THE UNIVERSITY OF CHICAGO

INCENTIVIZING FLEXIBILITY AND COOPERATION IN COMPUTER SYSTEMS USING FEEDBACK MECHANISMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

MUHAMMAD HUSNI SANTRIAJI

CHICAGO, ILLINOIS
AUGUST 2022

Dedication Text

First, I want to thank God for making me the luckiest person on Earth. He gives me ways to solve problems out of nowhere that I would never imagine I could solve by myself.

I thank my adviser, Hank Hoffmann, for believing in me and having unlimited patience in listening to me during our meeting.

I want to thank my wife, Dwi Kartika Sari, for her unlimited support during my study on another side of the Earth.

I also want to thank my family, My Mother, and Father, who make this Ph.D. journey possible.

My thanks to Haryadi Gunawi and Yohannes Surya, who allowed me to do world-class research.

I also want to thank my friends, lab mates, committee, Uchicago staff, and others who support me in finishing my thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Many computer systems and applications, from small embedded systems to large datacenters have deployment requirements. Meeting these requirements in a dynamic environment is challenging and requires flexibility from the application and the system.

Flexibility is the ability to trade-off the value of one measure space by adjusting the value of another measure space. For example, both DNN and approximate computing applications can reduce their runtime latency by sacrificing their output accuracy. However, managing this flexibility is difficult. Prior approaches do not incentivize flexibility and cooperation. In the single stakeholder scenario where applications come from one stakeholder, they do not cooperate with the application and system knobs which makes the deployment inefficient in terms of energy, output accuracy, and performance. In the multistakeholder scenario, they do not incentivize the flexibility of applications which make flexible application produce higher output error.

Our first contribution is GRAPE and MERLOT, a hardware feedback mechanism to meet latency requirements while reducing energy usage. GRAPE is a hardware control system for GPU that provides a soft guarantee to meet the performance requirements. Meanwhile, MERLOT is a real-time hardware scheduler that provides a hard real-time guarantee.

Our second contribution is ALERT, runtime management for Deep Neural Networks that decrease output error or energy usage while meeting latency requirements. ALERT achieves cooperation between application and system by coordinating the flexibility offered from both. ALERT uses a probabilistic feedback mechanism that predicts the energy, performance, and output accuracy of the applications during the runtime.

The third contribution is SIM, runtime management that incentivizes the flexibility of applications in the multistakeholder scenario. Prior approaches inadvertently disincentivize flexibility by forcing flexible applications to adapt to meet their deployment requirement, thus encouraging greedy behavior where every stakeholder deploys inflexible approaches that

consume as many resources as possible. SIM instead only enforces the adaptation for the application that holds the most resources. In each iteration, on behalf of the applications, SIM would make an application to a configuration that minimizes their output error such that the resource usage is either less than the application that holds most resources or higher as long as there are enough slack resources. SIM incentivize the deployment of flexible application by giving opportunity for all of the applications to fight for the slack resources by being flexible.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Many computer systems and applications, from small embedded systems to large datacenters have deployment requirements. For example, neural networks deployed in remote sensing infrastructure needs to minimize their output error while meeting latency deadlines and energy budgets. Meanwhile, a software application deployed in a large data center needs to minimize energy and meet Quality of Service (QoS) requirements. Meeting these deployment requirements in a dynamic environment is challenging and requires flexibility from the application and the system.

Flexibility is the ability to trade-off the value of one measure space by adjusting the value of another measure space. For example, both neural networks and approximate computing applications can reduce their runtime latency by sacrificing their output accuracy. Meanwhile, the computer hardware can run in different DVFS configurations to adjust its performance and energy usage. However, managing this flexibility is difficult. Prior approaches do not incentivize flexibility and cooperation. In the single stakeholder scenario where applications come from one stakeholder, they do not cooperate with the application and system knobs which makes the deployment inefficient in terms of energy, output accuracy, and performance. In the multistakeholder scenario, they do not incentivize the flexibility of applications which make flexible application produce higher output error.

In this dissertation, we incentivize the flexibility and cooperation in computer systems using feedback mechanisms. We focus on addressing these questions:

- How do we incentivize the hardware flexibility for interactive GPU applications?
- How do we incentivize the flexibility and cooperation of both application and system knobs for DNN deployment?

- How do we incentivize the flexibility and cooperation of anytime algorithms in the multistakeholder deployment scenario?

1.2 Contributions

This dissertation addresses the flexible management of applications and systems. We answer the first question by proposing GRAPE and MERLOT, a hardware feedback mechanism to meet latency requirements while reducing energy usage. We answer the second question by proposing ALERT, runtime management for Deep Neural Networks that decrease output error or energy usage while meeting latency requirements. We answer the third question by proposing SIM, runtime management that incentivizes the flexibility of applications in the multistakeholder scenario. We describe the overview of contributions below.

1.2.1 GRAPE

Many applications have performance requirements (e.g., real-time deadlines or quality-of-service goals) and we can save tremendous energy by tailoring resource usage so the application just meets its performance using the minimal resources. This problem is a classic constrained optimization: the performance goal is the constraint and energy consumption is the objective to be optimized. While several existing hardware approaches solve unconstrained optimizations (i.e., maximizing performance or minimizing energy), we are not aware of a hardware approach that minimizes GPU energy under an externally defined performance constraint. Therefore, we propose GRAPE, a hardware control system for GPUs that coordinates core usage, wavefront/warp action, core speed, and memory speed to deliver user-specified performance while minimizing energy. We implement GRAPE in VHDL (to demonstrate feasibility) and as an extension to GPGPU-Sim (for performance and power measurement).

1.2.2 MERLOT

Correct functioning of embedded systems requires strict timing guarantees. Traditionally, enforcing timing guarantees is the operating system's responsibility. The OS scheduler assigns sufficient resources to an application task to ensure it meets its deadline. Meeting hard real-time deadlines requires the scheduler to be conservative and allocate for the worst case timing; when behavior is not worst case, extra resources are allocated and energy is wasted. Some software schedulers reduce this energy waste by recognizing when an application is ahead of a worst case schedule and reclaiming unneeded resources, but they are fundamentally limited by (1) overhead and (2) a lack of visibility into low-level resource usage. Therefore, this paper advocates hardware assistance for energy management of hard real-time tasks. Specifically, we propose MERLOT, a hardware-based resource manager for GPUs that enforces software-specified timing guarantees with minimal energy. We implement MERLOT in VHDL and find that its performance, power, and area overheads are minuscule. We implement MERLOT in GPGPU-Sim to test timing and energy consumption and compare to two software-only approaches: one that always allocates for worst case timing and an intelligent approach that reduces resource usage when it recognizes better than worst case behavior.

1.2.3 ALERT

An increasing number of software applications incorporate runtime Deep Neural Networks (DNNs) to process sensor data and return inference results to humans. Effective deployment of DNNs in these interactive scenarios requires meeting latency and accuracy constraints while minimizing energy, a problem exacerbated by common system dynamics.

Prior approaches handle dynamics through either (1) system-oblivious DNN adaptation, which adjusts DNN latency/accuracy tradeoffs, or (2) application-oblivious system adaptation, which adjusts resources to change latency/energy tradeoffs. In contrast, this paper improves on the state-of-the-art by coordinating application- and system-level adaptation. ALERT, our runtime scheduler, uses a probabilistic model to detect environmental volatility and then simultaneously select both a DNN and a system resource configuration to meet latency, accuracy, and energy constraints. We evaluate ALERT on CPU and GPU platforms for image and speech tasks in dynamic environments.

1.2.4 SIM

Recent work has proposed shared sensing infrastructure, multi-tenant embedded systems that different scientists can apply to deploy neural networks to process information collected from a third-party sensor. Examples include the Array of Things in Chicago and the National Ecological Observatory. Scheduling on these devices is difficult. They want to accommodate as much science as possible while keeping the neural network accuracy as high as possible. They are all subject to dynamic workload changes (as networks from different scientists enter and exit the system). Prior work has proposed anytime networks that flexibly operate across a wide range of accuracy/latency tradeoffs to minimize neural network error in dynamic environments. These anytime networks achieve great results when deployed by a single stakeholder (e.g., in a traditional embedded system); however, we find that anytime networks by themselves are a poor solution to the problem of shared sensing infrastructure as the networks are deployed by different stakeholders (i.e., the different scientists). Specifically, we find that traditional multi-stakeholder schedulers and traditional embedded scheduling based on worst-case execution time sacrifice accuracy compared to an optimal schedule. Perhaps worse, we find that applying prior approaches to SSI disincentivizes the deployment of flexible, anytime networks and encourages greedy behavior where every stakeholder deploys networks that consume as many resources as possible. In this paper, we identify this problem and propose a scheduler that incentivizes Anytime networks by rewording stakeholders that deploy such flexible networks and detecting and punishing bad actors that are inflexible and consume the most resources.

CHAPTER 2

GRAPE: MINIMIZING ENERGY FOR GPU APPLICATIONS WITH PERFORMANCE REQUIREMENTS

2.1 Overview

Energy consumption is a first order concern for computing systems, from mobile devices (where it defines battery life) to supercomputers (where it determines operating costs). At the same time, ever-increasing performance demands have led to the adoption of GPU-based acceleration in a wide range of computing platforms. While GPUs deliver tremendous computational throughput, they consume a significant portion of total system energy. Therefore, this chapter studies hardware support for GPU energy reduction when executing applications with performance requirements.

Such applications are not required to achieve the best possible performance, but deliver results with predictable timing — often expressed as a latency or quality-of-service goal. Examples exist in mobile platforms — including video, media capture, and display where the system interacts with a user ([114]). At the other end of the computing spectrum, future supercomputer workloads will include interactive simulations and data analysis applications ([111, 100]). In these cases, applications should not run as fast as possible, but meet their performance requirements with minimal energy.

Many prior approaches manage resources for energy reduction. Some consider a single resource only, eg. CPU ([79]) or memory ([27]) frequency. Others coordinate multiple resources for greater energy savings ([44, 92, 53, 30, 113]). Finally, some approaches incorporate application-level knowledge (e.g., frame rates) to tailor resource usage to an application's frame-based performance requirements [97, 138]. While these approaches combine multi-component management with domain knowledge, they do so in software. We are not aware of a hardware approach that manages multiple components to meet performance requirements while minimizing energy.

A hardware resource management system has the potential to both remove the optimization burden from software and react more quickly than software can. Of course, providing a hardware solution presents several challenges:

- **Overhead:** Significant area, time, or power overhead will diminish any potential gains.
- **Unknown Applications:** To support different applications; hardware management should (1) rapidly detect applications' response to different resources and (2) react when this response changes (eg. it transitions from memory bound to compute bound).

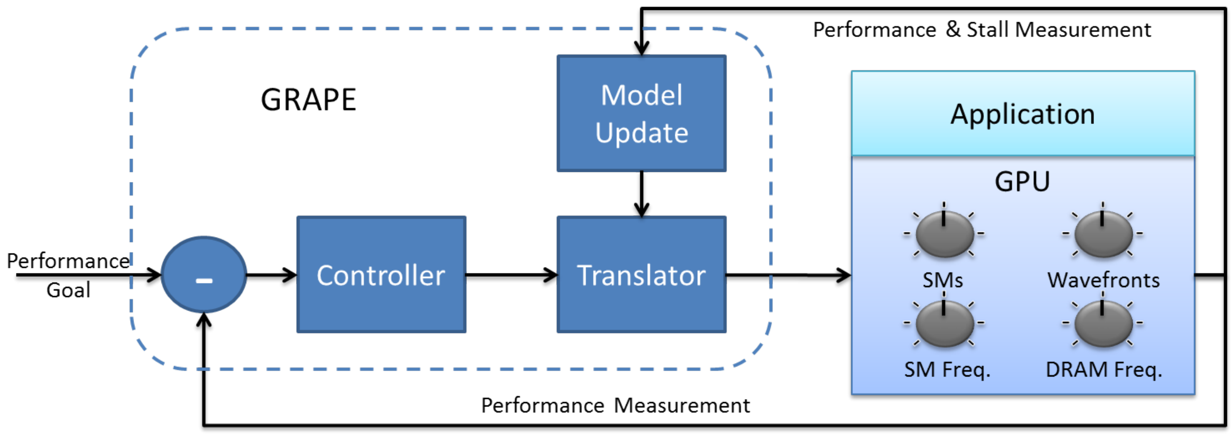


Figure 2.1: GRAPE Control Diagram

To provide hardware resource management, we introduce GRAPE (GPU Resource Adaptation for Performance and Energy). GRAPE is a hardware control system for GPUs designed to meet user-specified performance while minimizing energy through management of (1) streaming multiprocessors (SMs), (2) wavefronts/warps, (3) SM speed, and (4) DRAM speed, while keeping overhead low. Figure 2.2 illustrates the GRAPE controller's block diagram. The application provides a performance goal in the form of a target computation rate. This target is compared to the current performance and the difference is passed to a *controller*. The controller computes a signal indicating how much to speedup the application. This speedup signal is passed to a *translator*, which converts speedup into specific allocations of SMs, wavefront/warps, SM speed, and DRAM speed that deliver the controller-specified speedup with minimal energy. Each resource is adjusted and the application executes with the new resource configuration. GRAPE then observes both the stall behavior and the new performance of the application. The stall behavior will be used to update translation on the next iteration, while the performance is fed back into the controller and the process begins again. Compared to prior work, GRAPE provides three innovations:

- The domain knowledge comes at runtime in the form of a performance requirement. For this paper, the desired performance is expressed as instructions per second. GRAPE's design, however, is independent of any one metric; eg. it could be trivially modified to support floating point rate. GRAPE's general interface supports frame-based applications as well as potential future interactive applications.

- All management is performed in hardware. Whereas prior approaches for constrained optimization in GPUs require software support, GRAPE's hardware solution meets performance goals with minimal energy.

- GRAPE's control theoretic design provides some formal guarantees about its dynamic behavior, including guaranteed convergence to the desired performance and bounded convergence time. These guarantees make are appropriate for meeting soft real-time requirements.

We integrate GRAPE into GPGPU-Sim v3.2.2 ([7]) and GPUWattch ([80]) and then it using 17 benchmarks drawn from Rodinia ([20] and Parboil [118]). We compare to the strategy of *racing-to-idle*; ie. allocating all resources and transitioning to a low-power idle state when a task completes. We also implement GRAPE in VHDL to demonstrate its feasibility. The evaluation shows that GRAPE provides:

- **Low Overhead:** We synthesize the VHDL for an FPGA using Quartus to demonstrate feasibility and provide a rough estimate of overhead. The PowerPlay Early Power

Estimator shows that GRAPE needs 0.478 Watts to operate.

- **Performance Predictability:** Across a range of different targets (from 25% to 100% of maximum achievable performance), GRAPE meets the goal with only 0.75% average error.

- **Energy Efficiency:** At low performance targets, GRAPE consumes only 74% of the energy of race-to-idle. At higher performance targets, the energy savings diminishes; however, even at maximum performance GRAPE reduces energy consumption by 9.02% compared to allocating all resources.

- **Peak Power Reduction:** At low performance targets, peak power is only 40.29% of race-to-idle. At maximum performance, peak power is 87.48% of race-to-idle.

- **Competitiveness with Prior Work:** GRAPE's knowledge of performance requirements allows it to save substantial energy over Equalizer ([113]), a prior approach that is not aware of user-defined performance goals.

GRAPE is for applications with performance constraints; however, its low overhead allows it to be incorporated into many GPU designs. Overall this paper makes the following contributions:

- Developing a hardware control framework that adapts resource usage to meet application performance requirements with minimal energy.

- Evaluating the approach empirically.

- Releasing the code (both simulation and VHDL) as open source so others can expand or evaluate it [1].

1. Available at: https://github.com/grapemicro/GRAPE.git

To the best of our knowledge, GRAPE is the first approach to propose a hardware solution for reducing GPU energy while meeting user-defined performance goals.

2.2 Design and Implementation

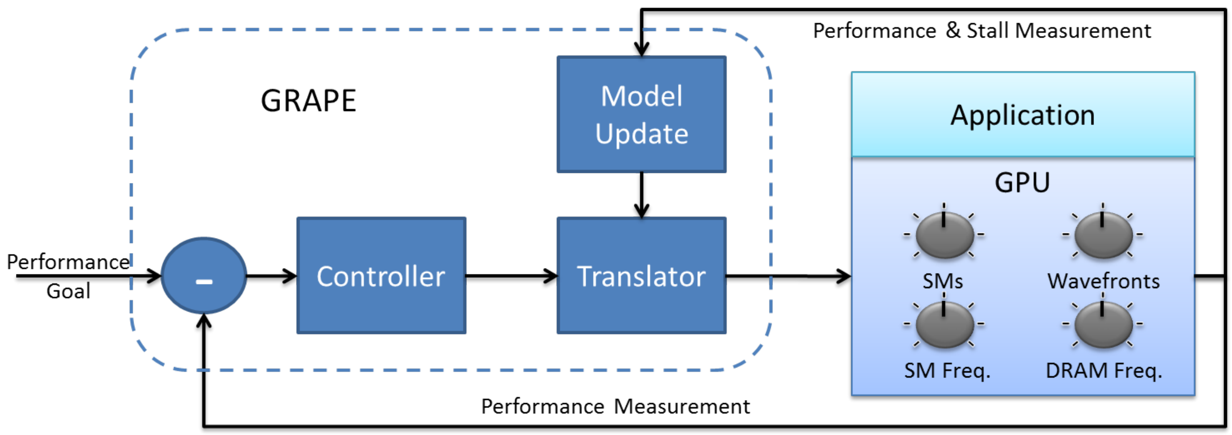


Figure 2.2: GRAPE Control Diagram

GRAPE is a hardware control system for GPUs designed to meet user-specified performance while minimizing energy through management of (1) streaming multiprocessors (SMs), (2) wavefronts/warps, (3) SM speed, and (4) DRAM speed, while keeping overhead low. Figure 2.2 illustrates the GRAPE controller's block diagram. The application provides a performance goal in the form of a target computation rate. This target is compared to the current performance and the difference is passed to a *controller*. The controller computes a signal indicating how much to speedup the application. This speedup signal is passed to a *translator*, which converts speedup into specific allocations of SMs, wavefront/warps, SM speed, and DRAM speed that deliver the controller-specified speedup with minimal energy. Each resource is adjusted and the application executes with the new resource configuration. GRAPE then observes both the stall behavior and the new performance of the application. The stall behavior will be used to update translation on the next iteration, while the performance is fed back into the controller and the process begins again.

We detail each GRAPE module in turn. For each of the three modules we give an intuitive overview and then formally specify its behavior in the form of equations and algorithms. The final subsection discusses GRAPE's hardware implementation. Table 5.1 summarizes the notation used throughout this section.

Table 2.1: Notation used in the paper.

Symbol	**Meaning**
Controller	
g_{user}	performance goal set by the user
t	time index
e	performance error
g	internal performance goal set by controller
α	control correction constant
$\hat{w}$	current workload
x	inverse workload
$\hat{x}$	a posteriori estimate of x
$\hat{x}^{-}$	a priori estimate of x
p	a posteriori performance variance estimate
p^{-}	a priori performance variance estimate
k	Kalman filter gain
h	current performance
s	general control speedup signal
Translator	
$cost$	power cost
sm	status of SM in GPU
nSM	number of SMs
nW	number of wavefronts
$DRAM_{stall}$	stall from SM to DRAM
SM_{stall}	stall in SM pipeline due to memory request
$DRAM_{threshold}$	stall threshold for DRAM
$SM_{threshold}$	stall threshold for SM
M	ordered set of memory configurations
$MEMindex$	index in M; ie. a specific configuration
$s_{MEMindex}$	speedup value of memory configuration
$c_{MEMindex}$	power cost of memory configuration
$f_{MEMindex}$	memory frequency of configuration
$SMindex$	SM configuration
$s_{SMindex}$	speedup of SM configurations
$c_{SMindex}$	power cost of SM configurations
$f_{SMindex}$	SM frequency of configuration
f_{max}	maximum SM frequency available
$cost_{temp}$	temporary cost for finding selection
$cost_{min}$	least cost for finding selection
Model Update	
$highbound_{MEMindex}$	upper bound on speedup for mem. config.
$lowbound_{MEMindex}$	lower bound on speedup for mem. config.
β	learning rate

2.2.1 The Controller

The controller determines how much to speed up the application at time t. It does so by computing the error between the desired behavior and the measured behavior. The controller accounts for both immediate behavior — eg. application performance is too low at this iteration — and long-term behavior — eg. the application initially ran too slowly and now needs extra speed to meet the overall performance target. Additionally, the controller tailors response to individual applications — or phases in applications — by continually estimating the application workload; ie. the application's instruction latency with minimal resources. GRAPE's controller has several strengths: (1) it uses a simple feedback model, which is easy to calculate in hardware at runtime, (2) it is provably convergent to the desired behavior, and (3) it is robust in the face of noise and model errors [81].

It is assumed that the controller executes at discrete time steps t, and the controller executes Algorithm 1 at each step. The algorithm has four inputs: (1) the user-specified performance goal g_{user}, (2) the current measured performance $h(t)$, (3) the number of instructions completed so far I, (4) and the elapsed time executing the application ℓ. The controller first sets an internal goal $g(t)$ allows GRAPE to correct for any errors it may have made previously — if GRAPE is initially too slow, it will speed up its own internal goals. GRAPE then computes the difference between its internal goal and the measured performance at the current time (Algorithm 1, line 2).

Phase Estimation. Next, GRAPE estimates the application workload at the current time $\hat{w}(t)$. The workload is a key parameter that tunes control response to the current application; it represents the number of instructions that the application would retire in a time step if allocated the minimal resources. As the application goes through phases, this value might change, so it is continually updated as part of the control action. Lines 4-9 of Algorithm 1 estimate workload using a standard one-dimensional Kalman filter formulation [128]. GRAPE uses a Kalman filter because it is specifically designed to provide accurate estimations in noise and it is exponentially convergent, meaning that the time it takes to converge to the correct estimation is proportional to the logarithm of the error between the initial estimate and the true value [16].

Algorithm 1 The Controller

```
Require: $g_{user}$                                ▷ application specified performance goal
Require: $h(t)$                                    ▷ instructions per second at time t
Require: $I$                                       ▷ completed instructions
Require: $\ell$                                    ▷ elapsed time
 1: procedure The Controller
        ▷ Update local goal based on global progress
 2:     $g(t) = g_{user} - \alpha(I/\ell - g_{user})$
        ▷ Error between new goal and current performance
 3:     $e(t) = g(t) - h(t)$
        ▷ Estimate application workload (i.e., phases)
 4:     $\hat{x}^{-}(t) = \hat{x}^{-}(t-1)$
 5:     $p^{-}(t) = p(t-1) + q(t)$
 6:     $k(t) = \frac{p^{-}(t)s(t-1)}{[s(t)]^2 p^{-}(t)+o}$
 7:     $\hat{x}(t) = \hat{x}^{-}(t) + k(t)[h(t) - s(t-1)\hat{x}^{-}(t)]$
 8:     $p(t) = [1 - k(t)s(t-1)]p^{-}(t)$
 9:     $\hat{w}(t) = \frac{1}{\hat{x}(t)}$
        ▷ Compute speedup
10:     $s(t) = s(t-1) + \hat{w}(t) \cdot e(t)$
11: return $s(t)$                                  ▷ speedup to apply at current time
12: end procedure
```

The last step in each control iteration is to use the error (from Algorithm 1 line 2) and the workload estimate (from line 9) to compute the speedup (line 10-11). The speedup is computed according to the Proportional Integral (PI) control law using standard techniques [41]. This speedup signal is then passed to the translator.

Algorithm 1 is constant time and a small number of instructions. It can easily be implemented in fixed-point arithmetic for hardware. Despite this simplicity, control adapts in several ways. First, by keeping an internal goal, separate from the externally specified goal, the controller can adapt to both errors it generates and to phases in application behavior that radically change the performance. Second, the Kalman filter provides fine grain customization of control by adapting to the current workload. Note that as workload increases, the controller's output speedup will also increase (line 10), which is consistent with intuition. Similarly, if the application suddenly entered a phase where it performed less work, then the controller would reduce speedup appropriately.

