Processor designs are moving away from homogeneity and are embracing heterogeneity. Such designs can take various forms like asymmetric CPU clusters targeting different power-performance trade-offs (e.g. big.LITTLE), programmable accelerators (e.g. GPU, DSP), fixed function accelerators (e.g. crypto) and custom logic (e.g. FPGA). Heterogeneous processors offer better performance and energy efficiency than regular multi-cores due to the shift towards specialization. Heterogeneity has become prevalent in computing environments from data centers to personal devices and across varied business environments from Wall street to entertainment.

Maximum performance and energy benefits can be achieved from a heterogeneous processor by matching application characteristics with hardware characteristics and vice versa. The challenge is that the magnitude of heterogeneity, as seen by the software or as exposed by the hardware, can vary at runtime, referred to dynamic heterogeneity. Such variation arises for different reasons such as processor-level reconfiguration in dynamic processors, performance variability due to hardware sharing, and the inability to use compute units at full performance in power-constrained architectures. It is a daunting task to address the varying needs of multiple applications – throughput for servers, minimum resource usage for cloud applications – in such a dynamic heterogeneous environment. The abstractions in current operating systems are not enough to capture the hardware intricacies or physical constraints, and application frameworks are not adaptable enough to sustain such an environment.

My vision is that *applications should achieve their performance and energy goals irrespective of the heterogeneous environment in which they run, without much added complexity in application development.* My research takes several steps towards this vision by addressing challenges raised by current and future trends in processor designs in the context of heterogeneity.

- First, I focus on dynamic processors that can reconfigure CPU cores at runtime by pooling resources from multiple cores. The new abstractions and mechanisms from my system ***Chameleon*** [1] enable parallel and sequential applications to benefit from dynamic processors easily.
- Second, I tackle processor set scaling: cores being logically added or removed at runtime from the system. ***Bolt*** [3] offers a low-latency software mechanism for applications to scale rapidly by one or more cores.
- Third, I target the problem of task placement in a shared accelerator-rich environment. ***Rinnegan*** [2] combines per-application stand-alone performance information with current system state to make better runtime placement decisions for applications.
- Finally, I am looking into power-constrained architectures, where processors are over-provisioned with respect to the power limit. With ***Firestorm***, I argue for introducing power and thermal awareness in systems to help the right set of applications to get their desired performance.

My research experience spans the software stack, including lower levels of the operating system that interact with the hardware [1], other parts of the operating system kernel such as the scheduler [1–3] and the memory manager [2, 4], and application runtimes [2]. In the rest of this document, I discuss my research contributions and other work in detail, and then outline my future research plans.

Research Contributions

Dynamic Processors: Parallel programs can achieve high performance on multi-core processors, whereas single-threaded (sequential) programs are limited by individual core performance. Architects proposed dynamic processors like Core Fusion, TRIPS, and WiDGET that can suit the needs of both sequential and parallel programs by changing core characteristics at runtime. These processors in their native state (split state) expose a large number of smaller cores suitable for parallel programs. At runtime, multiple smaller cores can be combined - pooling microarchitectural resources like the reorder buffer or caches from neighboring cores - to form a powerful core (fused state) suitable for sequential programs. Resource pooling can be observed even in current processors where single threaded performance can be achieved by borrowing power through Intel's turbo boost or by forcing a neighboring hyper-thread to idle state.

Current operating systems assume a static set of processor cores whereas fuse and split operations vary the number of cores exposed to the OS. Though hardware can reconfigure in hundreds of nanoseconds, the software overhead during reconfiguration is hundreds of milliseconds. Also, resource management in the OS treats every individual core as an independent execution context. However, on a dynamic processor

an execution context can now represent multiple cores in a fused state. Current systems do not allow a single thread to be scheduled on multiple cores at the same time nor do they have a way to account for a single thread for using multiple cores.

