MERLOT: Architectural Support for Energy-Efficient Real-time Processing in GPUs

Muhammad Husni Santriaji
University of Chicago, Dept. of Computer Science
santriaji@uchicago.edu

Henry Hoffmann
University of Chicago, Dept. of Computer Science
hankhoffmann@cs.uchicago.edu

***Abstract*—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. Compared to the approach that always allocates for worst case, MERLOT reduces energy by XX%. Compared to the intelligent software-only approach, MERLOT produces a YY% energy savings. MERLOT uses less energy than software-only approaches because it recognizes better than worst case behavior earlier, by monitoring hardware events that are not visible to software and quickly adjusting resource usage. [TODO: Fill in actual results.]**

I. Introduction

Systems such as self-driving cars have enormous computational requirements and incorporate GPUs ([15, 52]) and emerging GPU-like accelerators (e.g., for deep neural network based sensor processing [39]) 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 [14, 15, 24, 25, 37]. 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—*i.e.,* hard real-time requirements—software must be conservative and allocate resources for worst case behavior [7]. This conservatism is wasteful when inputs do not exercise the worst case execution path, consuming more energy than necessary [27].

Several software approaches have arisen to augment real-time schedulers with energy reduction [2, 5, 11, 17, 20, 23, 47, 53]. 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*, puts a limit on how much 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) *Hardware's 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 that a kernel was ahead of schedule until it completes.
2) *Hardware's 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 a kernel's software-specified timing requirements.

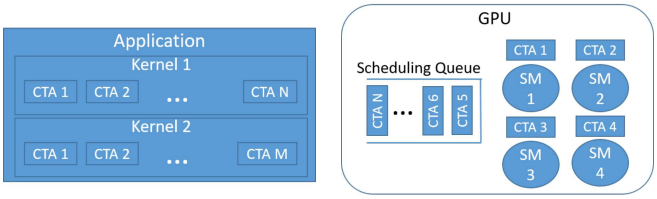
The challenge of hardware-based resource management for real-time systems, however, is that software has all the information about timing requirements and progress. For example, it is software that knows task periods and deadlines. Hardware resource management is also not appropriate for all platforms, 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 [16, 51].

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 [39]—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.

We implement MERLOT in VHDL and synthesize it 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 [3]. 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 when possible [2, 5, 17]. We test MERLOT with a wide range of applications and performance targets. Overall we find:

- MERLOT, like the software-only approaches, meets hard real-time deadlines.
- Compared to race-to-idle, MERLOT reduces energy consumption by [TODO: add the average energy savings] .
- Compared to the sophisticated software scheduler that transfers slack between kernels, MERLOT reduces energy consumption by [TODO: add average energy savings] .
- Compared to a prior hardware approach that provides only soft timing guarantees, MERLOT provides almost equivalent energy savings without missing deadlines. [TODO: Update with final numbers.]



(a) GPU Application (b) GPU Hardware Scheduling

Fig. 1: Structure of 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.

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 [TODO: add url for code repo] .

II. Background and Motivation

In this section we briefly review CPU-GPU interaction with a focus on what units are schedulable in software versus hardware. We then review how timing slack in GPU applications can be turned into energy savings with intelligent software scheduling. With this background in place, we argue that even greater energy saving is possible with a hardware assist to perform this slack transfer at an even finer granularity—using hardware schedulable units instead of software schedulable units.

A. GPU Scheduling Overview

Figure 1a 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-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 [26, 44], and real-time schedulers exist that manage applications and kernels to ensure predictable timing in GPU-augmented systems [14, 15, 24, 25].

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 1b). As CTAs complete, hardware is responsible for selecting the next CTA to run. While hardware-level schedulers have been proposed [48], we know of only one that can provide soft timing guarantees [46], and we 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; *e.g.,* 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.*

B. Opportunities for Energy Savings

Schedulers that provide hard timing guarantees are necessarily conservative [7]. The most conservative approach is to exploit dynamic power management (DPM) by always allocating all resources required to meet timing constraints and transitioning the system to a low-power idle state if a task is not worst case [11, 47]. Prior work, however, has shown that this *race-to-idle* strategy is never better than a more intelligent strategy that knows what resources are required for the current case [27]. The problem, of course, is determining that the current case can be slowed down without incurring timing violations.

1) Slack Transfer: A number of schedulers have been developed that save energy in hard real-time systems by exploiting *slack* [2, 5, 17, 23, 53]. If a task can be decomposed into parts and the scheduler can recognize that one part has finished ahead of schedule (*i.e.,* is not worst case) then this extra time—*i.e.,* slack—can be transferred to the next part of the task, allowing the scheduler to slow that part down without risking violation of the overall task's timing requirements.

