Proteus: Efficient Resource Use in Heterogeneous Architectures

Sankaralingam Panneerselvam and Michael M. Swift
Computer Sciences Department, University of Wisconsin–Madison
{sankarp, 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 on these accelerators can be highly variable due to issues like contention from multiprogramming or thermal constraints. In these systems, the decision of where to execute a task will have to consider not only stand-alone performance but also current system conditions and the program's performance goals such as throughput, latency or real-time deadlines.

We built Proteus, a kernel extension and runtime library, to perform scheduling and handle task placement in such dynamic heterogeneous systems. Proteus enables programs to perform task placement decisions by choosing the right processing units with the *libadept* runtime, since programs are best suitable to determine how to achieve their performance goals. System conditions such as load on accelerators are exposed by the *OpenKernel*, the kernel component of Proteus, to the *libadept* enabling programs to make an informed decision. While placement is determined by libadept, the OpenKernel is also responsible for resource management including resource allocation and enforcing isolation among applications. When integrated with StarPU, a runtime system for heterogeneous architectures, Proteus 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 [36, 62]. Major vendors like Intel and AMD have started packing CPU and GPU on the same chip [2, 9]. ARM's big.LITTLE architecture [3] packs processors made from different microarchitecture on the same die. IBM's Wire-Speed processor [32] and Oracle's M7 [49] are provisioned with accelerators for specific tasks like XML parsing, compression, encryption, and query processing.

In addition to static heterogeneity from different processing units, future systems will exhibit *dynamic heterogeneity* where the performance observed by programs on these processing units can vary over time for various reasons. First, *contention* for accelerators [34, 56] alters the performance benefits for programs. Given the maturity in programming models [4, 11, 14], many programs will try to leverage accelerators and compete for access to them. The growth in virtualization also promotes contention, as accelerators may by shared not just by multiple programs but also by multiple guest operating systems [34]. Second, *power and thermal limits* can prevent all processing units from being used at full power [38] and processing units performance (e.g., processor p-state) can vary as a result. More generally, future chips will have more Dark Silicon [31] where only a fraction of the logic can be used concurrently. Finally, the *data copy* overhead can greatly diminish performance benefits offered by certain accelerators [58].

Building systems to accomodate programs with different metrics such as throughput for servers, latency for interactive applications, deadlines for media applications, or minimum resource usage for cloud applications, in such a highly dynamic environment becomes extremely difficult. However, a major trend that can be leveraged in such heterogeneous systems is that same program task—parallel region of code or encrypting a data chunk—can often be executed on different processing units with different performance or energy efficiency. For example, a parallel region can be run on a GPU or multi-core CPUs and encryption can be performed using a crypto accelerator or AES instructions. Programs designed to be adaptable can thus be scheduled on alternate processing units to achieve their desired goals.

Current operating systems treat accelerators such as GPUs as external devices, and hence provide little support for applications to decide where to best run their code. Similarly, there is little support for heterogeneous processors (e.g., big.LITTLE) today. In research, there are proposed system schedulers to schedule tasks on heterogeneous cores (e.g., [57]) or coordinated CPU/GPU scheduling (e.g., [34, 47, 56]). These systems modify the OS kernel to address contention for resources, but do not support applications that can use multiple *different* processing units, such as running code either on a CPU or a GPU or using an accelerator.

Conversely, application runtimes for heterogeneous systems, automatically place application tasks on a processing unit from user mode [17, 21]. They offload user-defined functional blocks, such as encrypting a chunk of data, compressing an image, or a parallel computation, to the best accelerator or processing unit available. However, existing runtimes assume *static heterogeneity*, where the performance of

a processing unit does not vary over time due to contention from other applications or power limits.

The goal of our work is to efficiently execute programs on dynamically heterogeneous systems. This requires a system that *adapts* to current system conditions, such as load on different processing units. We propose that the operating system alone should *manage* shared resources, such as accelerators, by performing resource allocation, monitoring and isolation. Programs then *dynamically adapt* to available resources by *placing* tasks on suitable processing units to achieve their own performance goals.

We built **Proteus**, a decentralized system that extends Linux by separating resource management from task placement decisions, as a starting step towards this goal. The *OpenKernel* is a kernel component that provides **resource management**—allocating resources, enforcing scheduling policies over all processing units and publishing usage information such as accelerator utilization that enables programs to adapt to system conditions.

The *libadept* runtime layer is the user-mode component of Proteus that makes **task placement decisions**. It presents a simple interface to execute tasks and acts as an abstraction layer for the processing units present. The runtime also builds a performance model of program tasks on different processing units. At runtime, libadept combines this model with system state information from the OpenKernel to predict a task's performance on different processing units. Applications can specify a performance goal that the runtime uses to select the best unit for the task to achieve the application's goal, such as low latency or high throughput.

Applications that benefit more from using a specific processing unit will continue to use it even under contention, while those that benefit only slightly will migrate to other processing units [53]. As a result, even with fully decentralized placement decisions, applications with the highest speedup will receive access to the processing unit.

While designed for stand-alone use, Proteus can also act as an execution model for other programming systems. For example, we modified StarPU [21], which distributes tasks among heterogeneous processing units but is not aware of contention, to use Proteus to make task placement decisions based on system conditions. This enables existing applications written for StarPU to be easily ported to our system without any modifications.