One of the advantages of the GRAPE approach is that the control theoretic techniques presented here emit formal analysis, which heuristic techniques do not. While a rigorous mathematical analysis is beyond the scope of this paper (and also straightforward as GRAPE's controller is built on top of several standard mechanisms). The two major advantages of GRAPE's controller are: 1) it will converge to the desired performance (if achievable) and 2) the convergence time is bounded by the logarithm of the workload error estimate produced by the Kalman filter. Intuitively, GRAPE will hit the performance target and do so in a small number of steps (ie. invocations of Algorithm 1).

2.2.2 The Translator

The translator takes the generic speedup signal and produces a specific setting for the number of SMs, wavefronts, SM frequency, and memory frequency. Ideally, the translator would guarantee the desired speedup is achieved and minimize energy usage, which is properly an integer programming problem, and thus expensive to solve in hardware exactly even for small numbers of configurable resources. GRAPE, therefore, relies on a heuristic solution based on empirical observations.

At a high-level, the heuristic solution first finds the fastest combination of SMs and wavefronts for this application, it then selects the appropriate memory frequency based on the observed number of memory stalls, and finally reduces the SM frequency as much as possible while still achieving the speedup signal. Thus this heuristic still achieves the required speedup, but may sacrifice optimality to produce a simple implementation. Said another way, the heuristic will meet the required performance, but may use more energy than a true optimal solution.

Algorithm 2 details GRAPE's translation stage. It is broken into three distinct phases labeled with comments in the algorithm.

Algorithm 2 The Translator

```
Require: $s(t)$                                        ▷ speedup provided by controller
 1: procedure THE TRANSLATOR
      ▷ Compute the number of SMs
 2:     $nSM = 0$
 3:     for all $sm$ in the GPU do
 4:         if $sm ==$ active then
 5:             $nSM = nSM + 1$
 6:         end if
 7:     end for
      ▷ Compute the number of wavefronts
 8:     if $Cache_{stall} \geq Cache_{threshold}$ then
 9:         if $nW > 32$ then
10:             $nW = 32$
11:         end if
12:         $nW = nW - 4$
13:     else
14:         $nW = nW + 4$
15:     end if
      ▷ Compute the SM and memory frequencies
16:     if $DRAM_{stall} \geqslant SM_{stall}$ then
17:         $M = \{6, 7, 8\}$
18:     else
19:         $M = \{1, 2, 3, 4, 5, 6\}$
20:     end if
21:     $cost_{min} = \infty$
22:     for $MEMindex \in M$ do
23:         if $s_i \geq s(t)$ then
24:             $f_{SMindex} = \lceil f_{max} \cdot \frac{s(t)}{s_{MEMindex}} \rceil$
25:             $cost_{temp} = c_{SMindex} \cdot c_{MEMindex}$
26:             if $cost_{temp} \leq cost_{min}$ then
27:                 $cost_{min} = cost_{temp}$
28:                 $f_{SM} \leftarrow f_{SMindex}$
29:                 $f_{MEM} \leftarrow f_{MEMindex}$
30:             end if
31:         end if
32:     end for
33: return $nSM$                                         ▷ number of SMs to use
34: return $nW$                                ▷ number of wavefronts/warps
35: return $f_{SM}$                                            ▷ SM frequency
36: return $f_{MEM}$                                         ▷ DRAM frequency
37:
38: end procedure
```

Number of SM. The first phase (lines 2-7) simply counts how many SMs are *active*, meaning they have a CTA(Cooperative Thread Array) running. Empirically, it is always better to run as many SMs as possible. If the SM is not active, then GRAPE will set it to its lowest frequency setting.

Wavefront Scheduler. The next phase (in lines 8-15) determines the number of wavefronts to use. The key to determining the number of SMs is looking at the time the SM pipeline is spent stalling. If the stalls are above a threshold, then GRAPE limits wavefronts to 32 — 2/3 of its maximum capacity. We move wavefronts in steps of 4 per control action as we find it gives the best results empirically.

SM and DRAM DVFS. The final phase of translation (lines 16-32) set the SM and DRAM frequency. This phase uses two small tables that are stored in hardware and each indexed by an *id*. Each *id* has its own frequency, estimated speedups and costs (in power).

In this final phase, GRAPE is heuristically determining whether the application is compute or memory bound. It first checks if memory stalls are above a certain threshold (line 16). If they are, the application is considered memory bound, and GRAPE will only consider actuator settings that provide high memory frequency (line 17). Otherwise, GRAPE classifies the application as compute bound and only considers slower memory settings. At this stage, these settings are simply indexes into the tables mentioned above.

Once GRAPE has determined the settings to consider, it walks through those memory settings trying to find the slowest frequency setting that will meet or exceed the specified speedup (lines 21-31). Each step of the for loop matches a memory frequency setting to a corresponding SM frequency, determines whether or not that is above the required speedup, and then determines whether the cost (in power) is less than the lowest cost found so far. If the cost is lower, then GRAPE saves this new cost and the settings it found. After completing this final phase, the translator returns the number of SMs, number of wavefronts, the SM frequency, and the DRAM frequency.

Translation takes time proportional to the number of indices in the tables. In practice this tends to be a small number as hardware supports only a few frequency settings.

2.2.3 Actuator Model Update

The translator is reliant on models of frequency costs and speedups to produce good results. These values are not universal, however, and may differ for different applications. For example, a slight decrease in memory frequency may not have significant effect on a compute-bound benchmark, but it will for a memory-bound one. To account for these differences GRAPE updates these models on the fly. Initial actuator models assume that cost and performance between actions is linear, then after several decision period GRAPE will update the models to reflect actual behavior of the application under control.

We note that there is a complicated, non-linear relationship between the resources used, application workload, measured behavior and system noise. Rather than build a computationally expensive model, GRAPE adopts the approach of continually estimating this non-linear behavior with a series of linear models, including the Kalman filter presented above and the model update presented here. This approach is analogous to the way scientific applications model complicated, non-linear physical systems with iterative application of linear equations.

Every time GRAPE computes a new control action, it updates the table that stores its models of speedup and cost using Algorithm 3. This is a simple algorithm that updates the model as a function of its current value and the measured behavior. Line 2 computes a new estimate for the memory speedup in the last memory configuration used. The constant f_{max} represents the maximum frequency for SMs which is 800 MHz. β the learning rate which affects how fast the value changes. For example, $\beta = 1$ would always use the last measured value and ignore history. We set this value to 0.85 in our implementation. Lines 3-8 clamp the new value such that not overlap the higher and lower ID value to prevent overflow in the fixed-point hardware implementation. Line 9 updates the cost model, or power consumption,

Algorithm 3 Update Model

```
Require: $h(t)$                                          ▷ measured throughput
Require: $\hat{w}(t)$                                    ▷ estimated workload
Require: $cost(t)$                                       ▷ measured cost (power)
 1: procedure UPDATE MODEL
 2:     $s_{MEMindex} = \beta\cdot \frac{f_{max}}{f_{SMindex}} w(t)\cdot h(t) + (1-\beta)\cdot s_{MEMindex}$
 3:     if $s_{MEMindex} \geq highBound_{MEMindex}$ then
 4:         $s_{MEMindex} = highBound_{MEMindex}$
 5:     end if
 6:     if $s_{MEMindex} \leq lowBound_{MEMindex}$ then
 7:         $s_{MEMindex} = lowBound_{MEMindex}$
 8:     end if
 9:     $c_{MEMindex} = \beta \frac{cost(t)}{c_{SMindex}} + (1-\beta) c_{MEMindex}$
10: end procedure
```

of applying this system resource configuration.

GPU applications generally exhibit three distinct kinds of behavior depending on the resource that bottlenecks performance: compute, memory, or cache. Despite, this difference, however, we find that it is only necessary to update the model of memory response — the SM frequency response tends to keep linear for all types of benchmarks. Thus, we find that updating the SM frequency model is unnecessary. Therefore, to save area overhead, we only update the memory frequency model (using Algorithm 3) and keep the SM frequency model constant.

2.2.4 Hardware Implementation

GRAPE is designed to be implementable in hardware. While we do not have the resources to synthesize a GPU that includes GRAPE, we believe it is important to demonstrate that GRAPE can be implemented in hardware. We therefore implement (and release as open source) a VHDL implementation of GRAPE.

To get some specific numbers, we synthesize GRAPE for an FPGA using Quartus II software. The target FPGA device we use is DE2-115. We implement a fixed point package to perform multiplication and division in VHDL [13]. We synthesize the design and find that GRAPE requires 12,154 logic elements. TimeQuest timing analyzer shows that GRAPE's f_{max} is 1.35 MHz or 519 cycles overhead in GPU SM frequency. PowerPlay Early Power Estimator shows that GRAPE needs 0.178 Watts to operate. We share this implementation on the link below [2].

We implemented the dynamic frequency and SM actuator by masking the clock in GPGPU-Sim. Wavefront actuator implemented by swl scheduler. We edit the GPUWattch to count the dynamic voltage and leakage static power [17]. We use GTX480 model provided by those simulators. We model the DVFS overhead as 512 cycles [73]. These actuators increase the GPU power consumption by 0.3 W. One decision period for GRAPE is 8192 cycles. We include all of this overhead during simulation in GPGPU-Sim.

GRAPE samples the sensors every 4096 cycles. We assume there is no overhead in sampling the data. Control calculation is called early at 550 cycles earlier to reduce the error in calculation. Frequency overhead is 512 cycles, during this overhead period the simulator runs the application in previous frequency action.

Overall, we find that these results show GRAPE to be low overhead and easily implemented in hardware. The area, power, and timing would probably all improve if GRAPE was synthesized in ASIC or custom VLSI and added to a real GPU implementation. For the purposes of our evaluation we use these numbers from the FPGA for all experiments.

2.3 Evaluation

This section presents our empirical evaluation of GRAPE, using the experimental setup described in the previous section. We first measure GRAPE's ability to meet performance requirements. We then evaluate GRAPE's energy savings and peak power reduction. The section concludes by studying the impact of idle power on energy savings.

2. Available at: `https://github.com/grapemicro/GRAPE.git`

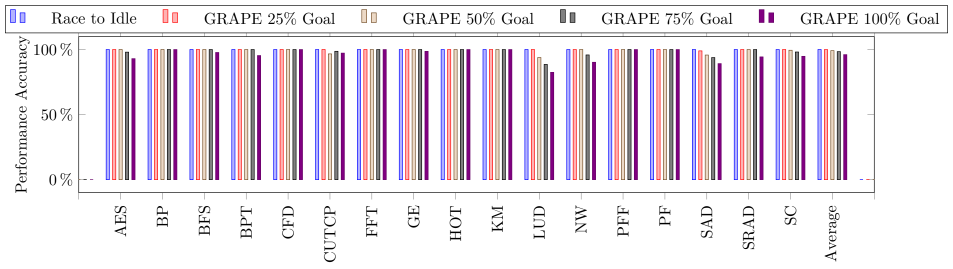


Figure 2.3: GRAPE Performance Accuracy

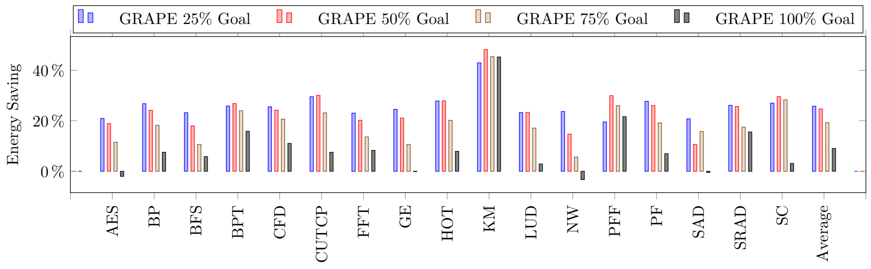


Figure 2.4: GRAPE energy savings compared to race-to-idle.

2.3.1 Performance Impact

For each benchmark we evaluate several performance goals. Specifically, we set a performance goal corresponding to $X\%$ of maximum performance where $X \in \{25, 50, 75, 100\}$. For example, FFT has a maximum performance of 408,346 MIPS (millions of instructions per second). The 25% goal for FFT means we set the performance at 102,086 MIPS.

We quantify error as the relative error expressed as a percentage. Relative error is the difference between the target and achieved performance divided by the target. We only count error if GRAPE runs the application below goal and count the error as zero if GRAPE runs the benchmark above the goal. Of course, running above the goal will incur additional energy costs, but we evaluate energy in the next section.

Figure 2.3 shows the relative error for each benchmark and performance target. GRAPE successfully maintains the performance goal achieving, on average, 99.25% of the desired performance; ie. 0.75% average error across all performance targets. We note that the race-to-idle strategy will never miss a target – the major advantage of this strategy – because it always completes all work in the default configuration and idles.

While GRAPE's accuracy is, in general, quite good, the results demonstrate two areas where GRAPE struggles. First, error increases as the performance target increases. Second, the errors are highest for the LUD and SAD benchmarks. As the performance target increases, GRAPE's margin for error decreases. While GRAPE's controller is self-correcting, at high performance targets, there may simply not be enough time to correct an error before the benchmark completes. This lack of time for the self-correction mechanism to take effect is the same issue that affects the LUD and SAD benchmarks. Both of these benchmarks consist of multiple kernels where the first kernel has very high parallelism and subsequent kernels have lower parallelism. GRAPE reduces resource usage for the high parallelism kernel, but then it is not physically possible to meet the performance target when the lower-parallelism kernels start to execute.

This pattern – highly parallel kernels followed by low-parallelism kernels – represents the worst case for GRAPE. Despite this worst case behavior, the results for SAD and LUD are still fairly good. We note that the results would improve if we ran these kernels in a loop, as that would allow GRAPE's self-correction mechanism to work over repeated application invocations. In addition, in future work, we could address this issue by combining GRAPE with static program analysis to provide GRAPE with the foreknowledge necessary to address this pattern.

2.3.2 Energy Impact

Figure 2.4 shows GRAPE's energy consumption and figure 2.5 shows its energy efficiency (performance/Watt). All numbers are normalized to the race-to-idle strategy. By geometric mean, targeting performance goals of 25%, 50%, 75% and 100% results in energy reductions of 25.76%, 24.66%, 19.25% and 9.02% compared to race-to-idle (higher is better). Meanwhile, targeting performance goal as 25%, 50%, 75% and 100% from default gives us 1.35×, 1.34×, 1.25× and 1.11× energy efficiency (MIPS/Watt, higher is better) compared to race-to-idle.

At the 100% performance target, race-to-idle is not actually idling the system at all. However, GRAPE's intelligent resource allocation strategies provide relative energy savings even when performance is not reduced. This is because GRAPE can reduce the energy consumed by unnecessary resources even when running at maximum performance. For example, GRAPE will reduce memory energy for compute bound benchmarks and reduce compute energy for memory bound benchmarks, compared to race-to-idle.

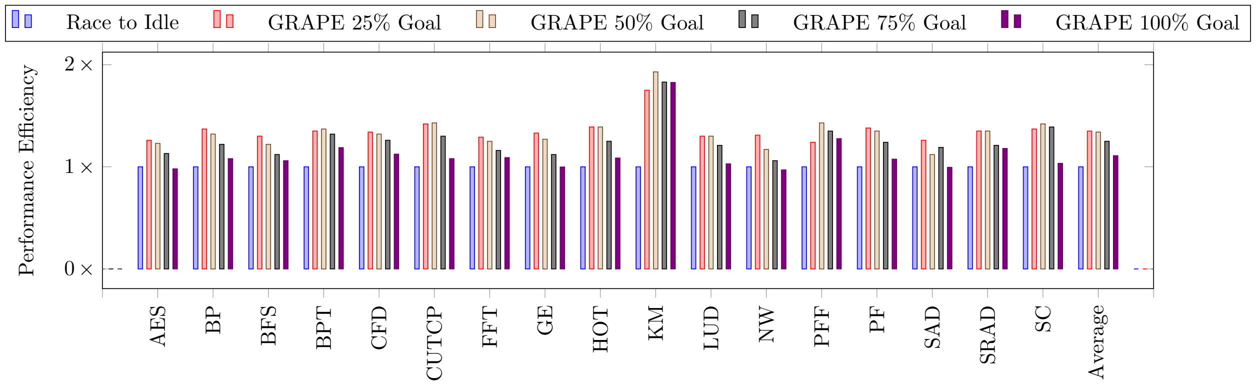


Figure 2.5: GRAPE energy efficiency (performance/Watt) compared to race-to-idle.

The kmeans benchmark gets the biggest benefit, as GRAPE increases its energy efficiency up to 1.93× and achieves energy saving of 48.18% compared to race-to-idle. GRAPE's wavefront scheduling successfully configures the most effective wavefront available. This increasing performance then turns into a reduction in resource usage. Streamcluster also benefits from this scenario, increasing energy efficiency up to 1.42× and achieves energy saving of 29.55%. GRAPE's generality also benefits computational benchmarks like Hotspot – increasing its performance efficiency up to 1.39× and achieves energy saving of 27.84%.

These results demonstrate the claim from the introduction: that careful tailoring of resource usage can greatly reduce energy consumption compared to strategies like racing-to-idle. Furthermore, these results demonstrate that it is possible to build a resource management strategy into hardware and achieve good results.

2.3.3 Power Impact

GRAPE not only decreases the energy consumption, it also decreases peak power consumption significantly compared to racing to idle. As we see in figure 2.6 GRAPE successfully manages the performance goal for 25%, 50%, 75% and 100% to give us 40.29%, 52.08%, 67.54% and 87.84% power reductions respectively.

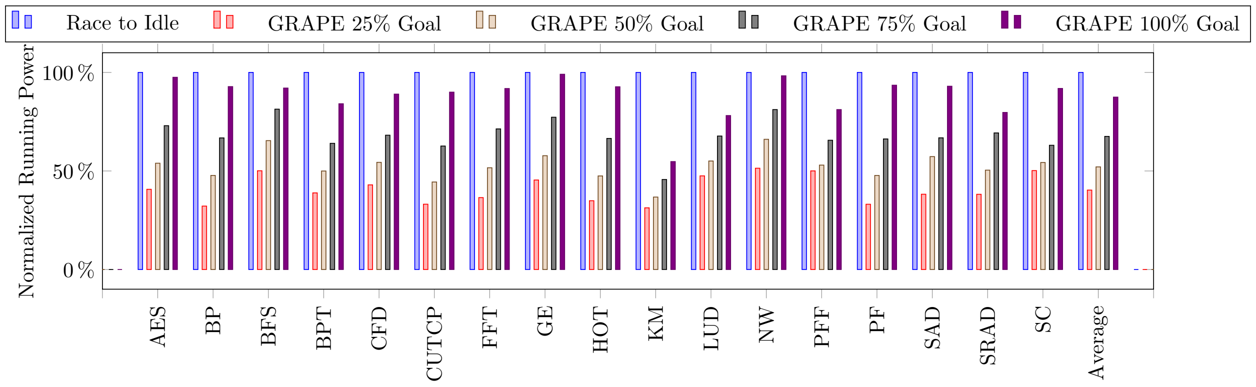


Figure 2.6: Average Power Consumption

2.3.4 Comparison with Prior Work

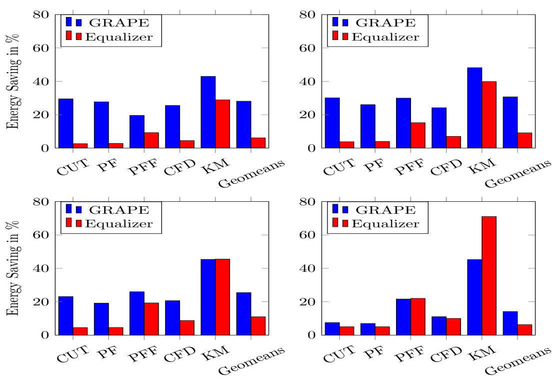


Figure 2.7: GRAPE comparison with Equalizer-to-idle in 25% goal (top-left chart), 50% goal (top-right chart), 75% goal (bottom-left chart) and unconstrained performance (bottom-right chart).

Figure 2.7 compares the energy savings of GRAPE to that of Equalizer [113], a comprehensive dynamic system which coordinates SM frequency, DRAM frequency, interconnect frequency, L2 frequency and number of CTA. While GRAPE performs constrained optimization (meeting performance with minimum energy), Equalizer is an unconstrained optimizer, it provides no performance guarantees, but generally tries to reduce energy without impacting performance. In this section, we compare GRAPE's constrained optimization approach to Equalizer-to-idle.

The results show that GRAPE's incorporation of performance requirements allows it to save substantial energy compared to Equalizer for all the targets less than 100%. For the 100% target, GRAPE is similar to Equalizer. We emphasize that GRAPE is not designed to improve on Equalizer, instead it solves a different problem: constrained optimization. These results, however, demonstrate that GRAPE can provide competitive behavior on unconstrained performance (bottom-right chart). Thus, GRAPE provides a new capability without diminishing existing capabilities.

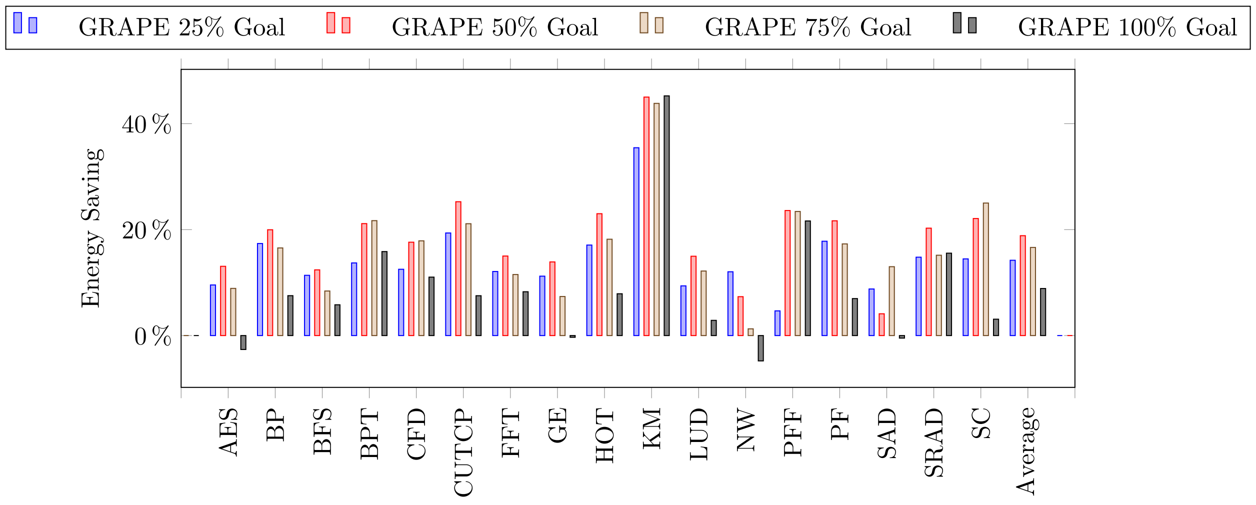


Figure 2.8: GRAPE energy saving compared to race-to-sleep.

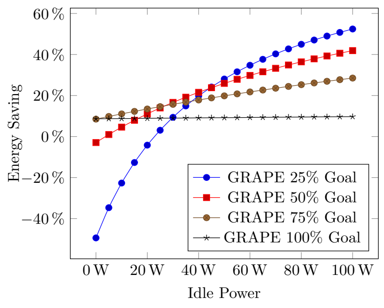


Figure 2.9: Energy Reduction in Varying Idle Power

2.3.5 Sensitivity to Idle Power

GRAPE's energy reduction is clearly sensitive to both the idle power consumption and the performance goal. We have already explored sensitivity to various performance goals in the above results. In this section we explore sensitivity to idle power.

Figure 2.8, compares GRAPE's energy savings to a race-to-sleep strategy. We assume sleeping GPU GTX480 power is 34.3265W, which all the SMs are in idle state and the voltage is minimum [80] and that the sleep state can be entered and exited with no overhead (likely an optimistic assumption). Compared to race-to-sleep, GRAPE can still increase the energy efficiency to 1.18 and decrease the energy consumption to 0.86×.

