charm++/AMPI [18] runtime system measures the load of different processors and balances the load accordingly.

In the context of process variation, runtime scheduling is even more important. For example, the runtime system should assign more work to faster cores and less work to slower ones. Otherwise, a multithreaded application will run at the pace of the slowest core. Figure 2 illustrates how load balancing is performed by migrating units of work among different cores. In this study, we assume that the runtime system has a way of assigning units of work to different cores, but our results do not depend on how it is done.

### A. Impact on Load Balance

In this paper, we assume that the runtime can achieve perfect load balance, but this is not always the case. Since the speeds of cores can be widely different, dividing a fixed number of similar tasks according to their speeds is not always possible. For instance, if there are three cores with frequencies of 1 GHz, 2 GHz, and 3 GHz, 7 equal tasks cannot be scheduled perfectly. In this case, at least one core has more work than the average load, and some other cores will be waiting for it to finish.

The work units cannot always be fine-grained enough to achieve perfect load balance. This is because scheduling and managing very fine-grained work units has high overhead. For example, in Charm++ and AMPI, usually a fixed number of tasks (e.g. *chares* or MPI ranks) are scheduled on different cores. In addition, in shared-memory approaches such as OpenMP and Intel TBB, chunks of iterations are scheduled by the runtime system. In this section, we strive to quantify the load imbalance due to work units not being fine-grained enough to be balanced perfectly on the chips with high variation.

We characterize the impact of the granularity of tasks (for a fixed computation) on load imbalance caused by variation in different configurations of a chip. A configuration is a subset of the cores on the chip, on which the parallel application will be run. We define the overdecomposition ratio as the ratio of the number of tasks to the number of cores. We also define our load imbalance metric using the ratio of maximum load to average load [19]:

$$\mathcal{I} = (\frac{L_{max}}{L_{avg}} - 1) \times 100 \quad (1)$$

where, $L_{max}$ is the maximum load of any core in the configuration, and $L_{avg}$ is the average load of all the cores in the configuration. These load values are obtained by appropriately scaling the assigned load with the frequencies of the corresponding cores. We use a greedy algorithm for load balancing, which is outlined in Algorithm 1. The algorithm assigns a *capacity* for work to each core, which is equal to its frequency (lines 2-4). In this way, more work will be assigned to faster cores. It then makes a heap (line 5), which has the core with the maximum available capacity at the top. The main loop (lines 6-14) removes the core with the maximum capacity from the heap, assigns a work unit to it, and adds it back to the heap. If there was not enough capacity on that core, the algorithm increases the capacity of all the cores by their frequencies (lines 8-10). This load balancing problem can be modeled as a variable size bin packing problem [20]. Theoretical analysis of this algorithm is left for future work.

```
1  Algorithm: VariationAwareGreedyLoadBalancing
   Input: N work units, C cores
2  for each core c ∈ C do
3  |  c.capacity ← c.frequency;
4  end
5  H ← MakeMaxHeap(C);
6  for each work_unit n ∈ N do
7  |  c ← RemoveMaxHeap(H);
8  |  while c.capacity < n.load do
9  |  |  IncreaseAllCapacities(C);
10 |  end
11 |  c.workUnits ← c.workUnits ∪ n;
12 |  c.capacity ← c.capacity − n.load;
13 |  AddMaxHeap(H,c);
14 end
```

**Algorithm 1:** Greedy variation-aware load balancing algorithm.

Figure 3 summarizes the load imbalance on various configurations of a 12-core chip. We simulated Algorithm 1 with different over-decomposition ratios for all the $2^{12} - 1$ configurations of enabled and disabled cores exhaustively, assuming that the work units are of equal size. The two bars correspond

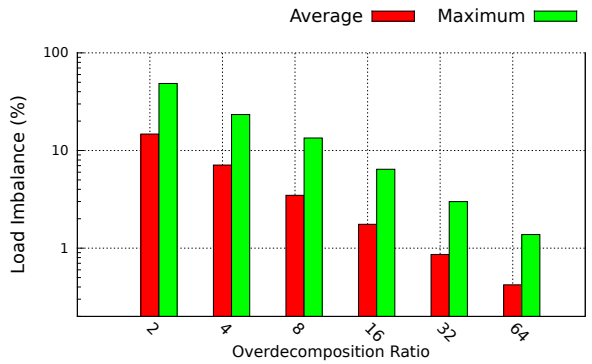



Fig. 3. The average and maximum load imbalance across all configurations with different over-decomposition ratios.

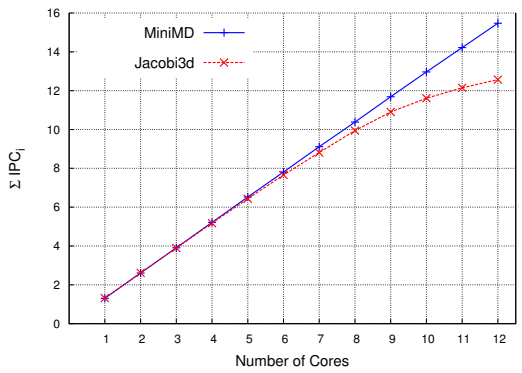



Fig. 4. Performance scaling with cores.