We performed an extensive set of experiments with GPUs and emulated heterogeneous CPUs on a variety of workloads. We show that Proteus enables StarPU to perform better than StarPU's native scheduling policies by 1.5-2x. This illustrates that Proteus is better than systems that rely on static performance. We also show that the performance of the Proteus's decentralized design is comparable to a centralized scheduler with global view of the system. Proteus provides isolation among applications though it allows applications to make task placement decisions. Finally, we show that applications can achieve customized goals in Proteus.

2. Design

The major design goals for Proteus are:

1. *Adaptability.* A program should continuously adapt to changing system conditions when choosing where to execute code.
2. *Efficiency.* Programs that achieve higher speedups on a processing unit should receive more access to it.
3. *Isolation.* The accelerators should be treated as shared resources and isolation among applications using them should be governed by a global policy irrespective of the runtimes used by the applications.
4. *Unintrusive.* Minimal OS changes should be required.

We designed Proteus 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 kernel, which is responsible for isolation, while the latter is pushed closer to applications. The application runtime identifies the best processing unit to execute each task. Unlike other systems that performs resource management and task placement in the kernel [34, 56, 57], Proteus divides the functionalities for the following reasons:

- Moving placement to applications simplifies the kernel, which only needs to handle scheduling [30].
- Selecting the best processing unit requires a performance model of the application. Moving this to application allows them to benefit from application-specific models.
- Applications have the best knowledge of their scheduling goals, such as real time for media applications, and hence are best equipped to choose between different processing units with different performance characteristics [6, 18].
- Some accelerators can be accessed directly from user-mode [32], so invoking a centralized system can increase latency [47].

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) Task in Proteus refers to a coarse-grained functional block such as a parallel region 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 (if possible). We use *placement* to mean selecting the processing unit for a task. Many systems, including Linux, have a per-core scheduler and a separate placement mechanisms for assigning threads to cores.

2.1 Platform Requirements

We designed Proteus to target heterogeneous systems with multiprogrammed workloads such as hand-held devices [19], cloud servers hosting virtual machines [1] and shared clusters [41].

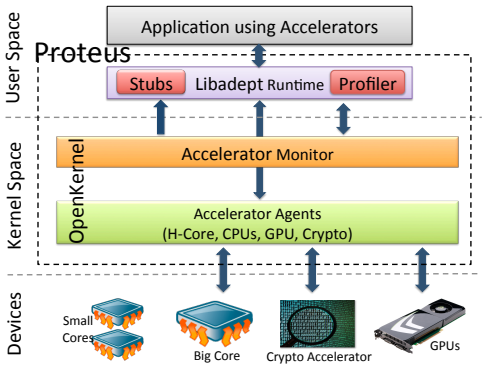

Figure 1. Proteus architecture

Heterogeneous. Proteus 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 [2], systems with discrete GPUs, or with on-chip accelerator devices [32, 61].

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

Proteus 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 Proteus are resource allocation, enforcing isolation and exposing system state 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.

2.2.1 Accelerator Agents

In order to support a wide variety of devices with different characteristics, Proteus attaches an agent to each unique type of processing unit. The agent acts as a sub-scheduler for a pool of similar processing units, such as small CPU cores. The resource allocation decisions are based on a policy suported by every agent. Policy types such as share-based and policy information like higher share for important applications are provided to the agent by the administrators. Policy enforcement allows Proteus to provide isolation even in the presence of a malicious application that could overload the accelerator device. Irrespective of the runtime—OpenCL or CUDA to access GPU—used by an application to offload task to a processing unit, the task has to reach the device through the agent in Proteus. After policy enforcement (like throttling after exhausting shares) the agent uses the device driver to launch tasks on accelerator devices. For asymmetric CPU cores, it creates a worker thread affinitized to each core, to which an application enqueue tasks. Proteus only exposes the ability to select a type of processing unit for a task, similar to processor affinity, but *does not* allow user-mode code to make scheduling or resource allocation decisions.

Agents are also responsible for sharing state information about the processing units it manages, such as the usage of different processing units, to applications. The agent tracks information like the utilization of a processing unit by every application, average size of tasks, and number of applications. This information is provided by the agents to the accelerator monitor, which publishes it to applications. Applications can use this information to predict when they can use the device and for how long.

2.2.2 Accelerator Monitor

The *accelerator monitor* is a centralized kernel service that publishes state information from various agents and enables applications to determine the best processing unit to run a task. 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 private to a 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

Proteus follows other runtimes for heterogeneous architectures [4, 11, 21, 52] 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 [4, 11], research systems [29, 44] and initiatives like HSAIL [8] 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 Proteus 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. These three services combined provide the ability for the runtime to make smart task placement decisions in order to achieve application specific goals.

Performance model. The runtime builds a model to predict the performance of program tasks on different processing units with the help of a task profiler. The model is built by executing the program's tasks on different processing units and extracting information like processing time per unit data from the task execution time and the amount of data the task operated on. In addition, the profiler measures overhead

of using a processing unit such as dispatch time and ata copying. These measurements form a model that can be used to predict performance of a task on any processing unit in the system. We later discuss applications that do not follow this assumption in § 3.2. We should also note that the model output is the stand-alone task performance. The model must be combined with the system state information to predict the actual latency to finish a task on a processing unit.

