Rinnegan: Efficient Resource Use in Heterogeneous Architectures

Sankaralingam Panneerselvam
University of Wisconsin-Madison
sankarp@cs.wisc.edu

Michael Swift
University of Wisconsin-Madison
swift@cs.wisc.edu

ABSTRACT

Current processors provide a variety of different processing units to improve performance and power efficiency. For example, ARM's big.LITTLE, AMD's APUs, and Oracle's M7 provide heterogeneous processors, on-die GPUs, and on-die accelerators. However, the performance experienced by programs using these processing units can vary widely due to contention from multiprogramming, thermal constraints and other issues. In these systems, the decision of where to execute a task must consider not only execution time of the task, but also current system conditions.

We built Rinnegan, a Linux kernel extension and runtime library, to perform scheduling and handle task placement in heterogeneous systems. The Rinnegan kernel extension monitors and reports the utilization of all processing units to applications, which then makes placement decisions at user level. The Rinnegan runtime provides a performance model to predict the speedup and overhead of offloading a task. With this model and the current utilization of processing units, the runtime can select the task placement that best achieves an application's performance goals, such as low latency, high throughput, or real-time deadlines. When integrated with StarPU, a runtime system for heterogeneous architectures, Rinnegan improves StarPU by performing 1.5-2x better than its native scheduling policies in a shared heterogeneous environment.

1. INTRODUCTION

Systems with heterogeneous processors and a variety of accelerators are becoming common as a solution to the power wall [27, 65]. Intel and AMD place CPUs and GPUs on the same chip [4, 32]. ARM's big.LITTLE architecture [5] includes processors with different microarchitectures on the same die. IBM's canceled WireSpeed processor [20] and Oracle's M7 [43] have accelerators for specific tasks like XML parsing, compression, encryption, and query processing.

In addition to static heterogeneity from different processing units, systems exhibit dynamic heterogeneity as the performance observed by programs can vary over time. Contention for accelerators [25, 56] alters the performance benefits for programs. With more mature programming models [12, 48, 49], more programs will try to leverage specialized processing units and compete for access to them. Virtualization can also lead to contention, as processing units may by shared not just by multiple programs but also by multiple guest operating systems [25]. Furthermore, the data copy and dispatching overhead can greatly diminish performance benefits offered by certain accelerators [59].

In this environment, applications that decide statically where to run code may perform poorly by waiting for a specific processing unit while there are idle resources in the system. For example, an encryption program that offloads work to the GPU may wait if the GPU is used by other programs while there are idle CPUs available. Ideally, applications would select the processing unit offering the performance, considering both the speedup it offers as well as the overhead of moving data, dispatching a task, waiting time for the unit to be available, and the application's share of the unit's time. With modern programming languages that can generate code for multiple targets, such as OpenCL [49], a programmer should be able to write the code once and let the system decide where it should execute.

However, in today's systems, it is difficult for applications to predict the utilization they can achieve on a processing unit: OS kernels generally do not tell applications how much CPU time or how many threads they can use. GPUs and other accelerator are often treated as external devices, and may lack any support for fairly sharing them between applications. It may be up to the device driver for an accelerator to make scheduling decisions, which may not be coordinated with CPU scheduling. While there have been research systems such as PTask and others [25, 41, 56] that perform scheduling for tasks on a single processing unit, they are unable to select between multiple possible units for a task. Conversely, application runtimes for heterogeneous systems can run a task on different processing units [6, 39], but support only static heterogeneity, where the performance of a processing unit does not vary over time.

The goal of our work is to efficiently execute programs on dynamically heterogeneous systems. To that end, we built *Rinnegan*, a system that (1) provides system-level scheduling support for non-CPU processing units such as GPUs and (2) extends heterogeneous runtimes with kernel support to make informed placement decisions. Rinnegan separates resource management, which is performed by the kernel and can enforce global share or priority policies, from task placement decisions, which are performed by the application runtime.

PACT '16, September 11-15, 2016, Haifa, Israel
© 2016 ACM. ISBN 978-1-4503-4121-9/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2967938.2967964

To support high-quality placement decisions, Rinnegan provides information to applications about their expected utilization of a processing unit: how much time they will receive, and with what delay. This allows an application runtime to make an informed decision about whether to use an accelerator; under contention, it may be better to run the task immediately on the CPU rather than waiting for a small share of the accelerator.

However, without control over how processing units are used, Rinnegan cannot accurately predict the utilization an application will receive: another application could monopolize a processing unit. Rinnegan therefore enforces a scheduling discipline on all processing units. For the CPU, this is already performed by the Linux kernel scheduler. For other processing units such as GPUs, Rinnegan adds an agent that can selectively delay or reorder tasks from different application to achieve desired policies, such as priorities, or proportional sharing.

Complementing this kernel support, Rinnegan provides a runtime library that automatically makes task placement decisions for applications. Rinnegan builds a performance model for the system that incorporates the time to transfer data to a processing unit (e.g., data copy to GPU memory) and the overhead of dispatching a task. For each of an application's tasks, Rinnegan builds a simple model predicting its runtime on each type of processing unit. At runtime, Rinnegan combines the utilization information from the OS with this model to predict how an application task will perform on each processing unit and selects the best place for it to execute.

A key advantage of Rinnegan's architecture is that it easily supports applications with different goals: the decision of what is "best" is determined different for each application, so some may choose highest throughput for batch tasks, while others may choose a processing unit that offers the lowest latency or best energy efficiency. Rinnegan also supports adaptable applications that can vary the behavior based on available resources; for example, by reducing fidelity or accuracy under heavy contention. In such cases, Rinnegan can call back into the application notifying it when performance goals are not met or when there are idle resources, allowing it to adapt its behavior.

An interesting benefit of Rinnegan's approach is that applications that achieve the best speedup on a processing unit will naturally use more of it. Applications that receive only a marginal speedup tend to prefer less contended resources, leaving only those that benefit greatly. As a result, even with fully decentralized placement decisions made indirectly by each applications, total system performance is high.

While useful as a stand-alone runtime, Rinnegan is best suited as an execution platform for heterogeneous programming systems. For example, we modified StarPU [6], which distributes tasks among heterogeneous processing units but is not aware of contention, to use Rinnegan. Unmodified StarPU applications, when run with Rinnegan, automatically make placement decision based on system conditions.

The Rinnegan approach advances over past heterogeneous schedulers and runtimes in the following ways. First, the fastest accelerator unit for a stand-alone application may always not be best when multiple programs compete for that accelerator. Runtimes assuming static heterogeneity, or systems that select a processing unit at compile time cannot adapt at runtime. Rinnegan shows that by exposing system state information, such as waiting time for accelerators, placement decisions can be made dynamically. Second, self-placement of tasks enables applications to optimize for different metrics (e.g., throughput for servers, deadlines for media applications, and resource usage for cloud applications). Rinnegan allows application to specify their own scheduling goals, rather than relying on a system-wide policy. Finally, the operating system performs scheduling for compute units to provide fairness between applications. Rinnegan separates scheduling and task placement, which helps achieve soft isolation among applications and simplifies the OS kernel by offloading the job of task placement to applications.

We performed an extensive set of experiments with GPUs and emulated heterogeneous CPUs on a variety of workloads. Rinnegan outperforms native StarPU by 1.5-2x when multiple applications contend for resources. We also show that the performance of the Rinnegan's decentralized design, in which each application makes their own placement decisions, is comparable to an idealized centralized scheduler with global view of the system. We also show that Rinnegan provides performance isolation among applications and enables applications to achieve application-specific performance goals.

2. DESIGN

We designed Rinnegan as a decentralized architecture separating *resource management* (how much access a process gets to a resource), from *task placement* (where tasks run) decisions as shown in Figure 1. The former is left to the OS kernel, which is responsible for isolation among applications, while the latter is pushed closer to applications. To support high-quality placement, the OS publishes information allowing an application to accurately predict its expected performance on all processing units in the system.

Compared to prior systems that perform both resource management and task placement in the kernel [25, 56, 57], this architecture offers several benefits:

- Application frameworks for heterogeneous systems already perform task placement, so this design fits well into existing systems.
- Applications can easily specify their own performance goals or modify their behavior in response to contention, as the placement policy is tightly coupled to the application.
- Performing placement in applications avoid adding complexity to the OS kernel and makes the Rinnegan more easily portable to other operating systems.