GRAPE saves energy by finding the best configuration and avoiding the high idle power in the GPU. If the idle power is very low then the energy saving and energy efficiency also become lower and if the idle energy is higher then the energy saving will be increasing too. We show the relation between different idle power and energy savings in Figure 2.9. However, due limitations in technology scaling, future processors are expected to decrease the dynamic power while the leakage power – and thus, idle power – increase [72, 60]. Therefore, we believe GRAPE will continue to be suitable and applicable to reduce future GPU energy consumption while maintaining performance goals.

2.4 Conclusion

GRAPE is a resource management system for interactive GPU applications. GRAPE takes a performance goal and then determines how to allocate resources to an application such that the performance goal is met and energy is minimized. GRAPE uses a computationally inexpensive control system which is easily realizable in hardware with low overhead. GRAPE is highly accurate in delivering performance yet it provides significant energy savings for applications with different performance goals. In addition, our results indicate that GRAPE is competitive with prior approaches for un- constrained optimization – meaning that GRAPE can have a positive benefit even for non-interactive applications. The combination of low-overhead and competitiveness with prior techniques means that GRAPE could be integrated into GPUs with almost no downside while providing significant energy savings for interactive applications.

CHAPTER 3

MERLOT: ARCHITECTURAL SUPPORT FOR ENERGY-EFFICIENT REAL-TIME PROCESSING IN GPUS

3.1 Overview

Systems such as self-driving cars have enormous computational requirements and incorporate GPUs ([33, 131]) and emerging GPU-like accelerators (e.g., for deep neural network sensor processing [99]) to meet those computational demands. A number of software schedulers have arisen for managing these GPU and accelerator resources to ensure complex embedded applications—like autonomous vehicles—meet the timing requirements necessary to ensure correct operation [33, 66, 65, 32, 87]. A secondary priority for these systems is minimizing energy consumption given the timing constraints.

There is a natural tension between timing guarantees and energy efficiency. When timing guarantees cannot be violated—ie. hard real-time requirements—software must conservatively allocate resources for worst case [14]. This conservatism is wasteful when inputs do not exercise the worst case execution path, consuming more energy than necessary [70].

Several software approaches have arisen to augment real-time schedulers with energy reduction [22, 112, 50, 12, 5, 38, 133, 55]. While these schedulers all differ in their details, the unifying theme is the exploitation of *timing slack*. If the scheduler can detect a task will terminate before its worst case schedule, the resulting slack can be converted into energy savings. In a GPU application consisting of multiple *kernels*, if an early kernel finishes ahead of schedule, then the slack can be *transferred* to the next kernel, slowing it down to save energy without risking the overall application timing.

Implementing slack transfer requires the application to signal sub-task completion to the software scheduler. This need for signaling, or *checkpointing*, limits the energy software can save through slack transfer. While kernels form a natural checkpoint, GPU applications may have only a few kernels (or even just one). Each kernel, however, consists of thousands of threads, making it tempting to use thread termination as a checkpoint. Signaling thread completion to a software scheduler, however, requires making significant changes to GPU code to trigger fine-grain communication between the GPU and CPU. Unfortunately, the overhead of this increased communication will quickly overwhelm any energy savings.

Consequently, this paper advocates hardware support for managing GPU resources to meet hard real-time deadlines with minimal energy by observing and managing timing slack at a sub-kernel level. This hardware support is not designed to replace software scheduling, but to complement it by extracting extra energy savings not available to software schedulers. There are two advantages to hardware-based resource management for real-time processing and one challenge to be addressed. The potential advantages are:

1. *Fast reaction time.* Hardware can perform fine-grained, low-overhead checkpointing to determine definitively when a kernel is not worst case and reduce its resource usage. Software cannot effectively detect or adapt to timing slack below the kernel level, because it cannot know a kernel is ahead of schedule until it completes.

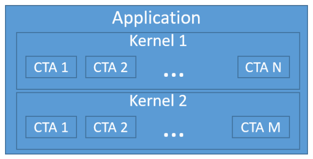
2. *Knowledge of resource usage.* Hardware has the most up-to-date information on a kernel's resource usage. Even a worst case kernel rarely requires all resources. One kernel may require more memory bandwidth, while a different one requires more compute. By observing usage, hardware can tailor resources to minimize energy while ensuring timing requirements.

The challenge of hardware-based resource management for real-time systems, however, is that software has all the information about timing requirements and progress. For example, it is software that knows task periods and deadlines. Hardware resource management is also not appropriate for all platforms, such as time-shared general-purpose processors, where tasks are not required to be decomposed into identical threads operating on different data. A hardware approach does make sense, though, for the growing class of specialized accelerators that have been proposed to increase performance and energy efficiency at the end of Dennard scaling [123, 34].

To enable hardware resource management in GPUs, we propose a small change in instruction set architecture that informs hardware of timing constraints. We show how to use this information to build a hardware resource allocator that provides hard timing guarantees with lower energy than state-of-the-art software-only approaches. While we think this approach is generally applicable to a number of accelerators—including recently released GPU-like accelerators for deep neural network based sensor processing [99]—we implement and evaluate it for GPUs.

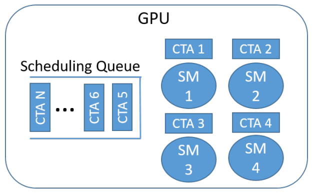
We call our system MERLOT. It consists of an interface allowing software to communicate timing requirements to hardware and a hardware resource manager that automatically adjusts (1) GPU frequency, (2) memory frequency, and (3) GPU core usage. The interface is a set of registers software uses to specify a GPU kernel's deadline and worst case execution time for a number of checkpoints hardware should take within the kernel. MERLOT begins executing the kernel with all resources available. It then samples execution—measuring progress as completed thread blocks—at fixed time intervals to determine (1) whether the current kernel is worst case and (2) which resources the kernel actually needs to meet its deadline. MERLOT then reduces the resources in use so that the software-specified deadline is met while energy consumption is minimized.

Figure 3.1 presents a high-level overview of a GPU program. Programmers create software applications that run on a traditional CPU and offload significant computation to the GPU, which provides a much faster and more energy efficient platform for highly parallel, regularly structured computations. The pieces of computation that are offloaded are called *kernels*, which correspond to parallel loops in traditional programs. Kernels themselves are broken up into *cooperative thread arrays* (CTAs), or blocks of threads that are executed in single-



(a)
b

Figure 3.1: GPU Application



(a)
b

Figure 3.2: GPU Hardware Scheduling

Figure 3.3: A GPU application with two kernels (a), each of which is divided into cooperative thread arrays (CTAs). The CTAs are scheduled in hardware (b), which assigns CTAs to SMs.

instruction multiple-data style on the GPU's *streaming multiprocessors* (SMs), analogous to CPU cores.

The kernel is the lowest-level software schedulable unit in a GPU program. Software running on a CPU (whether real-time or not) is responsible for launching the kernel on the GPU and synchronizing to ensure the kernel completes before the results are read from shared memory. Many software schedulers exist that allow GPUs to be shared among multiple processes [105, 67], and real-time schedulers exist that manage applications and kernels to ensure predictable timing in GPU-augmented systems [66, 65, 32, 33].

Note, however, that while software specifies CTAs, it has no control over—or even visibility into—how they are scheduled on the GPU hardware. While a GPU may have dozens of SMs, a typical kernel will have 100s to 1000s of CTAs. Therefore, GPU hardware maintains its own scheduling queue (as shown in Figure 3.2). As CTAs complete, hardware is responsible for selecting the next CTA to run. While hardware-level schedulers have been proposed [113], we know of only one that can provide soft timing guarantees [106], and we

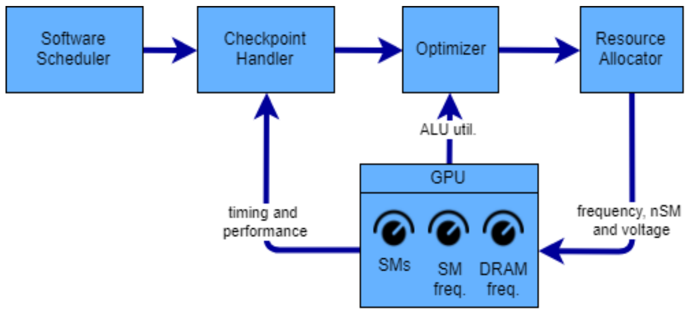


Figure 3.4: MERLOT block diagram

are not aware of any that support hard timing requirements.

Thus, in GPU-based systems there is divided knowledge. Software has information about application structure and kernel timing requirements that hardware does not. Hardware, however, can observe kernel progress at the much finer granularity of CTA completion. Additionally, hardware has knowledge about what resources are actually in use; eg. whether the kernel compute- or memory-bound. *The primary goal of this paper is to bridge this knowledge gap so that software can provide timing information to hardware, and hardware can use that information to reduce energy while ensuring software-specified timing requirements are met.*

Figure 3.4 illustrates MERLOT's design. When software launches a kernel on the GPU, it specifies a number of checkpoints (in units of CTAs) and a worst case completion time (in milliseconds) for each. These values are written into hardware registers on the GPU. As the kernel executes, MERLOT measures the time at each checkpoint to determine whether or not the kernel is ahead of its worst case timing.

If the kernel is ahead, then this timing slack can be transferred to the next checkpoint; ie. MERLOT adjusts hardware resources to slow the next set of CTAs down such that energy efficiency is maximized and the deadline is just met. MERLOT's hardware consists of three modules: (1) a *checkpoint handler* that tracks execution progress and determines available slack; (2) an *optimizer* that turns the slack into specific settings for the number of SMs to use, the SM frequency, and the DRAM frequency; and (3) a resource allocator that actually enforces the settings determined by the optimizer.

We implement MERLOT in VHDL and synthesize for an FPGA to show it can be implemented in hardware. To demonstrate it meets hard real-time requirements on a GPU, we integrate the design into GPGPU-Sim, a cycle-accurate simulator of an NVIDIA GTX 480 GPU [7]. We compare the timing guarantees and energy consumption to prior software-only approaches that either (1) race-to-idle—completing the kernel as fast as possible and then idling the GPU—or (2) transfer timing slack between kernels [12, 5, 38].

We test MERLOT with a wide range of applications and latency targets. MERLOT meets hard real-time deadlines. Compared to race-to-idle, MERLOT reduces energy consumption by 16.43%. Compared to the sophisticated software scheduler that transfers slack between kernels, MERLOT reduces energy consumption by 15.63%. Compared to a prior hardware approach that provides only soft timing guarantees, MERLOT provides almost equivalent energy savings without missing deadlines. MERLOT provides energy savings even when deadlines are large multiples of the worst case latency. Finally, experiments with modified power models show that MERLOT's energy savings will increase as GPUs incorporate more energy efficient features and lower-power idle states.

MERLOT's primary contribution is recognizing the potential for energy savings by incorporating hardware support into hard real-time scheduling. This energy savings arises from hardware's ability to quickly detect (1) when inputs are not worst case and (2) what resources are actually needed by the current task. Once 1 and 2 are known, hardware can scale back resource usage without violating the timing constraints. We demonstrate this benefit for GPUs, but we believe the idea is widely applicable to the plethora of accelerators that have recently been proposed for various specialized tasks. We release our modifications to GPGPU-Sim as open source so that others can recreate or extend our results [1].

1. https://github.com/santriaji/MERLOT.git

3.2 Design and Implementation

MERLOT provides architectural support for meeting hard real-time deadlines while minimizing energy through management of a GPU's (1) streaming multiprocessors (SMs or cores), (2) SM frequency, and (3) DRAM frequency. We emphasize that MERLOT's goal is not to replace existing software schedulers, but to augment them by ensuring the software-specified timing requirements are met while adjusting the specified resources to reduce energy. MERLOT operates on hardware-level structures that are simply not accessible to software. Thus, MERLOT allows existing software schedulers to focus on high-level scheduling decisions (eg. when to schedule a kernel), while hardware focuses on low-level resource management, for which it is better suited.

Figure 3.4 illustrates MERLOT's design. When software launches a kernel on the GPU, it specifies a number of checkpoints (in units of CTAs) and a worst case completion time (in milliseconds) for each. These values are written into hardware registers on the GPU. As the kernel executes, MERLOT measures the time at each checkpoint to determine whether or not the kernel is ahead of its worst case timing. If the kernel is ahead, then this timing slack can be transferred to the next checkpoint; ie. MERLOT adjusts hardware resources to slow the next set of CTAs down such that energy efficiency is maximized and the deadline is just met. MERLOT's hardware consists of three modules: (1) a *checkpoint handler* that tracks execution progress and determines available slack; (2) an *optimizer* that turns the slack into specific settings for the number of SMs to use, the SM frequency, and the DRAM frequency; and (3) a resource allocator that actually enforces the settings determined by the optimizer.

We detail each MERLOT module in turn. For each module we give an intuitive overview and then formally specify its behavior in the form of equations and algorithms. The final subsections discuss how MERLOT's hardware structures can be trivially repurposed for timing analysis instead of enforcement and then describe MERLOT timing guarantees and implementation issues. Table 5.1 summarizes the notation used throughout this section.

Table 3.1: Notation used in the paper.

Symbol	**Meaning**
$nCTA_j$	number of CTAs in checkpoint j
$wCTA_j$	WCET of checkpoint j
$t_{current}$	current time
o	timing overhead
t_{ahead}	timing slack
η	slowdown factor ranged from 0 to 1
f_{max}	maximum frequency id
λ	frequency id ranged from 0 to f_{max}
f_{crit}	critical frequency for energy efficiency
ALU_{util}	ALU utilization in GPU, ranged from 0 to 1
th_{ALU}	a threshold to distinguish memory and compute needs
sm	status of SM in GPU
nSM	number of SMs
f_{mem}	memory frequency of configuration
f_{SM}	SM frequency of configuration

3.2.1 Software-Hardware Interface

An exclusively software approach can only transfer slack between kernels. MERLOT allows further energy savings by tracking CTA completions in hardware as checkpoints within a kernel, allowing slack to be transferred within a kernel at a finer granularity than software alone can handle. To ensure timing, however, hardware must be informed of software's requirements.

The interface between software and hardware is a set of special purpose registers on the GPU that allow software to specify the number of checkpoints (in terms of number of CTAs to complete) and the deadline for each checkpoint (in terms of the worst case timing for that set of CTAs). A checkpoint j is a pair: the number of CTAs in a checkpoint $nCTA_j$ and the worst case timing for that set of CTAs $wCTA_j$. To keep the memory overhead low, we bound the number of checkpoints to 16 per application in this implementation.

We believe this is a flexible interface that supports a wide range of software schedulers. Using this simple interface, software can specify checkpoints per application or per kernel by simply dividing existing timing requirements among the checkpoints. More sophisticated schedulers with deep knowledge of the underlying hardware and data access patterns can even split timing requirements unevenly among CTAs. A methodology for worst case timing analysis of kernels, CTAs, and threads in GPU applications can be found in [10]. Notably, as MERLOT tracks individual CTAs' completion time, MERLOT can be trivially extended to produce per CTA timing reports (as described in Section III.E). Using this profiling mode, a developer could first use MERLOT to better understand CTA timing and then switch to enforcement mode using the deadlines produced from profiling.

3.2.2 Checkpoint Handler

Algorithm 4 lines 3–9 show pseudo-code for the checkpoint handler. Starting from the last checkpoint, MERLOT counts CTA completion in hardware. When the count is equal to a checkpoint count (ie. when the required number of CTAs has been completed), then MERLOT measures the current time $t_{current}$. It then calculates how far ahead of schedule it is t_{ahead} by subtracting $t_{current}$ from the worst case time for this checkpoint $wCTA_j$. (Although beyond the scope of the paper, this interface could easily be extended to detect when a kernel was *behind* the supposed worst case schedule and trigger a software interrupt on the CPU.) MERLOT then calculates how much time until the next checkpoint's worst case deadline $wCTA_{j+1}$. We note that if $t_{ahead} > 0$, then the system resources can be reduced without jeopardizing worst case timing. Thus, t_{ahead} represents the timing slack that hardware transfers from one set of CTAs to the other.

Given this timing information, MERLOT's checkpoint handler then computes the slowdown η that is permissible given the timing slack t_{ahead}. Note that this slowdown calculation (line 7 of Algorithm 4) includes the overhead o of changing hardware configurations—ie. the worst case timing (in milliseconds) of adjusting the number of SMs in use, their frequency and the DRAM frequency—which is set by the hardware developer. η is a fixed point number in the range of 0 to 1. Smaller η corresponds to more timing slack, meaning the hardware can reduce resources for the next checkpoint.

As a final step, the checkpoint handler transforms η into an *effective frequency* λ my multiplying η with the GPUs maximum clock frequency f_{max} and rounding to the nearest whole number. Thus λ represents a slower than maximum frequency that will still meet the specified checkpoint deadline. The checkpoint handler then passes this value to the optimizer.

3.2.3 Optimizer

The optimizer takes the effective frequency λ and converts it into a frequency for the SMs, a frequency for the DRAM, and the number of SMs to activate for the upcoming checkpoint (inactive SMs are power-gated to save energy). The optimizer here specifies frequencies as integer identifiers. For example, if the system supports 10 frequencies for SMs, then $0 \leq f_{SM} \leq 9$. The resource allocator will convert these identifiers into actual settings for the underlying hardware.

There are two challenges that complicate the optimization process. First, all processors (CPUs, GPUs, and other specialized accelerators) have some critical threshold beyond which additional slowdown actually causes higher energy consumption [95], so the optimizer must avoid reducing resources to the point that energy consumption is actually worse. Second, different applications will need different resources; some will require more memory bandwidth, while others will require more compute. The optimizer must deliver the resources that the application actually needs so that it can save energy.

Regarding the first challenge: while reducing the hardware resources in use will always reduce *power*, there is an *critical frequency threshold* f_{crit} beyond which additional slowdown will actually increase energy consumption [95, 106]. This threshold arises from the fact that chip power consumption consists of both a dynamic and static component. Dynamic power is decreased with decreased resource usage. While static power will remain constant if only frequency is changed. The critical frequency threshold is the point at which additional

Algorithm 4 Hardware Real-time Management

```
Require: $c_j[wCTA_j, nCTA_j]$                                  ▷ WCET for checkpoint j
Require: $t_{current}$                                          ▷ current elapsed time
Require: $ALU_{utilization}$
Require: $f_{max}$                                              ▷ maximum id for frequency scaling
Require: $f_{crit}$                                             ▷ frequency id that gives best energy
Require: $th_{ALU}$                                             ▷ ALU threshold that split memory and compute phase
Require: $CTA_{current}$                                        ▷ current number of CTA that already finished
Require: $o$                                                    ▷ Overhead
 1: procedure INITIALIZATION
 2:     $j \leftarrow 0$
 3: end procedure
 4: procedure CHECKPOINT HANDLER
 5:     if $nCTA_j = CTA_{current}$ then
 6:         $t_{ahead} = wCTA_j - t_{current}$
 7:         $t_{remaining} = wCTA_{j+1} - wCTA_j$
 8:         $\eta = \frac{t_{remaining+o}}{t_{remaining}+t_{ahead}+o}$
 9:         $\lambda = \lceil f_{max} \cdot \eta \rceil$
10:         $j++$
11:     end if
12: end procedure
13: procedure OPTIMIZER
14:     if $\lambda < f_{crit}$ then
15:         if $ALUutilization < th_{ALU}$ then
16:             $f_{SM} \leftarrow f_{crit} - 1$
17:             $f_{mem} \leftarrow f_{crit}$
18:         else
19:             $f_{SM} \leftarrow f_{crit}$
20:             $f_{mem} \leftarrow f_{crit} - 1$
21:         end if
22:     else
23:         if $ALUutilization < th_{ALU}$ then
24:             $f_{SM} \leftarrow \lambda$
25:             $f_{mem} \leftarrow \lambda + 1$
26:         else
27:             $f_{SM} \leftarrow \lambda$
28:             $f_{mem} \leftarrow \lambda$
29:         end if
30:     end if
31:     $nSM = 0$
32:     for all $sm$ in the GPU do
33:         if $sm ==$ active then
34:             $nSM = nSM + 1$
35:         end if
36:     end for
37: end procedure
38: procedure RESOURCE ALLOCATOR
39:     if $f_{mem} > f_{max}$ then
40:         $f_{mem} \leftarrow f_{max}$
41:     end if
42:     $Allocate(f_{SM}, f_{mem}, nSM)$
43: end procedure
```

slowdown increases energy consumption. If λ is below this value, then it is more energy efficient to run the processor at f_{crit} and then transition to a low-power idle state than it is to slowdown beyond f_{crit}.

Regarding the second challenge: prior work shows that memory- and compute-bound kernels can be distinguished by both their progress through the hardware scheduling queue and through their arithmetic and logical unit (ALU) utilization [113]. The exact threshold th_{ALU} for distinguishing these two types of kernels is hardware dependent; ie. it is a function of the SM core and memory design. Therefore, it must be determined by the hardware designer. On our experimental system, we determined this threshold to sit at 20% utilization, which is consistent with prior work [113].

MERLOT's optimizer uses these two observations to turn the effective frequency λ into an actual set of resources for the kernel to use, specified in pseudo-code in Algorithm 4, lines 10–28. Using nested conditionals, the optimizer accounts for four cases in lines 11–24. The cases correspond to whether or not the effective frequency λ is below the critical frequency f_{crit} and whether or not the kernel is memory-bound or compute-bound. For the SM frequency, if $\lambda < f_{crit}$, MERLOT simply uses f_{crit}, otherwise using λ. To set DRAM frequency, MERLOT relies on the observation that memory-bound applications should have DRAM set faster than the processor, while compute bound applications should have the DRAM set slower than the processor [113].

After adjusting SM and DRAM frequency, MERLOT sets the active number of SMs in Algorithm 4, lines 26–28. Power gating in our test system is not implemented well, so it is always most energy efficient to use all SMs, as long as there are CTAs to schedule. If there are not enough CTAs to saturate all SMs, then MERLOT records them as inactive, so that the resource allocator can set them to their lowest frequency and save energy. On a system with better power-gating, MERLOT could use SMs as another configuration parameter to tune the performance/energy tradeoffs to meet the required effective frequency with lower

Table 3.2: Resource Allocator Translation

SM frequency		*DRAM frequency*		Normalized Voltage
f_{SM}	frequency	f_{mem}	frequency	
7	700 MHz	7	924 MHz	1
6	600 MHz	6	792 MHz	0.9
5	500 MHz	5	660 MHz	0.83
4	400 MHz	4	528 MHz	0.76
3	300 MHz	3	396 MHz	0.7
2	200 MHz	2	264 MHz	0.62
1	100 MHz	1	132 MHz	0.55

energy.

One issue to note is that misclassifications may occur, but they will *not* cause the system to miss a deadline. The result of a misclassification is that energy efficiency will be lower than optimal. Misclassifications only affects energy (rather than timing) because the optimizer never reduces frequency below that determined by λ in the checkpoint handling procedure; ie. the frequency is always above the "safe" zone.

3.2.4 Resource Allocator

The *resource allocator* translates integer f_{SM} and f_{mem} from the optimizer to actual SM and DRAM frequency values and a voltage for each. This step is entirely hardware dependent. The hardware designer has to supply a table mapping the integer identifiers into actual frequency and voltage values. As an example, Table 3.2 shows this mapping for the evaluation system used in this paper. Lines in 30–31 check whether the frequency identifiers are out of bounds, which can only happen due to the increase in memory frequency in line 21.