Accelerator stubs. Proteus 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 the stub interface to launch a task. Stubs use information from the OpenKernel and the performance model from the profiler to predict the performance of the task on all available processing units. The processing unit that yields the best performance is chosen based on these predictions and the stub passes data to the implementation by handling data migration for processing units that lack coherent memory, such as GPUs.

Optimizer. The optimizer is a background service in the runtime and runs as part of the application. The optimizer act as a long term scheduler [55] aiming to achieve performance goals set by the user like a minimum throughput guarantee. The optimizer either requests more or fewer resources from OS, or notifies the application to increase or decrease workload based on resources available. For example, a compression algorithm may reduce its compression factor to keep up with the rate of incoming data, or a latency-sensitive applications may choose to give up on quality of the output [37, 51] by opting for a processing unit with the shortest queuing delay, while throughput-oriented programs may be willing to wait in order to process a larger volume of data.

3. Implementation

We implemented Proteus 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 Proteus, 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 heterogeneous CPU (with both standard and fast cores). These agents enforce a scheduling policy that reorders requests to achieve policy goals and export device usage information to the accelerator monitor. Table 1 shows the details exposed by the agents. Proteus does not yet handle directly accessible accelerators, but disengaged schedulers [47] can be extended to implement the agent functionality for such devices.

GPU agent. The GPU agent manages all GPUs present and currently relies on vendor-specific kernel drivers to submit tasks to the GPU units. GPU drivers are usually closed binaries that we cannot extend with agent functionality. We therefore move the GPU agent functionality into a separate kernel-mode component that receives scheduling information from an administrator and enforces scheduling policy. The runtime calls the GPU agent before offloading any tasks, which can then stall tasks to enforce the scheduling policy. To track device usage, libadept pass task execution time to the agent after the task complete. The runtime's assistance could be avoided if the agent could be better integrated with the GPU driver. Additional support like interfaces to add new scheduling policies might be needed.

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

Table 1. Details published by agents.

The agent calculates the utilization for each application with the information passed by the runtime. Short-lived processes do not accumulate much utilization (*current utilization* is zero) and so, agents also report the average task size on a processing unit 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.

The FIFO model cannot provide equal utilization if tasks have different execution times: a long task can monopolize the device because current GPU drivers only support context switching at kernel (i.e., task) granularity. The *Priority-based* policy prevents monopolization by allowing higher-priority applications to run before lower-priority ones. However, without preemption support from GPU drivers, low latency cannot be guaranteed if a long task is running. Current GPU drivers do not support preemption.

The *Share-based* policy provides strong performance isolation among applications. 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. Shares offered by this policy are specific to a processing unit. The shares unused by the CUSTOM class are given to applications in the NORMAL class, where the remaining shares are equally distributed amoung applications. Finally, BACKGROUND applications use the GPUs only when applications of the other two classes are not using the device.

The share-based policy is the default policy in Proteus. The agent throttles applications using more than their share. Thus, following a long task, an application will be blocked

from running more tasks to let other applications use the GPU. Since the shares are proportional [63], the share for a device is always normalized to the total shares of active applications using the device.

CPU agent. The CPU agent supports two classes of CPU cores, *standard* and *fast*, under the assumption that parallel code will run better on more standard cores, while mostly sequential code will run better on one or a few fast cores. The agent leverages Linux's native CFS scheduler [5] for scheduling on CPU cores. It also publishes state information like the # of threads and ratio of time obtained by threads at each priority level. This information helps to predict contention on fast 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 without contending with other parallel programs. The calculation is based on a min-funding [64] policy: all cores are allocated to applications based on their priority or share, and unused cores are redistributed. For example, if a single-threaded application uses a single core, and other cores it *could use* are instead given to parallel applications. This additional information enables parallel applications to tune their parallelism accordingly.

Accelerator monitor. The accelerator monitor aggregates information from all agents and exposes it to applications. It shares two pages of data with every Proteus application, which have read-only access to the pages. The *global data page* is mapped in all Proteus processes and has global information, such as the average task size on a GPU device. The *private page* has application-specific information, such as its maximum allowed utilization for each processing unit.

The monitor provides a simple interface that takes as parameters the visibility type, information identifier and a new value to be updated. When an agent's state changes, such as when new tasks are submitted or existing ones complete, it notifies the monitor. Agents calculate utilization information at periodic intervals of 25ms, which then gets pushed to the monitor. 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 Proteus application links to libadept, which consists of the *task profiler*, *optimizer* and *stubs*.

Performance model. The runtime includes a profiler that is responsible for building and maintaining a performance model for application tasks. The profiler predicts the task execution time by the amount of data processed by the task. If the model parameters—like processing time per unit data—are not available (start of a new application), the runtime executes the initial set of application's tasks on different processing units similar to the techniques employed in Qilin [45]. The task execution time over the amount of data passed to the task gives the processing time per unit data for that processing unit. We are assuming that the task execution is proportional to the amount of data it uses. The model is saved for use across program invocations. For short-lived applications, the model parameters are calibrated over multiple invocations of the application.

To detect if the performance model must be updated due to program changes, the profiler constantly measures the predicted task execution time against the actual execution time. The model calibration process is repeated if the variation is more than 25% for more than 10 predictions. The model calibration is performed again to recalculate model parameters. libadept runs a generic profiling task during system startup to measure the data copy latency and bandwidth for all processing units in the system. The data copy overhead is set to zero for devices with coherent access to memory (e.g., CPUs), because not explicit copy is required.