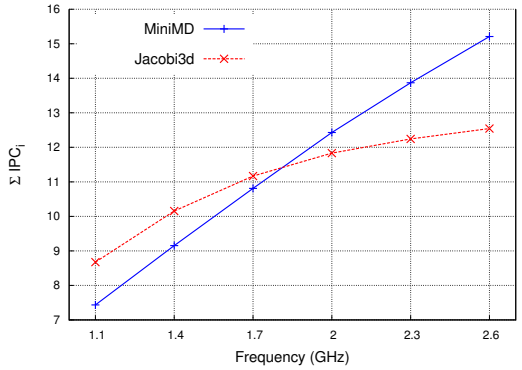



Fig. 5. Performance scaling with frequency.

to the average load imbalance across all the configurations and the maximum load imbalance for any configuration, respectively. As illustrated, more over-decomposition reduces the load imbalance rapidly. With an over-decomposition ratio of 16, the average load imbalance across all the configurations is 2%, while the maximum is 6%. Since an over-decomposition ratio of 16 is already in use for some current CHARM++ applications [21] and seems extensible to most applications, we conclude that load imbalance is not a fundamental problem for our approach. Therefore, in later sections, we assume that the runtime system can balance the load almost perfectly on any configuration chosen by the scheduler.

## V. PERFORMANCE AND POWER MODELING

To be able to find acceptable configurations, we need power and performance models that can evaluate each configuration accurately. Most models in the literature assume homogeneous chips [10], [11] or heterogeneous ones with only a few different processor types (such as GPUs and other accelerators) [12], [13]. Therefore, new models need to be developed for chips with high variation and heterogeneity. Building the models should require only a few (e.g. $O(n)$) samples, to be feasible for the runtime system to collect that many samples. In this section, we assume that the runtime system can balance the load perfectly.

Figures 4 and 5 compare the performance of miniMD and Jacobi3D with different number of cores and frequencies of a 12-core homogeneous systems. We use the insights from these plots to develop our models. For example, compute-bound applications, such as MiniMD, scale well with more cores and/or higher frequency. However, memory-bound applications do not scale beyond a certain point, since memory bandwidth becomes the bottleneck.

### A. Model 1

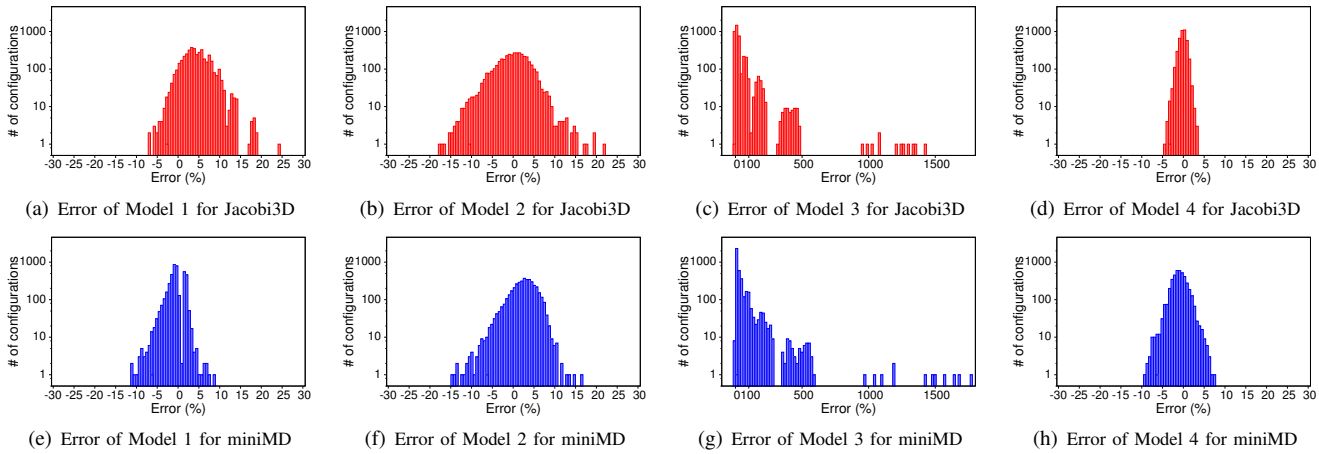
Our first model assumes that adding each core to the configuration improves the application's performance proportionate to the core's frequency. However, this improvement is different for various applications, and therefore performance samples are needed.

We assign a performance value to each core ($s_i$). The performance of a configuration is modeled as:

$$S_c = \sum_{i \in c} s_i \tag{2}$$

For each core $i$, $s_i$ is the performance (normalized IPC) of the application when running on that core alone. Therefore, $n$ samples can be used to obtain the $s_i$ values. Note that since $s_i$ values are proportionate to the frequencies of the corresponding cores (see Figure 5), one can sample the slowest and fastest cores and build a linear model to predict the other values. Hence, just two samples would suffice for this model as well.