3.2.5 Using MERLOT for Dynamic Timing Analysis

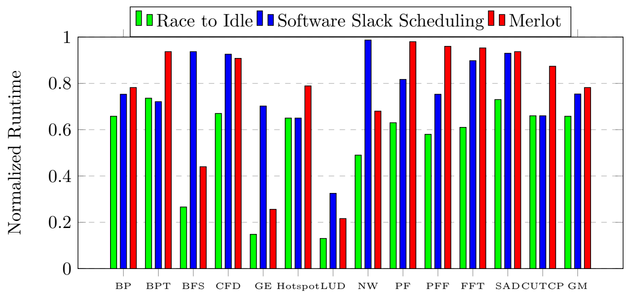
As mentioned earlier, MERLOT can not only enforce deadlines, it can be used to help developers understand how to set those deadlines in the first place. As shown in Algorithm 1, observes the completion time of each CTA before reconfiguring the GPU. Thus, at the cost of an additional register for storage, we can easily operate MERLOT in a profiling mode, where it simply updates a register with the longest completion time measured for each CTA (and the GPU resources are not changed). When the kernel completes, software on the CPU can query this register to get the measured CTA time. This profiling mode would be especially helpful during system design and development (before deployment). Developers can run all key kernels in profiling mode and record their observed CTA completion time. Software developers could even stress the system; for example, flushing all GPU caches and memories then run the kernels and measure CTA time using MERLOT's profiling ability. We emphasize that this support is only for dynamic timing analysis: it reports the worst measured time, but provides no guarantees that it finds the true worst case timing. Developers requiring true worst case timing bounds will need a static timing analysis tool.

3.2.6 Real-Time Considerations

We analyze MERLOT's hard real-time guarantees by following the example in [103]. This work shows a real-time scheduler guarantees that tasks will meet the deadline if the task set is schedulable. The task set is schedulable under EDF (earliest deadline first) if $C_1/P_1 + C_2/P_2 + ... + C_n + P_n \leq 1$, C is wcet and P is period. Then the task is schedulable under scaled value η if $C_1/P_1 + C_2/P_2 + ... + C_n + P_n \leq \eta$. Line 7 of Algorithm 4 reflects this schedulability test.

3.3 Evaluation

Because it is a simulator, GPGPU-Sim is entirely deterministic, which means that in simulation, all kernels have the same completion time. We therefore add some randomization to each program's timing properties. Specifically, for each benchmark we evaluate two different timing profiles, where the worst case timing is a factor of X times the average case timing when the application is allocated all resources and $X \in \{1.5, 2\}$. We note that these relative



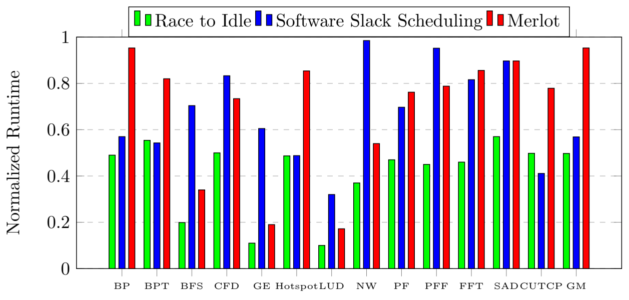
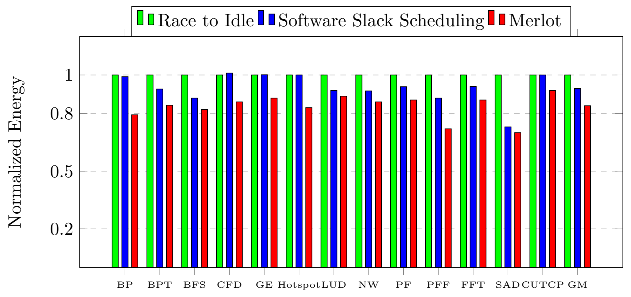


Figure 3.5: Running time normalized to deadline. The left figure shows the results where the worst case is 1.5× the average case and the right side shows the case where the ratio is 2.0×.



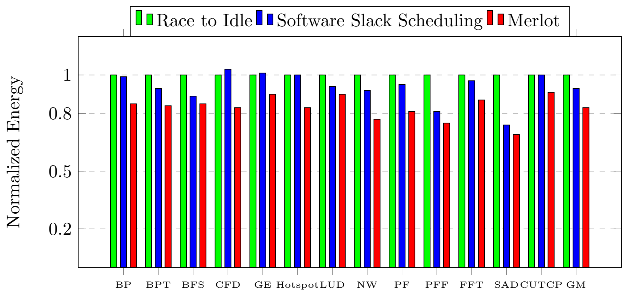


Figure 3.6: Energy normalized to the race-to-idle approach. The left figure shows the results where the worst case is 1.5× the average case and the right side shows the case where the ratio is 2.0×.

differences between the average and worst case timing are tighter than found in prior work [10].

The race-to-idle approach only takes advantage of dynamic power management, completing all work as fast as possible and then switching to the low-power idle state. Meanwhile the software approach will slowdown the configuration between the kernels, using a combination of slack transfer, dynamic voltage and frequency scaling and dynamic power management. This software scheduler runs as slow as possible such that the deadline is met by reducing the frequency of SMs and DRAM. MERLOT runs Algorithm 4 whenever the checkpoint is triggered. MERLOT is configured so that each application has 16 checkpoints.

3.3.1 Performance

Figure 3.5 shows how much faster MERLOT is done ahead of the worst case deadline (note the last set of bars on the x-axis is the geometric mean across all applications). Both race-to-idle, software and MERLOT never miss the deadlines. MERLOT in average runs 13.6% faster than the deadlines. We highlight several examples that illustrate the different behavior of the different schedulers.

As discussed in motivation, `backprop` has 2 kernels each of which take almost equal time, the first kernel is 11% longer than the second kernel. Software cannot configure the resources until the first kernel is finished, but MERLOT configures the resources at every checkpoint. Thus, MERLOT is closer to deadline—only 5% faster than necessary in 2.0X of ACET, while software is 43% faster than deadline. As we will see, this means that software incurs a higher peak power and higher total energy consumption.

`hotspot` is an example of an application with a single kernel. Race-to-idle and slack transfer are equivalent for this application because software cannot transfer slack in this application. Meanwhile, MERLOT can transfer the slack between CTA and slowdown the runtime to save power.

Because it is composed of many different kernels the software slack transfer approach has many chances to slowdown the `bfs` application. For this application, software actually drops the frequency below the critical threshold, which increases energy consumption. MERLOT's hardware-based approach is aware that it is not beneficial to slow down this much. Therefore the software approach is closer to the deadline—only 1.42× faster—and MERLOT is faster, but saves more energy as we will see in the next section.

3.3.2 Energy Saving

Figure 3.6 shows race-to-idle, software and MERLOT's energy consumption normalized to race-to-idle (note the last set of bars on the x-axis is the geometric mean across all applications). On average, software spends 93.4% energy and MERLOT saves 16.73% energy compared with race-to-idle with 16.43% in 1.5X and 17.03% in 2.0X. MERLOT consistently gives a better energy savings than the race-to-idle and software approaches. The main reason is that MERLOT simply has more opportunities to reduce resource usage. This fact is illustrated by the `backprop` and `hotspot` applications for which MERLOT provides 20.6% and 15.4% energy savings, respectively. In contrast, the software approach reduces `backprop`'s energy by just 1% with no savings for `hotspot`, as it has just a single kernel.

In previous subsection we saw that software approach is closer to deadline than MERLOT for the `cfd` and `bfs` applications. This timing occurs because the software is not aware of the critical frequency threshold and slows down the applications too much. This slowdown reduces peak power consumption, but actually increases energy use. The software approach uses 3.5% and 1.7% more energy in 1.5X and 2.0X for `cfd` than race-to-idle, while MERLOT saves 13.99% and 16.4% energy. In `bfs`, software saves 10.19% and 11.83% energy, which is worse than MERLOT which saves 17.68% and 15.05% energy.

3.3.3 Comparison with Prior Work

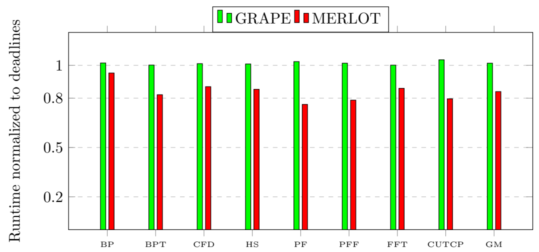


Figure 3.7: MERLOT and GRAPE Performance

GRAPE is a prior hardware approach to minimize energy in soft real-time systems [106]. GRAPE is based on control theory and cannot guarantee deadlines, but instead achieves close to the desired performance on average. As published numbers show, GRAPE will fail to meet deadlines if kernels within an application vary significantly in achievable performance. Another important difference is that GRAPE tracks application process using instructions

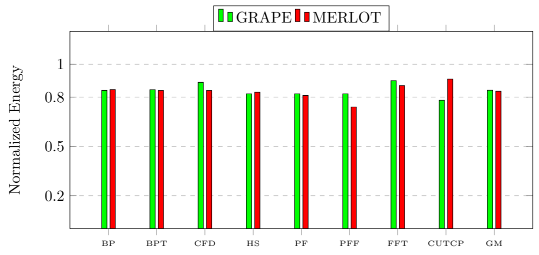


Figure 3.8: MERLOT and GRAPE Energy

retired, while MERLOT uses CTAs as checkpoints. Due to the different guarantees and progress metrics, the user interfaces are also different. GRAPE accepts a single number from a user: the target rate of instructions per second for a complete GPU application. In contrast, MERLOT requires each kernel to specify deadlines (in terms of number of checkpoints and worse-case timing for those checkpoints). This interface also affects the implementation, as GRAPE periodically checks progress and predicts whether or not the application is on schedule. In contrast, MERLOT triggers only at user-specified checkpoints and performs far less work, as it simply transfers any slack from the previous checkpoint to the next. This difference in workload can be see in the implementation details: GRAPE's FPGA implementation requires 18K logic elements and 63 multipliers, where MERLOT requires only 1K logic elements and no multipliers (see Section 2.2.4). Figure 3.7 compares the performance of MERLOT to GRAPE. Both target twice the default latency. Figure 3.8 compares the energy savings between those two. MERLOT reduces energy by transferring the slack of the task meanwhile GRAPE takes an average of the performance over time using a feedback control system. Figure 3.7 shows that while MERLOT never misses a deadline, GRAPE misses deadlines in all of the benchmarks. While MERLOT provides hard real-time guarantees, GRAPE's energy saving is slightly worse to MERLOT. MERLOT saves 16.38% energy consumption while GRAPE saves 15.96% energy saving on average. This is because MERLOT already know what is the best frequency to working on f_{crit} while GRAPE is oscillating to find the best energy configuration.

3.4 Conclusion

The computational demands of embedded systems like au- tonomous vehicles has created a need for GPU and GPU-like accelerators in these systems. While a great deal of work has been done to support real-time processing on GPU-equipped systems, energy management remains a concern. This paper argues that a simple hardware interface can allow software schedulers to make timing constraints known to hardware. Once those constraints are known, hardware can quickly detect when applications are running ahead of schedule and reduce their resource usage to save energy without violating their timing requirements. To test this insight we have proposed and implemented MERLOT. We find that MERLOT incurs negligible performance, power, and area overhead; but it can reduce GPU energy consumption substantially compared to sophisticated software-only schedulers. While implemented and tested on GPUs, we believe the insights are applicable to a wide variety of hardware accelerators that break software- specified tasks up into smaller hardware-schedulable units.

CHAPTER 4

ALERT: ACCURATE ANYTIME LEARNING FOR ENERGY AND TIMELINESS

4.1 Overview

Deep neural networks (DNNs) have become a key workload for many computing systems due to their high inference accuracy. This accuracy, however, comes at a cost of long latency, high energy usage, or both. Successful DNN deployment requires meeting a variety of user-defined, application-specific goals for latency, accuracy, and often energy in unpredictable, dynamic environments.

Latency constraints naturally arise with DNN deployments when inference interacts with the real world as a consumer—processing data streamed from a sensor—or a producer—returning a series of answers to a human. For example, in motion tracking, a frame must be processed at camera speed [57]; in simultaneous interpretation, translation must be provided every 2–4 seconds[85]. Violating these deadlines may lead to severe consequences: if a self-driving vehicle cannot act within a small time budget, life threatening accidents could follow [82].

Accuracy and energy requirements are also common and may vary for different applications in different operating environments. On one hand, low inference accuracy can lead to software failures [101, 119]. On the other hand, it is beneficial to minimize DNN energy or resource usage to extend mobile-battery time or reduce server-operation cost [58].

These requirements are also highly dynamic. For example, the latency requirement for a job could vary dynamically depending on how much time has already been consumed by related jobs before it[82]; the power budget and the accuracy requirement for a job may switch among different settings depending on what type of events are currently sensed [1]. Additionally, the latency requirement may change based on the computing system's current context; e.g., in robotic vision systems the latency requirement can change based on the robot's latency and distance from perceived pedestrians [29].

Satisfying all these requirements in a dynamic computing environment where the inference job may compete for resources against unpredictable, co-located jobs is challenging. Although prior work addresses these problems at either the application level or system level separately, each approach by itself lacks critical information that could be used to produce better results.

At the application level, different DNN designs—with different depths, widths, and numeric precisions—provide various latency-accuracy trade-offs for the same inference task [122, 59, 40, 54, 115]. Even more dynamic schemes have been proposed that adapt the DNN by dynamically changing its structure at the beginning of [36, 91, 121, 129] or during [75, 120, 49, 78, 127, 48, 124] every inference tasks.

Although helpful, these techniques are sub-optimal without considering system-level adaptation options. For example, under energy pressure, these application-level adaptation techniques have to switch to lower-accuracy DNNs, sacrificing accuracy for energy saving, even if the energy goal could have been achieved by lowering the system power setting (if there is sufficient latency budget).

At the system level, machine learning [4, 77, 102, 104, 116, 25, 26**?**] and control theory [61, 62, 93, 43, 107, 136] based techniques have been proposed to dynamically assign system resources to better satisfy system and application constraints.

Unfortunately, without considering the option of application adaptions, these techniques also reach sub-optimal solutions. For example, when the current DNN offers much higher accuracy than necessary, switching to a lower-precision DNN may offer much more energy saving than any system-level adaptation techniques. This problem is exacerbated because, in the DNN design space, very small drops in accuracy enable dramatic reductions in latency, and therefore system resource requirements.

A cross-stack solution would enable DNN applications to meet multiple, dynamic constraints.

However, offering such a holistic solution is non-trivial. The combination of DNN and system-resource adaptation creates a huge configuration space, making it difficult to dynamically and efficiently predict which combination of DNN and system settings will meet all the requirements optimally. Furthermore, without careful coordination, adaptations at the application and system level may conflict and cause constraint violations, like missing a latency deadline due to switching to higher-accuracy DNN and lower power setting at the same time.

4.1.1 Contributions

This paper presents ALERT, a cross-stack runtime system for DNN inference to meet user goals by simultaneously adapting both DNN models and system-resource settings.

Understanding the challenges We profile DNN inference across applications, inputs, hardware, and resource contention confirming there is a high variation in inference time. This leads to challenges in meeting not only latency but also energy and accuracy requirements. Furthermore, our profiling of 42 existing DNNs for image classification confirms that different designs offer a wide spectrum of latency, energy, and accuracy tradeoffs. In general, higher accuracy comes at the cost of longer latency and/or higher energy consumption. These trade-offs offered provide both opportunities and challenges to holistic inference management.

Run-time inference management We design ALERT, a DNN inference management system that dynamically selects and adapts a DNN and a system-resource setting together to handle changing system environments and meet dynamic energy, latency, and accuracy requirements.

ALERT is a feedback-based run-time. It measures inference accuracy, latency, and energy consumption; it checks whether the requirements on these goals are met; and, it then outputs both system and application-level configurations adjusted to the current requirements and operating conditions. ALERT focuses on meeting constraints in *any* two dimensions while

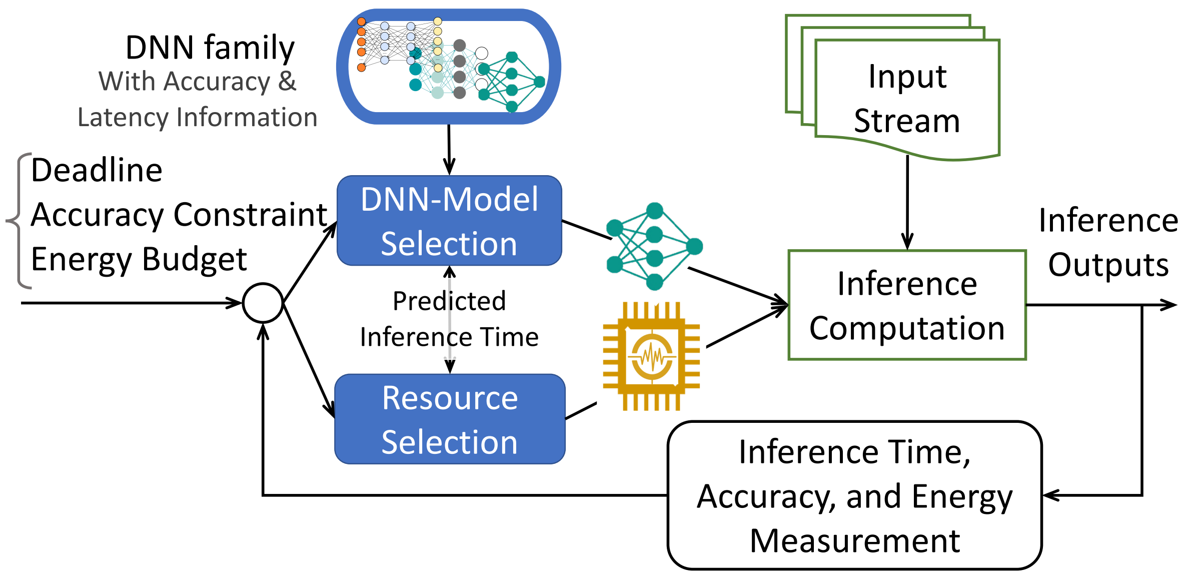


Figure 4.1: ALERT inference system

optimizing the third; e.g., minimizing energy given accuracy and latency requirements or maximizing accuracy given latency and energy budgets.

The key is estimating how DNN and system configurations interact to affect the goals. To do so, ALERT addresses three primary challenges: (1) the combined DNN and system configuration space is huge, (2) the environment may change dynamically (including input, available resources, and even the required constraints), and (3) the predictions must be low overhead to have negligible impact on the inference itself.

ALERT addresses these challenges with a *global slow-down factor*, a random variable relating the current runtime environment to a nominal profiling environment. After each inference task, ALERT estimates the global slow-down factor using a Kalman filter. The global slow-down factor's mean represents the expected change compared to the profile, while the variance represents the current volatility. The mean provides a single scalar that modifies the predicted latency/accuracy/energy for *every* DNN/system configuration—a simple mechanism that leverages commonality among DNN architectures to allow prediction for even rarely used configurations (tackle challenge-1), while incorporating variance into predictions naturally makes ALERT conservative in volatile environments and aggressive in quiescent ones (tackle challenge-2). The global slow-down factor and Kalman filter are efficient to implement and low-overhead (tackle challenge-3). Thus, ALERT combines the global slow-down factor with latency, power, and accuracy measurements to select the DNN and system configuration with the highest likelihood of meeting the constraints optimally.

We evaluate ALERT using various DNNs and application domains on different (CPU and GPU) machines under various constraints. Our evaluation shows that ALERT overcomes dynamic variability efficiently. Across various experimental settings, ALERT meets constraints while achieving within 93–99% of optimal energy saving or accuracy optimization. Compared to approaches that adapt at application-level or system-level only ALERT achieves more than 13% energy reduction, and 27% error reduction.

4.2 Design and Implementation

ALERT's runtime system navigates the large tradeoff space created by *combining* DNN-level and system-level adaptation. ALERT meets user-specified latency, accuracy, and energy constraints and optimization goals while accounting for run-time variations in environment or the goals themselves.

4.2.1 Inputs & Outputs of ALERT

ALERT's inputs are specifications about (1) the adaption options, including a set of DNN models $\mathbb{D} = \{d_i \mid i = 1 \cdots K\}$ and a set of system-resource settings, expressed as different power-caps $\mathbb{P} = \{P_j \mid j = 1 \cdots L\}$; and (2) the user-specified requirements on latency, accuracy, and energy usage, which can take the form of meeting constraints in any two of these three dimensions while optimizing the third. ALERT's output is the DNN model $d_i \in \mathbb{D}$ and the system-resource setting $p_j \in \mathbb{P}$ for the next inference-task input.

Formally, ALERT selects a DNN d_i and a system-resource setting p_j to fulfill *either* of these user-specified goals:[1]

Maximizing inference accuracy q (minimizing error) for an energy budget $\mathbf{E}_{\text{goal}}$ and inference deadline $\mathbf{T}_{\text{goal}}$:

$$\arg\max_{i,j} q_{i,j} \quad \text{s.t. } e_{i,j} \le \mathbf{E}_{\text{goal}} \wedge t_{i,j} \le \mathbf{T}_{\text{goal}} \tag{4.1}$$

Minimizing the energy use e for an accuracy goal $\mathbf{Q}_{\text{goal}}$ and inference deadline $\mathbf{T}_{\text{goal}}$.

$$\arg\min_{i,j} e_{i,j} \quad \text{s.t. } q_{i,j} \ge \mathbf{Q}_{\text{goal}} \wedge t_{i,j} \le \mathbf{T}_{goal} \tag{4.2}$$

Generality Along the DNN-adaptation side, the input DNN set can consist of any DNNs that offer different accuracy, latency, and energy tradeoffs; e.g., those in Figure **??**. In particular, ALERT can work with either or both of the broad classes of DNN adaptation approaches that have arisen recently, including: (1) traditional DNNs where the adaptation option should be selected prior to starting an inference task [36, 91, 121, 129, 35] and (2) anytime DNNs that produce a series of outputs as they execute [75, 120, 49, 78, 127, 48, 124]. These two classes are similar in that they both vary things like the network depth or width to create latency/accuracy tradeoffs.

On the system-resource side, ALERT uses a *power cap* as the proxy to system resource usage. Since both hardware [24] and software resource managers [134, 45, 23] can convert power budgets into optimal performance resource allocations, ALERT is compatible with many different schemes from both commercial products and the research literature.

4.2.2 ALERT Workflow

ALERT works as a feedback controller. It follows four steps to pick the DNN and resource settings for each input n:

1) Measurement. ALERT records the processing time, energy usage, and computes inference accuracy for $n-1$.

1. For space, we omit discussion of meeting energy and accuracy constraints while minimizing latency as it is a trivial extension of the discussed techniques and we believe it to be the least practically useful.

2) Goal adjustment. ALERT updates the time goal T_{goal} if necessary, considering the potential latency-requirement variation across inputs. In some inference tasks, a set of inputs share one combined requirement (e.g., in the NLP1 task in Table **??**, all the words in a sentence are processed by a DNN one by one and share one sentence-wise deadline) and hence delays in previous input processing could greatly shorten the available time for the next input [1, 68]. Additionally, ALERT sets the goal latency to compensate for its own, worst-case overhead so that ALERT itself will not cause violations.

3) Feedback-based estimation. ALERT computes the expected latency, accuracy, and energy consumption for every combination of DNN model and power setting.

4) Picking a configuration. ALERT feeds all the updated estimations of latency, accuracy, and energy into Eqs. 4.1 and 4.2, and gets the desired DNN model and power-cap setting for n.

The key task is step 3: the estimation needs to be accurate and fast. In the remainder of this section, we discuss key ideas and the exact algorithm of our feedback-based estimation.

4.2.3 Key Ideas of ALERT Estimation

Strawman Solving Eqs. 4.1 and 4.2 would be trivially easy if the deployment environment is guaranteed to match the training and profiling environment: we could estimate $t_{i,j}$ to be the average (or worst case, etc) inference time $t_{i,j}^{\text{prof}}$ over a set of profiling inputs under model d_i and power setting p_j. However, this approach does not work given the dynamic input, contention, and requirement variation.

Next, we present the key ideas behind how ALERT estimates the inference latency, accuracy, and energy consumption under model d_i and power setting p_j.

How to estimate the inference latency $t_{i,j}$? To handle the run-time variation, a potential solution is to apply an estimator, like a Kalman filter [84], to make dynamic predictions based on recent history about inferences under model d_i and power p_j. The problem is that most models and power settings will not have been picked recently and hence would have no recent history to feed into the estimator. This problem is a direct example of the challenge imposed by the large space of combined application and system options.