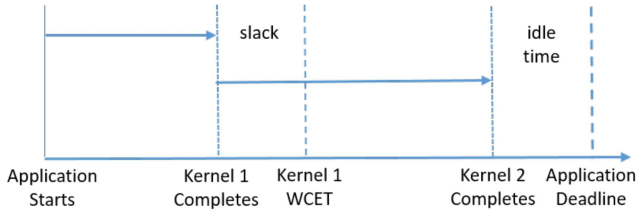


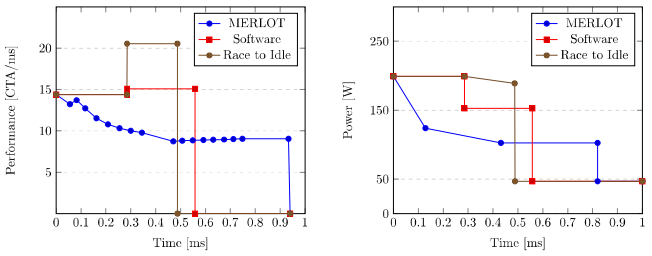
Fig. 2: Slack transfer for energy savings in the GPU application from Figure 1a. The application has a deadline. When kernel 1 finishes ahead of its worst case timing, the resulting slack can be transferred to kernel 2, which can now use a reduced energy resource allocation without risking a timing violation, even if it encounters a worst case input.

Figure 2 illustrates this concept of *slack transfer* as applied to the example GPU application from Figure 1a. In this case, the application has a hard timing deadline. A software scheduler can monitor the execution of kernel 1 to determine that it finished ahead of its worst case timing. This slack is transferred to kernel 2, as the scheduler allocates fewer resources—slowing down kernel 2—without violating the deadline on the application itself. If kernel 2 is also not worst case, then the system can idle the GPU until the next application task is ready to be scheduled.

2) Fine-grain Slack Transfer: Figure 2 highlights the energy savings possible with slack transfer, but it also hints that more energy savings may be available by incorporating hardware-level knowledge into the transfer process. Consider that if the application consisted of only a single kernel, there would be no opportunity to transfer slack. Furthermore, even in the multi-kernel scenario, our intuition is: if kernel 1 is not worst case, then that property is probably observable early in the execution of kernel 1's CTAs, so additional energy could have been saved by reducing resource usage as soon as CTAs complete ahead of the worst case schedule. Given that a single kernel can consist of 100s–1000s of CTAs, there is significantly more opportunity to transfer slack at this level.

We test this intuition empirically using the `backprop` application (an important part of deep neural network computation [10]). `backprop` consists of two kernels. To illustrate the potential energy savings we run it on an instrumented GPU simulator to measure its performance and power as a function of time (see Section IV for details on our experimental setup). We compare three scheduling algorithms that guarantee timing while reducing energy. The first, race-to-idle, simply allocates all GPU resources to the application and then idles the GPU until the deadline (when, presumably, the next task will be scheduled). The second software approach transfers slack from the first kernel to the second, reducing energy usage. The third is the proposed contribution of this paper, MERLOT, which transfers slack at the CTA-level in hardware.

Figure 3a shows the performance (measured as completed CTAs per millisecond) over time, while Figure 3b shows the power over time. First, the two different kernels are easily visible in the time series data: the first kernel naturally



(a) Performance (b) Power

Fig. 3: `backprop`'s performance (a) (in CTAs completed per millisecond) and power (in Watts) (b) for three different scheduling algorithms. Each scheduler meets the timing requirement of completing the application in 1 ms. Energy is the area under each curve in (b). The software slack scheduler reduces energy by just 1% compared to race-to-idle, but MERLOT achieves a 13% reduction.

achieves a lower CTA/ms than the second under race-to-idle. Additionally, we see that all three approaches meet the 1 ms deadline. Considering power consumption, race-to-idle has the highest peak power, which it maintains for the duration of the second kernel. The software slack scheduler recognizes the first kernel completes ahead of schedule and reduces power for the second kernel—by reducing its resource usage from what would be required if the second kernel started at the first kernel's worst case completion time. MERLOT's hardware-based approach recognizes that kernel 1 is not worst case early in its execution (because it is observing CTA completion) and reduces kernel 1's resource usage immediately. MERLOT continues with reduced energy during kernel 2, as it again recognizes the system is ahead of a worst case schedule. Compared to race-to-idle, the more sophisticated software scheduler reduces energy by just 1%, but MERLOT reduces energy by 13%.

C. Challenges for Fine-grain Slack Transfer

The above results demonstrate the energy saving potential of hardware-based slack transfer on GPUs. Achieving these benefits, however, requires addressing two challenges: overhead and resource allocation.