I built ***Chameleon*** [1] (ASPLOS '12), an extension to the Linux OS that supports dynamic processors with three new capabilities. First, the low-cost *processor proxy* mechanism allows fine-grain reconfiguration, as often as a scheduling quantum. The mechanism sets up an agent called a proxy to virtualize only the external interfaces like interrupts of an offline CPU on another active CPU. The proxy does not schedule or run threads, unlike a VCPU. Second, a new *execution object* abstraction represents an execution context, which may be a single core or fused set of cores. It encapsulates details such as the number of cores, and hardware properties needed to represent an execution context. Every thread requesting an execution object is scheduled by forging the desired execution context by using the proxy mechanism. Finally, resource accounting in scheduling policies should be aware of threads using execution object (e.g. threads scheduled on four cores in a fused state). The *cluster scheduler* creates virtual threads on all constituent CPUs of the execution object to represent the main thread. The main thread can be scheduled to run on an execution object only when all virtual threads are at the top of their corresponding run queues.

Chameleon can reconfigure in 2μs, 100,000 times faster than native Linux, and provides a flexible trade-off between the performance of single-threaded and parallel programs. Overall, Chameleon allows applications to transparently and easily leverage dynamic processors based on their demand for processing.

Processor Set Scaling: There are many benefits to scaling the number of cores in the system dynamically, such as energy savings by turning off cores during low utilization, adding more compute resources during a surge in application demand, turning on/off compute units in a power constrained architecture, and virtual machine scaling.

Hotplug is the mechanism in Linux to support processor set scaling. The mechanism is inherently slow compared to entering or exiting from a processor sleep state. The major overhead is caused by the software due to three major limitations. First, hotplug is *synchronous*. All software operations during the scaling event are carried out in the critical path. Second, it is *pessimistic*. It clears or reinitializes all software structures after every scaling event. Finally, it is *serial* and does not allow multiple scaling events to happen concurrently. The high scaling cost incurred in current systems is amortized by performing scaling at coarser time scales as done in power management policies for mobile devices.

Bolt [3] (USENIX ATC '15) is a new mechanism I built over the Linux hotplug infrastructure for low latency scaling. Bolt classifies software operations during hotplug as critical or non-critical based on their impact on system correctness. Critical operations (e.g. redirecting interrupts or migrating applications threads from an offline core) impact the correctness of the system if they are not handled during the scaling event. However, non-critical operations (e.g. freeing memory regions) do not affect correctness even if they are not handled. Bolt handles non-critical operations asynchronously by deferring such operations and removing them from the critical path. Bolt also overcomes the serial nature of hotplug through its new bulk interface that can add or remove multiple cores at once. The new interface improves performance by avoiding redundant operations (e.g. initializing scheduling structures once rather than for every core individually) and concurrently executing operations (e.g. hardware initialization) on different cores. The scaling latency in Bolt is 400-500μs, achieving 20x speedup over native hotplug mechanism. Faster scaling can enable aggressive policies and allow applications to scale rapidly without violating SLAs.

Accelerator Rich Architectures: The provisioning of accelerators such as GPU, DSP, and video encoders is becoming common in architectural designs. There are two major trends that make task placement harder in accelerator rich systems. First, a single task can run on different compute units like a parallel task running on CPUs or GPU. Second, mature programming models such as OpenCL and C++ AMP allow applications to automatically make use of accelerators. When more applications are accessing accelerators, contention for accelerators can arise. The task placement decision has to consider the performance trade-off on different compute units where the task can run. However, a key challenge is that the trade-offs does not remain the same due to sharing. Current systems and runtimes use standalone performance information to make task placement decisions assuming a single application makes the most use of accelerators. However, it is clear from the above trend that static decisions are not sufficient in a shared environment. A contended accelerator might perform worse than an idle CPU.