Idea 1: Handle the large selection space with a single scalar value. To make effective online estimation for *all* combinations of models and power settings, ALERT introduces a *global slow-down factor* ξ to capture how the current environment differs from the profiled environment (e.g., due to co-running processes, input variation, or other changes). Such an environmental slow-down factor is independent from individual model or power selection. It can fully leverage execution history, no matter which models and power settings were recently used; it can then be used to estimate $t_{i,j}$ based on $t_{i,j}^{\text{prof}}$ for all d_i and p_j combinations.

Applying a *global* slowdown factor for *all* combinations of application and system-level settings is crucial for ALERT to make quick decisions for every inference task. Although it is possible that some perturbations may lead to different slowdowns for different configurations, the slight loss of accuracy here is out-weighed by the benefit of having a simple mechanism that allows prediction even for configurations that have not been used recently.

This idea is also novel for ALERT, as previous cross-stack management systems all use much more complicated models to estimate and select different setting combinations (e.g., using model predictive control to estimate combinations of settings [86]). ALERT's global slowdown factor is based on several unique features of DNN families that accomplish the same task with different accurarcy/latency tradeoffs. We categorize these features as: (1) similarity of code paths and (2) proportionality of structure. The first is based on the observation that DNNs do not have complex conditional code dependences, so we do not need to worry about the case where different inputs would exercise very different code paths. Thus, what ALERT learns about latency, accuracy, and energy for one input will always inform it about future inputs. The second feature refers to the fact that as DNNs in a family scale in latency, the proportion of different operations tend to be similar, so what ALERT learns about one DNN in the family generally applies to other DNNs in the same family. These properties of DNNs do not hold for many other types of software, where different inputs or additional functionality can invoke entirely different code paths, with different resource requirements or responses.

How to estimate the accuracy under a deadline? Given a deadline $\mathbf{T}_{\text{goal}}$, the inference accuracy delivered by model d_i and power setting p_j is determined by three factors, as shown in Eq. 4.3: (1) whether the inference result, which takes time $t_{i,j}$, can be generated before the deadline $\mathbf{T}_{\text{goal}}$; (2) if yes, the accuracy is determined by the model d_i;[2] (3) if not, the accuracy drops to that offered by a backup result q_{fail}. For traditional DNN models, without any output at the deadline, a random guess will be used and q_{fail} will be much worse than q_i. For anytime DNN models that output multiple results as they are ready, the backup result is the latest output [75, 120, 49, 78, 127, 48, 124], which we discuss more in Section 4.2.5.

$$q_{i,j}[\mathbf{T}_{\text{goal}}] = \begin{cases} q_i & \text{, if } t_{i,j} \leq \mathbf{T}_{\text{goal}} \\ q_{\text{fail}} & \text{, otherwise} \end{cases} \tag{4.3}$$

A potential solution to estimate accuracy $q_{i,j}$ at the deadline $\mathbf{T}_{\text{goal}}$ is to simply feed the estimated $t_{i,j}$ into Eq. 4.3. However, this simple approach fails to account for two issues. First, while DNNs are generally well-behaved, significant tail effects are possible (see Figure **??**). Second, Eq. 4.3 is not linear, and is best understood as a step function, where a failure to complete inference by the deadline results in a worthless inference output (q_{fail}). Combined, these two issues mean that for tail inputs, inference will produce a worthless result; i.e., accuracy is not proportional to latency, but can easily fall to zero for tail inputs. The tail will, of course, be increased if there is any unexpected resource

2. Since it could be infeasible to calculate the exact inference accuracy at run time, ALERT uses the average training accuracy of the selected DNN model d_i, denoted as q_i, as the inference accuracy, as long as the inference computation finishes before the specified deadline.

contention. Therefore, the simple approach of using the mean latency prediction fails to account for the non-linear affects of latency on accuracy.

Idea 2: handle the runtime variation and account for tail behavior To handle the run-time variability mentioned in Section 4, ALERT treats the execution time $t_{i,j}$ and the global slow-down factor ξ as *random variables* drawn from a normal distribution. ALERT uses a recently proposed extension to the Kalman filter to adaptively update the noise covariance [2]. While this extension was originally proposed to produce better estimates of the mean, a novel approach in ALERT is using this covariance estimate as a measure of system volatility. ALERT uses this Kalman filter extension to predict not just the mean accuracy, but also the likelihood of meeting the accuracy requirements in the current operating environment. Section 4.4.3 shows the advantages of our extensions.

How to minimize energy or satisfy energy constraints? Minimizing energy or satisfying energy constraints is complicated, as the energy is related to, but cannot be easily calculated by, the complexity of the selected model d_i and the power cap p_j. As discussed in Section **??**, the energy consumption includes both that used during the inference under a given model d_i and that used during the inference-idle period, waiting for the next input. Consequently, it is not straightforward to decide which power setting to use.

Idea 3. ALERT leverages insights from previous research, which shows that energy for latency-constrained systems can be efficiently expressed as a mathematical optimization problem [71, 18, 76, 93]. These frameworks optimize energy by scheduling available configurations in time. Time is assigned to configurations so that the average performance hits the desired latency target and the overall energy (including idle energy) is minimal. The key is that while the configuration space is large, the number of constraints is small (typically just two). Thus, the number of configurations assigned a non-zero time is also small (equal to the number of constraints) [71]. Given this structure, the optimization problem can be solved using a binary search over available configurations, or even more efficiently with a hash table

[93].

The only difficulty applying prior work to ALERT is that work assumed there was only a single job running at a time, while ALERT assumes that other applications might contend for resources. Thus, ALERT cannot assume that there is a single system-idle state that will be used whenever the DNN is not executing. To address this challenge, ALERT continually estimates the system power when DNN inference is idle (but other non-inference tasks might be active), $p_{DNNidle}$. With this estimating DNN-idle power, Eq. 4.1 is transformed into:

$$\arg\max_{i,j} q_{i,j}[\mathbf{T}_{\text{goal}}] \quad \text{s.t. } p_{i,j} \cdot t_{i,j} + p_{DNNidle} \cdot t_{DNNidle} \leq \mathbf{E}_{\text{goal}} \tag{4.4}$$

4.2.4 ALERT Estimation Algorithm

Global Slow-down Factor ξ. As discussed in Idea-1, ALERT uses ξ to reflect how the run-time environment differs from the profiling environment. Conceptually, if the inference task under model d_i and power-cap p_j took time $t_{i,j}$ at run time and took $t_{i,j}^{\text{prof}}$ on average to finish during profiling, the corresponding ξ would be $t_{i,j}/t_{i,j}^{prof}$. ALERT estimates ξ using recent execution history under any model or power setting.

Specifically, after an input $n-1$, ALERT computes $\xi^{(n-1)}$ as the ratio of the observed time $t_{i,j}^{(n-1)}$ to the profiled time $t_{i,j}^{\text{prof}}$, and then uses a Kalman Filter[3] to estimate the mean $\mu^{(n)}$ and standard deviation $\sigma^{(n)}$ of $\xi^{(n)}$ at input n. ALERT's formulation is defined in Eq. 4.5, where $K^{(n)}$ is the Kalman gain variable; R is a constant reflecting the measurement noise; $Q^{(n)}$ is the process noise capped with $Q^{(0)}$. We set a forgetting factor of process variance $\alpha = 0.3$ [2]. ALERT initially sets $K^{(0)} = 0.5$, $R = 0.001$, $Q^{(0)} = 0.1$, $\mu^{(0)} = 1$, $\sigma^{(0)} = 0.1$, following the standard convention [84].

3. A Kalman Filter is an optimal estimator that assumes a normal distribution and estimates a varying quantity based on multiple potentially noisy observations [84].

$$\begin{cases} Q^{(n)} = \max\{Q^{(0)}, \alpha Q^{(n-1)} + (1\text{-}\alpha)(K^{(n-1)}y^{(n-1)})^2\} \\ K^{(n)} = \dfrac{(1-K^{(n-1)})\sigma^{(n-1)} + Q^{(n)}}{(1-K^{(n-1)})\sigma^{(n-1)} + Q^{(n)} + R} \\ y^{(n)} = t_{i,j}^{(n-1)}/t_{i,j}^{\text{prof}} - \mu^{(n-1)} \\ \mu^{(n)} = \mu^{(n-1)} + K^{(n)}y^{(n)} \\ \sigma^{(n)} = (1-K^{(n-1)})\sigma^{(n-1)} + Q^{(n)} \end{cases} \tag{4.5}$$

Then, using $\xi^{(n)}$, ALERT estimates the inference time of input n under any model d_i and power cap p_j: $t_{i,j}^{(n)} = \xi^{(n)} * t_{i,j}^{\text{prof}}$.

Accuracy. As discussed in Idea-2, ALERT computes the estimated inference accuracy $\hat{q}_{i,j}[\mathbf{T}_{\text{goal}}]$ by considering $t_{i,j}$ as a random variable that follows normal distribution with its mean and standard deviation computed based on that of ξ:

$$\begin{aligned} \hat{q}_{i,j}[\mathbf{T}_{goal}] &= E(q_{i,j}[\mathbf{T}_{goal}] \mid t_{i,j}^{(n)}) \\ &= E(q_{i,j}[\mathbf{T}_{goal}] \mid \xi^{(n)} \cdot t_{i,j}^{\text{prof}}) \\ \xi^{(n)} &\sim \mathcal{N}(\mu^{(n)}, (\sigma^{(n)})^2) \end{aligned} \tag{4.6}$$

Energy. As discussed in Idea-3, ALERT predicts energy consumption by separately estimating energy during (1) DNN execution: estimated by multiplying the power limit by the estimated latency and (2) between inference inputs: estimated based on the recent history of computer idle power using the Kalman Filter in Eq. 4.7. $\phi^{(n)}$ is the predicted DNN-idle power ratio, $M^{(n)}$ is process variance, S is process noise, V is measurement noise, and $W^{(n)}$ is the Kalman Filter gain. ALERT initially sets $M^{(0)} = 0.01$, $S = 0.0001$, $V = 0.001$.

$$\begin{cases} W^{(n)} = \dfrac{M^{(n-1)} + S}{M^{(n-1)} + S + V} \\ M^{(n)} = (1 - W^{(n)})(M^{(n-1)} + S) \\ \phi^{(n)} = \phi^{(n-1)} + W^{(n)}(p_{\text{idle}}/p_{i,j}^{(n-1)} - \phi^{(n-1)}) \end{cases} \tag{4.7}$$

ALERT then predicts the energy by Eq. 4.8.

$$e_{i,j}^{(n)} = p_{i,j} \cdot \xi^{(n)} \cdot t_{i,j}^{\text{prof}} + \phi^{(n)} \cdot p_{i,j} \cdot (\mathbf{T}_{goal} - (\xi^{(n)} \cdot t_{i,j}^{\text{prof}})) \tag{4.8}$$

4.2.5 Integrating ALERT with Anytime DNNs

An anytime DNN is an inference model that outputs a series of increasingly accurate inference results—o_1, o_2, ... o_k, with o_t more reliable than o_{t-1}. A variety of recent works [75, 120, 49, 78, 127, 124] have proposed DNNs supporting anytime inference, covering a variety of problem domains. ALERT easily works with not only traditional DNNs but also Anytime DNNs. The only change is that q_{fail} in Eq. 4.3 no longer corresponds to a random guess. That is, when the inference could not generate its final result o_k by the deadline $\mathbf{T}_{\text{goal}}$, an earlier result o_x can be used with a much better accuracy than that of a random guess. The updated accuracy equation is below:

$$q_{\cdot,j} = \begin{cases} q_k & \text{, if } t_{k,j} \leq \mathbf{t}_{\text{goal}} \\ q_{k-1} & \text{, if } t_{k-1,j} \leq \mathbf{t}_{\text{goal}} < t_{k,j} \\ & \quad \ldots \\ q_{\text{fail}} & \text{, otherwise} \end{cases} \tag{4.9}$$

Existing anytime DNNs consider latency but not energy constraints—an anytime DNN will keep running until the latency deadline arrives and the last output will be delivered to the user. ALERT naturally improves Anytime DNN energy efficiency, stopping the inference sometimes before the deadline based on its estimation to meet not only latency and accuracy, but also energy requirements.

Furthermore, ALERT can work with a set of traditional DNNs and an Anytime DNN together to achieve the best combined result. The reason is that Anytime DNNs generally sacrifice accuracy for flexibility. When we feed a group of traditional DNNs and one Anytime DNN to construct the candidacy set $\mathbb{D}$, with Eq. 4.6, ALERT naturally selects the Anytime DNN when the environment is changing rapidly (because the expected accuracy of an anytime DNN will be higher given that variance), and the regular DNN, which has slightly higher accuracy with similar computation, when it is stable, getting the best of both worlds.

In our evaluation, we will use the nested design from [124], which provides a generic coverage of anytime DNNs.

4.2.6 Limitations of ALERT

ALERT's prediction, particularly the Kalman Filter, relies on the feedback from recent input processing. Consequently, it requires at least one input to react to sudden changes. Additionally, the Kalman filter formulations assume that the underlying distributions are normal, which may not hold in practice. If the behavior is not Gaussian, the Kalman filter will produce bad estimations for some amount of time. Fortunately, after 2–3 such bad predictions, the estimated variance will increase, which will then trigger ALERT to pick anytime over traditional DNNs or pick a low-latency traditional DNN over high-latency ones, because the former has a better chance to produce results at latency deadlines and hence a higher expected accuracy under high variance. So—worst case—ALERT will choose a DNN with slightly less accuracy than what could have been used with the right model of randomness. We evaluate ALERT's ability to handle non-normal distributions in Section 4.4.3.

ALERT provides probabilistic, not hard, guarantees. As ALERT estimates not just average timing, but the distributions of possible timings, it can provide arbitrarily many nines of assurance that it will meet latency or accuracy goals; e.g., scheduling for slow-down factors of up to three standard deviations corresponds to meeting the goals 99.7% of the time. Providing 100% guarantees, however, requires much more conservative configuration selection—hurting both energy and accuracy—a property shared by all systems that have to choose between probabilistic and hard guarantees [15].

How the inference behaves ultimately depends not only on ALERT, but also on the DNN models and system-resource setting options. As we will evaluate in Section 4.4, ALERT helps make the best use of supplied DNN models, but does not eliminate the difference between different DNN models.

4.3 Implementation

We implement ALERT for both CPUs and GPUs. On CPUs, ALERT adjusts power through Intel's RAPL interface [24], which allows software to set a hardware power limit. On GPUs, ALERT uses PyNVML to control frequency and builds a power-frequency lookup table. ALERT can also be applied to other approaches that translate power limits into settings for combinations of resources [45, 52, 23, 134].

In our experiments, ALERT considers a series of power settings within the feasible range with 2.5W interval on our test laptop and a 5W interval on our test CPU server and GPU platform, as the latter has a wider power range than the former. The number of power buckets is configurable.

ALERT incurs small overhead in both scheduler computation and switching from one DNN/power-setting to another, just 0.6–1.7% of an input inference time. We explicitly account for overhead by subtracting it from the user-specified goal (see step 2 in Section 4.2.2).

Users may set goals that are not achievable. If ALERT cannot meet all constraints, it prioritizes latency highest, then accuracy, then power. This hierarchy is configurable.

4.4 Results

We apply ALERT to different inference tasks on both CPU and GPU with and without resource contention from co-located jobs. We set ALERT to (1) reduce energy while satisfying latency and accuracy requirements and (2) reduce error rates while satisfying latency and energy requirements. We compare ALERT with both oracle and state-of-the-art schemes and evaluate detailed design decisions.

	Run-time environment setting		
Default	Inference task has no co-running process		
Memory	Co-locate with memory-hungry STREAM [90] (@CPU)		
	Co-locate with Backprop from Rodinia-3.1 [21] (@GPU)		
Compute	Co-locate with Bodytrack from PARSEC-3.0 [11] (@CPU)		
	Co-locate with the forward pass of Backprop [21] (@GPU)		
	Ranges of constraint setting		
Latency	0.4x–2x mean latency* of the largest Anytime DNN		
Accuracy	Whole range achievable by trad. and Anytime DNN		
Energy	Whole feasible power-cap ranges on the machine		
Task	Trad. DNN	Anytime [124]	Fixed deadline?
Image Classifi.	Sparse ResNet	Depth-Nest	Yes
Sentence Pred.	RNN	Width-Nest	No
Scheme ID	DNN selection		Power selection
Oracle	Dynamic optimal		Dynamic optimal
Oracle$_{\text{Static}}$	Static optimal		Static optimal
App-only	One Anytime DNN		System Default
Sys-only	Fastest traditional DNN		State-of-Art[?]
No-coord	Anytime DNN w/o coord. with Power		State-of-Art[?]
ALERT	ALERT default		ALERT default
ALERTAny	ALERT w/o traditional DNNs		ALERT default
ALERTTrad	ALERT w/o Anytime DNNs		ALERT default

Table 4.1: Settings and schemes under evaluation (* measured under default setting without resource contention)

Plat.	DNN	Work.	ALERT	ALERT Any	Sys-only	App-only	No-coord	Oracle	ALERT	ALERT Any	Sys-only	App-only	No-coord	Oracle
			Energy in Minimizing Energy Task						Error Rate in Minimizing Error Task					
CPU1	Sparse Resnet	Idle	0.64	0.68	1.08^{19}	1.19	0.94^{1}	0.64	0.91	0.92	1.35	1.02^{3}	0.91^{3}	0.89
		Comp.	0.57	0.58	0.80^{19}	1.30	1.39^{1}	0.57	0.38	0.39	0.51	1.35^{24}	0.39^{6}	0.36
		Mem.	0.53	0.55	0.76^{19}	1.43	1.37^{2}	0.53	0.34	0.34	0.46	1.47^{28}	0.39^{2}	0.33
	RNN	Idle	0.61	0.65	1.01^{30}	1.34	0.95^{2}	0.61	0.87	0.87	0.87	0.87^{21}	0.87^{14}	0.86
		Comp.	0.60	0.57	0.93^{30}	1.21	1.26^{5}	0.60	0.42	0.44	0.50	0.46^{28}	0.46^{23}	0.42
		Mem.	0.54	0.56	0.95^{31}	1.45	1.24^{9}	0.54	0.45	0.45	0.50	0.57^{28}	0.54^{27}	0.44
CPU2	Sparse Resnet	Idle	0.93	0.88	0.96^{20}	0.99	1.18	0.91	0.68	0.68	0.97	0.79^{2}	0.71^{24}	0.66
		Comp.	0.59	0.57	0.60^{23}	1.00	1.01	0.58	0.58	0.57	0.85	0.74^{16}	0.71^{29}	0.55
		Mem.	0.38	0.37	0.39^{19}	0.65	0.63^{13}	0.38	0.24	0.82	0.32	0.33^{17}	0.75^{31}	0.21
	RNN	Idle	0.87	0.99	0.80^{34}	1.04	1.00^{6}	0.83	0.84	0.85	0.99	0.89^{14}	0.89^{1}	0.84
		Comp.	0.60	0.60	0.55^{34}	0.99	0.86^{7}	0.60	0.51	0.52	0.60	0.53^{21}	0.54^{17}	0.52
		Mem.	0.52	0.51	0.43^{33}	0.70	0.85^{14}	0.52	0.26	0.27	0.31	0.28^{21}	0.27^{17}	0.26
GPU	Sparse Resnet	Idle	0.97	0.99	0.92^{20}	1.36	1.37	0.92	0.90	0.92	1.22	1.09^{2}	1.74^{12}	0.86
		Comp.	0.96	0.97	0.94^{20}	1.66	1.77	0.89	0.32	0.34	1.28	1.21^{23}	2.50^{18}	0.30
		Mem.	0.97	1.01	0.91^{20}	1.39	1.43	0.91	0.89	0.92	1.22	1.11^{2}	1.81^{14}	0.86
Harmonic mean			0.64	0.64	0.73^{27}	1.11	1.08^{4}	0.62	0.46	0.47	0.63	0.67^{16}	0.63^{15}	0.45

Table 4.2: Average energy consumption and error rate normalized to *Oracle*$_{\text{Static}}$, smaller is better. (Each cell is averaged over 35–40 constraint settings; superscript: # of constraint settings violated for >10% inputs and hence excluded from energy average.)

4.4.1 Methodology

Experimental setup. We use the three platforms listed in Table **??**: *CPU1*, *CPU2*, and *GPU*. On each, we run inference tasks[4], image classification and sentence prediction, under three different resource-contention scenarios: (1) no contention (Default); (2) contention with a computation-intensive job (Memory); (3) contention with a memory-intensive job (Compute). We then evaluate a number of management schemes' ability to meet latency, accuracy, and energy constraints. Table 4.1 lists the details.

Schemes under evaluation. We give ALERT three different DNN sets: traditional DNN models (ALERTTrad), an Anytime DNN (ALERTAny), and both (ALERT).

We compare with two $Oracle_*$ schemes that have perfect predictions for every input under every DNN/power setting (i.e., impractical). The "Oracle" allows DNN/power settings to change across inputs, representing the best possible results; the "$\text{Oracle}_{\text{Static}}$" has one fixed setting across inputs, representing the best results without dynamic adaptation.

Finally, we compare with three state-of-the-art approaches: the "App-only" conducts adaptation only at the application level through an Anytime DNN [124]; the "Sys-only" conducts adaptation only at the system level following an existing resource-management system that minimizes energy under soft real-time constraints [93][5] and uses the fastest candidate DNN to avoid latency violations; the "No-coord" uses both the Anytime DNN for application-level adaptation *and* the power-management scheme [93] to adapt power, but with these two working independently.

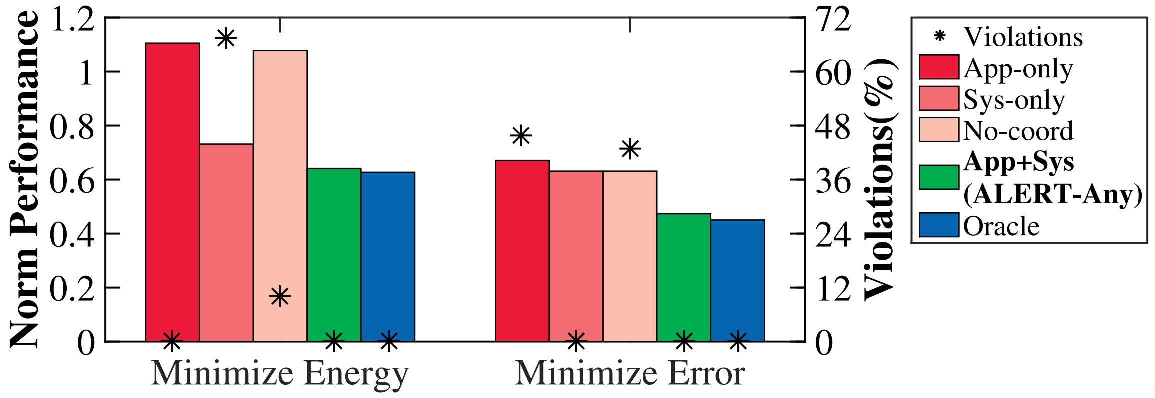


Figure 4.2: Result Summary: average performance normalized to Oracle$_{\text{Static}}$ (Smaller is better; Details in Table 4.2)

4.4.2 Overall Results

Table 4.2 shows the results for all schemes for different tasks on different platforms and environments. Each cell shows the average energy or accuracy under 35–40 combinations of latency, accuracy, and energy constraints (the settings are detailed in Table 4.1), normalized to the Oracle$_{\text{Static}}$ result. Figure 4.2 compares these results, where lower bars represent better results and lower *s represent fewer constraint violations. ALERT and ALERT$_{\text{Any}}$ both work very well for all settings. They outperform state-of-the-art approaches, which have a significant number of constraint violations, as visualized by the many superscripts in Table 4.2 and the high * positions in Figure 4.2. ALERT outperforms Oracle$_{\text{Static}}$ because it adapts to dynamic variations. ALERT also comes very close to the theoretically optimal Oracle.

Comparing with Oracles. As shown in Table 4.2, ALERT achieves 93-99% of Oracle's energy and accuracy optimization while satisfying constraints. Oracle$_{\text{static}}$, the baseline in Table 4.2, represents the best one can achieve by selecting 1 DNN model and 1 power setting for all inputs. ALERT greatly out-performs Oracle$_{\text{static}}$, reducing its energy consumption

4. For GPU, we only run image classification task there, as the RNN-based sentence prediction task is better suited for CPU [135].