Terminology. (i) We use the words *application* and *program* interchangeably. (ii) The term *processing units* refers to all compute units including CPUs, GPUs, and accelerators. (iii) A *task* in Rinnegan refers to a coarse-grained functional block such as a parallel kernel or a function, such as encrypting a block of data, that executes independently and to completion. (iv) We use the term *scheduling* to mean selecting the next task to execute on a processing unit, and for deciding how long that task may run. We use *placement* to mean selecting the processing unit for a task.

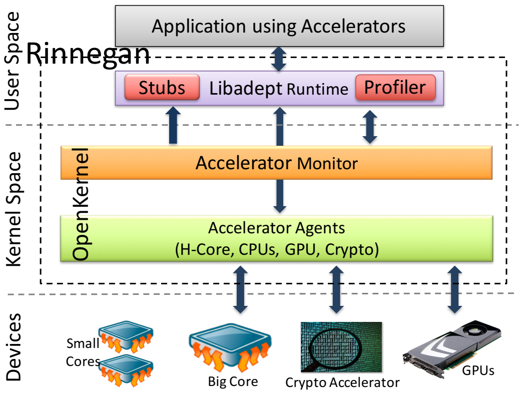

Figure 1: Rinnegan architecture

2.1 Platform Requirements

We designed Rinnegan to target heterogeneous systems with multiprogrammed workloads such as desktops and laptops, mobile devices [21], and shared clusters [34].

Heterogeneous. Rinnegan target architectures that provide heterogeneous processing units in the form of discrete or on-chip accelerators, and both single-ISA and multi-ISA heterogeneous processors. Target systems include unified CPU/GPU chip, such as AMD's APUs [4], systems with discrete GPUs, or with on-chip accelerator devices [20, 64].

Multiprogrammed. Without multiprogramming, a single application can take complete control of the hardware and thus assume the heterogeneity is static and predictable. In contrast, multiprogramming leads to contention and variable performance as applications come and go.

2.2 Resource Management

Rinnegan promotes shared resources like accelerators as first-class entities through its resource management support in the kernel. The major responsibilities of the kernel layer in Rinnegan are resource allocation, enforcing isolation and exposing usage information about processing units. We call this kernel component the *OpenKernel* because of the transparency it provides to applications. The OpenKernel consists of *Accelerator Agents* which act as resource managers and the *Accelerator monitor*, which publishes usage information from agents.

Accelerator agents. In order to support a wide variety of devices with different characteristics, Rinnegan attaches an agent to each unique type of processing unit, such as CPUs, GPUs, and each type of accelerator. Agents perform two key tasks. First, agents act as the scheduler for a type of processing unit and enforce global policy about how much and when each process can use a processing unit. Second, agents gather utilization information about processing units.

Agents implement scheduling policies by controlling when tasks are dispatched to processing units; by delaying tasks from one application, a unit can be used by others. This allow agents to implement global scheduling policies. For example, agents can use shares to provide proportional-share scheduling that is consistent across all processing units, which can provide performance isolation and also guarantee performance for critical applications. Alternatively, shares can also be allocated separately for each unit for finer-grained control. Maximum shares for an application is set by the administrator.

Agents gather and share usage information about the processing units they manage. The agent tracks information needed to predict performance, such as utilization by every application, average size of tasks, and number of applications. This information is provided by the agents to the accelerator monitor (described next) which then publishes it to applications.

Accelerator monitor. The accelerator monitor is a kernel service that publishes usage information from various agents. Some information, such as total utilization of a processing unit, is published globally to all applications. Other information, such as an application's expected share of a unit, is specific to a process; the monitor calculates and shares this information separately with each process. The per-application information reflects the priority, share, or scheduling guarantees of an application. For example, processing units that have been reserved for exclusive use by one application will show up as busy for all other applications.

2.3 Task Placement

Rinnegan follows other runtimes for heterogeneous architectures [6,12,46,49] and employs a task-based programming interface. The programmer or compiler generates task implementations for multiple processing units (not necessary for single-ISA systems). There are several runtimes [12, 49], research systems [17,37] and initiatives like HSAIL [30] that allow code to be written once and then compiled into binaries optimized for different compute devices. For more varied architectures, such as fixed-function accelerators, a programmer may have to code multiple implementations.

The user-mode runtime layer of Rinnegan is contained in the *libadept* library and provides three key services. First, the runtime builds a performance model for every program task on each processing unit, which allows it to predict performance. Second, it provides a simple interface for placing tasks on the best available processing unit. Finally, it allows applications to specify performance goals that guide the placement decision.

Performance model. The runtime builds a model to predict the performance of program tasks on different processing units. Rinnegan measures basic system performance, such as the overhead to dispatch a task and to copy data to processing units that lack coherent memory. For each application, the runtime measures the performance of an application's tasks on different processing units; this allows it to calculate processing time per unit data from the task execution time and the amount of data operated on. These measurements form a model of uncontended (stand alone) performance that can be used to calculate turnaround time of a task on any processing unit in the system, including dispatch overhead, data copying, and actual execution. The stand-alone performance from the model is later combined with utilization information from the accelerator monitor to predict the expected turnaround time even in the presence of contention.

Accelerator stubs. Rinnegan abstracts different processing units into a single procedural interface, so application developers reason about invoking a task but not *where* the task should run. Applications invoke a *stub* interface to launch a task. The overall launch process involves two stages: (a) choosing the right processing unit (b) dispatching task to the chosen unit. The stub considers a processing unit for task offload only if an implementation of the task for that processing unit is available. Stubs use information from the

OpenKernel and the performance model from the profiler to predict the performance of the task on (runnable) processing units. The processing unit that yields the best performance is chosen based on these predictions, and the task is offloaded to that unit by the stub. The second stage deals with the actual dispatch of the task. The Rinnegan runtime contains dispatch and data transfer routines specific to an agent (and thus specific to a processing unit). The stub handles transferring data to the unit if necessary (e.g., it lacks coherent memory and the data is not already at the unit) and then dispatching the task using these routines.

Rinnegan applications can specify performance goals to libadept, which are used by the stub in selecting the best placement for a task. For example, an application may specify maximum throughput, indicating it is willing to wait for an accelerator that provides a high speedup, while others may seek minimum latency and prefer a lower performing CPU that is available immediately.

Optimizer. For tasks that have specific performance targets, such as a desired long-term throughput, Rinnegan provides an *optimizer* that ensures these goals are met. The optimizer is part of the runtime and runs as a thread within the application. It monitors application performance by invoking a *tracker* function provided by the application, which returns the current performance of the application. If performance is below the desired level, the optimizer can either request more resources from the OS (e.g., increase priority or share), or invoke a callback registered by the application to reduce the amount of work (trade-off quality of the results for performance) so performance is acceptable. For example, a compression application may decrease compression to sustain a throughput when insufficient CPU is available. The optimizer can also detect that performance is above what is desired, and again either invoke the OS to release resources or notify the application that it can do more work.

3. IMPLEMENTATION

We implemented Rinnegan as an extension to the Linux 3.4.4 kernel. The code consists of accelerator agents for CPU and GPU, the accelerator monitor, and the libadept shared library linked to user-mode applications. The OpenKernel and the libadept runtime consists of around 2000 and 3400 lines of code respectively. In this section we describe the implementation of Rinnegan, beginning with how agents schedule tasks and then discuss about user-mode task placement.

3.1 Accelerator Agents

We implemented accelerator agents for GPUs and single-ISA heterogeneous CPUs (with standard and fast cores). These agents enforce a scheduling policy by delaying requests, and pass processor usage information to the accelerator monitor. Table 1 shows the statistics generated by the agents. Rinnegan does not yet handle directly accessible accelerators, but disengaged schedulers [41] can be extended to implement the agent functionality for such devices.

GPU agent. The GPU agent manages any GPUs present. As GPU drivers are usually closed binaries that we cannot extend with agent functionality, we therefore implement the GPU agent functionality in a separate kernel-mode component that receives scheduling information (policy, priorities/shares) from an administrator and enforces scheduling policy. The libadept runtime invokes the GPU agent before offloading tasks, which can stall tasks to enforce the scheduling policy. To track device usage (e.g., task size), libadept passes task execution time to the agent after it completes.