Measuring application progress at the CTA level necessarily requires more fine grain monitoring. GPU software interfaces do not currently support observing the status of a single CTA. It might be possible, however, for the programmer to augment GPU code such that each thread signals completion to the CPU through an interrupt or a write to shared memory. The problem is that the huge number of threads in the kernel would quickly overwhelm software, if each independently signaled termination. We therefore argue that sub-kernel slack transfer requires hardware support. Adding this support is reasonable because hardware GPU schedulers already monitor CTA progress (see Figure 1b). Of course, the additional hardware for resource management must be low-overhead in terms of its area, performance, and power.

Resource allocation is a challenge because hardware must be able to tell with certainty how to adjust a kernel's resource usage without violating timing constraints. For this challenge we assume it is beyond software's scope to provide hints about what resources the application might use. It is up to hardware to determine an appropriate resource allocation at runtime.

III. MERLOT Design & 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 (*e.g.,* when to schedule a kernel), while hardware focuses on low-level resource management, for which it is better suited.

Figure 4 illustrates MERLOT's design. When a software scheduler 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 checkpoint. 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; *i.e.,* 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 MERLOT timing guarantees and implementation issues. Table I summarizes the notation used throughout this section.

A. Software-Hardware Interface

As mentioned in Section II, 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

TABLE I: 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

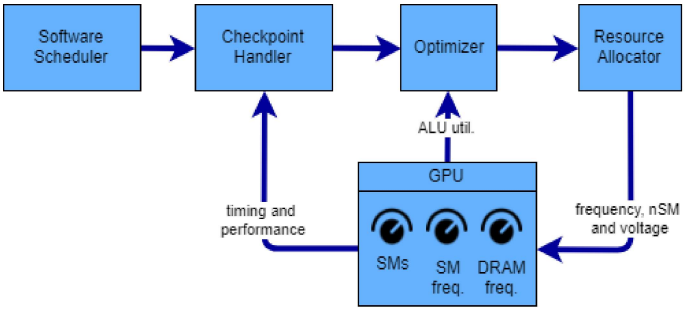


Fig. 4: MERLOT block diagram

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. MERLOT does assume, however, that software can provide the necessary timing analysis for GPU programs; finding the worst case timing is beyond the scope of this paper. A methodology for worst case timing analysis of kernels, CTAs, and threads in GPU applications can be found in [4].

B. Checkpoint Handler

The pseudo-code for the checkpoint handler is shown in Algorithm 1, lines 3–9. Starting from the last checkpoint, MERLOT counts CTA completion in hardware. When the count is equal to a checkpoint count (*i.e.,* 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 1) includes the overhead o of changing hardware configurations—*i.e.,* 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.

C. 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 [38], 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 [38, 46]. 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 is constant. The critical frequency threshold is the point at which additional slowdown increases energy consumption. If λ is below this value, then it is more energy efficient to run