I built ***Rinnegan*** [2] (HotPar '12 and ongoing work), a system with a kernel extension and a runtime library, to perform scheduling and task placement in a shared heterogeneous environment. Rather than relying on static performance information alone, Rinnegan combines them with current system state

information including the load on accelerators while making task placement decisions. *OpenKernel*, the kernel component in Rinnegan, allows resource managers (e.g. GPU driver, CPU scheduler) to share the system state information with applications. *Libadept*, user-mode runtime, estimates the stand-alone performance of application tasks on different compute units. The runtime combines the performance estimate with system state information during task offloads to make better task placement decisions. Rinnegan also decouples task placement from scheduling contrary to current systems and perform task placement in applications rather than in the kernel. This enables adaptability in applications (e.g. tradeoff quality for performance due to lack of sufficient resources) and making custom task placement decisions based on the performance goals of applications. Resource management, including scheduling and resource allocation, is still performed in the kernel to ensure proper isolation among applications.

Rinnegan's runtime when integrated with existing programming models like StarPU, boosts their performance up to 2x compared to StarPU's policies that are unaware of contention. The amalgamation of static and runtime information in Rinnegan enables applications to make better task placement decision and achieve better performance in a shared heterogeneous environment.

Power-Constrained Architectures: The compute capacity of current processors is over-provisioned in that the compute units – CPUs, GPU – cannot be used at full performance without exceeding the power limit. The maximum performance of all compute units are capped to stay within the power limit. Current processors only support opportunistic solutions like Intel's Turbo Boost to increase the performance when extra thermal headroom is available due to idle cores. However, high priority applications may not receive more power since hardware is unaware of application semantics. Under such physical limits, an ideal system should multiplex power among compute units – by turning on/off or increasing/decreasing p-state – based on the importance of applications (e.g. run GPU at higher performance than CPUs when GPU-bound program has more priority than CPU-bound programs).

The major reason for enforcing power limits is to control heat dissipation. Though cooling devices like heat sinks and fans are available, the type of cooling in processors varies based on form factors (mobile devices or server farms) or can even vary dynamically (convertible laptops). This can result in thermal hotspots (or overheating) in processor packages. Current processors handle this by throttling all compute units, and therefore all active applications, to avoid any thermal meltdown. Increased power density by deploying high power on fewer compute units and insufficient cooling are the main reasons for thermal hotspots. Malicious or background applications can thus trigger throttling to impact primary applications.

I am building ***Firestorm*** (ongoing work), a system that promotes power and thermal capacity as primary resources in the system. It operates with the goal of choosing the right processor configuration to meet the application goals within the physical limits. Firestorm abstracts power in terms of power tokens and effective power distribution is achieved by forcing applications to acquire tokens before they run. Token collection is dictated through a policy based on application priority. Thus, performance achievable is directly proportional to the amount of tokens collected by the application. Firestorm also models thermal usage of each application based on the compute unit(s) and the p-state at which they are used. The policy caps the tokens obtainable by background applications to limit their thermal usage. This is done to run the primary application without getting throttled. Firestorm thus introduces power and thermal awareness and enables the right set of applications to get more performance in such architectures.

Other Research: Storage-class memory (SCM) technologies are byte addressable storage devices that inherit the performance properties of RAM and persistence properties of storage. ***Aerie*** [4] revisits the software storage stack to evaluate its suitability for the new storage technologies. The work argues for a new file system architecture with direct access to storage for applications since the software overhead dominates the data access latency in current file systems. The new architecture supports application specific storage interface to better exploit the storage performance. I implemented the SCM manager in Aerie that is responsible for allocation, mapping and protection of SCM. A slab allocator manages the storage space by partitioning them into equal sized chunks, the mapping functionality is responsible for memory mapping the storage space into application's address space and the protection enforcement deals with setting appropriate page table protection bits based on the file ACLs.

Future Work

The focus on accelerator-centric computing brings out many opportunities in designing systems. I would like to continue identifying new challenges and build systems for new processor designs, and also explore new ways of programming with accelerators. I am also interested in venturing into new fields such as Internet of Things and explore the potential of applying concepts like resource management in such areas.