The current implementation of libadept does not account for the contention in I/O bus. So, the data copy is considered to have a fixed overhead in addition to the per-byte cost. We also found that our current model did not suit workloads with data-dependent execution time, like *Sphyraena* (Table 4) that performs SQL select query execution. The execution time was dependent on the selectivity of the operators which our model did not consider. We are investigating more mature profilers that could provide better models [33, 43, 48].

Accelerator stubs. The stub interface is responsible for finding the right processing unit for a given task. It requires multiple implementations of the task for different processing units. The code can be identical for the single-ISA architectures or if a compiler can generate code for multiple architectures. Currently, we rely on OpenCL [11] to generate versions of task for GPU and CPUs. But, application developers can also provide hand-optimized implementation for processing units. Task setup entails registering a task identifier with a task object that points to a set of implementations.

Applications launch tasks with the `Accelerate()` function, which takes a task identifier and boolean variable to denote a synchronous or an asynchronous offload. This function uses the model from the profiler and the utilization information from the monitor to choose the best processing unit for the task. The actual task latency is predicted as:

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

The utilization factor implicitly captures the running time as well as the waiting time of the task. The profiler provides the predicted latency and the agent provides the utilization. The stub chooses the current utilization or the maximum utilization value (Table 1) based on the policy (FIFO or share-

based) employed by the agent. The maximum utilization reflects the share given to the application.

Dispatching task onto a GPU involves informing the GPU agent about the task start, managing any data movement 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. libadept calculates data movement cost, based on profiler measurements, when predicting task performance.

To dispatch tasks onto CPU cores, libadept starts worker threads for each CPU class in the system during program startup (e.g., standard and fast), and dispatches tasks to workqueues serviced by the worker threads. The runtime sets affinity mask for these threads to the CPUs in the class.

For asynchronous tasks, stubs return an event handle to the application and queue the task to an internal scheduler that maintains a central queue. Tasks are processed (the prediction process) in-order and then pushed onto processing unit-specific task queues. The application, though, must enforce data dependencies between tasks. Task execution may not happen immediately upon queuing to the central queue. Scheduler keeps the number of pending tasks for a processing unit low to maintain predictor accuracy by avoiding a long latency between selecting a unit and executing a task.

Optimizer. Proteus enables applications to target different performance goals, such as throughput guarantees or minimum execution time (best performance) or energy efficiency. The optimizer expects two inputs: the *expected performance* (e.g. 100 MB/Sec or 10 Frames/Sec) and, a *tracker function* that measures application performance. The optimizer spawns a thread to periodically (every 500ms) invoke the tracker function. libadept implements trackers to record amount of data processed per second and number of tasks run per second. Applications can choose from them or provide a custom performance tracker function.

The optimizer operates in three phases when the tracker function indicates that an application is not meeting its goal. First, the optimizer tries to spread tasks across more processing units. Applications can thus offload multiple (asynchronous) tasks at once to different units. Second, it attempts to increase shares on different processing units by calling into the OS (agents). This phase expects the application to be in CUSTOM class, 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, the optimizer notifies the application via a registered callback function, and the application can 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.

The above three phases are performed in reverse if an application exceeds its goals. This allows the application to modify its workload to increase its workload like frame rate. Applications without any goal are run by default for best performance (minimum execution time) and the optimizer does not run in this case. Applications belonging to other scheduling classes can still leverage the optimizer but a consistent performance cannot be guaranteed. The application performance is mostly best-effort with the option of application reducing fidelity for increased performance.

```
        /*** Libadept ***/
void Accelerate(Task *task, bool sync) {
  /* 1. Iterate through all processing units
     2. Calculate actual task latency on all units
     3. Let punit be processing unit offering minimum
        latency */
  Schedule(task, punit);
}
        /*** Application ***/
void TaskGpu(void *args) {/* Task logic on GPU */}
void TaskCPU(void *args) {/* Task logic on CPU */}
int main() {
    ...
  Task *task = IntializeAcceleratorTask(SIMD, TaskGPU
      , TaskCPU, args);
  Accelerate(task, ASYNC);
  WaitForTask(task);
    ...
}
```

Listing 1. Program using libadept

Data movement. When the optimizer chooses a processing unit for the task, the stub must migrate dependent data to the destination accelerator. Switching between different processing units hurts performance, as data moves back and forth. 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.

Proteus implements mechanisms to avoid both causes of task migration. For tasks near the break-even point, Proteus 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 [42]. The major source of this problem is the staleness in the published information [28]. Proteus tries to reduce the staleness by making the agents update the utilization information at regular intervals rather

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

Table 2. System configurations

Applications	Task Size Ratio	Task Speedup 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

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

than waiting for the tasks to complete. Proteus also takes a two-step approach to resolve the ping-pong problem. First, libadept identifies the problem by detecting when the predicted runtime is different than the actual runtime. This indicates that the utilization obtained from the monitor was wrong. Applications then resolve the issue without any communication between them. Second, on detection of the problem, libadept temporarily uses the *actual* utilization from the last task instead of predicted utilization. Thus, the first applications to use the unit runs quickly and observe high utilization, while those arriving later experience queuing delay from congestion and hence lower utilization. Applications with higher delays will tend to choose a different processing unit, while those with less delay stay put.