Algorithm 1 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 ← 0
 3: procedure CHECKPOINT HANDLER
 4:     if nCTA_j = CTA_current then
 5:         t_ahead = wCTA_j − t_current
 6:         t_remaining = wCTA_{j+1} − wCTA_j
 7:         η = (t_remaining + o) / (t_remaining + t_ahead)
 8:         λ = ⌈f_max · η⌉
 9:         j++
10: procedure OPTIMIZER
11:     if λ < f_crit then
12:         if ALUutilization < th_ALU then
13:             f_SM ← f_crit − 1
14:             f_mem ← f_crit
15:         else
16:             f_SM ← f_crit
17:             f_mem ← f_crit − 1
18:     else
19:         if ALUutilization < th_ALU then
20:             f_SM ← λ
21:             f_mem ← λ + 1
22:         else
23:             f_SM ← λ
24:             f_mem ← λ
25:     nSM = 0
26:     for all sm in the GPU do
27:         if sm == active then
28:             nSM = nSM + 1
29: procedure RESOURCE ALLOCATOR
30:     if f_mem > f_max then
31:         f_mem ← f_max
32:     Allocate(f_SM, f_mem, nSM)
```

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 [48]. The exact threshold th_{ALU} for distinguishing these two types of kernels is hardware dependent; *i.e.,* 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 [48].

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 1, 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 [48].

TABLE II: 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

After adjusting SM and DRAM frequency, MERLOT sets the active number of SMs in Algorithm 1, 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 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; *i.e.,* the frequency is always above the "safe" zone.

D. 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 II 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.

E. Hardware Implementation

MERLOT is designed to be implementable in hardware. While we do not have the resources to synthesize a GPU that includes MERLOT, we believe it is important to demonstrate that MERLOT can be implemented in hardware. We therefore implement MERLOT in VHDL.

To get some specific numbers, we synthesize MERLOT's VHDL implementation for an FPGA using Quartus II software. The target FPGA device we use is CYCLONE-4E. We implement a fixed point package to perform multiplication and division in VHDL [6]. We synthesize the design and find that MERLOT requires 1,032 logic elements and 6 registers. The TimeQuest timing analyzer shows that MERLOT's f_{max} is 7.18 MHz or 98 cycles overhead in GPU SM frequency. The

PowerPlay Early Power Estimator shows that MERLOT needs 0.371 Watts to operate.

The FPGA implementation demonstrates the feasibility of MERLOT's design. To test its capabilities with real GPU applications, we integrate it into a widely used, open source GPU Simulator, GPGPU-Sim [3]. We implement the dynamic frequency and SM actuator by masking the clock in GPGPU-Sim. We edit GPUWattch to count the dynamic voltage and leakage static power [8]. We use GTX480 model provided by those simulators.[1] We model the overhead of reconfiguring the system as 512 cycles based on prior results [29]. During reconfiguration, we stall the simulator. Based on our FPGA implementation, MERLOT increases the GPU power consumption by 0.4 W. We include all of this overhead in our reported results.

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

F. Real-Time Considerations

We analyze MERLOT's hard real-time guarantees by following the example in [41]. 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 1 reflects this schedulability test.

IV. Methodology

A. Baseline Architecture

MERLOT's timing is evaluated on GPGPU-Sim v3.2.2, a cycle-accurate GPU Simulator [3] and its energy is evaluated using GPUWattch [31], which models GPU power and energy consumption. This tool set has been proven to deliver realistic timing and energy that is within a few percent of the timing on real hardware. Modifying such a cycle accurate simulator is the closest we can get to evaluating MERLOT's hardware design without implementing the hardware itself. Our baseline implementation in GPGPU-Sim uses the configuration in Table III.

In this implementation, we have a set of SM actuators, SM frequency actuators and memory frequency actuators. SM actuators have 16 members covering operation from idle to 15 SMs. We assume that SM frequency actuator has 5 P-states, ranging from a peak of 700 MHz to a minimum of 300 MHz, with step size of 100 MHz. These settings align with the GTX 480's existing DVFS settings. DRAM frequency actuators have 5 P-states, ranging from a peak of 924 MHz to 396 MHz. We use the 45 nm predictive technology models [43] to scale the voltage with frequency (from 1 V to 0.7 V).