Table 1: Details published by agents. (# - Number of)

Agent	Details Published	Visibility
CPU	# active threads at each priority level	Global
	Schedule time ratio for each priority	Global
	# standard cores per application	Private
GPU	Current utilization by an application	Private
	Maximum share allowed for an application	Private
	Average task size	Global
	# active applications	Global
	Runqueue status of each priority queue	Global

The agent calculates GPU utilization by each application from the information passed by the runtime. As short-lived processes do not accumulate much utilization (e.g., *current utilization* is zero), agents also report the average task size on each GPU and the number of applications using the device. We implement three different policies in the agent: *FIFO*, *Priority-based* and *Share-based.*

The FIFO policy is the default policy in GPU drivers and exposes a simple FIFO queue for scheduling tasks from different applications. However, this policy cannot enforce performance isolation if tasks have different execution times: a long task can monopolize the device because current GPU drivers only support context switching at the granularity of an application kernel (i.e., task).

The *Priority-based* policy prevents monopolization by reordering requests to allow higher-priority applications to run before lower-priority ones. However, without preemption support that is not supported in current GPUs, low latency cannot be guaranteed if a long task is running.

The default policy in Rinnegan is *Share-based.* This policy provides soft performance isolation by guaranteeing execution time to each application. The share distribution is maintained by Rinnegan over coarse time scales on the order of hundreds of milliseconds. This is due to the lack of preemption support in GPUs, wherein long-running tasks with low shares can temporarily exceed their share distribution. This policy defines three application classes: CUSTOM, NORMAL and BACKGROUND. In the CUSTOM class, designed for applications with strict performance requirements, applications reserve an absolute share of one or more GPUs, and the total shares on a device cannot exceed 100. Possession of 50 shares on a GPU guarantees the application a minimum of 50% time on the device. The shares unused by the CUSTOM class are given to applications in the NORMAL class, where the remaining shares are distributed proportionally among applications according to the shares they possess. Finally, BACKGROUND applications use the GPUs only when applications of the other two classes are not using the device. To implement this policy, the agent monitors the utilization by each process, and throttles applications using more than their share. Thus, after executing a long task, an application will be blocked from running more tasks to let other applications use the GPU.

CPU agent. The CPU agent supports two classes of CPU cores, *standard* and *fast*, under the assumption that parallel code runs best by having as many cores as possible, while mostly sequential code prefers one or a few fast cores. Rather than providing a new scheduler, the agent uses Linux's native CFS scheduler [13], which already provides priorities and shares. The CPU agent publishes usage information including the number of threads and fraction of time given to threads at each priority level. This information allows the monitor to predict how much CPU time a thread will get on each class of cores. The share based policies (described above) for strong performance guarantees can be implemented by dynamically mapping the amount of shares to a `nice priority` value. This leverages the fact that every priority level has a time slice ratio over other levels.

Additionally, for standard cores, the agent provides hints on the number of standard cores a program can use exclusively, similar to scheduler activations. The calculation is based on a min-funding [67] policy: all cores are allocated to applications based on their priority or share, and unused cores are redistributed. For example, a single-threaded application can use only a single core and other cores it *could use* are instead given to parallel applications. This enables parallel applications to avoid time shares cores in most cases.

Accelerator monitor. The accelerator monitor aggregates information from all agents and publishes it to applications. While Linux already publishes CPU utilization information under `/proc`, we need a higher-performance mechanism given the frequency of access. The monitor shares two pages of data with application using Rinnegan, which have read-only access to the pages. The *global data page* is mapped in all Rinnegan processes and has information about the whole system, such as the average task size on a GPU. The *private page* has process-specific information, such as its maximum allowed utilization (e.g., what fraction of the time it can expect) for each processing unit.

Agents invoke the monitor with the visibility of information, an identifier for the information, and a new value (e.g., <public, GPU average task size, 40 ms>). Agents calculate utilization information at periodic intervals of 10ms, which they immediately push to the monitor. All information calculated by the agents are moving averages. We currently do not synchronize access between the monitor and applications. However, the data is entirely scalar values used as scheduling hints, so races are benign.

3.2 libadept Adaptive Runtime

Every Rinnegan application links to libadept, which consists of the performance model, stubs, and the optimizer.

Performance model. The runtime builds and maintains a performance model for each of the application's tasks. The profiler predicts the task execution time by the amount of data processed by the task; we assume that task execution time is proportional to the amount of data processed, which is true for many applications. For applications where this is not the case, a more sophisticated performance model should be used; we experienced this with the *Sphyraena* workload (Table 4) that executes SQL select queries. The execution time is dependent on the selectivity of the operators which our model did not consider. Rinnegan also allow individual applications to provide their own performance model. During task dispatch, libadept invokes the application to get the predicted task performance on different processing units, which it then combines with information from the monitor. This allows the flexibility of using a custom performance model for every application, such as more mature models [24, 36, 42].

During application startup, we execute an initial set of offloaded tasks from the application on all processing units, similar to Qilin [38]. This generates a set of parameters—the processing time per unit data—that are later used to predict execution time. The model is saved for use across program invocations. For short-lived applications, the developer can request that the model be generated incrementally over multiple invocations of the program, rather than as one step the first time the program runs. A program with a fewer number of tasks is considered short-lived and the profiling stage should not adversely impact such applications.

With this initial data, the performance model generates predictions for task execution time. libadept continuously monitors the accuracy of predictions; if it is off by more than 25% for more than 10 predictions, libadept recalibrates the model by running tasks on different processing units to generate a new set of measurement for the model. Rinnegan does not expect the model prediction to be 100% accurate and accommodates some error in prediction. The amount of error that can be handled depends on the speedup ratio between processing units. More details on the impact of errors in performance model is analyzed in Section 4.6.

libadept runs a generic profiling task during system startup to measure system performance, such as the latency and bandwidth of copying data to/from a GPU and latency of dispatching tasks. For processing units with coherent memory, such as CPUs, the performance model assumes no copy is required. The current implementation of libadept does not account for the contention in I/O bus. This is approximated by a fixed overhead for transfers in addition to the per-byte cost.

Accelerator stubs. Rinnegan stubs are implemented as a single `Accelerate` function that can launch any task on any processing unit. Applications register a task by providing a set of implementations (one per processing unit type). They then invoke `Accelerate` with the task object and its parameters. The `Accelerate` function invokes the performance model to determine where to run the task, moves data to the selected processing unit if necessary, and launches the task on the unit. While we use OpenCL [49] to generate implementations for CPU and GPU, they can also be hand written for better performance

`Accelerate` invokes the performance model to predict the task execution time, and then computes the expected turnaround time as follows:

$$Latency \;=\; Overhead + Data\ Copy \;+\; (Predicted\ Task\ Execution \;*\; (100\ /\ Utilization))$$

The term utilization comes from the agent and is the fraction of time the process can expect to receive on the processing unit. It captures both expected wait time, as wait time is inversely proportional to utilization, as well as its share.

The stub dispatches tasks to a GPU by informing the GPU agent about the start of the tasks, managing any data movement if required and finally invoking the OpenCL or CUDA runtime to submit the task to the kernel driver. For processing units with coherent memory, no movement is required. For those without coherence, such as discrete GPUs, stubs explicitly migrate data to and from the GPU's memory when necessary.

To dispatch tasks onto CPU cores, libadept starts worker threads affinitized to each core in each CPU class (e.g., standard and fast) during program startup. At runtime, it adds tasks to workqueues serviced by the desired worker threads.

The `Accelerate` function also takes a flag to indi-

Table 2: System configurations

Name	System Configuration	Accelerators
asym_config	12 cores in 2 Intel Xeon X5650 chips, 12 GB RAM	2 fast cores–one per socket–at 62.5% duty cycle Remaining 10 cores at 37.5% duty cycle
simd_config	12 cores in 2 Intel Xeon X5650 chips, 12 GB RAM	*GPU-Bulky* (GPU-B): NVIDIA GeForce 670 with 2 GB GDDR5 RAM *GPU-Wimpy* (GPU-W): NVIDIA GeForce 650 with 512 MB GDDR5 RAM

cate whether the task should run synchronously or asynchronously. Asynchronous tasks return an event handle that can be used to wait for their completion. The tasks are queued to an internal scheduler that maintains a central queue. Tasks are processed (the prediction process) in-order from the central queue and then pushed onto processing unit-specific task queues. The application must enforce any data dependencies between tasks. Task execution may not happen immediately upon queuing to the central queue. The internal scheduler keeps the length of the processing unit specific queues low to maintain predictor accuracy by avoiding a long latency between selecting a unit and executing a task.