Figures 6(a) and 6(e) show the error of Model 1 for Jacobi3D and miniMD. Each point corresponds to a configuration. All of the configurations of our 12-core chip (total number of configurations = 4095) are simulated and evaluated exhaustively. Positive values mean that the model overestimated the performance, while negative values illustrate that the model underestimated the performance. As can be seen in the figure, this model is accurate for miniMD but not for Jacobi3D. The reason is that, with more cores running at the same time, memory bandwidth becomes the bottleneck for Jacobi3D and the performance does not improve proportionately with the number of cores. The $s_i$ measurements do not capture this effect since the performance of each core is sampled executing alone. This does not capture resource contention, such as memory bandwidth, that occur in the presence of other active cores. However, for a compute intensive code such as miniMD, this problem is not significant and sampling individual cores




(a) Error of Model 1 for Jacobi3D (b) Error of Model 2 for Jacobi3D (c) Error of Model 3 for Jacobi3D (d) Error of Model 4 for Jacobi3D

(e) Error of Model 1 for miniMD (f) Error of Model 2 for miniMD (g) Error of Model 3 for miniMD (h) Error of Model 4 for miniMD

Fig. 6. Distribution of errors of different models for Jacobi3D and miniMD performance. The number of configurations on y-axis is shown in log scale. Model 4 performs very well, with average prediction errors of only $1.6\% and 0.7\%$ across all the configurations for miniMD and Jacobi3D, respectively.

separately gives accurate performance predictions. In general, the disadvantage of Model 1 is that it does not consider the bottlenecks that limit the performance when there are multiple cores running at the same time.

### B. Model 2

The second model tries to avoid the drawbacks of the first model by using a random set of samples that potentially have multiple cores running at the same time. Furthermore, it assumes that the application execution time has a memory component ($T_{mem}$) that does not become faster with more cores.

Model 2 uses the sum of the frequencies as the variable that determines the application's performance for each configuration. It uses $n$ random samples to fit a linear function of the following form:

$$F_c = \sum_{i \in c} f_i$$
$$t_c = \frac{T_{comp}}{F_c} + T_{mem}$$

In this equation, $t_c$ is the predicted execution time of configuration $c$. $T_{comp}$ and $T_{mem}$ are constants that represent the computation and memory time of the application (which are found by function fitting). $F_c$ is the sum of the frequencies of the cores of the configuration.

Figures 6(b) and 6(f) show the error of Model 2 for Jacobi3D and miniMD. The results illustrate that this model is not very accurate in predicting the performance. Similarly, one might fit a linear function of the sum of the frequencies with the following form:

$$S_c = a_1 F_c + a_2 \quad (3)$$

However, we verified that this model does not predict well and has similar accuracy to Model 2. These models do not work because they fail to take into account the number of cores that are executing the parallel application.

### C. Model 3

To have more accurate predictions than Model 2, one might consider second degree curve fitting. In this model, the runtime samples $n$ random configurations as before, but fits a second degree curve to predict the performance:

$$S_c = a_1 (F_c)^2 + a_2 F_c + a_3 \quad (4)$$

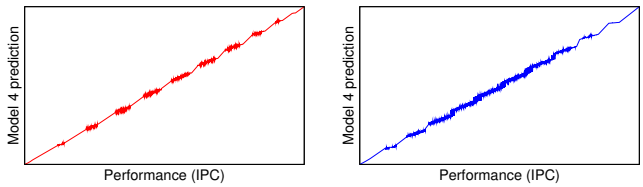
Figures 6(c) and 6(g) show the prediction error of Model 3. There are extremely large error values for some configurations, which make the use of this model impractical. This model suffers from *overfitting*; although the curve is very close to the few sample values, it cannot capture the trend and can be very inaccurate for other configurations.

### D. Model 4

Previous models could not capture various aspects of the configurations simultaneously (such as memory bandwidth limitations and frequency sensitivity). Considering Figures 4 and 5, we observe that both the frequencies and the number of cores influence the performance. Moreover, just adding the frequencies does not capture the effect of additional cores. For example, two 1 GHz cores are not the same as one 2 GHz core. Therefore, an accurate model needs to consider the frequencies as well as the number of cores in each configuration.

To consider both the frequencies and number of cores, Model 4 fits a linear function for each possible number of cores, i.e. one line for configurations with only one core, one line for configurations with two cores, and so on. Therefore, a 12-core chip will have 12 linear functions. A line for $k$-core configurations is a linear function of the sum of frequencies of the $k$ cores of each configuration.