[1]While old, no more recent open-source simulator is available.

TABLE III: GPGPU-Sim Configuration.

Field	Value
Shader Core	Fermi 15 Shader Cores, 5-Stage Pipeline,
Resources / Core	Max 1536 Threads, 32 kB Shared Memory, 32768 Registers 48 warps
L1 Caches / Core	32kB 8-way L1 Data Cache, 8 kB 4-way texture, 8kB 4-way constant cache, 64B Line Size
L2 Unified Cache	786 kB, 128B Line size, 8-way associative
Scheduling	GTO Scheduling, Load Balance scheduling
Interconnect	1 crossbar, 16B Channel Width,
DRAM Model	FR-FCFS (128 RQS/MC), 4B Bus Width, 4 DRAM-banks/MC, 2 kB page size, 4 burst size, 8 MCs

TABLE IV: Benchmarks Used.

	Applications	Abbr.	Resource Need
1	Backpropagation [9]	BP	Memory
2	Fast Fourier Transform [50]	FFT	Memory
3	Needelman-Wunch [9]	NW	Unsaturated
4	Breadth First Search [9]	BFS	Cache
5	B+ Tree [9]	BPT	Memory
6	CFD Solver [9]	CFD	Memory
7	Gaussian Elimination [9]	GE	Unsaturated
8	Hotspot [9]	HOT	Compute
9	LU Decomposition [9]	LUD	Unsaturated
10	Particle Filter Float [9]	PFF	Memory
11	Pathfinder[9]	PF	Compute
12	Sum of Absolute Diff. [50]	SAD	Unsaturated

B. Benchmarks

We evaluate MERLOT using the benchmarks in Table IV. For every benchmark we measure the timing data from the simulator and get the power data from GPUWattch.

C. Points of Comparison

We compare MERLOT to two different scenarios: (1) the race-to-idle heuristic which runs each task in the fastest configuration available and then puts the system into a low-power idle state until the next job is to be processed [27] and (2) a software approach which uses a combination of slack transfer, DVFS and low-power idle states to adjust the configuration between the kernels. This second approach is based on implementing existing software approaches (*e.g.,* [2, 5, 17, 23, 53]) for the GPU. We set the idle power to 23.31 W, which is the idle power when the frequency and voltage of both GPU and DRAM is set as low as possible.

V. Experimental Evaluation

Because it is a simulator, GPGPU-Sim is entirely deterministic, which means that in simulation, all kernels have the

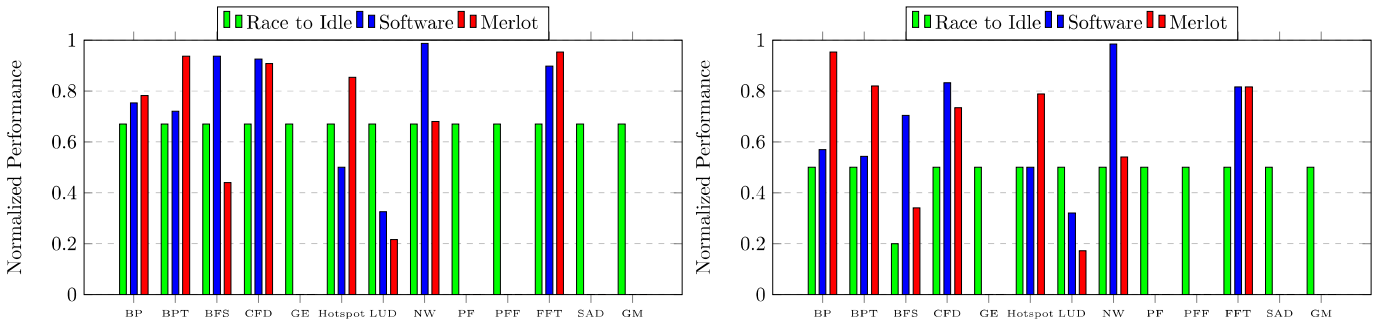


Fig. 5: Running time normalized to race-to-idle. The left figure shows the results where the worst case is $1.5\times$ the average case and the right side shows the case where the ratio is $2.0\times$.

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 differences between the average and worst case timing are tighter than found in prior work [4]. The tighter differences creates fewer opportunities for slack transfer and, we believe, means that the reported energy savings will err on the conservative side.

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 will run the algorithm 1 whenever the checkpoint is triggered. MERLOT is configured so that each application has 16 checkpoints.

A. Performance

[TODO: Change "Software" in the charts to "Software Slack Scheduling."] .

Figure 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 X% faster than the deadlines. We highlight several examples that illustrate the different behavior of the different schedulers.

As discussed in Section II, `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, 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 (see Section III-C), 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 $1.29\times$ faster—and MERLOT is faster, but saves more energy as we will see in the next section.

B. Energy Saving

Figure 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 $X\%$ energy and MERLOT saves $X\%$ energy compared with race-to-idle. 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 13% and 14.6% 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 1.7% more energy for `cfd` than race-to-idle, while MERLOT saves 23% energy. In `bfs`, software saves 10.1% energy, which is worse than MERLOT which saves 13.3% energy.

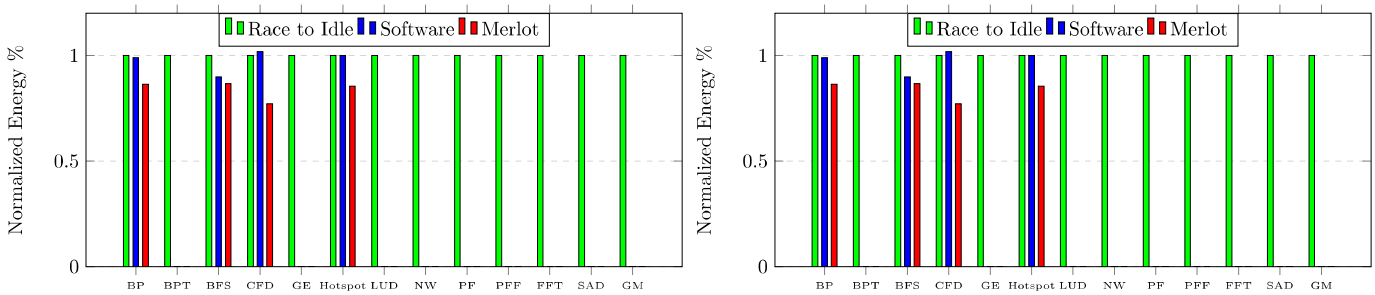


Fig. 6: Energy normalized to the race-to-idle approach. The left figure shows the results where the worst case is $1.5\times$ the average case and the right side shows the case where the ratio is $2.0\times$.

C. Comparison with Prior Work

Figure 7 compares the performance of MERLOT to that of GRAPE, a hardware approach to minimize energy in soft real-time systems. Figure 8 compares the energy savings between those two. MERLOT reduces energy by predicting the slack of the task meanwhile GRAPE takes an average of the performance over time using a feedback control system. Figure 7 shows that while MERLOT never misses a deadline, GRAPE misses deadlines in LUD and NW, an application that has high variation in slack. While MERLOT provides hard real-time guarantees, MERLOT's energy saving is similar to GRAPE. MERLOT saves 17.3% energy consumption while GRAPE saves 17.66% energy saving on average.