Optimizer. Rinnegan enables applications to target different performance goals, such as throughput guarantees or minimum execution time (best performance). The optimizer expects two inputs: the performance goal (e.g., 100 MB/Sec or 10 Frames/Sec) and, a tracker function that measures and returns current application performance. The optimizer, part of the libadept runtime, spawns a thread that periodically (every 500ms) invokes the tracker function. The tracker function is to be implemented by every application and it is registered as a callback with the runtime during application start-up. The function is responsible for returning the current application performance (same performance metric — throughput or latency — as that of the target performance) on invocation. Applications without a performance goal always seek maximum performance (minimum execution time) and the optimizer does not run in this case.

The optimizer operates in three phases when the tracker function indicates that an application is not meeting its goal. Each phase is carried out only when the preceding phase fails to improve performance to meet the target. First, the optimizer tries to spread tasks across more processing units. Applications can thus offload multiple (asynchronous) tasks at once to different units. This is not the default behavior, because the libadept tries to minimize resource cost by ensuring performance on minimum resources. Second, for applications with appropriate privileges, it invokes the operating system to increase priority for desired processing units. This phase targets the application to be in CUSTOM class share-based scheduler, and is important if applications need to meet strict performance goals (e.g., throughput or soft real-time). If the second phase does not succeed, for example if the application lacks the privilege to increase its share or all the shares are already allocated, the optimizer notifies the application via a registered callback function (different from the tracker function). The application uses this callback function, which provides achieved performance, to modify its workload. For example, a game could reduce its frame rate or resolution. Upon availability of more resources (when other applications exit), the application automatically gets additional resources since the shares are proportional.

When an application performs above its goal or the optimizer observes that an application is not using its maximum utilization, it performs the same phases in reverse. This allows the application to modify its workload to increase its workload. e.g., frame rate, or to release resources to the OS by relinquishing its shares.

```
void TaskGPU(void *args) {/* Task logic on GPU */}
void TaskCPU(void *args) {/* Task logic on CPU */}
int main() {
  Task *tasks[10];
  for (int i = 0; i < 10; i++) {
     tasks[i] = IntializeAcceleratorTask(SIMD,
         TaskGPU, TaskCPU, args);
     Accelerate(task[i], ASYNC);
  }
  for (int i = 0; i < 10; i++)  {
     WaitForTask(tasks[i]);
  }
}
```

Listing 1: Program using libadept

Data movement. When launching tasks, the stub may need to migrate the task's data to the selected GPU if it lacks coherent access to memory. Moving a task back and forth between processing units can hurt due to excess data copies. This can occur when a task is at the break-even point between two processing units, so a small change in utilization can push the task to switch processors. It can also occur when there are rapid changes in the utilization of a unit and predictions of performance are inaccurate.

Rinnegan implements mechanisms to avoid both causes of task migration. For tasks near the break-even point, Rinnegan dampens movement with a *speedup threshold*: tasks only migrate if the expected speedup is high enough to quickly amortize the cost of data movement. Thus, small performance improvements that require expensive copies are avoided. In addition, for subsequent task offloads, stubs make an aggregate decision that determines where to dispatch the next group of tasks, rather than making the dispatch decision for each task before switching to the new unit. In this *task aggregation*, group size grows with the data movement overhead.

Stability. Rapid changes in utilization can occur when multiple applications simultaneously make an offload decision: they may all run tasks on the same processing unit, leading to poor performance and then all migrate away from the unit. This *ping-pong problem* is common to distributed control systems, such as in routing [35]. Rinnegan takes a two-step approach to resolve the ping-pong problem. First, libadept detects when the predicted runtime is different than the actual runtime. This indicates that the utilization obtained from the monitor was wrong. Second, libadept temporarily uses the *actual* turnaround time of the last task instead of predicted turnaround time. Thus, the first applications to use the unit runs quickly and observe high performance, while those arriving later experience queuing delay from congestion and hence lower performance. Applications with higher delays will tend to choose a different processing unit, while those with less delay stay put.

Table 3: GPU application characteristics. Speedups are relative to the CPU alone, and size is relative to *AES*.

Applications	Task Size	Task Speedup Ratio	
	Ratio	GPU-B	GPU-W
AES	**1**	32	22
LBM	3.33	1.4	0.5
DXT	5.6	16	4.5
lavaMD	20	27	7.5
Grep	33	10	3.3
Histogram	83	12	4

Programmer effort. Rinnegan exposes new interfaces for programmers. We created a simple task-based programming model to help prototype a complete system from programming model to system software. Though writing a program using the new interfaces can be done without much effort as shown in Listing 1, it still requires program modifications for the new interface. To avoid this extra work, we integrated the libadept runtime with the StarPU [6] heterogeneous runtime system. libadept is used as an execution platform; we invoke `Accelerate` from StarPU's dispatch function. However, we did not need to make any changes to the StarPU's interfaces meant for application development. It already incorporates a profiling stage, that we use for Rinnegan' performance model. Thus programs already written for the StarPU model can leverage the Rinnegan's dynamic task placement without any code changes.

4. EVALUATION

We address four major questions in our evaluation: (a) How beneficial is adaptation in a shared environment? (b) Can Rinnegan satisfy application-specific performance goals? (c) Can Rinnegan isolate applications from each other? (d) How well a decentralized scheduler performs? We also evaluate the overhead and accuracy of individual components of Rinnegan.

The experiments in this section focus on understanding the behavior of Rinnegan when applications of different characteristics (speedup, size of the task, long running/short lived) are run together. We evaluate the performance of Rinnegan against other systems such as StarPU and c-sched (a centralized system we wrote to be similar to an Oracle) and also with different scheduling policies. The primary goal is to show how Rinnegan handles resource contention, which can occur in real systems.

4.1 Experimental Methods

Table 2 lists the configurations used in our experiments. We disable Turbo Boost and hyperthreading to avoid performance variability.

Platform. We emulate an asymmetric CPU (*asym_config*) using Intel's clock-modulation feature [31] (in our platforms DVFS cannot be used for a single core) to slow down all cores but the *fast* cores. Similar mechanisms have been used in past research to emulate asymmetric processors [3]. We wanted a single infrastructure with different forms of heterogeneity (asymmetric cores and several accelerators). So, we chose to emulate the asymmetry rather than using a separate real hardware such as big.LITTLE processors [5]. Though none of our experiments make use of both forms of heterogeneity, we believe this infrastructure will serve best for the extension of this work. We emulate powerful cores by setting ten cores to run at 62.5% duty cycle and the remaining two at 37.5%. So, the speedup of fast core is 1.6x over slow core [5]. Where noted, we run some applications at 100% to emulate applications-specific speedups. The *simd_config* configuration comprises two GPUs of different performance and 12 CPU cores. Applications access these processing units through OpenCL, which uses the NVIDIA OpenCL SDK for GPUs and Intel OpenCL SDK for CPUs.

Table 4: Workloads

Name	Description
Blackscholes [9]	Mathematical model for a financial market
Dedup [9]	Deduplication
Pbzip [51]	file compression
AES [1]	AES-128-ECB mode encryption, OpenCL
LBM [63]	Fluid dynamics simulation, OpenCL
DXT [47]	Image Compression, OpenCL
lavaMD [14]	Particle simulation, OpenCL
Grep [60]	Search for a string in a set of files, OpenCL and OMP for CPU
Histogram [60]	Finding the frequency of dictionary words in a list of files, OpenCL and OMP for CPU
Sphyraena [7]	Select queries on a sqlite database, CUDA
EncFS [56]	FUSE based encrypted file system, CUDA
Truecrack [66]	Password cracker, CUDA
x264 [40]	Video Encoder, OpenCL

The GPU workloads are run simultaneously for all experiments performed on *simd_config*. We run workloads continuously, and present the relative throughput (tasks/second) for applications on the system under test compared to running all the applications on GPU-B. For a few experiments, those shown in Figure 3 and Figure 5, we allow some applications to finish in order to show how the system reconfigures.. The application properties are shown in Table 3.

Workloads. We run the workloads listed in Table 4 in a multi-programmed environment to create contention for hardware resources. We select workloads with specific characteristics, such as tasks suitable for fast cores and varying task sizes to demonstrate Rinnegan's capabilities and to exercise different forms of heterogeneity.

Dedup, *Pbzip* and *Blackscholes* are parallel applications with tasks that can benefit from running on powerful CPU cores. We use six workloads for GPUs, described in Table 4. These workloads demonstrate the effect of task size and implementation style. For OpenCL programs, we compile to both GPU and CPU code. The same set of applications was also ported on to StarPU runtime. For CUDA programs, we use a separate implementation for the CPU. Table 3 shows the speedup for these applications when running on GPUs and their average task sizes (relative execution time). The speedup shown is relative to using all 12 CPUs at full performance. We report the average of five runs and variation was below 2% unless stated explicitly.