Programmer effort. The stub exposes new interfaces for programmers to write their applications. We created a simple task-based programming model to help prototype a complete system from the programming model to the system software. Though writing a program using the new interfaces can be done without much effort as shown in Listing 1, it still requires the program to be rewritten for the new interface. To avoid this extra work, we have started intergrating libadept runtime with currently available runtimes for heterogeneous architectures, StarPU [21]. libadept is used as an execution model - the ability of stubs to combine performance model with the system state to predict task performance has been ported to StarPU's dispatch mechanism. We did not make any changes to the StarPU's interfaces meant for application development. Thus all programs already written for the StarPU model can leverage the smartness of libadept and OpenKernel without any code change. We should note that the tasks (implementations) could rely on other runtimes [11, 27, 52] or be hand-optimized code for a particular accelerator.

4. Evaluation

We try to answer the four major questions through our evaluation results: (a) How beneficial is adaptation in a shared environment? (b) How good is the performance of a decentralized system? (c) Can Proteus isolate the impact of malicious or non-cooperative applications? (d) Can Proteus provide application-specific performance? Finally, we also evaluate the overhead and accuracy of individual components.

Name	Description
Blackscholes [24]	Mathematical model for a financial market
Dedup [24]	Deduplication
Pbzip [12]	file compression
AES [16]	AES-128-ECB mode encryption, OpenCL
LBM [60]	Fluid dynamics simulation, OpenCL
DXT [10]	Image Compression, OpenCL
lavaMD [26]	Particle simulation, OpenCL
Grep [59]	Search for a string in a set of files, OpenCL and OMP for CPU
Histogram [59]	Finding the frequency of dictionary words in a list of files, OpenCL and OMP for CPU
Sphyraena [22]	Select queries on a sqlite database, CUDA
EncFS [56]	FUSE based encrypted file system, CUDA
Truecrack [15]	Password cracker, CUDA
x264 [46]	Video Encoder, OpenCL

Table 4. Workloads

4.1 Experimental methods

Table 2 lists the configurations we use for our experiments. We disable Turbo Boost and hyperthreading to avoid performance variability and to enable control over power usage.

Platform. Processors with a mix of CPU types were not available, so we emulate an asymmetric CPU (*asym_config*) using Intel's clock-modulation feature [39] (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 [20]. We emulate powerful cores by setting all but two 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 [3]. The *simd_config* configuration models a collection of GPU devices and multi-core CPUs. Applications access these processing units through OpenCL, which uses the NVIDIA OpenCL SDK for GPUs and Intel OpenCL SDK for CPUs.

The GPU workloads are run simultaneously for all experiments performed on *simd_config*. Performance (system throughput) is measured as the number of tasks executed over a period of 60 seconds. We restart workload if it finishes before the experiment is completed. We allow early workload completion only for results shown in Figure 3 and Figure 6. We normalize the number of tasks executed based on the task size ratio as 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 like tasks suitable for fast cores, varying task sizes

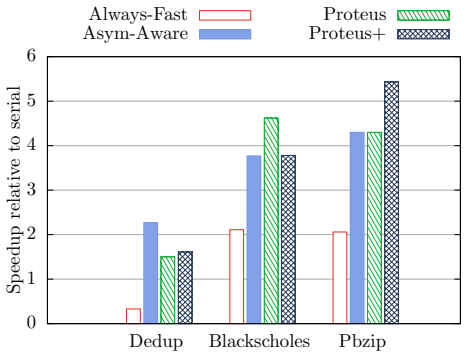

Figure 2. Contention for fast cores.

and real applications to demonstrate Proteus'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 Proteus is the ability to dynamically adapt to run-time conditions. We evaluate how well applications adapt to a set of common scenarios.

Contention for fast cores. Proteus 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* modifies the Linux scheduler to execute tasks on normal cores but migrate them to a powerful core if it becomes idle, and (c) *Proteus*, where the stub decides where to run tasks based on speedup achievable. To demonstrate the ability of Proteus to prioritize fast cores for applications with better speedup, we made *Pbzip* to receive a speedup of 2.65x on the faster cores. This was done by modifying the scheduler to increase the duty cycle to 100% only when *Pbzip* runs on the faster core. The Proteus+ configuration runs *Pbzip* at higher speedup.

Figure 2 shows performance normalized to serial versions of the applications running on a regular core. The overall performance of Proteus and *Asym-Aware* is comparable though both configurations perform differently on *Dedup*

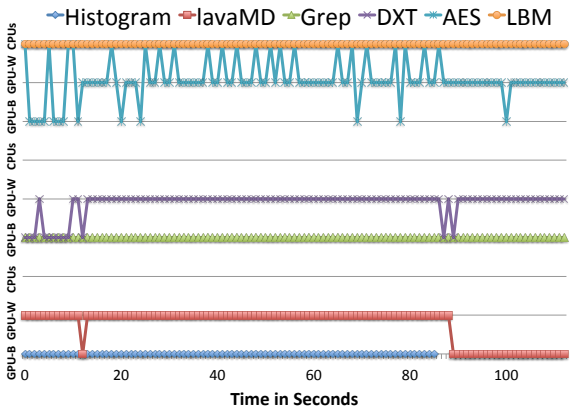

Figure 3. Proteus task placement with FIFO policy.

and *Blackscholes*. This is because of the variation in task sizes for *Dedup* in both configurations. The *Always-Fast* performs poorly because it under-utilizes the regular CPU cores while waiting for the fast CPUs. Proteus+ runs application with the best speedup, *Pbzip* on the fast cores, which gives application a 27% speedup compared to Proteus. Corresponding performance drop can be observed for *Blackscholes*. The results show the high speedup setup only for Proteus+ configuration since performance was similar for other configurations for both the setup.