5. Specifically, this adaptation uses a feedback scheduler that predicts inference latency based on Kalman Filter.

by 3–48% while satisfying accuracy constraints (36% in harmonic mean) and reducing its error rate by 9-66% while satisfying energy constraints (54% in harmonic mean).

Figure **??** shows a detailed comparison for the energy minimization task. The figure shows the range of performance under all requirement settings (i.e., the whiskers). ALERT not only achieves similar mean energy reduction, its whole range of optimization behavior is also similar to Oracle. In comparison, $\text{Oracle}_{\text{Static}}$ not only has the worst mean but also the worst tail performance. Due to space constraints, we omit the figures for other settings, where similar trends hold.

ALERT has more advantage over $\text{Oracle}_{\text{static}}$ on CPUs than on GPUs. The CPUs have more empirical variance than the GPU, so they benefit more from dynamic adaptation. The GPU experiences significantly lower dynamic fluctuation so the static oracle makes good predictions.

ALERT satisfies the constraint in 99.9% of tests for image classification and 98.5% of those for sentence prediction. For the latter, due to the large input variability (NLP1 in Figure **??**), some input sentences simply cannot complete by the deadline even with the fastest DNN. There the Oracle fails, too.

Note that, these Oracle schemes not only have perfect—and hence, impractical—prediction capability, but they also have no overhead. In contrast, ALERT is running on the same machines as the DNN workloads. *All results include ALERT's run-time latency and power overhead.*

Comparing with State-of-the-Art. For a fair comparison, we focus on ALERTAny, as it uses exactly the same DNN candidate set as "Sys-only", "App-only", and "No-coord". Across all settings, ALERTAny outperforms the others.

The System-only solution suffers from not being able to choose different DNNs under different runtime scenarios. As a result, it performs much worse than ALERTAny in satisfying accuracy requirements or optimizing accuracy. For the former (left side of Table 4.2 and

Plat.	Work.	ALERT	Any	Trad	ALERT	Any	Trad
		Minimize Energy Task			Minimize Error Task		
CPU1	Idle	0.64	0.68	0.65^1	0.91	0.92	0.93
	Comp.	0.57	0.58	0.65^6	0.38	0.39	0.41
	Mem.	0.53	0.55	0.53^3	0.34	0.34	0.35
CPU2	Idle	0.93	0.88	0.95^1	0.68	0.68	0.69
	Comp.	0.59	0.57	0.60^4	0.58	0.57	0.59
	Mem.	0.38	0.37	0.40^8	0.23	0.24	0.32
GPU	Idle	0.97	0.99	0.95	0.90	0.92	0.89
	Comp.	0.97	1.01	0.96	0.89	0.92	0.89
	Mem.	0.96	0.97	0.95	0.32	0.34	0.32
Harmonic mean		0.66	0.66	0.67^3	0.47	0.48	0.50

Table 4.3: ALERT normalized average energy consumption and error rate to $Oracle_{\text{Static}}$ @ Sparse ResNet (Smaller is better)

Figure 4.2), it creates accuracy violations in 68% of the settings as shown in Figure 4.2; for the latter (right side of Table 4.2 and Figure 4.2), although capable of satisfying energy constraints, it introduces 34% more error than ALERTAny.

The Application-only solution suffers from not being able to adjust to the energy requirements. As a result, it consumes 73% more energy in energy-minimizing tasks (left side of Table 4.2 and Figure 4.2) and introduces many energy-budget violations particularly under resource contention settings (right side of Table 4.2 and Figure 4.2).

The no-coordination scheme is worse than both System- and Application-only. It violates constraints in both tasks with 69% more energy and 34% more error than ALERTAny. Without coordination, the two levels can work at cross purposes; e.g., the application switches to a faster DNN to save energy while the system makes more power available.

4.4.3 Detailed Results and Sensitivity

Different DNN candidate sets. Table 4.3 compares the performance of ALERT working with an Anytime DNN (Any), a set of traditional DNN models (Trad), and both. At a high level, ALERT works well with all three DNN sets. Under close comparison, ALERTTrad violates more accuracy constraints than the others, particularly under resource contention on CPUs, because a traditional DNN has a much larger accuracy drop than an anytime

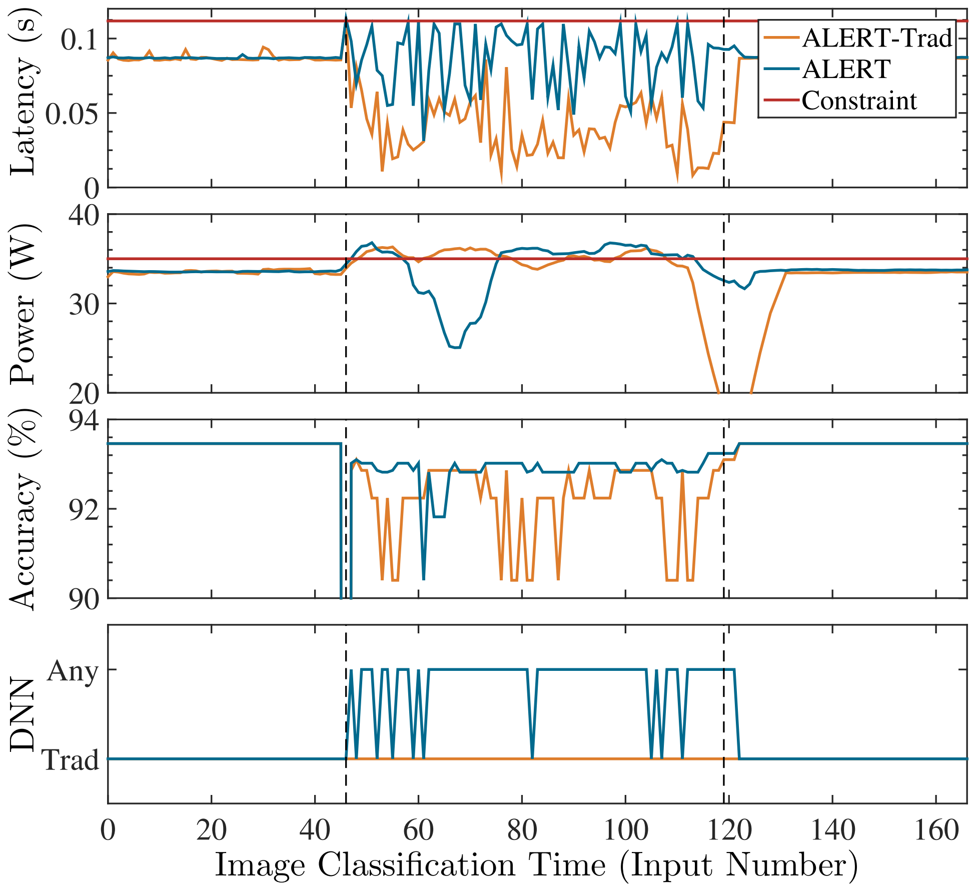


Figure 4.3: Minimize error rates w/ latency, energy constraints @ CPU1. ALERT in blue; $\text{ALERT}_{\text{Trad}}$ in orange; constraints in red. Memory contention occurs from about input 46 to 119; Deadline: 1.25× mean latency of largest Anytime DNN in Default; power limit: 35W.

DNN when missing a latency deadline. Consequently, when the system variation is large, ALERTTrad selects a faster DNN to meet latency and thus may not meet accuracy goals. Of course, ALERTAny is not always the best. As discussed in Section 4.2.5, Anytime DNNs sometimes have lower accuracy then a traditional DNN with similar execution time. This difference leads to the slightly better results for ALERT over ALERTAny.

Figure 4.3 visualizes the different dynamic behavior of ALERT (blue curve) and $\text{ALERT}_{\text{Trad}}$ (orange curve) when the environment changes from Default to Memory-intensive and back. At the beginning, due to a loose latency constraint, ALERT and $\text{ALERT}_{\text{Trad}}$ both select the biggest traditional DNN, which provides the highest accuracy within the energy budget. When the memory contention suddenly starts, this DNN choice leads to a deadline miss and an energy-budget violation (as the idle period disappeared), which causes an accuracy dip. Fortunately, both quickly detect this problem and sense the high variability in the expected latency. ALERT switches to use an anytime DNN and a lower power cap. This switch is effective: although the environment is still unstable, the inference accuracy remains high, with slight ups and downs depending on which anytime output finished before the deadline. Only able to choose from traditional DNNs, $\text{ALERT}_{\text{Trad}}$ conservatively switches to much simpler and hence lower-accuracy DNNs to avoid deadline misses. This switch does eliminate deadline misses under the highly dynamic environment, but many of the conservatively chosen DNNs finish before the deadline (see the Latency panel), wasting the opportunity to produce more accurate results and causing $\text{ALERT}_{\text{Trad}}$ to have a lower accuracy than ALERT. When the system quiesces, both schemes quickly shift back to the highest-accuracy, traditional DNN.

Overall, these results demonstrate how ALERT always makes use of the full potential of the DNN candidate set to optimize performance and satisfy constraints.

Sensitivity to latency distribution. ALERT assumes a Gaussian distribution. However, ALERT is still robust for other distributions, as explained in Section 4.2.6. As shown in Figure **??**, the observed ξs (red bars) are indeed not a perfect fit for Gaussian distribution (blue lines) in all scenarios, which confirms ALERT's robustness.

4.5 Conclusion

This paper demonstrates the challenges behind the important problem of ensuring timely, accurate, and energy efficient neural network inference with dynamic input, contention, and requirement variation. ALERT achieves these goals through dynamic and coordinated DNN model selection and power management based on feedback control. We evaluate ALERT with a variety of workloads and DNN models and achieve high performance and energy efficiency.

CHAPTER 5
SIM: RUNTIME MANAGEMENT FOR ANYTIME NEURAL NETWORK IN SHARED SENSING INFRASTRUCTURE

5.1 Overview

Remote sensing is the process of gathering data from a distance. Shared sensing infrastructure (SSI) is an emerging class of remote sensing system that allows multiple, independent stakeholders to deploy their workload and share the sensors and associated compute. For example, the Array of Things (AoT) provides a shared sensing infrastructure for scientists to measure urban and environmental phenomena in the city of Chicago [19]. The National Ecological Observatory Network (NEON) provides a shared sensing infrastructure allowing scientists to collect ecological data from remote sites around the world [42]. Due to both privacy concerns [37, 89] and bandwidth limitations [9], the raw, sensed data cannot leave the SSI. Consequently, the sensor data must be processed locally, typically with some sort of machine learning to detect the phenomena of interest from within the raw data. The output of these machine learning algorithms are then communicated from the sensor.

Like many embedded workloads, SSI workloads process sensor data under latency constraints to avoid missing important data. For example, a meteorologist using NEON would want to detect lightning phenomena that last for only 20-93 ms [88, 31]. Additionally, the inference must be done with minimal error. Workloads like a gunshot detector [96] and a forest fire detector [74] should avoid making a false alarms to call the emergency response unit while avoiding false negatives that would fail to detect these events. While meeting latency constraints is well-studied in single-stake holder systems. The multitenancy in SSI adds novel system dynamics because the stakeholders can come and go. For example, a car detection workload might only run during rush hour [56] and flood detection might only run when there is a rain [98].

Designing an SSI that allows multiple stakeholders to do in-sensor inference while meeting latency requirements and minimizing error prediction has 2 main challenges. The first is *dynamics*: the SSI should ensure that the workloads still meet latency constraint even when the computing environment is changing; e.g., a new stakeholder deploys an additional workload on the SSI. The second is *stakeholder behavior*: when stakeholders have the freedom to choose how to deploy their workload, they can either (1) launch a greedy workload that consumes most resources and inhibits other workloads or (2) deploy a workload that consumes just enough resources such that all other workloads can meet their latency requirements with minimal error. Ideally, the SSI should prevent greedy behavior and incentivize users who cooperate by helping the system meet latency constraints while minimizing the overall inference error. In summary, SSI's unique combination of embedded requirements with multi-tenancy means that a novel solution is required to address these challenges.

A promising prior approach to handling the dynamics challenge is to deploy *anytime* machine learning to process sensor data. Anytime machine learning return a sequence of results over time, the more time they are given the lower the error [83, 3]. This behavior makes anytime machine learning a **flexible** application because it can trade the runtime latency with error output. Thus, anytime machine learning are a good match for meeting latency constraints in systems with unpredictable dynamics, provided that all the workloads are deployed by a single stakeholder. If there is just one stakeholder, then it is in their interest to carefully design the system to minimize the overall error [47, 126]. Recent work pairs anytime machine learning (for neural networks) with timing based scheduling (such as scheduling for the worst case execution time of every workload's earliest output) [130, 6]. This approach has two problems when deployed on SSI. First, it fails to minimize error because it only ensures schedulability of the highest error output for each workload. Second, and perhaps more importantly for SSI, if a workload submits an incorrect timing and error profile, then that workload can effectively steal resources from other stakeholders, decreasing its error while increasing the error of cooperative workloads. **Cooperative** workloads in this case is a flexible anytime machine learning that allows the scheduler to limit their incremental output such that the scheduler can ensure other application to meet their latency deadline.

Another approach would be to use anytime machine learning with the type of fair resource allocation found on typical multi-tenant server-class systems [39, 8]. While this approach prevents any workload from accessing more resources, it fails to minimize inference error. The problem is that fair schedulers assume that any additional resource can be put to beneficial use by the workload to which it is allocated. When systems are judged by throughput, this assumption holds. However, for SSI, just increasing throughput is not sufficient: workloads must meet a latency deadline and if latency deadlines are met, the system will be judged by overall error. The problem with fairly allocating resources to anytime inference machine learning is that they are discrete and non-linear. Thus, it is not always the case that more resources reduces error for an anytime workload. In practice, fair resource allocation may not be enough resources for some workloads to meet their latency deadlines, while others might have more resources than they can use to reduce error (i.e., they hit a point of diminishing returns.

This paper proposes SIM, a runtime manager for anytime machine learning in shared sensing infrastructure, to meet latency requirements while incentivizing flexibility and cooperation. **Flexibility** is the ability to trade runtime latency and output quality, while **cooperation** is the decision to let the scheduler limit their incremental output. We differ from the previous approach in multiple aspects. First, instead of scheduling based on a fixed runtime profile, SIM updates the time profile online, thus giving better results because the application adapts to the dynamic environment. Second, unlike prior approaches that punish flexibility and cooperation, SIM incentivizes flexibility by auctioning resources only to flexible and cooperative ANN. SIM also incentivizes cooperation by protecting them from a greedy applications. In each iteration, SIM monitors the resource usage and output of applications and compares it with the most greedy application. Envy happens if the cooperative application can improve the output by swapping the resource with a greedy application. SIM will resolve the envy by setting a new, improved output goal for the flexible application. SIM then monitors the feedback. If the output goal cannot be met, SIM enforces the greedy application to adapt. If the greedy application cannot adapt, it will be kicked from SSI because the application is not flexible enough.

We evaluate our proposed solution in GPU using different kinds of Anytime Neural Networks. Our evaluations show that we incentivize application flexibility by decreasing the error prediction of a flexible anytime neural network by 20% on average compared to an approach that is not aware of untruthful stakeholders. In the scenario where all stakeholder is cooperative and deploys the flexible application, SIM decreases the error prediction by 21% compared to an approach that schedules based on fixed profile time and by 16% compared to a fair scheduler. Our solution only differs from the optimal solution by 5%.

In summary, we provide the following contributions:

1. We describe the potential problem in SSI, where the current state-of-the-art runtime manager mismanages Anytime machine learning workload. It can lead to higher error prediction, and an untruthful stakeholder can exploit the SSI.

2. We present SIM, an Anytime workload runtime management that meets latency deadlines and incentivizes the flexibility and cooperation between stakeholders.

5.2 Motivation

This section demonstrates how a single stakeholder can use Anytime machine learning to handle dynamics. Then we describe the challenge of deploying the Anytime machine learning in a multistakeholder scenario.

5.2.1 Handling dynamics in single stakeholder scenario

In an embedded multitenancy system, the number of applications that need to use the computation simultaneously can change dynamically. A stakeholder who wants their application to meet latency deadlines must ensure that the embedded system can provide enough resources. Since an embedded system has a fixed amount of resources, the application needs to be flexible to trade runtime latency and accuracy to meet the latency deadline with dynamic resource allocation. This single stakeholder scenario guarantees the flexibility and cooperation between applications. The stakeholder can design which application to be flexible and which application needs to cooperate to ensure all applications meet their latency deadlines.

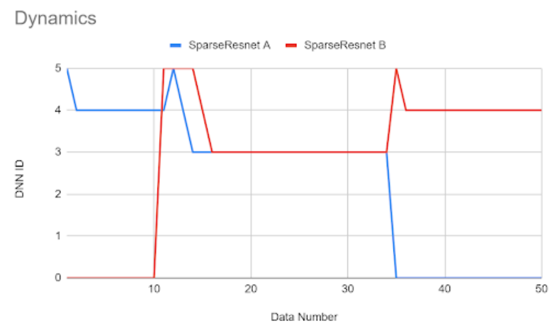


Figure 5.1: Using ANN to handle multitenancy

Figure 5.1 shows how the Anytime machine learning (AML) adapts to dynamics in a scenario where all applications are designed and deployed by the same stakeholder. In the beginning, AML ran alone and could produce a low error by improving its output until all the timeshares were consumed. Then, a co-located application is deployed that interferes with the AML. Suddenly both AMLs have to share the timeshare. The stakeholder has to limit both AML incremental output so that there is enough timeshare to handle both AMLs. Now, both AMLs run with a higher error output but less runtime latency. When an application is no longer running, the stakeholder reallocates the slack timeshare to the running AML such that it can produce a lower error.

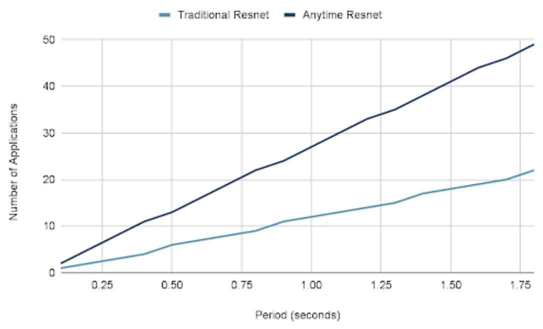


Figure 5.2: Schedulability Analysis of Anytime Neural Network vs Traditional Neural Network

The deployment of the Anytime algorithm benefits the SSI as its success is determined by how many sciences it can provide [19]. Figure 5.2 shows the number of applications SSI can guarantee to meet their deadlines across different periods. The flexible AML can trade their latency with error output. Therefore, the SSI can schedule them such that all of them meet their latency deadline. Meanwhile, the traditional DNN only produces one output with low error and high runtime latency. The scheduler cannot adjust traditional DNN runtime latency and can only guarantee the latency deadlines of fewer applications.

5.2.2 Handling dynamics in multistakeholder scenario

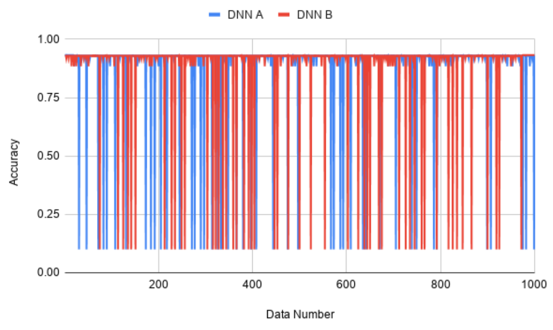


Figure 5.3: Two Flexible DNN without Coordination

Even though the AML can adapt to the dynamic environment, in multi-stakeholder scenarios, AMLs need to cooperate such that AMLs can meet latency deadlines. When multiple stakeholders run together in SSI, they will compete for resources. Figure 5.3 shows a scenario where two stakeholders deploy Anytime Cifar Spareseresnet DNNs [137]. These are image classification neural networks trained using CIFAR dataset and can incrementally improve their output quality. Each stakeholder controls their AML, but they do not cooperate. Both AMLs are blocking each other's resources, which eventually makes them miss many deadlines. Figure 5.4 shows the advantages of cooperation. Here a scheduler makes the AML cooperate by limiting the incremental output of each AML.

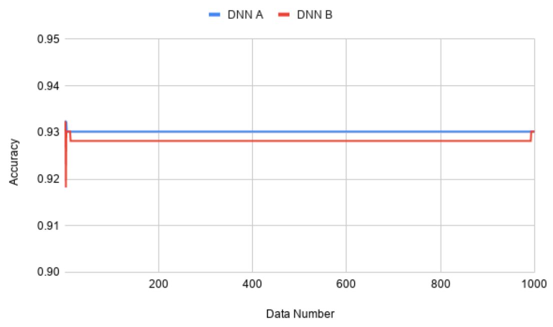


Figure 5.4: Two Flexible DNN with Coordination

5.2.3 Challenge to handling dynamics in multistakeholder scenario

There are prior approaches to meeting the latency deadlines of applications by limiting or early exiting the AML. However, those schedulers assume that all applications are designed and deployed by a single stakeholder. In SSI, applications come from different stakeholders, and it is no longer reasonable to assume that all stakeholders would be cooperative. An AML can be **untruthful** such that they are flexible but do not want to cooperate. This untruthful stakeholder can exploit the scheduler to increase their intent while sacrificing other stakeholders that have truthful behavior. **Truthful** behavior is when the application is flexible and wants to cooperate.

There is recent work in the embedded system to meet latency deadlines of AMLs [132]. The scheduler uses dynamic programming to minimize the error output given the information of WCET and the expected error output of each incremental output of the AML. Application with low error value will have high priority as it decreases the overall error of the system. If a single stakeholder designs and deploys all applications, they can validate the application's error value. However, there is no way to validate the error value in a multi-stakeholder scenario. A stakeholder with malicious intent can self-declare their application error output as low as possible and get high priority during the deployment. **For the rest of the paper, we call a stakeholder with malicious intent an untruthful stakeholder**. The untruthful stakeholder exploits the SSI by either lying about their error output value (see figure 5.5) or by hiding their flexible configuration (see figure 5.6).

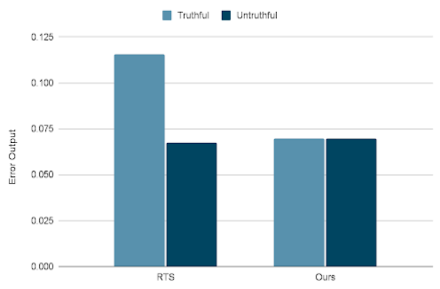


Figure 5.5: Comparison of Embedded Scheduling vs Ours

Figure 5.5 shows how the embedded scheduler works in SSI when a cooperative stakeholder is running together with an untruthful stakeholder. The untruthful stakeholder exploits the SSI by declaring a deficient error output for each of their incremental output. Since the untruthful application has a lower error, the embedded scheduler will limit the incremental output of the cooperative application to make room for the untruthful application. Then the cooperative stakeholder would run in the high error output.

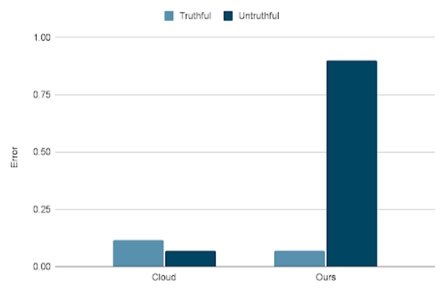


Figure 5.6: Comparison of Embedded Scheduling vs Ours

Figure 5.6 shows that the prior scheduler and resource allocation methods disincentivize a cooperative deployment. The error output of the flexible application is consistently lower than the similar application that does not cooperate. Here, the untruthful stakeholder did not declare their flexible configuration and forced the scheduler to configure the AML only. Meanwhile, our potential solution incentivizes a cooperative flexible application deployment, and the error of a flexible application is lower than the inflexible one.

Incentivizing the deployment of the cooperative application is essential. We can summarize that the SSI has the highest utility when the stakeholders deploy their application as a truthful flexible application. SSI can maximize the number of applications running while meeting their latency deadline. However, the best deployment strategy for the stakeholder is to deploy the application as untruthful by either hiding their flexible information or falsifying their error output. Disincentivizing flexibility will make all stakeholder deploy their application untruthfully.