4.2 Adaptation

A major benefit of Rinnegan compared to existing heterogeneous runtimes is its ability to use application-specific performance models to select the best placement for a task, but at the same time dynamically adapting to runtime conditions. We evaluate how well applications adapt to a set of common contention scenarios.

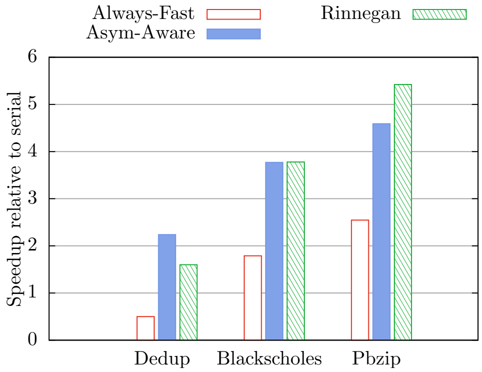

Figure 2: Contention for fast cores.

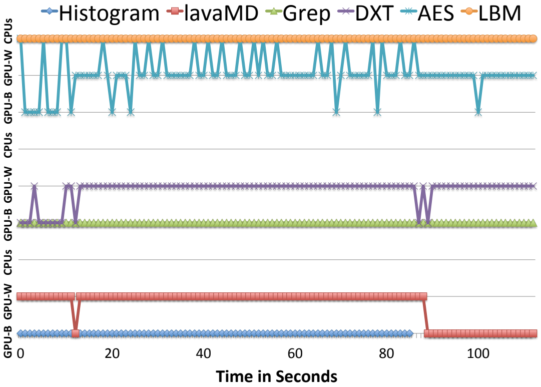

Figure 3: Rinnegan task placement with FIFO policy.

Contention for fast cores. Rinnegan allows applications to accelerate important code regions on a small number of fast cores. We run multiple parallel applications concurrently that have tasks suitable for a fast CPU: the compression phase in *Dedup* and *Pbzip*, and the financial calculation in *Blackscholes*. The speedup of fast core over regular core is 1.6x. Using *asym_config*, we compare three configurations: (a) *Always-Fast* is a compile-time static policy that always runs all tasks on the fast cores, (b) *Asym-Aware* is similar to Global Task Scheduling (GTS [33]) in which it modifies the Linux scheduler to execute tasks on normal cores but migrate them to a powerful core if it becomes idle, and (c) *Rinnegan*, where the stub decides where to run tasks based on speedup achievable. To demonstrate the ability of Rinnegan to prioritize fast cores for applications with better speedup, we made *Pbzip* receive a speedup of 2.65x on the faster cores. This speedup boost was applied for all three configurations. This was done by modifying the scheduler to increase the duty cycle to 100% only when *Pbzip* runs on the faster core.

Figure 2 shows performance normalized to serial versions of the applications running on a regular core. The overall performance of Rinnegan and *Asym-Aware* is comparable though both configurations perform differently on *Dedup* and *Pbzip*. Rinnegan runs the application with the best speedup, *Pbzip*, on the fast cores, which gives that application an 18% speedup compared to *Asym-Aware*. The *Asym-Aware* greedily schedules task on faster cores as they become available without any regard to their speedup. Rinnegan trades-off the performance of other applications for the high speedup application *Pbzip*. The *Always-Fast* performs poorly because it under-utilizes the regular CPU cores while waiting for the fast CPUs. Overall, the system throughput is 3% higher in Rinnegan than *Asym-Aware*.

Contention for accelerators. A key goal of Rinnegan is to manage contention for shared accelerators. In current systems, applications cannot choose between processing units and thus must wait for the contended accelerator. Rinnegan, though, allows applications to use other processing units *if* they are faster than waiting. As a result, applications with large benefit from an accelerator tend to use it preferentially, while applications with lesser benefit will migrate their work to other processing units. Task size also plays a role with the FIFO policy: large tasks dominate the usage on bulky GPU. We ran all the OpenCL GPU workloads concurrently on *simd_config* using the FIFO policy.

In the *Native* system, all applications offload tasks to GPU-B, which leaves GPU-W and the CPUs idle. In contrast, Rinnegan achieves 1.8x better performance because it makes use of GPU-W and the CPU to run tasks as well. To explain these results, Figure 3 plots the processing unit used by each applications from a run where we launch all apps at the same time. *LBM* uses the CPU only exclusively because it gets low performance on the GPUs. In contrast, *Histogram* and *Grep* always uses GPU-B, because of their relatively larger tasks. This causes long delays for other applications, and hence *lavaMD* and *DXT* move to GPU-W. When *Histogram* completes, *lavaMD* switches to GPU-B and shares the device with *Grep*. On the other hand, *AES* switches execution between CPU and GPUs since the amount of data to encrypt varies with every task offloaded by the program. For smaller data, *AES* runs on the CPU to avoid costly data copies. The three column stacks labeled *FIFO* in Figure 6 shows the percentage of time each application gets on different processing units. The native stack at the left shows the default behavior where all tasks are offloaded to GPU-B.

Rinnegan as an execution engine. We integrated Rinnegan as the execution engine for StarPU, a runtime for heterogeneous architectures, to analyze the impact of contention awareness in the runtime. We rewrote all six GPU workloads to run on the StarPU runtime. We compare three different configurations for this experiment. We use two of StarPU's native policies [6, 62]: *DMDA* (deque model data aware) offloads task to the best performing processing unit taking into account the previously queued tasks from the local application and *Eager* employs a task stealing approach where worker threads of any processing unit can run a task as long as the implementation for that unit is available. The third configuration is StarPU with Rinnegan, where task offload decisions are made by Rinnegan based on input from OpenKernel. Rinnegan makes StarPU aware of other applications in the system.

The *DMDA* policy is representative of application runtimes [6, 55] for heterogeneous architectures where tasks are offloaded to the processing unit that performs best for that application (inclusive of all tasks from an application) in isolation. As a result, DMDA always offloads tasks to GPU-B for our workloads since it offers highest speedup. Since the task placement decisions consider only individual task performance but not contention at accelerators (due to other applications), DMDA's performance suffers by not utilizing the lower performing processing units such as GPU-W and CPUs when GPU-B is busy.

The eager policy, similar to runtimes such as Cilk [10] and others [58], employ a task stealing approach for task distribution in heterogeneous architectures. With this policy, the

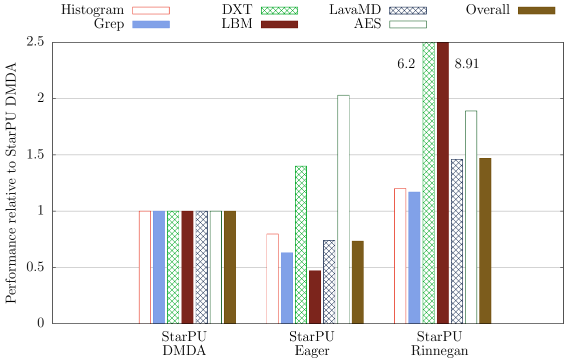

Figure 4: Throughput of StarPU-based systems.

Table 5: Stand-alone performance of CUDA workloads. W/S - Words/s; Q/S - Queries/s; F/S - Frames/s

	Truecrack	Sphyraena	EncFS (read)	x264
GPU-B	320 W/S	38 Q/S	340 MB/S	15 F/S
GPU-W	230 W/S	13 Q/S	190 MB/S	-
CPU	15 W/S	0.2 Q/S	13 MB/S	-

StarPU runtime for each application spawns worker threads for every compute unit in the system (12 workers for every CPU, 2 workers for each GPU). Every worker thread pulls task from the task pool when they are free without any regard to task speedup on that processing unit. As a result, when the GPU is temporarily busy, CPU worker threads pull many tasks that could have run at higher speedup on a GPU. Thus, the eager policy provides the lowest performance of the configurations.

We show the individual applications throughput and the overall system throughput relative to StarPU DMDA in Figure 4. Rinnegan is able to utilize the less-loaded processing units and thus provide better throughput than other policies. As a result, the overall system throughput of StarPU+Rinnegan is 2x and 1.5x better than native policies of StarPU (Eager and DMDA) respectively.