In general, each line needs at least two samples but more samples can make it more accurate by using regression tools to fit the best possible line for all the samples. To keep the number of samples needed low, we use only two samples. Hence, $2n$ samples are needed, which is more than other models but still low enough for the runtime system to collect with negligible overhead. As a heuristic, we choose the configuration with the minimum sum of frequencies and the configuration with the maximum sum of frequencies for sampling. Finding the




(a) Jacobi3D (b) miniMD

Fig. 7. Model 4 predictions as a function of actual (simulated) performance.

configuration with minimum (or maximum) sum of frequencies is easily done by choosing the needed number of cores from the list of cores sorted by frequencies.

The runtime system builds the model as follows.

$\forall k \in (1..n)$:

$K = \{c | c \text{ has } k \text{ cores}\}$, where $c$ is a configuration

$$
\begin{aligned}
c^k_{min} &= c \in K | \sum_{i \in c} f_i \text{ is minimum} \\
c^k_{max} &= c \in K | \sum_{i \in c} f_i \text{ is maximum} \\
F^k_{min} &= \sum_{i \in c^k_{min}} f_i \\
F^k_{max} &= \sum_{i \in c^k_{max}} f_i \\
Y^k_{min} &= \text{performance of } c^k_{min} \\
Y^k_{max} &= \text{performance of } c^k_{max} \\
a^k_1 &= \frac{(Y^k_{max} - Y^k_{min})}{(F^k_{max} - F^k_{min})} \\
a^k_2 &= Y^k_{min} - a_1 F^k_{min}
\end{aligned}
$$

The performance of any configuration with $k$ cores is then predicted by a simple linear formula:

$$S^k_c = a^k_1 F_c + a^k_2 \tag{5}$$

Figures 6(d) and 6(h) show the accuracy of Model 4 for Jacobi3D and miniMD. Compared to other models, the points are closer to the zero error line, meaning that this model is much more accurate.

For each application, Figure 7 illustrates the performance predictions of Model 4 for the chip configurations as a function of their actual performance obtained by simulation. There are some jitters but both of the functions are mostly monotonic. This means that, given two configurations, the model predicts higher performance for the configuration that is actually faster. Therefore, Model 4 is accurate at comparing configurations.

### E. Summary of Performance Models

Figures 8 and 9 compare the accuracy of the discussed models. Model 4 is superior to others in all of the metrics. The average error of Model 4 for miniMD is 1.6%, and it is 0.7% for Jacobi3D. In the worst case, the maximum error of Model 4 is 9.2% for miniMD and 4.3% for Jacobi3D. Thus,

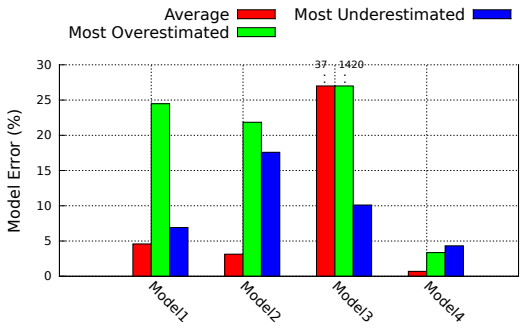



Fig. 8. Prediction accuracy of different models for Jacobi3d. Numbers on top of the Model 3 bars represent the values that are beyond the plotted range.

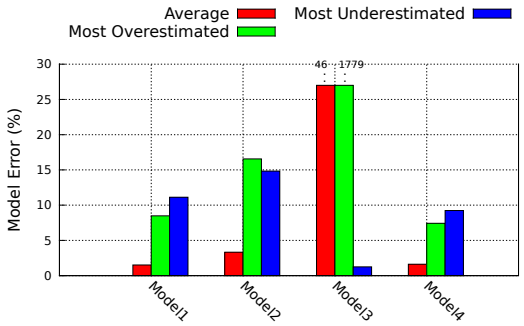



Fig. 9. Prediction accuracy of different models for MiniMD. Numbers on top of some bars represent the values that are beyond the plotted range.

we conclude that Model 4 is sufficiently accurate for predicting the performance of various configurations of a heterogeneous chip.

### F. Modeling Dynamic Power

Model 4 predicts performance accurately but dynamic power also varies with configuration and needs to be predicted. The dynamic power of a processor core can be formulated as follows:

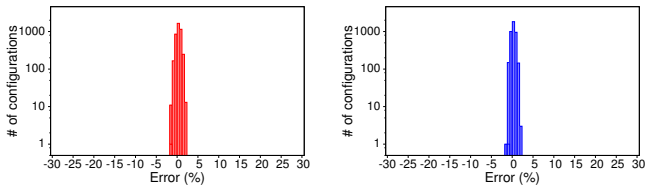
$$D_i = \alpha_i C V^2 f_i \tag{6}$$

In this formula, $f_i$ is the frequency of the core, $V$ is its voltage, $C$ is its capacitance, and $\alpha_i$ is the activity of the core. Except the activity level of the core $\alpha_i$, which varies in different configurations, other parameters are constants. Therefore, $\alpha_i$ needs to be taken into account for accurate dynamic power predictions.