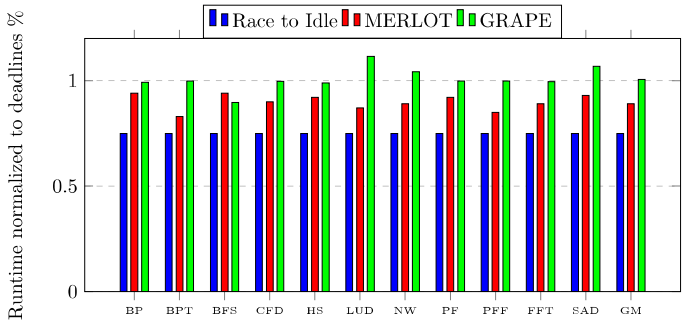


Fig. 7: MERLOT and GRAPE Performance

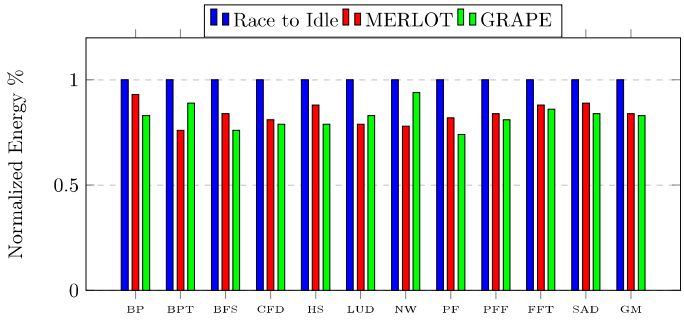


Fig. 8: MERLOT and GRAPE Energy

VI. Related Work

Heterogeneous systems have become common place because they allow computation to be performed on the most energy efficient processor. Heterogeneous designs include processors with a single-instruction set and different core types as well as general purpose processors paired with accelerators, like GPUs. The importance of such architectures has led to extensive research on their timing and modeling [1, 18]. A great deal of recent research addresses scheduling techniques that guarantee timing while minimizing energy on such heterogeneous systems [12, 19, 21, 22, 32–34, 36, 40, 42, 45, 49].

As this paper concerns GPU resource management, we focus on GPU-related real-time research. GPUs are attractive because they provide high compute power in a small form factor that is appropriate for embedded and automotive workloads. There are several approaches that provide software support for real-time processing on GPUs. Ptasks is a framework for OS management of GPUs that supports scheduling tasks across multiple GPUs [44]. Given a GPU task to schedule, Ptasks selects the "best" GPU on which to run the task, but it provides no timing guarantees. Gdev is another approach for making the GPU a "first-class" object supported by an OS scheduler [26]. Gdev provides fairness and some quality-of-service guarantees, but does not enforce timing.

Kato et al. propose several frameworks which combine OS scheduling support for GPUs with timing guarantees [24, 25]. TimeGraph is a device-driver based approach that supports priority-based GPU scheduling to ensure hardware isolation of time-critical GPU tasks [25]. RGEM is a user-space approach that supports real-time GPU scheduling, by ensuring the highest priority tasks are assigned to the GPU first [24]. RGEM also breaks long copy operations into *chunks* that can be pre-empted. Elliott et al. propose GPUSync, which provides real-time scheduling across one or more GPUs and supports sophisticated operations like copying data from one GPU to another [14]. GPUSync has been used as a platform for real-time vision applications like those required in autonomous vehicles [15]. GPES is another software framework designed to support real-time processing on GPUs by making long-running kernels pre-emptible [54]. While all of these approaches benefit from the fact that GPUs are more energy efficient that CPUs for tasks like computer vision, none of these approaches explicitly address or reduce the energy of the GPUs they manage. Maghazeh et al. do propose an energy-

aware technique for scheduling tasks across both CPU and GPU [37], but it is a software technique and suffers from the same drawbacks as other software techniques: specifically, that software has limited opportunities to take advantage of better than worst case behavior for energy savings as described in Section II.

MERLOT differs from this prior work on GPU scheduling by enforcing software defined timing requirements *in hardware* to reduce energy consumption compared to software-only approaches. Hardware has more information about how resources are used by a kernel and it can make faster decisions than software. Thus, where software must be conservative to provide hard timing guarantees, hardware has more opportunities to observe execution and reduce resources to save energy when inputs are not worst case. In that sense, the prior software approaches and our proposed hardware approach should be complementary: software systems like GPUSync can focus on orchestrating computation across multiple GPUs and CPUs, while delegating low-level timing and resource management of individual kernels to hardware, which is better equipped to handle these tasks.

Because it works at the hardware-level, MERLOT is similar to prior approaches that guarantee timeliness for various hardware structures. Examples include predictable memory controllers [13, 28], network-on-chip design to support predictable latency [30], and several processor designs that add timing instructions into the instruction set architecture, including PRET [35] and FlexPRET [55]. MERLOT shares the most commonalities with these last approaches as it directly exposes control of timing to software in the form of registers that specify latency requirements for kernels launched on the GPU. The major differences between PRET/FlexPRET and MERLOT are that (1) MERLOT is designed for GPUs rather than embedded CPUs and (2) MERLOT not only delivers predictable timing but does so while tailoring resource usage to reduce energy consumption.