5.3 Preliminaries

This section describes Anytime machine learning (AML), an analysis showing why prior approaches are not suitable to manage AML in terms of Pareto optimality and envy-freedom. Finally, we summarized this analysis into the design principles for SIM.

5.3.1 Anytime Neural Network

Anytime machine learning (AML) uses principles from incremental algorithms, and approximate computing [?]. AML provides multiple outputs in multiple time increments. In other words, an anytime machine learning system produces a series of increasingly accurate inference results—the more time available, the more accurate the result.

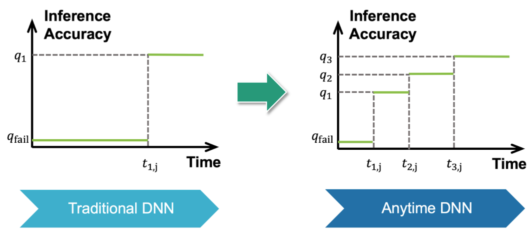


Figure 5.7: Traditional and Anytime DNN

Figure 5.7 shows the differences between a traditional neural network and an anytime neural network (AML). AML is a neural network that produces a series of increasingly accurate outputs over time. In contrast, a traditional neural network is a neural network that only produces one output at the end of time. AML is robust and can adapt to system dynamics. The earlier, less accurate result will be used when the inference cannot produce the output by the deadline. Therefore, scheduling AML by its worst-case execution time (WCET) while providing more protection in case the WCET calculation is inaccurate eliminates the potential improvement of the output quality when the runtime latency is faster than the WCET. It is better to schedule AML by using a dynamic time profile.

AML's incremental output is discrete and nonlinear. Scheduling AML by allocating the time resources would waste unnecessary timeshare. The error output of AML would not decrease unless the additional timeshare is enough to produce the subsequent incremental output. Therefore it is more efficient to schedule AML by limiting its incremental output rather than scheduling them by time resources.

5.3.2 Managing Anytime Neural Networks

An AML would want to minimize the error output by using as many resources as possible. Therefore, the deployment of multiple AMLs together requires cooperation between the AMLs. A scheduler is needed to control AMLs such that they are not competing for the resources. Meanwhile, cooperation is needed such that the scheduler has something to be controlled. AML is cooperative if they provide their outputs to the scheduler representing their error-accuracy tradeoff and letting the scheduler limit their incremental output.

Pareto optimal is a condition where no stakeholder's application can decrease their error output without worsening the other error output. Meanwhile, envy-free is a situation where stakeholders cannot decrease their errors when they swap each other's resources. Scheduling in multitenancy would aim to achieve both Pareto optimal and envy-free as a desirable condition [**?**]. There are multiple ways to manage AMLs to meet latency requirements. Here, we will analyze prior approaches used in embedded systems to meet latency deadlines in terms of Pareto optimality and envy-freedom.

Managing by Profile

A real-time scheduler solves the problem of managing multiple applications to meet their deadline. It makes sure that applications pass the schedulability analysis:

$$\sum_{i=0}^{n} \frac{C_i}{T_i} \leq 1 \tag{5.1}$$

C_i is the worst-case execution time (WCET) for application i. T_i is deadline of application i. Based on equation 5.1, the ratio between worst-case running time and the deadline should not be higher than one. If the sum is less than 1, the system has enough timeshare to run all of the applications. If it is higher than 1, then it means that the system is overloaded, and adjustment is needed. The value of C_i needs to be adjusted. The approximate algorithm gives the scheduler ability to adjust the C_i.

Prior approaches manage the flexibility of AML by limiting the output they produce. They decide when AML should do early exiting such that the worst-case execution time to do early exiting C_i is schedulable for the system. An embedded system's specific scheduler goal is to meet latency deadlines while minimizing error output. The developer assigns each application's incremental output with an error value. Then a management mechanism such as reinforcement learning [132] and probabilistic scheduling [126] would be embedded in the scheduler to prioritize the application and output that minimize the overall error output. For example, the lowest error application will allocate all the resources needed. Then the higher error application will be allocated the slack of this application.

The scheduler managed by the profile and flexibility would converge into a Pareto optimal condition but not envy-free. It is Pareto optimal because a cooperative AML cannot decrease its error without making a non-cooperative application's error output higher. Meanwhile, it is not envy-free because a cooperative AML can decrease its error output if the allocated timeshare is swapped with the non-cooperative application.

Fair Scheduling

A fair scheduler would divide the available resource evenly based on how many applications are available [**?** 39, 8]. Dividing resources evenly is not suitable for hardware utilization because each application requires different resources. The fair scheduler would converge into an envy-free condition but not Pareto optimal. It is envy-free because the resource allocated to one stakeholder is the same; therefore, no stakeholder can improve their output by swapping with other stakeholders' resources. Meanwhile, it is not Pareto optimal since each application would require a different timeshare, and an application might not have a good timeshare. At the same time, the others are allocated too many resources.

Notation	Description
C_i	Worst Case Execution Time (WCET) of application i
T_i	deadline of application i
l_i	runtime latency of application i
n_m	number of times the application run produces output m
c_m	output of application i

Table 5.1: Notation used in this paper

5.3.3 SIM Design Principle

Building on the information and analysis from previous subsections, we have two critical ideas for building SIM. First, AML should be scheduled by its output and dynamic time profile. Second, the flexibility and cooperation of stakeholders are exploited by untruthful stakeholders because the prior scheduler is either not envy-free or not Pareto optimal.

To handle dynamics SIM employs a monitoring block that updates the timing profile of AML's incremental output. SIM also schedules the AML by limiting its output. This output limit is set as a goal in SIM's feedback mechanism, and SIM then monitors the output as feedback to decide the subsequent actions. If an application cannot achieve the output goal, then SIM will enforce the greedy application to be flexible and cooperative by reducing their output goal to release timeshare resources.

To achieve envy freedom, SIM would set the goal of each AML as if they are allocated to the most greedy AML that uses most of the resources. However, to avoid oscillation and decrease performance overhead, SIM relaxed the resource swapping from equal timeshare to slightly less timeshare.

To achieve Pareto optimality, SIM would auction the slack resources to the flexible application. This auction also incentivizes flexibility since SIM will allocate the resources (by increasing their output goal) to the currently most flexible application.

5.4 Design and Implementation

Figure 5.8 shows our approaches to handling multiple stakeholders in SSI. When a new stakeholder i launch the application to SSI, they have to declare their latency deadline D_i and their AML's output that can be used to tradeoff latency and output error, SIM then will monitor the application to get the timing resource required for each output. This information is then used to schedule the applications. The decision-maker will decide the AML's output based on their flexibility and resource usage. SIM then embedded a feedback mechanism to this AML. The feedback mechanism will dynamically adjust the output knobs to minimize errors while meeting the latency deadlines.

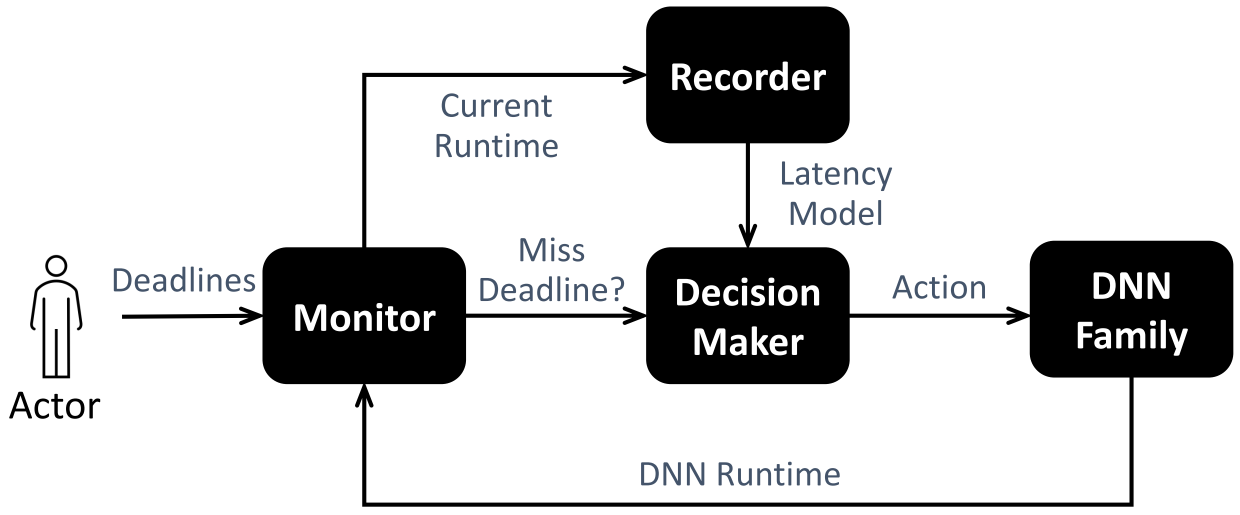


Figure 5.8: Algorithm Building Block

We start our discussion by describing the input that is needed from the AMLs. Then the monitoring block. The auction decision-maker. Last is the local feedback mechanism to manage flexibility.

5.4.1 Input of The System

SIM's inputs are the workload's name i and its deadline D_i. They must also declare their specifications as shown in Table 5.2. While the latency profile is needed for the scheduling,

Output	Latency	Error
c_0	l_0	e_0
c_1	l_1	e_1
...	...	...
c_{max}	l_{max}	e_{max}

Table 5.2: Model of Workload

Output	Average Latency	Number
No Output	-	n_{missed}
0	l_0	n_0
1	l_1	n_1
...	...	...
m	l_n	m_n

Table 5.3: Runtime Statistic of Application

our monitor system would measure it during runtime because it is better suited for the dynamic phase and environment. We do not require the expectation error of the application. A stakeholder can put the error information. However, it would not be used for calculation but as a relative preference of their other output. An application is considered untruthful if it cannot provide the output to be limited later.

5.4.2 Monitoring Block

Monitoring block duty is to measure the application's runtime statistic and maintain the record table for each auction period. Table 5.2 shows the runtime statistic to be maintained. When an auction is performed, SIM will look for this data table for their process and reset the table.

5.4.3 SIM Workflow

The algorithm starts by getting the runtime statistic from the monitoring block (line 2). Then determines which application is the most greedy and how much timeshare it is used (line 3). This statistic is shown in table 5.3. Given this statistic, the algorithm calculates

Algorithm 5 SIM Workflow

```
while TRUE do
    Get runtime statistics from the monitoring block. (Table 5.3)
    Determine which application is greedy.
    if Output targets are not met then
        Force the greedy application to adapt.
    else
        Auction the slack timeshare.
        Adjust the output target for the auction winner.
    end if
    for each application do
        Resolve the envy-freedom.
    end for
    for each application do
        Assign output target to local controller
    end for
end while
```

the output realization of each application. Then compare it with the previously assigned output target (line 4).

When any workload misses the output target, SIM will fix the system by releasing some resources from the most greedy application. In line 5, SIM forces the application that uses the most resource to adapt by reducing its output target, which reduces its resource allocation. If they cannot adapt, then they have to leave the system.

When all the goals are met, each application can claim the slack resources if any slack is available. However, SIM would grant the resources to those who claim the least. Only flexible applications can bid for the slack resource since they have to provide their following output profile.

$$\begin{aligned} bid_i &= l_{next} - l_{greedy} \\ resource_{increment} &= l_{next} - l_{current} \end{aligned} \tag{5.2}$$

SIM calculates bid_i and $resource_{increment}$ for each application i based on equation 5.2. On behalf of the application, SIM does the auction where the winner is the one with the least bid, and the $resource_{increment}$ is less than slack. Once the winner is decided, SIM increases the output target and resource for the winner.

Line 10-11 shows the resource allocation for other applications. Using the model from table 5.2, SIM resolves the envies of the application. SIM assigns output goal such that the expected resource usage is less than the previous resource used by the greedy application l_{greedy}. In prior approaches, the flexible application is disincentivized because they are allocated the slack resource of the non-flexible or higher priority application. Here, SIM, on behalf of each application, would set the application target as highest as possible, but the expected resource usage is below the last most used resource.

5.4.4 Properties of SIM

Incentivize Cooperation

SIM incentivizes cooperative behavior and punishes untruthful behavior. Whenever an application cannot meet the target output, SIM looks for the greedy workload instead of the one that can be controlled. When this greedy workload cannot be controlled, the agent kicks this workload from the system instead of allocating more time. The mechanism encourages cooperation. A cooperative stakeholder can get more resources by bidding for slack resources using their subsequent output.

Convergence

The mechanism converges when all the slack resources are claimed and applications meet their target output. SIM would auction the slack resources until all of them are claimed or when all the applications cannot decrease their output error without increasing their resource usage above the most greedy one. On the opposite, SIM would iteratively decrease the resource allocation of the greedy application when there is an application that cannot meet its target output.

Rationality

The mechanism is individually rational, meaning that when the stakeholder is truthful, they will not be at a loss. In each iteration, SIM solves envies by setting the output target of all applications as if they are allocated the same resources as the most greedy one. Since the outputs and resources are discrete, this will ensure an envy-free condition.

Robustness

The mechanism is robust from misbehaving. A stakeholder cannot lie about their utility, and SIM does not schedule and allocates resources based on their utility. There is no incentive to hide their flexibility. A stakeholder must declare their output stages. If an application is punished with no more minor output stage, it will be kicked from SSI. A stakeholder cannot acquire more resources if they do not declare their increased output.

5.4.5 Feedback Mechanism

Our feedback mechanism is embedded into each of application locally. It solves the following:

$$\begin{aligned} &minimize\ \mathbf{e} \\ &s.t. l_i \leq D_i \\ &o_i \leq g_i \end{aligned} \tag{5.3}$$

Algorithm 6 shows the runtime of Anytime Workload. SIM controls the ANN by limiting its output. ANN is an incremental algorithm that keeps reducing the error output over time. ANN will stop producing a new output when they reach their output limit c_{lim}.

Algorithm 6 Runtime of Anytime Machine Learning

```
Require: $C_{limit}$
Ensure: $c_{done}$
Ensure: $l_{elapsed}$
1: for $c_j$ in $C_{limit}$ do
2:     IncrementalInference
3:     $c_{done} \leftarrow c_j$
4: end for
```

5.5 Experimental Setup

This experiment aims to show that SIM can meet the latency deadline of all the workloads while incentivizing flexible applications. We use multiple Anytime Machine Learning applications, from image processing to speech recognition, to simulate the arbitrary scientist's Neural Network application.

5.5.1 Platform and Benchmark

We evaluate SIM on Nvidia GeForce RTX2080MQ GPU. A mobile GPU with 8 Giga Bytes of GDDR6 memory and 46 streaming multiprocessors with a maximum power of 90 Watts. We believe this mobile GPU can simulate the environment in a shared sensing infrastructure.

We test our experiment by mimicking a shared infrastructure. We evaluate the ability to minimize the error of the DNNs with multiple latency deadlines to mimic the actual deployment of SSI with a camera sensor. For each experiment, we quantify the expected error calculation based on whether the AML can finish before the deadline or not. If AML can finish before the deadline, we will record the error as the training accuracy of the selected configuration. If it cannot finish before the deadline, we quantify it as the error of the last output.

Table 5.4 shows the AML workloads used in this paper. We adopt nested architecture [125] on classic neural networks to get these AMLs.We also provide the symbol used in this paper. For example, later, SII would mean that we run a single Sparseresnet colocated with

2 Imagenet together.

Base Network	Task	Symbol
Sparse ResNet [137]	Image classification on CIFAR	S
Sparse ResNet	Image classification on ImageNet	I
RNN	Word prediction	N
SASRec [63]	Recommendation system	R

Table 5.4: Anytime Workloads Description

5.5.2 Point of Comparison

We compared our scheduler with different kinds of the current state-of-the-art approaches.

- Local adaptation. In this scenario, each AML controls itself locally to maximize its accuracy while meeting its latency deadlines without considering other DNNs. They react to the difference between their runtime latency and latency deadline.

- Worst case approach (embedded system). In this scenario, the system manager will profile the DNNs. Then determine each AML's configuration WCET based on the most prolonged observed latency. The manager then reconfigures the DNNs to meet the latency deadlines while maximizing accuracy based on profiling information.

- Fair time share allocation (cloud system). In this scenario, the system manager will allocate a fair share of computing utilization for all DNNs. The resource manager will divide and allocate the computation timeshare based on the number of DNNs available. Then each DNN will control itself to maintain its latency deadline using the same step as in local adaptation but limited by the timeshare from the resource manager.

- Oracle. In this scenario, the system manager knows which configurations can meet the latency deadlines while minimizing the harmonic mean of error prediction. We create this scenario to evaluate how far we differ from the optimal solution.

5.6 Evaluation

5.6.1 Overall Results

This section evaluates our runtime management to meet latency deadlines while incentivizing flexibility and cooperation. First, we evaluate our system to minimize errors and meet latency deadlines when all workloads are flexible by quantifying the harmonic mean error and latency misses. Next, we introduce a non-flexible Neural Network which is not flexible and relies on other workloads. Last, we show a scenario about our system handling the dynamic environment.

Our System in Flexible Workloads

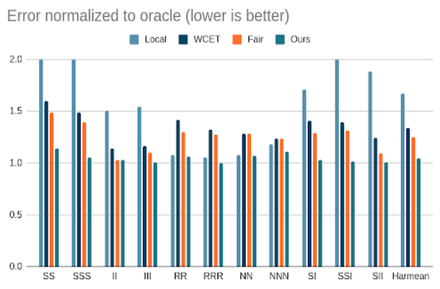


Figure 5.9: Error of Cooperative Workloads

Comparison in general. Figure 5.9 shows our system in minimizing error under latency constraints. We already included the violated latency calculation in this result. The result is normalized to the oracle, where the lower the value is, the better. Our system minimizes error by only 5% more than the Oracle, which is the best across all of the baseline.

Compared to local adaptation. Local adaptation gives high errors, and the worst result because they will compete for timeshare. Without coordination, Anytime Neural Network runtime is limited by their latency deadline. So they will occupy the computation without knowing other colocated ANN also needs to run. This colocated ANN misses its latency constraints because it simply cannot run its work. SIM decreases the error by 63% compared with the local adaptation.

Compared to WCET scheduling. Applications met their latency constraints but with low quality because scheduler schedules were based on their WCET. The worst-case rarely happens, and Anytime DNN should be scheduled by their average case instead since they can adapt by sacrificing their accuracy. SIM is better because we manage the configurations dynamically by monitoring the quality and latency and responding when there is a change in the dynamics. SIM is better by 23% compared with this approach.

Compared with fair scheduling. Fair scheduling does not regard any resource need of applications and only allocates a fair timeshare for all DNNs. All truthful stakeholders here are not incentivized to fulfill the remaining slack timeshare available. Our approach is different here, such that we allocate the remaining timeshare for DNNs that can increase their utilization but with the least additional resources. Because of this policy, we encourage the stakeholders to design and deploy their DNN as flexibly as possible. SIM is better by 16% compared with this approach to decrease error.

Compared with oracle. The oracle knows the best configuration from the beginning. We slightly worse than this approach because we need time to discover the best action for the system.

Scenario of a untruthful stakeholder

In this scenario, we add an untruthful stakeholder into the multitenancy. This untruthful stakeholder will not cooperate with other ANNs and will not sacrifice their quality of prediction. This untruthful stakeholder relies on other ANN to do the adaptation for the untruthful stakeholder's benefit.

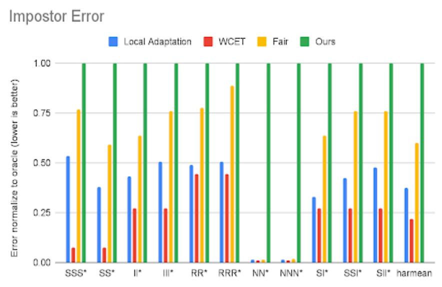


Figure 5.10: Error of untruthful stakeholder

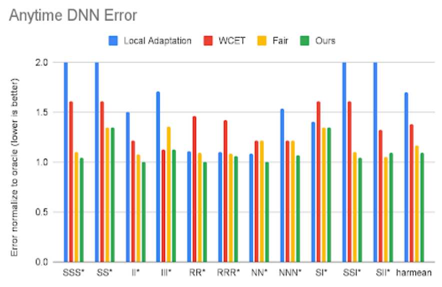


Figure 5.11: Error of truthful stakeholder

Comparison in general. We show that our approach can punish the deployment of untruthful stakeholders compared to all other baselines. By punishing the untruthful stakeholder we decrease the error prediction of the truthful stakeholder thus incentivizing them.

Compared with local adaptation. Without coordination, ANNs in the local adaptation baseline would be untruthful stakeholders. They are competing with each other to use as many resources as possible to minimize their error prediction. Both good and untruthful stakeholders have high errors because they will compete for time share and miss their latency constraints. Contrary, we are coordinating the DNNs by limiting their timeshare and configuration. We prevent the competing behaviour thus we have less error prediction compared to this baseline.

Compared with scheduling by WCET. In the WCET baseline, they assume all of the stakeholders are truthful stakeholders. The scheduler will look for the profiling configuration of each DNNs to meet latency deadlines and minimize error prediction. Since the untruthful stakeholder is not reconfigurable, the scheduler will force the truthful stakeholder to reduce their quality when the timeshare is not enough. We can see from the result, the error prediction of untruthful stakeholders are the best in this scenario. The untruthful stakeholder game the system, having small error prediction by sacrificing the truthful stakeholder. Our solution differs in the term of meeting the latency deadlines. Instead of looking for any configuration which comes from the truthful stakeholder, we detect which DNN is actually consuming resources such that other DNN missed the deadline. By doing this, we prevent the bad behavior of DNN and force the DNN to use only the fair enough amount for their runtime.

Compared with fair scheduling. In a fair baseline, the errors of both truthful stakeholders and untruthful stakeholders are quite similar. This happened because the system manager allocates the same time share for all DNNs regardless of their behavior. Our approach differs from this such that we punish the bad behavior of the system. When a DNN is dominating other DNN to sacrifice their accuracy, we detect this by asking the bad behavior to provide a configuration that is not dominating. If they cannot provide that configuration, then we will kick them out of the system and allocate the timeshare for other truthful stakeholder DNNs. We argue that just providing a fair timeshare is bad for the system in general. First, there is no incentive to be a truthful stakeholder because regardless of the strategy they will be allocated the same timeshare as a untruthful stakeholder. The second, it actually hurt the global accuracy prediction when all of the DNNs are truthful stakeholders. We will cover the explanation in the next session.

Compared with oracle. We assume Oracle would want to kick the untruthful stakeholder. Here our result is slightly differ from the oracle because we need time to detect which one is the untruthful stakeholder.

5.6.2 Overhead

Our algorithm only has a really small overhead. The overhead is less than 1% of the inference latency.

5.7 Related Works

Much prior work focuses on designing networks that meet latency requirements under all timeshare variations. Some focus on designing efficient networks that achieve high accuracy with fewer operations [46, 69, 51]. Others propose networks supporting anytime [127, 125, 47, 78] inference and cascade design [75, 120, 49, 48], which produce a series of increasingly accurate outputs over time. These works are already incorporated into our system.

Many prior works meet latency constraints using control theory ([109, 110, 64]), real-time scheduler ([108, 94]) or neural network ([28, 117]). Those approaches solve different scenarios that would not be suitable for SSI. SSI consists of multiple application that comes from different stakeholders. We already show in this paper that an untruthful stakeholder can exploit it.

5.8 Conclusion

This paper demonstrates the potential problem that arises when Anytime machine learning are deployed in an embedded multi-tenancy system. Our work solves this problem by building a scheduler that incentivizes flexibility and cooperation between applications. We evaluate our work with multiple varieties of Anytime machine learning and meet latency goals with low error.

CHAPTER 6
FUTURE WORKS

While this thesis explores the mechanism to manage the flexibility of computer system and application, it is still a tip of iceberg and many things can be done.

6.1 More Complex Resources

In this thesis, we already described the mechanism to incentivize flexibility in real time environment. For the future, we can extend this work into more complex resource allocation. The resource allocation should be extended not only for timeshare but also the computation allocation and memory allocation.