The dynamic power of a configuration is the sum of the dynamic powers of the cores:

$$D^c = \sum_{i \in c} \alpha^c_i C V^2 f_i = C V^2 \sum_{i \in c} \alpha^c_i f_i \tag{7}$$

We strive to adopt our performance model (Model 4) to predict dynamic power due to the following observations. First, dynamic power has similar properties to performance in general. Dynamic power is higher when there are more cores and when the cores have higher frequencies. Second, the activity level of each core is correlated with performance, and hence, correlated with the sum of frequencies. We therefore formulate our dynamic power model as follows:




(a) Jacobi3D (b) miniMD

Fig. 10. Distribution of errors of Model 4 for power consumption prediction.

$$
\begin{aligned}
Z^k_{min} &= \text{ dynamic power of } c^k_{min} \\
Z^k_{max} &= \text{ dynamic power of } c^k_{max} \\
b^k_1 &= \frac{(Z^k_{max} - Z^k_{min})}{(F^k_{max} - F^k_{min})} \\
b^k_2 &= Z^k_{min} - a_1 F^k_{min}
\end{aligned}
$$

The dynamic power of a configuration with $k$ cores is then predicted by a simple linear formula:

$$D^k_c = b^k_1 F_c + b^k_2 \tag{8}$$

Figure 10 illustrates the accuracy of our model for predicting the dynamic power of all of the configurations of a 12-core chip. The errors of the model are all before 2%, which means the accuracy is very high. Furthermore, since the accuracy of the model is higher for dynamic power than performance, one can conclude that predicting dynamic power is easier than performance.

## VI. Model Driven Scheduling

The runtime system should intelligently select the frequency domains of the chip to execute the application intelligently in order to meet the performance, power, and energy demands/constraints. There are various tradeoffs that need to be considered. For example, choosing a configuration with too many cores for a memory-bound application might not improve the performance much, but it can consume excessive power and energy. In addition, a configuration might be fast but have high power consumption.

The number of configurations can be prohibitively large for the runtime system to try exhaustively. For a chip with $n$ frequency domains, there are $2^n - 1$ configurations since each domain can be turned on or off (with at least one domain on). This exponential growth is due to the heterogeneity of the chips. Otherwise, analogous homogeneous chips with $n$ cores have only $n$ distinct configurations. In the previous section, we developed a model to predict the running application's performance on any configuration of the cores of a heterogeneous chip. In this section, we use the model to solve the scheduling problem in the presence of variation, given the performance and power constraints.

### A. Efficient Configuration Space Exploration

Using our performance and power models, the runtime system can evaluate many configurations quickly (with only a few samples), but exhaustive exploration is not always practical. For our example processor chip with 12 frequency domains, the runtime only needs to evaluate the models for $(2^{12} - 1 = 4095)$ configurations. However, future processors will have more frequency domains and the number of configurations increases exponentially with the number of frequency domains. Therefore, exploring all of the configurations in the runtime can be impractical.

We propose the use of integer linear programming (ILP) by the runtime system for finding the best (or very close to the best) configuration given the performance and power constraints. Using ILP (in many cases) needs linear objective functions and constraints. Our performance and power models are linear for configurations with the same number of cores, but the overall functions are not linear. To solve this issue, we setup separate ILP problems for configurations with different number of cores (36 ILP problems for a 36-core chip). Therefore, to maximize performance given a power budget, an ILP problem for a given number of cores ($k$) is formulated as follows :

*Parameters*

$x_i$ : binary variable indicating whether core $i$ is used
$F = \sum_{i \in \text{all cores}} x_i f_i$
$\mathcal{P}$ : power budget of the chip
$p^s_i$ : static power of core $i$

*Objective function*

Maximize performance:

$$S^k_c = a^k_1 F + a^k_2 \tag{9}$$

*Constraints*

Only configurations with $k$ cores:

$$\sum_{i \in \text{all cores}} x_i = k \tag{10}$$

Cap total power according to budget:

$$\sum_{i \in \text{all cores}} x_i p^s_i + b^k_1 F + b^k_2 \leq \mathcal{P} \tag{11}$$

After solving these ILPs, the runtime system needs to compare the results and choose the best one. Note that in our formulation, we considered performance as the main objective metric that needs to be maximized given a power budget. One can similarly minimize power given performance constraints. In this case, the roles of Equations 9 and 11 are switched, and performance becomes a constraint, while power becomes the objective function.

Note that if one wants to minimize energy without any performance constraints, our ILP framework in this form cannot be used since the objective function will not be linear anymore. Energy minimization without performance constraints is left for future work.

### B. Incorporating DVFS

Previous studies suggest Dynamic Voltage and Frequency Scaling (DVFS) for energy-efficient computing for some cases, such as for memory bound applications. Our framework can incorporate DVFS as well. We only need more binary variables and constraints that indicate at which DVFS level each core

should operate. The constraints make sure that the solver does not choose illegitimate conditions, such as a core operating at two DVFS levels simultaneously.