Contention for accelerators. A key goal of Proteus is to manage contention for a shared accelerator. In current systems, applications cannot choose between processing units and thus must wait for the contented accelerator. Proteus, 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, Proteus 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 5

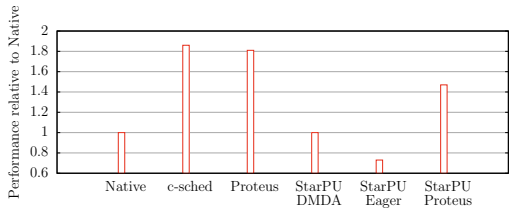

Figure 4. Performance (throughput) of different systems.

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 alone.

Proteus as an execution model. We integrated Proteus as the execution engine for StarPU, a runtime for heterogeneous architectures, to analyze the impact of contention awareness in the runtime. The results are shown in Figure 4.

We ran the same experiment previously mentioned running all six workloads simultaneously on four different systems with a total of six different configurations. c-sched is a simple centralized system that we describe more in Section 4.3. We tested two policies with StarPU [13]: DMDA offloads task to the best performing processing unit 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. All six workloads were rewritten to run on StarPU runtime.

StarPU+Proteus outperforms the native policies of StarPU (Eager and DMDA) by 1.5 and 2x. Ideally, the StarPU on Proteus should have performance similar to c-sched and Proteus. When *LBM* with StarPU runs on CPUs, it suffers performance degradation due to interference from worker threads in StarPU runtime. This results in the reduced throughput for the configuration. The performance of other applications were similar to that of c-sched and Proteus.

4.3 Decentralized vs. Centralized

We want to evaluate the performance of Proteus against a centralized system to understand how much performance is lost due to a decentralized architecture.

Centralized scheduler. We want to build a simple system that behaves like a real centralized systems like PTask or Pegasus but without the complexity and kernel modifications of those systems. We built a centralized scheduler (c-sched) aware of tasks from all applications and can make perfect task placement decisions. c-sched is built on top of libadept and it maintains task information from all applications on a shared page (page mapped onto all applications address space). c-sched consists of global queue per processing unit rather than per application queues. This enables c-sched to have a global view of the system and to predict the wait time for every processing unit accurately. However, it expects applications to be cooperative and the system can be easily gamed by providing erroneous speedup values.

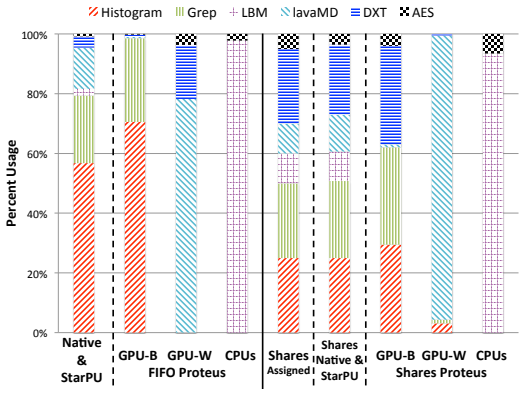

Figure 5. Percentage of time spent on different devices with various policies. Native & StarPU and Shares Native & StarPU columns uses only GPU-B. Shares Assigned column is input.

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 Proteus 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. Proteus performs well even without global knowledge of all workloads by using the average task size and average number of applications metrics from the agents rather than utilization information. This shows that Proteus employing a decentralized design performs as good as a centralized system.

Varying mix of workloads. We want to compare Proteus on a more varied set of workloads. We created workloads by varying three different parameters: Task sizes (large and small), speedup(high and low), and longevity of applications (long running or short-lived). We generated eight synthetic workloads with manually generated performance model for different configurations. All eight workloads were ran and the system throughput was measured. We compared the results from c-sched, which could see all tasks and sleect the best ones for each workload against Proteus. Overall, we found that Proteus performed similar to c-sched whereas the native system performed 40% lower than c-sched. Despite the lack of global knowledge, Proteus achieves performance equal to an omniscient global scheduler.

4.4 Preserving Isolation

Proteus only does placement in user-mode, but leaves scheduling and policy enforcement in the kernel to protect against malicious applications. To demonstrate this, we manually allotted CUSTOM shares on GPU-B in the ratio of

25:25:25:10:10:5 to *Histogram*, *Grep*, *DXT*, *lavaMD*, *LBM* 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.

Proteus applications. The three stacks on the right labeled *Shares Proteus* in Figure 5 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 application 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. Finally, Proteus guarantees isolation by enforcing shares and makes best effort to provide better performance by migrating to alternate processing units.

StarPU applications. The leftmost stack in Figure 5 shows the utilization received by each application when no policy is enforced. StarPU uses DMDA policy that makes it similar to the native system where all tasks are run on GPU-B. The Shares native and starpu column towards the right shows the performance when share-based policy is enforced by the agents. There are two key points: (a) The native system and StarPU do not provide isolation. (b) As seen on the fourth stack, isolation is provided even when applications are not built using libadept.