4.3 Application-Specific Goals

The libadept library allows applications to set their own performance goals through an optimizer that guides offload decisions. We demonstrate Rinnegan's ability to support application-specific performance goals by running the CUDA applications—*x264*, *Sphyraena*, and *EncFS*—with their own goals in CUSTOM class and *Truecrack* in BACKGRND class. Native performance of application is shown in Table 5. The goals for each application are:

x264	15 Frames/Sec, can degrade quality for performance.
EncFS	275MB/Sec on sequential reads.
Sphyraena	20 Queries/Sec. minimum
Truecrack	Background: use GPU only when unused.

Of these applications, *x264* can adapt by reducing fidelity and *Sphyraena* uses asynchronous tasks and can spread its work across multiple processing units.

We started the applications in their respective classes where libadept assigns default shares to applications (15 shares on GPU-B each for applications in CUSTOM class and no shares assigned for BACKGRND class applications). We let libadept request shares automatically from agents after the initial assignment. We expect Rinnegan to automatically adapt—find the right set of processing units, or adjust shares, or callback to applications to alter configurations to adjust performance—without any manual intervention. The adaptation of the system is shown in form of a time graph in Figure 5. We start *x264*, *Sphyraena* and *Truecrack* at time zero, and *EncFS* at time 60. When applications finish, we do not restart them so as to release resources for other applications' use.

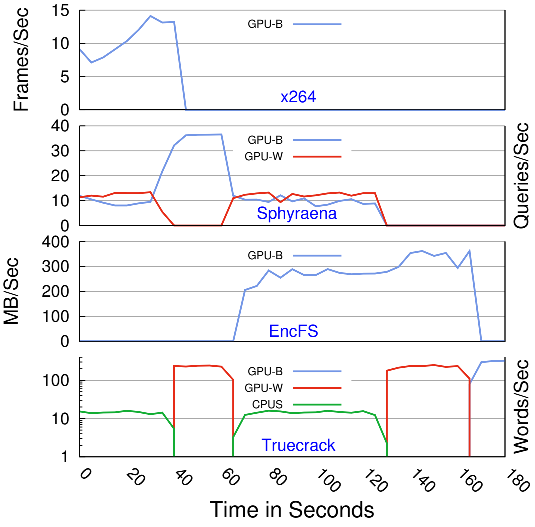

Figure 5: CUDA workload adaptation.

As the application runs, the optimizer detects the lag in performance and attempts to use multiple GPUs. *x264*, as a synchronous application, cannot leverage multiple GPUs. However, *Sphyraena* spreads its tasks across both GPU-B and GPU-W. *x264* receives around 80 shares on GPU-B and *Sphyraena* gets 20 on GPU-B and 100 shares (all) on GPU-W. Around 5th second, the runtime (through optimizer) notifies applications to reduce fidelity to improve performance. *x264* adapts by reducing quality and is able to reach 13 FPS. *Truecrack*, being a background application, runs on CPU since the accelerators (GPUs) are occupied.

When x264 completes, *Sphyraena* receives the whole GPU-B and achieves its goal of more than 20 queries/sec, and *Truecrack* adapts by moving to GPU-W, which greatly increases its throughput. We start *EncFS* at 60 seconds and the optimizer adjusts the shares such that both applications achieve their goals by giving 80 shares of GPU-B to *EncFS*, and remaining of GPU-B and the whole GPU-W were given to *Sphyraena*. Truecrack is forced to move to the CPUs at this point, as both GPUs are totally saturated. Only when *Sphyraena* completes around 120 seconds does *Truecrack* move to GPU-W, and then to GPU-B when *EncFS* completes at 160 seconds. These results demonstrate how applications adapt to the changing use of the system, as well as how the optimizer allows applications to achieve their goals by spreading the work, increasing the share of a processing unit, or reducing the workload.

4.4 Preserving Isolation

Rinnegan only does placement in user-mode and leaves scheduling and policy enforcement in the kernel to protect against poorly behaved applications. To demonstrate Rinnegan's ability to isolate the performance of different applications, we allotted CUSTOM shares on GPU-B in the ratio of 25:25:25:10:10:5 to *Histogram*, *Grep*, *DXT*, *lavaMD*, *LBM*

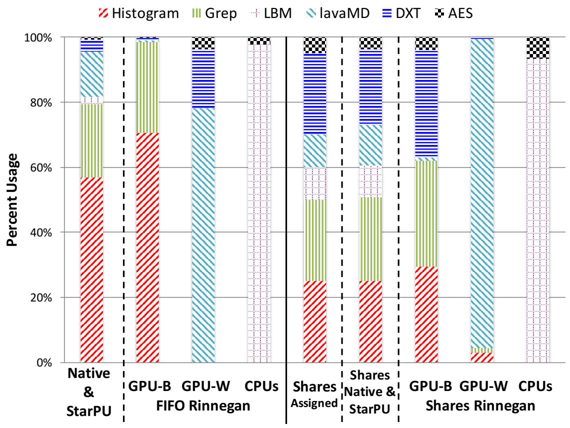

Figure 6: Percentage of time spent on different devices with various policies. Native & StarPU and Shares Native & StarPU columns uses only GPU-B. Shares Assigned column is input.

and *AES*. Such a share ratio was used to show that (a) applications with large tasks (*Histogram*) can be constrained, (b) small tasks (*LBM*) can receive guaranteed share, and (c) isolation is achieved even in the presence of varied task sizes.

Rinnegan applications. The three stacks on the right labeled *Shares Rinnegan* in Figure 6 show the portion of each processing unit used by each application. *Histogram* gets major portion of GPU-B because of its large task sizes in FIFO-based policy. With share-based scheduling, *Histogram*, *Grep* and *DXT* evenly share GPU-B, while *lava-MD* uses GPU-W since it yields better performance than the guaranteed 10 shares on GPU-B. We observe that three applications (*Histogram*, *Grep* and *DXT*) receive more than their assigned 25% on GPU-B (30% each) because other applications decided to offload tasks on GPU-W or CPUs. So, these three active applications enjoy equal share on GPU-B. Also, applications with lower shares of GPU-B, such as *lavaMD* and *LBM*, offload tasks to GPU-W and CPUs respectively since the performance is better than on their 10% of GPU-B.

StarPU applications. We also show how Rinnegan can isolate applications that are not using Rinnegan by running StarPU version of our GPU workloads. However, we modified the workloads to notify the agents about their task execution time. The leftmost stack in Figure 6 shows the utilization received by each application with StarPU's native uses DMDA policy that runs all tasks on GPU-B. The *Shares native & starpu* column in the center-right shows the performance when share-based policy is enforced by the Rinnegan GPU agent: the resulting utilization almost exactly tracks the shares assigned. These results show both that native Linux and StarPU alone cannot or do not enforce performance isolation, while Rinnegan can provide isolation even for applications not actively using it for task placement. We note that without GPU driver support, Rinnegan relies on applications to call into the agent to know start and completion time of the task, and not all applications may do this. However, this functionality can be enforced either through GPU driver modifications or disengaged scheduling [41].

4.5 Decentralized vs. Centralized

Task placement in Rinnegan is decentralized: individual applications make placement decisions based on the state information exposed by the monitor. However, the information can be stale due to the delay between when it is generated and when applications place a task based on the information. In contrast, task placement in a centralized architecture is performed by a single central entity for all applications. It has global knowledge of all tasks dispatched so far and can make accurate decisions for future tasks. We compare the behavior of Rinnegan against a centralized system to understand how much performance is lost due to a decentralized architecture.

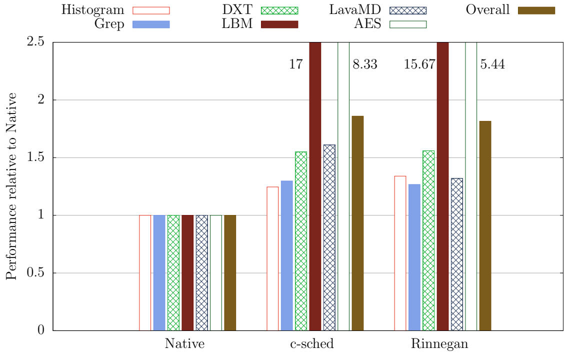

Figure 7: Centralized vs. Decentralized systems.