*Parameters*

$x_{ij}$ : binary variable indicating core $i$ at DVFS level $j$ is used or not

$F = \sum_{i \in \text{ all cores}} \sum_{j \in \text{ all DVFS levels}} x_{ij} f_{ij}$

$\mathcal{P}$ : power budget of the chip

$p_i^s$ : static power of core $i$

*Objective function*

Maximize performance:

$$S_c^k = a_1^k F + a_2^k \tag{12}$$

*Constraints*

Only configurations with $k$ cores:

$$\sum_{i \in \text{all cores}} \sum_{j \in \text{ all DVFS levels}} x_{ij} = k \tag{13}$$

Only one DVFS level is selected per core:

$$\sum_{j \in \text{ all DVFS levels}} x_{ij} \leq 1, \forall i \in [1..n] \tag{14}$$

Cap total power according to budget:

$$\sum_{i \in \text{all cores}} \sum_{j \in \text{ all DVFS levels}} x_{ij} p_i^s + b_1^k F + b_2^k \leq \mathcal{P} \tag{15}$$

Further study and evaluation of DVFS in our framework is left for future work. For the chips we evaluate, the static power is high and therefore, operating more cores at lower frequencies is not very well justified, even for memory-bound applications. It is most often more beneficial to turn as many cores off as possible and operate the rest at full speed. However, DVFS might be beneficial in other cases.

### C. Incorporating Communication Performance

So far we have assumed that on-chip communication performance of different configurations does not differ significantly, but this can be incorporated in our framework as well. Depending on several factors, such as the application's communication pattern, network design, and the routing algorithm, a communication model needs to be included in the ILP framework.

As an example, we consider a case where the application has a heavy all-to-all communication pattern, the network topology is a 2D mesh, and a minimal adaptive routing algorithm is used. In this case, one heuristic is to use the cores that are farther apart on the chip. This lets the application use more of the network links and have less congestion.

To handle this case in our framework, communication performance needs to be incorporated in the objective function as a linear expression. We assign a communication score ($e_i$) to each core on the chip. Figure 11 shows an example of possible core assignments. The cores closer to the corners and sides are assigned higher scores because they can potentially use more of network's links to send and receive messages. Using this model, we extend the object function of the ILP as follows:

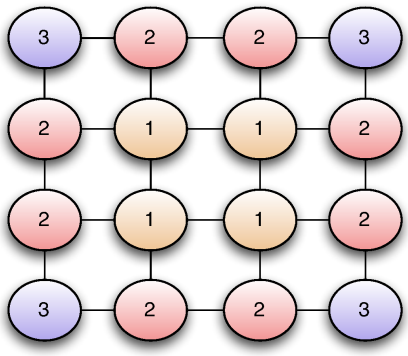



Fig. 11. Assigning communication scores to different cores on a chip.

$$S_c^k = \mathcal{E}_1(a_1^k F + a_2^k) + \mathcal{E}_2(\sum_{i \in \text{all cores}} x_i e_i) \tag{16}$$

In this equation, $\mathcal{E}_1$ and $\mathcal{E}_2$ are constants that give weights to computation and communication performance depending on the application. They can be tuned offline, or online by the runtime system using measurements.

Further study and evaluation of communication models is beyond the scope of this study and is left for future work.

### D. Adapting to Application Phases

Some HPC applications have multiple phases with different characteristics. Our previous study demonstrates how application phases can be recognized and exploited effectively in the runtime system [22]. One could use our scheduling framework for different phases separately as well. In this case, the runtime needs to migrate the tasks and turn cores on and off when changing the configurations. The overheads of this reconfiguration and task migration should also be considered. Extensive study of of this feature is left for future work.

## VII. Evaluation

In this section, we evaluate our ILP-based scheduling framework by comparing it against two heuristic based scheduling algorithms. The goal of these algorithms is to maximize the performance of a parallel application under a given power budget for the chip. Evaluation and analysis of cases that consider other aspects such as DVFS and communication constraints is left for future work.

The first heuristic, called the Min heuristic, chooses as many cores as possible with the lowest frequencies as possible such that the selected configuration remains within the given power budget and the performance is maximized. Although this heuristic does not consider static power for core selection, most often the slower cores also have lower static power. A previous study confirms this correlation using silicon measurements [14]. Hence, this heuristic strives to choose as many low power cores as possible. The second heuristic, called the Max heuristic, chooses as many of the highest frequency cores as

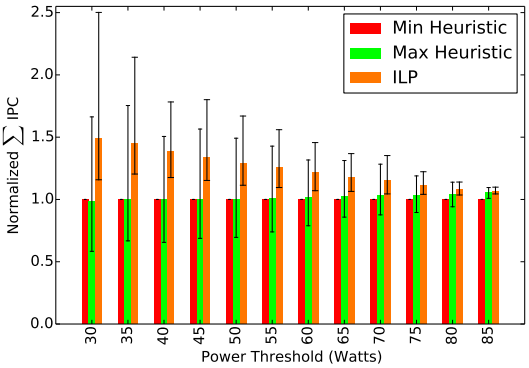



Fig. 12. Comparison of our ILP-based scheduling approach to simple heuristics for miniMD.