4.5 Application-Specific Goals

The libadept library allows applications to set their own performance goals through an optimizer that guides offload decisions. We demonstrate Proteus's ability to support application-specific performance goals by running the following applications–*x264*, *Sphyraena*, and *EncFS*—with their own goals in CUSTOM class. We also run *Truecrack* on BKGRND class. Native performance of application is shown in Table 5. The goal of each application is listed below.

- *x264:* Frame rate of 15 Frames/Sec where encoding quality can be given up for performance.
- *EncFS:* Bandwidth of 275MB/Sec on sequential reads.
- *Sphyraena:* Minimum throughput of 30 Queries/Sec.

We expect Proteus 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 from applications in achieving their goals. We started the applications with no shares but let libadept to gather shares. The adaptation of the system is shown in form of a timegraph in Figure 6.

	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	-

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

The first phase until time 40 shows the ability of runtime to enable applications to adapt themselves in reducing fidelity to achieve desired performance goals. x264 has the ability to trade-off its output quality for better performance (FPS). During the initial phase, x264 and sphyraena were run and the goals were set as 15 FPS and 30 Queries/Sec.

On application start-up, libadept allocates default shares for each application. As the optimizer runs, it detects the lag in performance and attempts to use multiple GPUs. x264 being a synchronous application cannot leverage multiple GPUs. However, Sphyranea spreads its tasks across both GPU-B and GPU-W. x264 gets around 80 shares on GPU-B and sphyraena gets 20 on GPU-B and complete GPU-W before per-device shares are exhausted. Around 5th second, the runtime notifies applications to reduce fidelity to improve performance. x264 adapts by reducing quality and is able to reach performance up to 13 FPS. On the other hand, Sphyraena does not adapt and its performance is limited to 22 Queries/Sec.

Encfs cannot run at all in the first phase since all custom shares are consumed. We started Encfs after x264 completed which marks the second phase (from 45th second). Sphyraena continues to run (we reduced the goal to 20 Queries/Sec) along with EncFS. These two goals can be accommodated within the capacity of the system. The optimizer adjusts the shares such that both applications achieve their goals. 85 shares of GPU-B were given to EncFS, and remaining of GPU-B and the whole GPU-W were given to Sphyraena. The Truecrack application in BKGRND class uses the accelerators only when they are not available else it resorts to running on the CPUs.

4.6 Overhead and Accuracy

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

Overheads. The primary overhead in Proteus 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, Proteus predicts that task size is a linear function of input data size, which is not true for all applica-

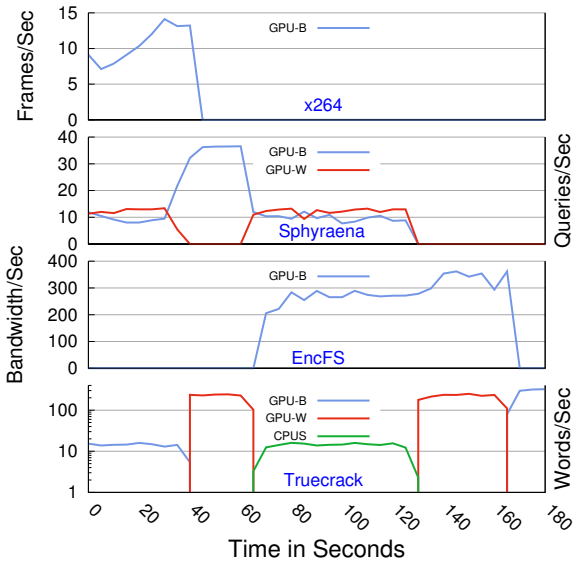

Figure 6. CUDA application performance.

tions (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 increased from 5-100% the probability of making a incorrect decision increases from 0.5% to 5%. For the same range of error rates, the error probability varies from 2.3% to 25% for applications with 2x speedup. This shows that error in profiler prediction does not impact placement for tasks with better speedups. Similar analysis based on task sizes showed that the impact of profiler is low for larger tasks.

Reducing data movement. Proteus reduces the amount of data movement by limiting the frequency with which tasks move between a processing units with task aggregation and the speedup threshold. When we randomly varied the utilization between 65-75%, LBM's preference flipped between GPU-B and the CPUs and performance suffered. 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 Proteus against our c-sched centralized scheduler. Because it has knowledge of all applications, c-sched does not have the ping-pong problem. For the contention experiments described in § 4.2, we compared the task movements of Proteus 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. Proteus'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[21], Merge[44], Harmony[29], Lithe[54] and others). Proteus shares the similarity of automatically selecting where to run a task. Unlike these runtimes, Proteus supports multi-programmed systems where there may be contention for processing units. In addition, Proteus exposes system-wide resource usage to applications to enable them to self-select processing units. Runtimes [25, 35] for databases help in task scheduling on heterogeneous architectures. 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. Proteus 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.

Resource scheduling. Many past systems provide resource management for heterogeneous systems (e.g., PTask[56], Barrelfish[23], Helios[50], Pegasus [34]). In contrast to these systems, where the OS transparently handles scheduling and resource allocation, Proteus cooperatively involves the application. 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[7], Scheduler activations[18]), where applications or the system can adjust the degree of parallelism. Proteus enables a similar outcome, but over longer time scales. Unlike scheduler activations, Proteus does not notify applications during change in resource allocation, but allows applications to monitor their utilization.