Direct Accessible Accelerators: Processors are beginning to support new interfaces to access on-chip shared accelerators as co-processors. Accelerators like the GPU in AMD Kaveri, crypto accelerator in IBM Power 8 and database accelerators in SPARC M7 can be accessed directly from applications through new instructions in the ISA and virtual memory support in accelerators. Such integration of accelerators enables low latency access since the OS is bypassed and improves programmability by avoiding explicit data copies. However, accelerators are virtualized by processors (not software) in such designs and it can impact performance of application tasks on accelerators as well as isolation among applications in the system. Memory virtualization is achieved by the use of TLB to cache the virtual to physical address translations. Few designs also provision data caches to avoid memory accesses. I am interested in studying the impact of cache and TLB misses in the context of short tasks that runs for tens or hundreds of microseconds. I want to explore designs of prefetcher and TLB designs by leveraging the predictable data access patterns for tasks running on accelerators. Also, access to such accelerators is not arbitrated by a resource manager due to kernel bypass. This could impact any isolation guarantees in terms of performance in the system. New scheduler interfaces should be supported in the hardware to preserve low latency access and provide isolation support. However, inputs to the scheduler such as policy type and application priorities (or shares) should be set by the operating system. I plan to explore the idea of generic hardware-software co-designed scheduler interfaces applicable for any form of direct accessible devices.

Interfacing Accelerators: The CPU has been the primary computation engine for most applications. Even with the advent of accelerators, the application thread on the CPU is responsible for co-ordinating computations on accelerators and also transferring data from/to external devices. However, this approach could prove to be costly for applications involving the participation of many accelerators and IO devices to accomplish a single task. Such applications include gesture recognition, augmented reality applications, streaming application over middleboxes, and online deduplication. The legacy co-ordinated approach can lead to redundant data copies, traffic in the interconnect, increased latency and loss in energy efficiency. There are many hardware improvements that obviate the participation of CPU during data movement. They are direct communication through RDMA support or Intel SCIF interfaces, programmable devices like NetFPGA to perform tasks like inline data filtering, and protection and virtualization through SR-IOV and IOMMU. Rather than a CPU-centric application design, I want to explore a graph-style design where data can flow through different computation engines without much hand-holding from the main application thread. This raises some interesting challenges such as how to design communication interfaces for accelerators (and IO devices) to interact with each other, data abstraction and flow control mechanism to handle the variation in compute capability of the source and destination devices.

Smart Homes: Interesting opportunities arise when Internet of Things and cloud analytics are combined. I plan to explore the idea of smart power management in this context. If every watt of power consumption can be attributed to the correct device at all times, such power measurement frameworks can provide a great deal of information such as total energy consumption, device usage pattern, and even energy wastage. The trend in today's households is provisioning multiple power sources such as the power grid with consistent supply of power (pricing can vary based on time of the day), free power through solar panels or fuel based power generators that are usually used as a backup. Interesting policies (e.g. tradeoff device availability to stay within energy budget, schedule activities to run only on solar power based on weather trends or minimizing energy bills by avoiding grid for solar power during grid peak times) can be implemented by combining information from the power measurement framework and the properties of the power sources (cost and supply consistency). Fine-granular power measurement and power multiplexing mechanisms are the two key pieces of the new power management system. The multiplexing mechanism enables devices to draw power from any power source based on the global policy. The overall system can be viewed as a central controller co-ordinating the actions of local power agents based on the policy goals. Such a system raises interesting challenges such as notion of resource containers, constructing cheap power sensors, aggregation and analytics over data from sensors, scaling to larger environments like enterprises, and establishing the new power management framework in current households.

References

[1] Sankaralingam Panneerselvam and Michael M. Swift. Chameleon: Operating System Support for Dynamic Processors. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, 2012.

[2] Sankaralingam Panneerselvam and Michael M. Swift. Operating Systems Should Manage Accelerator. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism*, HotPar '12, 2012.

[3] Sankaralingam Panneerselvam, Michael M. Swift, and Nam Sung Kim. Bolt: Faster Reconfiguration in Operating Systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, 2015.

[4] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.