Of the vast array of prior work hardware approaches for reduced GPU energy consumption, the two most similar are Equalizer [48] and GRAPE [46]. Equalizer adds hardware resource management to minimize GPU energy consumption, but provides no timing guarantees—actually reducing performance in many cases. GRAPE provides hardware support for soft real-time guarantees while minimizing energy consumption. GRAPE uses a control theoretic design and, in some, cases it cannot deliver the requested timing. Unlike these approaches, MERLOT provides hard real-time guarantees and delivers the required performance on GPU applications which fail timing with GRAPE. MERLOT, however, must be more conservative than GRAPE and cannot reduce energy consumption by the same amount. For example, GRAPE will detect cache-limited applications adjust CTA scheduling to further reduce energy consumption, but this can cause timing violations if GRAPE misclassifies an application.

VII. Conclusion

The computational demands of embedded systems like autonomous 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.

References

[1] Y. Abe, H. Sasaki, S. Kato, K. Inoue, M. Edahiro, and M. Peres. “Power and Performance Characterization and Modeling of GPU-Accelerated Systems”. In: *IPDPS*. 2014.

[2] H. Aydi, P. Mejía-Alvarez, D. Mossé, and R. Melhem. “Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems”. In: *RTSS*. 2001.

[3] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. “Analyzing CUDA workloads using a detailed GPU simulator”. In: *ISPASS*. IEEE. 2009.

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

[5] E. Bini, G. C. Buttazzo, and G. Lipari. “Minimizing CPU energy in real-time systems with discrete speed management”. In: *ACM Trans. Embedded Comput. Syst.* 8.4 (2009).

[6] D. Bishop. “Fixed point package user’s guide”. In: *Packages and bodies for the IEEE* (2006), pp. 1076–2008.

[7] G. C. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency: Predictability vs. Efficiency*. Springer, 2006.

[8] Y. Cao. *Predictive Technology Model for Robust Nanoelectronic Design*. Springer Science & Business Media, 2011.

[9] S. Che and M. Boyer. “Rodinia: A Benchmark Suite for Heterogeneous Computings”. In: *IISWC*. 2009.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. “Rodinia: A Benchmark Suite for Heterogeneous Computing”. In: *IISWC*. 2009.

[11] J.-J. Chen, M.-J. Kao, D. Lee, I. Rutter, and D. Wagner. "Online Dynamic Power Management with Hard Real-Time Guarantees". In: *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*. Ed. by E. W. Mayr and N. Portier. Vol. 25. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 226–238.

[12] J. Chen, A. Schranzhofer, and L. Thiele. "Energy minimization for periodic real-time tasks on heterogeneous processing units". In: *IPDPS*. 2009.

[13] L. Ecco, A. Kostrzewa, and R. Ernst. "Minimizing DRAM Rank Switching Overhead for Improved Timing Bounds and Performance". In: *ECRTS*. 2016.

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

[15] G. A. Elliott, K. Yang, and J. H. Anderson. "Supporting Real-Time Computer Vision Workloads Using OpenVX on Multicore+GPU Platforms". In: *RTSS*. 2015.

[16] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. "Dark silicon and the end of multicore scaling". In: *ISCA*. 2011.

[17] S. Funk, V. Berten, C. Ho, and J. Goossens. "A Global Optimal Scheduling Algorithm for Multiprocessor Low-power Platforms". In: *RTNS*. 2012.

[18] C. Gerum, O. Bringmann, and W. Rosenstiel. "Source level performance simulation of GPU cores". In: *DATE*. 2015.

[19] J. Henkel, V. Narayanan, S. Parameswaran, and J. Teich. "Run-time adaption for highly-complex multi-core systems". In: *CODES+ISSS*. 2013.

[20] K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. C. Buttazzo. "Applying real-time interface and calculus for dynamic power management in hard real-time systems". In: *Real-Time Systems* 47.2 (2011), pp. 163–193.

[21] W. Huang, J. Chen, and J. Reineke. "MIRROR: symmetric timing analysis for real-time tasks on multicore platforms with shared resources". In: *DAC*. 2016.

[22] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann. "POET: A Portable Approach to Minimizing Energy Under Soft Real-time Constraints". In: *RTAS*. 2015.

[23] R. Jejurikar and R. Gupta. "Dynamic Slack Reclamation with Procrastination Scheduling in Real-time Embedded Systems". In: *Proceedings of the 42Nd Annual Design Automation Conference*. DAC '05. Anaheim, California, USA: ACM, 2005, pp. 111–116.