Adaptive Systems. Proteus is inspired by many previous works in the area of adaptive systems. Odyssey [51] was built as an adaptation platform for mobile systems. Proteus employs a similar architectures for shared heterogeneous systems and uses alternate processing units as a mode of adaptation. Application heartbeats [37] tunes applications according to available resources. However, it does not handle global resource management. We think the heartbeats work is orthogonal to Proteus. Similar adaptive techniques have been employed in databases [40] where better interfaces between OS and databases are designed for better performance.

6. Conclusion

Heterogeneity in various forms will be prevalent in future architectures and the maturity in programming models is go-

ing to make it easier to program accelerators. Proteus exposes available heterogeneity with the OpenKernel, but uses libadept to conceal it 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 Proteus 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.

References

[1] Amazon EC2. https://aws.amazon.com/ec2/.

[2] AMD Kaveri. http://www.amd.com/us/products/desktop/processors/a-series/Pages/nextgenapu.aspx.

[3] ARM big.LITTLE Processing. http://www.arm.com/products/processors/technologies/biglittleprocessing.php.

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

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

[6] Frame Rate Target Control. http://www.amd.com/en-us/innovations/software-technologies/technologies-gaming/frtc.

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

[8] HSA Intermediate Language. https://hsafoundation.app.box.com/s/m6mrsjv8b7r50kqeyyal.

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

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

[11] 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.

[12] Parallel Implementation of bzip2 . http://compression.ca/pbzip2/.

[13] Task Scheduling Policy. http://starpu.gforge.inria.fr/doc/html/Scheduling.html.

[14] The OpenACC Application Program Interface. http://www.openacc-standard.org/.

[15] Truecrack. https://code.google.com/p/truecrack/.

[16] Advanced encryption standard (AES). http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

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

[18] 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. *ACM TOCS*, 10(1):53–79, Feb. 1992.

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

[20] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *Proc. of the 32nd ISCA*, pages 298 – 309, June 2005.

[21] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proc. 15th Euro-Par*, pages 863–874, Aug. 2009.

[22] 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*, GPGPU '10, pages 94–103, 2010.

[23] A. Baumann, P. barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proc. 22nd SOSP*, Oct 2009.

[24] 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.

[25] S. Breßand G. Saake. Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. *Proc. VLDB Endow.*, 6(12).

[26] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. IISWC*, pages 44–54, 2009.

[27] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), January–March 1998.

[28] M. Dahlin. Interpreting Stale Load Information. In *IEEE Transactions on Parallel and Distributed Systems*, volume 11, Oct. 2001.

[29] G. F. Diamos and S. Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proc 17th HPDC*, pages 197–200, 2008.

[30] 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*, SOSP '95, pages 251–266, 1995.

[31] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proc. ISCA*, June 2011.

[32] 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*, 54(1):3:1 –3:11, january-february 2010.

[33] D. Grewe and M. F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Proc. 20th CC/ETAPS*, pages 286–305, 2011.

[34] 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*, pages 3–3, 2011.

[35] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*

[36] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, pages 33–38, July 2008.

[37] 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*, ASPLOS XVI, pages 199–212, 2011. ACM.

[38] Intel Corp. Intel turbo boost technology 2.0. `http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`.

[39] Intel Corp. Thermal protection and monitoring features: A software perspective. `http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/54118.htm`, 2005.

[40] Jana Giceva and Tudor-ioan Salomie and Adrian Schpbach and Gustavo Alonso and Timothy Roscoe. COD: Database / Operating System Co-Design. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[41] 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*, ISCA '15, pages 158–169, 2015.

[42] A. Khanna and J. Zinky. The revised ARPANET routing metric. In *Proceedings of ACM SIGCOMM*, 1989.

[43] 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*, ICS '13, 2013.

[44] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *Proc. 13th ASPLOS*, pages 287–296, 2008.

[45] 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*, MICRO 42, 2009.

[46] Marth, Erich and Marcus, Guillermo. Parallelization of the x264 encoder using OpenCL. `http://li5.ziti.uni-heidelberg.de/x264gpu/`.

[47] K. Menychtas, K. Shen, and M. L. Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. In *Proc. 19th ASPLOS*, 2014.

[48] 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*, ASPLOS '15, pages 267–281, 2015.

[49] 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.

[50] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proc. of the 22nd SOSP*, Oct. 2009.

[51] 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*, SOSP '97, 1997.

[52] NVidia, Inc. CUDA toolkit 4.1. `http://www.developer.nvidia.com/cuda-toolkit-41`, 2011.

[53] A. Odlyzko. Paris metro pricing for the internet. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, pages 140–147, 1999.

[54] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with lithe. In *Proc PLDI*, pages 376–387, 2010.

[55] 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*, HotPar'10, pages 10–10, 2010.

[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*, pages 233–248, 2011.

[57] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proc. EuroSys*, 2010.

[58] M. Shoaib Bin Altaf and D. Wood. Logca: A performance model for hardware accelerators. *Computer Architecture Letters*, PP(99):1–1, 2014.

[59] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: integrating a file system with gpus. In *Proc. 18th ASPLOS*, pages 485–498, 2013.

[60] 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.

[61] 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.

[62] M. B. Taylor. Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse. DAC '12.

[63] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 1–11, Nov. 1994.

[64] 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)*, IWOOOS '96, 1996.