6.2 Enforcing Flexibility

The design of flexible application is difficult. Sometime stakeholder is not actually have bad intent. We want to extend our work such that it treats the inflexible application better instead of kicking them out of system.

6.3 Automatic Cooperation

Currently, the mechanism to meet goals while optimizing something need to be designed together. However, this is not applicable in real deployment. For example in serverless computing, the goals of the application and the system are contradict each other. The applications want to meet performance requirements while minimizing the cost. Meanwhile the provider want to minimize the energy usage while maintaining the QoS of their tenants. In the next research, I want to explore the requirement to build a serverless computing framework that can handle multiple control problems.

CHAPTER 7

REFERENCES

[1] Baidu AI. 2018. Apollo Open Vehicle Certificate Platform. Online document, `http://apollo.auto`.

[2] S. Akhlaghi, N. Zhou, and Z. Huang. 2017. Adaptive adjustment of noise covariance in Kalman filter for dynamic state estimation. In *IEEE Power Energy Society General Meeting*.

[3] Irina Alam and KT Lau. 2017. Approximate adder for low-power computations. *International Journal of Electronics Letters* 5, 2 (2017), 158–165.

[4] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. 2012. Siblingrivalry: online autotuning through local competitions. In *CASES*.

[5] Hakan Aydi, Pedro Mejía-Alvarez, Daniel Mossé, and Rami Melhem. 2001. Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems. In *RTSS*.

[6] Iljoo Baek, Matthew Harding, Akshit Kanda, Kyung Ryeol Choi, Soheil Samii, and Ragunathan Raj Rajkumar. 2020. CARSS: Client-Aware Resource Sharing and Scheduling for Heterogeneous Applications. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 324–335. `https://doi.org/10.1109/RTAS48715.2020.00008`

[7] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*. IEEE.

[8] S.K. Baruah. 1995. Fairness in periodic real-time scheduling. In *Proceedings 16th IEEE Real-Time Systems Symposium*. 200–209. `https://doi.org/10.1109/REAL.1995.495210`

[9] Pete Beckman, Rajesh Sankaran, Charlie Catlett, Nicola Ferrier, Robert Jacob, and Michael Papka. 2016. Waggle: An open sensor platform for edge computing. In *2016 IEEE SENSORS*. 1–3. `https://doi.org/10.1109/ICSENS.2016.7808975`

[10] Adam Betts and Alastair Donaldson. 2013. Estimating the WCET of GPU-accelerated applications using hybrid analysis. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. IEEE, 193–202.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*.

[12] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe Lipari. 2009. Minimizing CPU energy in real-time systems with discrete speed management. *ACM Trans. Embedded Comput. Syst.* 8, 4 (2009).

[13] David Bishop. 2006. Fixed point package user's guide. *Packages and bodies for the IEEE* (2006), 1076–2008.

[14] Giorgio C Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. 2006. *Soft Real-Time Systems: Predictability vs. Efficiency: Predictability vs. Efficiency*. Springer.

[15] Giorgio C Buttazzo, Giuseppe Lipari, Luca Abeni, and Marco Caccamo. 2006. *Soft Real-Time Systems: Predictability vs. Efficiency: Predictability vs. Efficiency*. Springer.

[16] Liyu Cao and Howard M. Schwartz. 2004. Analysis of the Kalman filter based estimation algorithm: an orthogonal decomposition approach. *Automatica* 40, 1 (2004), 5–19.

[17] Yu Cao. 2011. *Predictive Technology Model for Robust Nanoelectronic Design*. Springer Science & Business Media.

[18] Aaron Carroll and Gernot Heiser. 2013. Mobile Multicores: Use Them or Waste Them. In *HotPower*.

[19] Charlie Catlett, Pete Beckman, Nicola Ferrier, Howard Nusbaum, Michael E. Papka, Marc G. Berman, and Rajesh Sankaran. 2020. Measuring Cities with Software-Defined Sensors. *Journal of Social Computing* 1, 1 (2020), 14–27. `https://doi.org/10.23919/JSC.2020.0003`

[20] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 44–54.

[21] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*.

[22] Jian-Jia Chen, Mong-Jen Kao, D.T. Lee, Ignaz Rutter, and Dorothea Wagner. 2014. Online Dynamic Power Management with Hard Real-Time Guarantees. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 25)*, Ernst W. Mayr and Natacha Portier (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 226–238. `https://doi.org/10.4230/LIPIcs.STACS.2014.226`

[23] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. 2011. Pack & Cap: adaptive DVFS and thread packing under power caps. In *MICRO*.

[24] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: memory power estimation and capping. In *ISLPED*.

[25] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ASPLOS*.

[26] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *ASPLOS*.

[27] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. 2011. MemScale: Active Low-power Modes for Main Memory. *SIGPLAN Not.* 47, 4 (March 2011), 225–238. https://doi.org/10.1145/2248487.1950392

[28] Yi Ding, Nikita Mishra, and Henry Hoffmann. 2019. Generative and Multi-Phase Learning for Computer Systems Optimization. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 39–52. https://doi.org/10.1145/3307650.3326633

[29] Piotr Dollar, Christian Wojek, Bernt Schiele, and Pietro Perona. 2011. Pedestrian detection: An evaluation of the state of the art. *TPAMI* (2011).

[30] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, and Michael F. P. O'Boyle. 2010. A Predictive Model for Dynamic Microarchitectural Adaptivity Control. In *MICRO*.

[31] Joseph R. Dwyer and Martin A. Uman. 2014. The physics of lightning. *physrep* 534, 4 (Jan. 2014), 147–241. https://doi.org/10.1016/j.physrep.2013.09.004

[32] G. A. Elliott, B. C. Ward, and J. H. Anderson. 2013. GPUSync: A Framework for Real-Time GPU Management. In *RTSS*.

[33] Glenn A. Elliott, Kecheng Yang, and James H. Anderson. 2015. Supporting Real-Time Computer Vision Workloads Using OpenVX on Multicore+GPU Platforms. In *RTSS*.

[34] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *ISCA*.

[35] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Mobicom*.

[36] Michael Figurnov, Maxwell D Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry P Vetrov, and Ruslan Salakhutdinov. 2017. Spatially Adaptive Computation Time for Residual Networks.. In *CVPR*. 7.

[37] Michael Fisher, Michael Fradley, Pascal Flohr, Bijan Rouhani, and Francesca Simi. 2021. Ethical considerations for remote sensing and open data in relation to the endangered archaeology in the Middle East and North Africa project. *Archaeological Prospection* 28, 3 (2021), 279–292. `https://doi.org/10.1002/arp.1816` arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/arp.1816

[38] Shelby Funk, Vandy Berten, Chiahsun Ho, and Joël Goossens. 2012. A Global Optimal Scheduling Algorithm for Multiprocessor Low-power Platforms. In *RTNS*.

[39] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, et al. 2020. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 681–697.

[40] Soheil Hashemi, Nicholas Anthony, Hokchhay Tann, R Iris Bahar, and Sherief Reda. 2017. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In *DATE*. 1474–1479.

[41] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. 2004. *Feedback Control of Computing Systems.* John Wiley & Sons.

[42] Eve-Lyn S. Hinckley, Gordon B. Bonan, Gabriel J. Bowen, Benjamin P. Colman, Paul A. Duffy, Christine L. Goodale, Benjamin Z. Houlton, Erika Marín-Spiotta, Kiona Ogle, Scott V. Ollinger, Eldor A. Paul, Peter M. Vitousek, Kathleen C. Weathers, and David G. Williams. 2016. The soil and plant biogeochemistry sampling design for The National Ecological Observatory Network. *Ecosphere* 7, 3 (2016), e01234. https://doi.org/10.1002/ecs2.1234 arXiv:https://esajournals.onlinelibrary.wiley.com/doi/pdf/10.1002/ecs2.1234

[43] Henry Hoffmann. 2015. JouleGuard: energy guarantees for approximate applications. In *SOSP*.

[44] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E. Miller, Sabrina M. Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, Anantha P. Chandrakasan, and Srinivas Devadas. 2012. Self-aware computing in the Angstrom processor. In *DAC*.

[45] Henry Hoffmann and Martina Maggio. 2014. PCP: A Generalized Approach to Optimizing Performance Under Power Constraints through Resource Management. In *ICAC*. 241–247.

[46] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV]

[47] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. 2019. Dynamic adaptive

DNN surgery for inference acceleration on the edge. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1423–1431.

[48] Hanzhang Hu, Debadeepta Dey, Martial Hebert, and J Andrew Bagnell. 2019. Learning anytime predictions in neural networks via adaptive loss balancing. In *AAAI*.

[49] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Multi-Scale Dense Convolutional Networks for Efficient Prediction. In *CoRR*.

[50] Kai Huang, Luca Santinelli, Jian-Jia Chen, Lothar Thiele, and Giorgio C. Buttazzo. 2011. Applying real-time interface and calculus for dynamic power management in hard real-time systems. *Real-Time Systems* 47, 2 (01 Mar 2011), 163–193. `https://doi.org/10.1007/s11241-011-9115-z`

[51] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size. *arXiv preprint arXiv:1602.07360*. arXiv:arXiv:1602.07360

[52] C. Imes and H. Hoffmann. 2016. Bard: A unified framework for managing soft timing and power constraints. In *SAMOS*. 31–38.

[53] Connor Imes, David H.K. Kim, Martina Maggio, and Henry Hoffmann. 2015. POET: a portable approach to minimizing energy under soft real-time constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 75–86. `https://doi.org/10.1109/RTAS.2015.7108419`

[54] Shubham Jain, Swagath Venkataramani, Vijayalakshmi Srinivasan, Jungwook Choi, Pierce Chuang, and Leland Chang. 2018. Compensated-DNN: energy efficient low-precision deep neural networks by compensating quantization errors. In *DAC*. 1–6.

[55] Ravindra Jejurikar and Rajesh Gupta. 2005. Dynamic Slack Reclamation with Procrastination Scheduling in Real-time Embedded Systems. In *Proceedings of the 42Nd Annual Design Automation Conference* (Anaheim, California, USA) *(DAC '05)*. ACM, New York, NY, USA, 111–116. https://doi.org/10.1145/1065579.1065612

[56] Hong Ji, Zhi Gao, Tiancan Mei, and Bharath Ramesh. 2020. Vehicle Detection in Remote Sensing Images Leveraging on Simultaneous Super-Resolution. *IEEE Geoscience and Remote Sensing Letters* 17, 4 (2020), 676–680. https://doi.org/10.1109/LGRS.2019.2930308

[57] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: scalable adaptation of video analytics. In *ACM SIGCOMM*. 253–266.

[58] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay

Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*.

[59] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Proteus: Exploiting numerical precision variability in deep neural networks. In *ICS*. 23.

[60] A.B. Kahng. 2013. The ITRS design technology and system drivers roadmap: Process and status. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*. 1–6.

[61] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. 2009. Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In *ICAC*.

[62] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. 2014. Adaptive resource provisioning for virtualized servers using Kalman filters. *TAAS* (2014).

[63] Wang-Cheng Kang and Julian McAuley. 2018. Self-attentive sequential recommendation. In *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 197–206.

[64] Tejas Kannan and Henry Hoffmann. 2021. Budget RNNs: Multi-Capacity Neural Networks to Improve In-Sensor Inference Under Energy Budgets. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 143–156. `https://doi.org/10.1109/RTAS52030.2021.00020`

[65] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. 2011. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In *RTSS*.

[66] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *USENIX ATC*.

[67] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott A. Brandt. 2012. Gdev: First-Class GPU Resource Management in the Operating System. In *USENIX ATC*.

[68] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. 2018. Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems. In *ICCPS*. 287–296.

[69] Ivan Khokhlov, Egor Davydenko, Ilya Osokin, Ilya Ryakin, Azer Babaev, Vladimir Litvinenko, and Roman Gorbachev. 2020. Tiny-YOLO object detection supplemented with geometrical data. *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)* (May 2020). https://doi.org/10.1109/vtc2020-spring48590.2020.9128749

[70] David H. K. Kim, Connor Imes, and Henry Hoffmann. 2015. Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics. In *CPSNA*.

[71] D. H. K. Kim, C. Imes, and H. Hoffmann. 2015. Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics. In *ICCPS*.

[72] Nam Sung Kim, Todd Austin, David Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. 2003. Leakage current: Moore's law meets static power. *computer* 36, 12 (2003), 68–75.

[73] Wonyoung Kim, M.S. Gupta, Gu-Yeon Wei, and D. Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *High Performance*

Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on. 123–134. https://doi.org/10.1109/HPCA.2008.4658633

[74] Zachary Langford, Jitendra Kumar, and Forrest Hoffman. 2018. Wildfire Mapping in Interior Alaska Using Deep Neural Networks on Imbalanced Datasets. In *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. 770–778. https://doi.org/10.1109/ICDMW.2018.00116

[75] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. 2016. Fractalnet: Ultra-deep neural networks without residuals. *arXiv preprint arXiv:1605.07648* (2016). arXiv:arXiv:1605.07648

[76] Etienne Le Sueur and Gernot Heiser. 2011. Slow Down or Sleep, That is the Question. In *USENIX ATC*.

[77] Benjamin C Lee and David Brooks. 2008. Efficiency trends and limits from comprehensive microarchitectural adaptivity. *ASPLOS* (2008).

[78] Hankook Lee and Jinwoo Shin. 2018. Anytime Neural Prediction via Slicing Networks Vertically. *arXiv preprint arXiv:1807.02609* (2018).

[79] C. Lefurgy, X. Wang, and M. Ware. 2008. Power capping: a prelude to power shifting. *Cluster Computing* 11, 2 (2008), 183–195.

[80] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: enabling energy optimizations in GPGPUs. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 487–498.

[81] W.S. Levine. 2005. *The control handbook*. CRC Press.

[82] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. 2018. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *ASPLOS*. 751–766.

[83] J.W.S. Liu, Wei-Kuan Shih, Kwei-Jay Lin, R. Bettati, and Jen-Yao Chung. 1994. Imprecise computations. *Proc. IEEE* 82, 1 (1994), 83–94. `https://doi.org/10.1109/5.259428`

[84] Jun S Liu and Rong Chen. 1998. Sequential Monte Carlo methods for dynamic systems. *Journal of the American statistical association* (1998).

[85] ATLAS LS. 2010. What is Simultaneous/Conference Interpretation? Online document, `https://atlasls.com/what-is-simultaneousconference-interpretation/`.

[86] Martina Maggio, Alessandro Vittorio Papadopoulos, Antonio Filieri, and Henry Hoffmann. 2017. Automated control of multiple software goals using multiple actuators. In *FSE*.

[87] Arian Maghazeh, Unmesh D. Bordoloi, Adrian Horga, Petru Eles, and Zebo Peng. 2014. Saving energy without defying deadlines on mobile GPU-based heterogeneous systems. In *CODES+ISSS*.

[88] Brian Magi, Thomas Winesett, and Daniel Cecil. 2016. Estimating Lightning from Microwave Remote Sensing Data. *Journal of Applied Meteorology and Climatology* 55 (07 2016). `https://doi.org/10.1175/JAMC-D-15-0306.1`

[89] Maria Maniadaki, Athanasios Papathanasopoulos, Lilian Mitrou, and Efpraxia-Aithra Maria. 2021. Reconciling Remote Sensing Technologies with Personal Data and Privacy Protection in the European Union: Recent Developments in Greek Legislation and Application Perspectives in Environmental Law. *Laws* 10, 2 (2021). `https://doi.org/10.3390/laws10020033`

[90] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE TCCA Newsletter* (Dec. 1995), 19–25.

[91] Mason McGill and Pietro Perona. 2017. Deciding how to decide: Dynamic routing in artificial neural networks. *arXiv preprint arXiv:1703.06217* (2017). arXiv:arXiv:1703.06217

[92] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. 2011. Power management of online data-intensive services. *ISCA* (2011).

[93] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. 2018. CALOREE: Learning Control for Predictable Latency and Low Energy. In *ASPLOS*.

[94] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. 2018. CALOREE: Learning Control for Predictable Latency and Low Energy. *SIGPLAN Not.* 53, 2 (mar 2018), 184–198. `https://doi.org/10.1145/3296957.3173184`

[95] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. 2002. Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling. In *ICS*. `https://doi.org/10.1145/514191.514200`

[96] Alex Morehead, Lauren Ogden, Gabe Magee, Ryan Hosler, Bruce White, and George Mohler. 2019. Low Cost Gunshot Detection using Deep Learning on the Raspberry Pi. In *2019 IEEE International Conference on Big Data (Big Data)*. 3038–3044. `https://doi.org/10.1109/BigData47090.2019.9006456`

[97] N.C. Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M.T. Kandemir, R. Iyer, and C.R. Das. 2015. Domain knowledge based energy management in handhelds. In *High Performance Computer Architecture (HPCA), 2015 IEEE*

21st International Symposium on. 150–160. https://doi.org/10.1109/HPCA.2015.7056029

[98] Edoardo Nemni, Joseph Bullock, Samir Belabbes, and Lars Bromley. 2020. Fully Convolutional Neural Network for Rapid Flood Segmentation in Synthetic Aperture Radar Imagery. *Remote Sensing* 12, 16 (2020). https://doi.org/10.3390/rs12162532

[99] NVIDIA. 2017. NVIDIA TensorRT 3 Dramatically Accelerates AI Inference for Hyperscale Data Centers. https://nvidianews.nvidia.com/news/nvidia-tensorrt-3-dramatically-accelerates-ai-inference-for-hyperscale-data-center

[100] Tapasya Patki, David K. Lowenthal, Anjana Sasidharan, Matthias Maiterth, Barry L. Rountree, Martin Schulz, and Bronis R. de Supinski. 2015. Practical Resource Management in Power-Constrained, High Performance Computing. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/2749246.2749262

[101] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *SOSP*.

[102] Paula Petrica, Adam M Izraelevitz, David H Albonesi, and Christine A Shoemaker. 2013. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *ISCA*.

[103] Padmanabhan Pillai and Kang G. Shin. 2001. Real-time Dynamic Voltage Scaling for Low-power Embedded Operating Systems. In *Proceedings of the Eighteenth ACM*

Symposium on Operating Systems Principles (Banff, Alberta, Canada) *(SOSP '01)*. ACM, New York, NY, USA, 89–102. `https://doi.org/10.1145/502034.502044`

[104] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. 2001. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *MICRO*.

[105] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: operating system abstractions to manage GPUs as compute devices. In *SOSP*.

[106] M. H. Santriaji and H. Hoffmann. 2016. GRAPE: Minimizing energy for GPU applications with performance requirements. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. `https://doi.org/10.1109/MICRO.2016.7783719`

[107] Muhammad Husni Santriaji and Henry Hoffmann. 2016. GRAPE: Minimizing energy for GPU applications with performance requirements. In *MICRO*.

[108] Muhammad Husni Santriaji and Henry Hoffmann. 2016. GRAPE: Minimizing energy for GPU applications with performance requirements. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. `https://doi.org/10.1109/MICRO.2016.7783719`

[109] Muhammad Husni Santriaji and Henry Hoffmann. 2018. Formalin: Architectural Support for Power Performance Aware GPU. In *2018 IEEE Conference on Control Technology and Applications (CCTA)*. 1494–1501. `https://doi.org/10.1109/CCTA.2018.8511572`

[110] Muhammad Husni Santriaji and Henry Hoffmann. 2018. MERLOT: Architectural Support for Energy-Efficient Real-Time Processing in GPUs. In *2018 IEEE Real-Time*

and Embedded Technology and Applications Symposium (RTAS). 214–226. https://doi.org/10.1109/RTAS.2018.00030

[111] Vivek Sarkar, William Harrod, and Allan E. Snavely. 2009. Software challenges in extreme scale systems. *Journal of Physics: Conference Series* 180, 1 (2009), 012045. http://stacks.iop.org/1742-6596/180/i=1/a=012045

[112] Marcus T. Schmitz, Bashir M. Al-Hashimi, and Petru Eles. 2004. *System-Level Design Techniques for Energy-Efficient Embedded Systems.* Kluwer Academic Publishers, Norwell, MA, USA.

[113] A. Sethia and S. Mahlke. 2014. Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution. In *MICRO*.

[114] Alex Shye, Yan Pan, Benjamin Scholbrock, J. Scott Miller, Gokhan Memik, Peter A. Dinda, and Robert P. Dick. 2008. Power to the people: Leveraging human physiological traits to control microprocessor frequency. In *MICRO*.

[115] Hyeonuk Sim, Saken Kenzhegulov, and Jongeun Lee. 2018. DPS: dynamic precision scaling for stochastic computing-based deep neural networks. In *DAC*. 13.

[116] Srinath Sridharan, Gagan Gupta, and Gurindar S Sohi. 2013. Holistic run-time parallelism management for time and energy efficiency. In *ICS*.

[117] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2013. Holistic Run-Time Parallelism Management for Time and Energy Efficiency. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (Eugene, Oregon, USA) *(ICS '13)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2464996.2465016

[118] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-M Hwu. 2012. Parboil: A revised benchmark

suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* (2012).

[119] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *ASE*.

[120] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung. 2016. BranchyNet: Fast inference via early exiting from deep neural networks. In *CVPR*.

[121] Andreas Veit and Serge Belongie. 2018. Convolutional Networks with Adaptive Inference Graphs. In *ECCV*.

[122] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. 2014. AxNN: energy-efficient neuromorphic systems using approximate computing. In *ISLPED*.

[123] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation cores: reducing the energy of mature computations. In *ASPLOS*.

[124] Chengcheng Wan, Henry Hoffmann, Shan Lu, and Michael Maire. [n.d.]. Orthogonalized SGD and Nested Architectures for Anytime Neural Networks. In *ICML 2020, to appear*.

[125] Chengcheng Wan, Henry Hoffmann, Shan Lu, and Michael Maire. 2020. Orthogonalized SGD and nested architectures for anytime neural networks. In *International Conference on Machine Learning*. PMLR, 9807–9817.

[126] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. 2020. {ALERT}: Accurate Learning for Energy and Timeliness. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 353–369.

[127] Yan Wang, Zihang Lai, Gao Huang, Brian H Wang, Laurens van der Maaten, Mark Campbell, and Kilian Q Weinberger. 2018. Anytime Stereo Image Depth Estimation on Mobile Devices. *arXiv preprint arXiv:1810.11408* (2018). arXiv:arXiv:1810.11408

[128] Greg Welch and Gary Bishop. [n.d.]. *An Introduction to the Kalman Filter.* Technical Report TR 95-041. UNC Chapel Hill, Department of Computer Science.

[129] Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. 2018. Blockdrop: Dynamic inference paths in residual networks. In *CVPR*. 8817–8826.

[130] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 392–405.

[131] Kecheng Yang, Glenn A. Elliott, and James H. Anderson. 2015. Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *RTNS*.

[132] Shuochao Yao, Yifan Hao, Yiran Zhao, Huajie Shao, Dongxin Liu, Shengzhong Liu, Tianshi Wang, Jinyang Li, and Tarek Abdelzaher. 2020. Scheduling Real-time Deep Learning Services as Imprecise Computations. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 1–10. https://doi.org/10.1109/RTCSA50079.2020.9203676

[133] P. G. Zaykov, G. Kuzmanov, A. M. Molnos, and K. Goossens. 2013. Run-Time Slack Distribution for Real-Time Data-Flow Applications on Embedded MPSoC. In *2013 Euromicro Conference on Digital System Design*. 39–47. https://doi.org/10.1109/DSD.2013.15

[134] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. In *ASPLOS*.

[135] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. Deepcpu: Serving rnn-based deep learning models 10x faster. In *ATC*. 951–965.

[136] Yanqi Zhou, Henry Hoffmann, and David Wentzlaff. 2016. CASH: Supporting IaaS customers with a sub-core configurable architecture. In *ISCA*.

[137] Ligeng Zhu, Ruizhi Deng, Michael Maire, Zhiwei Deng, Greg Mori, and Ping Tan. 2018. Sparsely Aggregated Convolutional Networks. *CoRR* (2018).

[138] Yuhao Zhu and Vijay Janapa Reddi. 2013. High-performance and energy-efficient mobile web browsing on big/little systems. In *HPCA*.