Centralized scheduler. We built a simple centralized scheduler, *c-sched*, that behaves similarly to PTask [56] or Pegasus [25] but without the complexity and kernel modifications of those systems. libadept maintains task queues per processing unit for each application to track all dispatched tasks from that application. With c-sched, these queues are made global by implementing them on a shared page accessible by all applications, so the per-applications task queues becomes system-wide task queues. This enables c-sched to have a global view of the system and to predict the wait time for every processing unit accurately. Thus the c-sched scheduler is aware of tasks from all applications and can make perfect task placement decisions. However, it requires applications to be cooperative and can be easily gamed by providing erroneous speedup values.

Contention results. Using the same six GPU applications as used in previous experiments with the FIFO policy, we compare the individual applications and the total system throughput of c-sched against Rinnegan. We also consider a native system that offloads task to GPU-B, the highest speedup processing unit. The results, shown as the c-sched and Rinnegan bars in Figure 7, demonstrate that despite being decentralized Rinnegan still performed within 2.5% of a perfect centralized scheduler. Rinnegan performs close to the optimal scheduler is less prone to gaming and supports application-level adaptation, which is difficult in a centralized system. To achieve adaption, application-specific constraints and adaptation techniques would have to be conveyed to the centralized service, making it more complex.

Short-lived applications. Task placement for short-lived applications is hard without the global knowledge of the system since the utilization information can fluctuate. We generated such applications that start, run a small task for 0.5 – 2ms and then exit. We generated small tasks with varying inputs for *AES* and *DXT* workloads. We ran an experiment running multiple such applications periodically to test the overall throughput of the system in terms of tasks processed per second. The c-sched system should provide the best performance since it has a global knowledge of the system. The performance of Rinnegan is only 3.5% lower than c-sched whereas the native system that offloads all tasks to GPU-B is 20% slower than c-sched. Rinnegan performs well even without global knowledge of all workloads by using the average task size and average number of applications metrics rather than utilization information.

4.6 Overhead and Accuracy

We separately measured the overhead of Rinnegan's mechanisms and the accuracy of its profiler.

Overheads. The primary overhead in Rinnegan comes from stubs, which must decide where to dispatch tasks. The overhead of stubs ranged between 1μs when choosing between fast and regular CPU cores to 2μs for selecting among different GPUs. Aggregation can reduce this by changing the dispatch decision less often.

Task profiler accuracy. We measure the difference between the task profiler's predicted run time and the actual task latency including the wait time. Across all our experiments, the prediction error is between 8–16%. The error came from two sources. First, Rinnegan predicts that task size is a linear function of input data size, which is not true for all applications (e.g., *Sphyraena*). Second, we found that the data copy latency to the GPU varied due to contention for the PCIe bus.

To understand the importance of accurate performance predictions, we built an analysis tool to observe the impact of profiler error on task placement. For applications with 10x speedup, when the error rate increases from 5–100% the probability of making an incorrect decision increases by only 0.5%–5%. For the same range of error with applications getting only a 2x speedup, the error probability varies from 2.3%–25%. This shows that error in profiler prediction does not impact placement for tasks with better speedups.

Reducing data movement. Rinnegan reduces the amount of data movement by limiting the frequency with which tasks move between processing units with task aggregation and the speedup threshold. To measure the benefit of Rinnegan's mechanisms for limiting data movement, we disabled the mechanisms and randomly varied *LBM*'s maximum utilization for GPU-B between 65–75%. Its preference flips between GPU-B and the CPUs and performance suffers as a result. Compared to a system without the threshold and aggregation mechanisms, the speedup threshold improves performance by 5%, and task aggregation improves performance by an additional 60%.

Ping-pong problem. We also investigated the impact of the ping-pong problem by comparing Rinnegan against our c-sched centralized scheduler. Because it has global knowledge of the utilization of all processing units, c-sched does not have the ping-pong problem. For the contention experiments described in Section 4.2, we compared the task movements of Rinnegan to c-sched. We found that both systems had similar amounts of task movement. Because of varied task size, applications saw different processing unit utilization and hence made different scheduling decisions. To force a ping-pong problem, we ran five copies of the same *Grep* workload, which offloaded tasks of the same size. Rinnegan's ping-pong avoidance mechanism helped in stabilizing task placement sooner, resulting in the same task throughput same as the c-sched. Without the mechanism, the system took 8-10 task offloads to stabilize, as compared to 2-3 with ping-pong prevention.

5. RELATED WORK

Runtimes for heterogeneous architectures. Many systems provide runtime layers to aid applications in using heterogeneous units (e.g., StarPU [6], OmpSs [18, 53–55], Merge [37], Harmony [17], Lithe [50]). Rinnegan shares the goal of automatically selecting where to run a task based on predicted performance on different compute units. Unlike these runtimes, Rinnegan supports multi-programmed systems where there may be contention for processing units by exposing system-wide resource usage to applications. Runtimes for databases help in task scheduling on heterogeneous architectures [11, 26]. However, they assume the system resources are not shared with other applications.

Other systems abstract the presence of heterogeneous processing units. For example, PTask [56] provide an abstraction for GPU programming that aids in scheduling and data movement. Rinnegan differs by exposing multiple types of processing units, not just GPUs, and by exposing usage to applications to self-select a place to execute. In contrast, PTask manages allocation and scheduling in the kernel.

Some runtimes [55, 58] employ task-stealing for better utilization of all compute units in the system. Sbîrlea et. al in their runtime [58] perform stealing without regard to the speedup of the stealer destination over the task's current placement whereas the OmpSs runtime [55] does not consider the contention caused by other applications. As a result, the stolen task might perform worse at the destination compute unit. Rinnegan employs a task steering approach in contrast to the stealing approach in which tasks are dispatched to an appropriate processing unit based on its standalone performance and also the current load at the processing unit.

Resource scheduling. Many past systems provide resource management for heterogeneous systems (e.g., PTask [56], Barrelfish [8], Helios [44], Pegasus [25]). In contrast to these systems, where the OS transparently handles scheduling and resource allocation, Rinnegan cooperatively involve the participation of applications. It exposes processing unit utilization via the accelerator monitor to allow applications to predict where to run their code, instead of making the choice for the application.

Other systems provide contention-aware scheduling (Grand central dispatch [22], Scheduler activations [2]), where applications or the system can adjust the degree of parallelism. Rinnegan enables a similar outcome, but over longer time scales. Unlike scheduler activations, Rinnegan does not notify applications during change in resource allocation, but allows applications to monitor their utilization. Baymax [15] offers QoS guarantees for critical applications in a shared heterogeneous environment. Unlike Rinnegan, it employs a centralized architecture and focuses on tail latency as the key metric.

Adaptive Systems. Rinnegan is inspired by many previous works in the area of adaptive systems. Odyssey [45] was built as an adaptation platform for mobile systems. Rinnegan employs a similar architecture for shared heterogeneous systems and uses alternate processing units as a mode of adaptation. Application heartbeats [29] tunes applications according to available resources, but does not address heterogeneous platforms nor provide kernel-level scheduling control over non-CPU processing units. Similar adaptive techniques have been employed in databases [23] where new interfaces between OS and databases are designed for better performance. Varuna [61] tackles a similar problem but targets only parallel programs. Performance is improved by dynamically adapting the parallelism in the applications based on the runtime conditions. However, the system runs strictly at user level and does not offer isolation across applications due to the lack of resource management.

Multi Level Scheduling. Many systems [16, 19, 28, 52] distribute resource management across two levels. The first system layer is responsible for resource allocation to ensure isolation among applications. The second layer lies within every application, which decide for themselves how to manage these allocated resources. These systems are generally designed for homogeneous platforms and are concerned with the number/duration of CPUs available but not the type. In most heterogeneous systems, though, applications or runtimes decide where to run their tasks (based on type of processing unit) but are oblivious to the shared environment. Rinnegan integrates the two designs — resource allocation by system layer and task placement by applications — by allowing the system to manage resources and also enable applications to make informed task placement decisions aware of contention.

6. CONCLUSION

Heterogeneity in various forms will be prevalent in future architectures. Rinnegan exposes available heterogeneity with the OpenKernel along with the system state information, and uses libadept to conceal the complexity from application programmers. The kernel retains control over resource management but provides application-level code with the flexibility to execute wherever is best for the application. The decentralized design employed by Rinnegan performs well and at the same time does not require extensive changes to the kernel. As future work, we plan to explore how to better handle power-limited heterogeneous systems.

Acknowledgements

This work is supported in part by National Science Foundation (NSF) grants CNS-1302260, CNS-1117280 and CCF-1533885. We would like to thank our shepherd and the anonymous reviewers for their invaluable feedback. Swift has a significant financial interest in Microsoft.