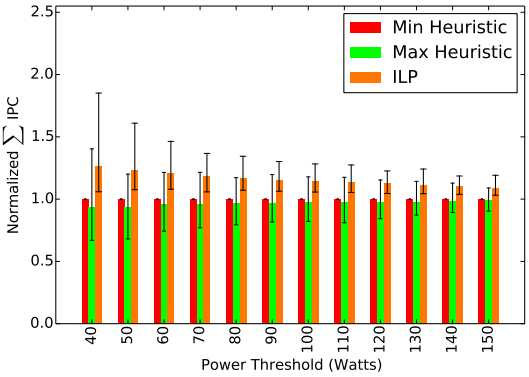



Fig. 13. Comparison of our ILP-based scheduling approach to simple heuristics for Jacobi3D.

possible, while staying within the power budget. In contrast to the first one, this heuristic therefore most often choose the highest power consuming cores and probably fewer cores.

Figures 12 and 13 compare our ILP framework to the heuristics for miniMD and Jacobi3D applications. The bars represent the average benefits of the different schemes across 100 chips, and are normalized to the MIN heuristic. The vertical lines on the ILP and the MAX heuristic bars illustrate the maximum and minimum benefit obtained with the corresponding approach across all the chips for the given power budget. MiniMD's power caps are lower since it consumes less power in general compared to Jacobi3D. One can conclude that MIN and MAX heuristics have similar results, while intelligent ILP scheduling can be considerably better. ILP finds configurations that are up to 1.85 times faster for Jacobi3D and up to 2.5 times faster for miniMD. On average, for all the cases we examined (various power caps for 100 chips), ILP scheduling is 25% faster for miniMD and 16% faster for Jacobi3D as compared to the MIN heuristic.

The results indicate that the benefit of intelligent ILP scheduling is more with lower power budgets. This is because with lower power caps (that still allow multiple cores to be selected), there are more choices. On the other hand, if the power budget allows many of the cores to be selected, the different configurations chosen by different schemes have many overlapping cores and are similar. If there is enough power, all cores will be chosen by all of them and there is no other choice for the ILP. One can also see that the benefit of intelligent ILP scheduling can be much higher for compute-bound applications such as miniMD, since they are more sensitive to the frequencies of the chosen cores (as illustrated in Figure 5).

Table II presents an example scheduling case, demonstrating that the ILP is choosing cores intelligently and its choices are different from the heuristics. In this case, the power cap is 40 W and the schemes strive to find the highest performing configuration for a 36-core chip running miniMD. The cores are numbered by their frequencies in increasing order. One can conclude that ILP is choosing the cores intelligently resulting in higher performance within the power budget.

TABLE II. EXAMPLE SCHEDULING CASE COMPARING VARIOUS SCHEMES

| Scheme | Selected Cores | Relative Performance |
|---|---|---|
| MIN Heuristic | $0, 1, 2, 3, 4, 5, 6$ | 1 |
| MAX Heuristic | $31, 32, 33, 34, 35$ | 0.97 |
| ILP | $7, 13, 14, 17, 21, 24, 31$ | 1.19 |

Integer program optimization is an NP-hard problem and can be computationally very expensive in the worst case, but it is very fast for our scheduling framework. Our ILP solver took only 37 ms on average across all the chips and power budgets we examined. The underlying simplex algorithm performed only 217 iterations on average across all the nodes of the branch-and-bound tree for the corresponding ILP. This is negligible compared to the typical execution time of HPC applications, which can run for up to several days in many cases. We used the state-of-the-art Gurobi [23] optimizer for solving the ILPs.

## VIII. RELATED WORK

Process variation has been explored from the manufacturing and circuit perspective [1], [24], [25], [26]. Dighe et al. [14] measured the process variation of Intel TeraFLOPS experimental chips and studied the optimal operating point of different applications. We use the valuable insights of these studies for our assumptions about the properties of the processor chips.

The proposed Exascale architectures such as Runnemede [8] and Echelon [9] consider hundreds of cores on a chip, which are arranged in many frequency domains (with power gating). Furthermore, those architectures are over-provisioned, and need extensive power management to stay within the power budget [8]. However, previous studies do not provide scheduling algorithms that meet these constraints. A framework like ours seems essential for such architectures, as it provides the necessary runtime component to perform scheduling and power management depending on the application characteristics.

Previous studies have explored scheduling in the presence of process variation for multiprogrammed environments [3], [6], [7]. For example, Winter et al. [7] use a Hungarian optimization algorithm to assign different sequential programs to the best matching cores. Karpuzcu et al. [3] use heuristics to map different multithread applications to the many-core

chip's cores. However, in HPC environments, a single parallel application almost always runs on the whole chip, and the previous algorithms cannot be used. Since all the threads of the application usually have similar behavior, switching threads among cores is not very useful. In addition, previous [7] work either ignored the interference of different cores (e.g. memory bandwidth contention), or assumed that load balancing is not possible and the parallel application's speed is determined by the slowest core [3]. To the best of our knowledge, this is the first study to propose a variation-aware scheduling approach for HPC systems.