[24] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. "RGEM: A Responsive GPGPU Execution Model for Runtime Engines". In: *RTSS*. 2011.

[25] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. "TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments". In: *USENIX ATC*. 2011.

[26] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt. "Gdev: First-Class GPU Resource Management in the Operating System". In: *USENIX ATC*. 2012.

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

[28] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh. "A predictable and command-level priority-based DRAM controller for mixed-criticality systems". In: *RTAS*. 2015.

[29] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. "System level analysis of fast, per-core DVFS using on-chip switching regulators". In: *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. 2008, pp. 123–134.

[30] A. Kostrzewa, S. Tobuschat, S. Saidi, and R. Ernst. "Supporting Suspension-based Locking Mechanisms for Real-Time Networks-on-chip". In: *RTNS*. 2016.

[31] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. "GPUWattch: enabling energy optimizations in GPGPUs". In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 487–498.

[32] A. A. Lifa, P. Eles, and Z. Peng. "On-the-fly energy minimization for multi-mode real-time systems on heterogeneous platforms". In: *ESTIMedia*. 2015.

[33] D. Liu, J. Spasic, G. Chen, and T. Stefanov. "Energy-efficient mapping of real-time streaming applications on cluster heterogeneous MPSoCs". In: *ESTIMedia*. 2015.

[34] D. Liu, J. Spasic, P. Wang, and T. Stefanov. "Energy-Efficient Scheduling of Real-Time Tasks on Heterogeneous Multicores Using Task Splitting". In: *RTCSA*. 2016.

[35] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. "A PRET microarchitecture implementation with repeatable timing and competitive performance". In: *ICCD*. 2012.

[36] J. Liu, K. Li, D. Zhu, J. Han, and K. Li. "Minimizing Cost of Scheduling Tasks on Heterogeneous Multicore Embedded Systems". In: *ACM Trans. Embedded Comput. Syst.* 16.2 (2017).

[37] A. Maghazeh, U. D. Bordoloi, A. Horga, P. Eles, and Z. Peng. "Saving energy without defying deadlines on mobile GPU-based heterogeneous systems". In: *CODES+ISSS*. 2014.

[38] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. "Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling". In: *ICS*. 2002.

[39] NVIDIA. *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-centers. 2017.

[40] S. Pagani, M. Shafique, H. Khdr, J. Chen, and J. Henkel. “seBoost: Selective boosting for heterogeneous manycores”. In: *CODES+ISSS*. 2015.

[41] P. Pillai and K. G. Shin. “Real-time Dynamic Voltage Scaling for Low-power Embedded Operating Systems”. In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP ’01. Banff, Alberta, Canada: ACM, 2001, pp. 89–102.

[42] T. Pop, P. Pop, P. Eles, and Z. Peng. “Optimization of Hierarchically Scheduled Heterogeneous Embedded Systems”. In: *RTCSA*. 2005.

[43] *Predictive Technology Model*. http://ptm.asu.edu/.

[44] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. “PTask: operating system abstractions to manage GPUs as compute devices”. In: *SOSP*. 2011.

[45] A. Roy, H. Aydin, and D. Zhu. “Energy-Aware Standby-Sparing on Heterogeneous Multicore Systems”. In: *DAC*. 2017.

[46] M. H. Santriaji and H. Hoffmann. “GRAPE: Minimizing energy for GPU applications with performance requirements”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–13.

[47] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 2004.

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

[49] J. Spasic, D. Liu, and T. Stefanov. “Energy-efficient mapping of real-time applications on heterogeneous MPSoCs using task replication”. In: *CODES*. 2016.

[50] J. A. Stratton and C. Rodrigues. “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing”. In: *IMPACT*. 2012.

[51] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. “Conservation cores: reducing the energy of mature computations”. In: *ASPLOS*. 2010.

[52] K. Yang, G. A. Elliott, and J. H. Anderson. “Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms”. In: *RTNS*. 2015.

[53] P. G. Zaykov, G. Kuzmanov, A. M. Molnos, and K. Goossens. “Run-Time Slack Distribution for Real-Time Data-Flow Applications on Embedded MPSoC”. In: *2013 Euromicro Conference on Digital System Design*. 2013, pp. 39–47.

[54] H. Zhou, G. Tong, and C. Liu. “GPES: a preemptive execution system for GPGPU computing”. In: *RTAS*. 2015.

[55] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. “FlexPRET: A processor platform for mixed-criticality systems”. In: *RTAS*. 2014.