7. REFERENCES

[1] "Advanced Encryption Standard (AES)," http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '91. New York, NY, USA: ACM, 1991, pp. 95–109.

[3] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating Amdahl's Law Through EPI Throttling," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 298–309. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2005.36

[4] "AMD A-Series Desktop APUs," http://www.amd.com/us/products/desktop/processors/a-series/Pages/nextgenapu.aspx.

[5] ARM Limited, "big.LITTLE Technology: The Future of Mobile," www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Proceedings of the 15th International Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 5704. Delft, The Netherlands: Springer, Aug. 2009, pp. 863–874.

[7] P. Bakkum and K. Skadron, "Accelerating SQL Database Operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10, 2010, pp. 94–103.

[8] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629579

[9] C. Bienia and K. Li, "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors," in *Proc. 5th Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, Aug. 1995, pp. 207–216.

[11] S. Breß and G. Saake, "Why It is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS," *Proc. VLDB Endow.*, vol. 6, no. 12.

[12] "C++ AMP : Language and Programming Model," http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf.

[13] "CFS Scheduler," https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, pp. 44–54.

[15] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," pp. 681–696, 2016.

[16] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, K. Asanović, and J. D. Kubiatowicz, "Tessellation: Refactoring the os around

explicit resource containers with continuous adaptation," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, May 2013, pp. 1–10.

[17] G. F. Diamos and S. Yalamanchili, "Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems," in *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, ser. HPDC '08. New York, NY, USA: ACM, 2008, pp. 197–200. [Online]. Available: http://doi.acm.org/10.1145/1383422.1383447

[18] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[19] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An Operating System Architecture for Application-level Resource Management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95, 1995, pp. 251–266.

[20] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson, "Introduction to the wire-speed processor and architecture," *IBM Journal of Research and Development*, vol. 54, no. 1, pp. 3:1 –3:11, january-february 2010.

[21] A. Frumusanu, "The Samsung Exynos 7420 Deep Dive - Inside A Modern 14nm SoC," http://www.anandtech.com/show/9330/exynos-7420-deep-dive/2.

[22] "Grand Central Dispatch," http://developer.apple.com/library/ios/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html.

[23] J. Giceva, T.-i. Salomie, A. Schupbach, G. Alonso, and T. Roscoe, "COD: Database / Operating System Co-Design," in *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[24] D. Grewe and M. F. P. O'Boyle, "A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL," in *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, ser. CC'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 286–305.

[25] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: coordinated scheduling for virtualized accelerator-based systems," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 2011, pp. 3–3.

[26] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-oblivious Parallelism for In-memory Column-stores," *Proc. VLDB Endow.*

[27] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *IEEE Computer*, pp. 33–38, Jul. 2008.

[28] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308.

[29] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic Knobs for Responsive Power-aware Computing," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 199–212.

[30] "HSA Intermediate Language," https://hsafoundation.app.box.com/s/m6mrsjv8b7r50kqeyyal, May 2013.

[31] Intel Corporation, "Thermal Protection And Monitoring Features: A Software Perspective," http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/54118.htm, 2005.

[32] "Intel Sandy Bridge," http://software.intel.com/en-us/blogs/2011/01/13/a-look-at-sandy-bridge-integrating-graphics-into-the-cpu.

[33] B. Jeff, "big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling," http://www.arm.com/files/pdf/big_LITTLE_technology_moves_towards_fully_heterogeneous_Global_Task_Scheduling.pdf, Nov. 2013.

[34] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a Warehouse-scale Computer," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015, pp. 158–169.

[35] A. Khanna and J. Zinky, "The Revised ARPANET Routing Metric," in *Symposium Proceedings on Communications Architectures &Amp; Protocols*, ser. SIGCOMM '89. New York, NY, USA: ACM, 1989, pp. 45–56. [Online]. Available: http://doi.acm.org/10.1145/75246.75252

[36] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer, "An Automatic Input-sensitive Approach for Heterogeneous Task Partitioning," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, 2013.

[37] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: A Programming Model for Heterogeneous Multi-core Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 287–296.

[38] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009.

[39] "Qualcomm MARE: Enabling Applications for Heterogeneous Mobile Devices," https://developer.qualcomm.com/downloads/whitepaper-qualcomm-mare-enabling-applications-heterogeneous-mobile-devices, Apr. 2014.

[40] E. Marth and G. Marcus, "Parallelization of the x264 encoder using OpenCL," http://li5.ziti.uni-heidelberg.de/x264gpu/.

[41] K. Menychtas, K. Shen, and M. L. Scott, “Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’14. New York, NY, USA: ACM, 2014, pp. 301–316.

[42] N. Mishra, H. Zhang, J. D. Lafferty, and H. Hoffmann, “A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’15, 2015, pp. 267–281.

[43] T. P. Morgan, “Oracle Cranks Up The Cores To 32 With Sparc M7 Chip,” http://www.enterprisetech.com/2014/08/13/oracle-cranks-cores-32-sparc-m7-chip/, Aug. 2014, enterpriseTech.

[44] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, “Helios: Heterogeneous Multiprocessing with Satellite Kernels,” in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP ’09. New York, NY, USA: ACM, 2009, pp. 221–234.

[45] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, “Agile Application-aware Adaptation for Mobility,” in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’97, 1997.

[46] NVidia, Inc., “CUDA Toolkit 4.1,” http://www.developer.nvidia.com/cuda-toolkit-41, 2011.

[47] “NVIDIA OpenCL SDK,” http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/OpenCL/website/samples.html.

[48] “The OpenACC Application Program Interface,” http://www.openacc-standard.org/.

[49] “OpenCL - The open standard for parallel programming of heterogeneous systems,” http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf.

[50] H. Pan, B. Hindman, and K. Asanović, “Composing Parallel Software Efficiently with Lithe,” in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’10. New York, NY, USA: ACM, 2010, pp. 376–387.

[51] “Parallel Implementation of bzip2,” http://compression.ca/pbzip2/.

[52] S. Peter, A. Schüpbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe, “Design Principles for End-to-end Multicore Schedulers,” in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, ser. HotPar’10, 2010, pp. 10–10.

[53] J. Planas, R. M. Badia, E. AyguadÃl’, and J. Labarta, “Self-Adaptive OmpSs Tasks in Heterogeneous Environments,” in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 138–149.

[54] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta, “AMA: Asynchronous Management of Accelerators for Task-based Programming Models,” *Procedia Computer Science*, vol. 51, pp. 130–139, 2015.

[55] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, “SSMART: Smart Scheduling of Multi-architecture Tasks on Heterogeneous Systems,” in *Proceedings of the Second Workshop on Accelerator Programming Using Directives*, ser. WACCPD ’15. New York, NY, USA: ACM, 2015, pp. 1:1–1:11.

[56] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, “PTask: Operating System Abstractions To Manage GPUs as Compute Devices,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 233–248.

[57] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, “A Comprehensive Scheduler for Asymmetric Multicore Systems,” in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys ’10. New York, NY, USA: ACM, 2010, pp. 139–152.

[58] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, “Mapping a Data-flow Programming Model Onto Heterogeneous Platforms,” in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, ser. LCTES ’12. New York, NY, USA: ACM, 2012, pp. 61–70.

[59] M. Shoaib Bin Altaf and D. Wood, “LogCA: A Performance Model for Hardware Accelerators,” *Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2014.

[60] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, “GPUfs: integrating a file system with GPUs,” in *Proc. 18th ASPLOS*, 2013, pp. 485–498.

[61] S. Sridharan, G. Gupta, and G. S. Sohi, “Adaptive, Efficient, Parallel Execution of Parallel Programs,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14, 2014, pp. 169–180.

[62] “StarPU Task Scheduling Policy,” http://starpu.gforge.inria.fr/doc/html/Scheduling.html.

[63] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W. Hwu, “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing,” *Center for Reliable and High-Performance Computing*, 2012.

[64] N. Sun and C.-C. Lin, “Using the Cryptographic Accelerators in the UltraSparc T1 and T2 Processors,” http://www.oracle.com/technetwork/server-storage/archive/a11-014-crypto-accelerators-439765.pdf, Nov. 2007.

[65] M. B. Taylor, “Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse,” ser. DAC ’12.

[66] “Truecrack,” https://code.google.com/p/truecrack/.

[67] C. A. Waldspurger and W. E. Weihl, “An Object-oriented Framework for Modular Resource Management,” in *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS ’96)*, ser. IWOOOS ’96, 1996.