In their studies on overprovisioned HPC data centers, Sarood et al. [27] and Patki et al. [28], have proposed frameworks that distribute power to nodes such that the performance is maximized under a given power budget for the data center. However, they do not propose how the chosen power budget for each node will be used for achieving maximum performance from the node, which is the focus of this work. The proposed framework in this paper can be combined with their frameworks for maximizing the performance of future HPC data centers with a fixed power budget.

## IX. Conclusion and Future Work

Process variation causes performance and power heterogeneity among various cores of a many-core chip. We studied various performance and power models for such chips. Based on the models, we developed a novel scheduling framework that uses integer linear programming (ILP) to explore the configuration space efficiently. This enables intelligent HPC runtime systems to enforce different performance and power constraints and schedule work units effectively. We also illustrated how different constraints, such as communication performance, can be enforced in our framework.

There are many future research directions based on our study. More in depth evaluation and analysis of different HPC applications on heterogeneous chips needs to be performed. Furthermore, the use of our framework to find different configurations for different application phases needs to be studied. In addition, we assumed that tasks of the application have the same amount of work but this is not always the case. Variation-aware load balancing of complex applications that have different workloads in different tasks needs further exploration. Moreover, the development and evaluation of new scheduling constraints dependent on the application and system characteristics requires much research.

## References


[1] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proc. of DAC*, 2003.

[2] J. Torrellas, "Extreme-scale computer architecture: Energy efficiency from the ground up," in *Proc. of DATE*, 2014.

[3] U. R. Karpuzcu, A. Sinkar, N. S. Kim, and J. Torrellas, "EnergySmart: Toward energy-efficient manycores for Near-Threshold Computing," in *Proc. of HPCA*, 2013.

[4] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT press, 1999.

[5] B. Lewis and D. J. Berg, *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., 1998.

[6] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *Proc. of ISCA*, 2008.

[7] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *Proc. of PACT*, 2010.

[8] N. Carter et al., "Runnemede: An architecture for ubiquitous high-performance computing," in *Proc. of HPCA*, Feb 2013, pp. 198–209.

[9] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[10] J. L. Gustafson, "Reevaluating Amdahl's Law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988.

[11] A. B. Downey, "A parallel workload model and its implications for processor allocation," *Cluster Computing*, vol. 1, no. 1, pp. 133–145, 1998.

[12] A. Lastovetsky and R. Reddy, "On performance analysis of heterogeneous parallel algorithms," *Parallel Computing*, vol. 30, no. 11, pp. 1195 – 1216, 2004.

[13] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Proc. of ISCA*, 2012.

[14] S. Dighe et al., "Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core TeraFLOPS processor," *Solid-State Circuits, IEEE Journal of*, vol. 46, no. 1, pp. 184–193, Jan 2011.

[15] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. of SC*, 2011.

[16] U. R. Karpuzcu, K. B. Kolluru, N. S. Kim, and J. Torrellas, "VARIUS-NTV: A Microarchitectural Model to Capture the Increased Sensitivity of Manycores to Process Variations at Near-Threshold Voltages," in *Proc. of DSN*, 2012.

[17] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. of MICRO*, 2009.

[18] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," in *Proc. of SC*, 2014.

[19] H. Menon and L. Kalé, "A distributed dynamic load balancer for iterative applications," in *Proc. of SC*, 2013.

[20] J. Kang and S. Park, "Algorithms for the variable sized bin packing problem," *European Journal of Operational Research*, vol. 147, no. 2, pp. 365 – 372, 2003.

[21] Y. Sun, G. Zheng, C. M. E. J. Bohm, T. Jones, L. V. Kalé, and J. C. Phillips, "Optimizing fine-grained communication in a biomolecular simulation application on Cray XK6," in *Proc. of SC*, 2012.

[22] E. Totoni, J. Torrellas, and L. V. Kale, "Using an adaptive HPC runtime system to reconfigure the cache hierarchy," in *Proc. of SC*, 2014.

[23] "Gurobi Optimization Inc. Software, 2014," http://www.gurobi.com/.

[24] K. Kuhn, M. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, S. Ma, A. Maheshwari, and S. Mudanai, "Process technology variation," *Electron Devices, IEEE Transactions on*, vol. 58, no. 8, pp. 2197–2208, Aug 2011.

[25] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, Nov 2005.

[26] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (NTV) design: Opportunities and challenges," in *Proc. of DAC*, 2012.

[27] O. Sarood, A. Langer, A. Gupta, and L. V. Kale, "Maximizing Throughput of a Data Center Under a Strict Power Budget," in *Proc. of SC*, 2014.

[28] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "Exploring Hardware Overprovisioning in Power-constrained, High Performance Computing," in *Proc. of ICS*, 2013.