HeteroMates: Providing High Dynamic Power Range on Client Devices using Heterogeneous Core Groups

Vishal Gupta*, Paul Brett†, David Koufaty†, Dheeraj Reddy† , Scott Hahn†, Karsten Schwan*, Ganapati Srinivasa‡
*Georgia Institute of Technology, Atlanta, GA
†Intel Labs, Hillsboro, OR
‡Intel Corporation, Hillsboro, OR
Email: firstname@cc.gatech.edu, firstname.lastname@intel.com

***Abstract*—This paper presents *HeteroMates*, a solution that uses heterogeneous processors to extend the dynamic power/performance range of client devices. By using a mix of different processors, HeteroMates offers both high performance and reduced power consumption. The solution uses *core groups* as the abstraction that groups a small number of heterogeneous cores to form a single execution unit. Group heterogeneity is exposed as multiple *heterogeneity (H) states*, an interface similar to the P-state interface already used for frequency scaling. An H-state controller governs H-state transitions based on dynamic policies maximizing performance or minimizing power consumption, while a 'core switcher' transparently migrates tasks to the appropriate core, i.e., the one matching the chosen H-state. Experimental evaluations use real-world client applications and a unique experimental testbed comprised of heterogeneous cores and a shared uncore component. Results show that core groups can provide significant performance improvements while also lowering energy consumption for a diverse set of applications when compared to homogeneous processor configurations. Also demonstrated is the importance of 'uncore' power in total SoC power consumption and the need for uncore power scalability when seeking to extend a platform's dynamic power range.**

***Keywords*-heterogeneous; client; power; uncore**

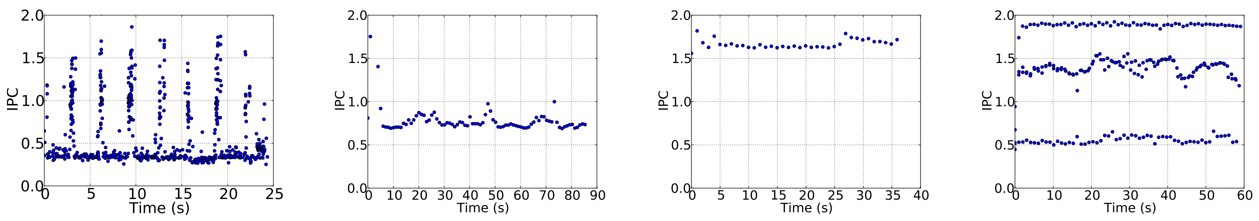
I. Introduction

The ubiquity of handhelds is causing an unprecedented increase in the range of performance demands imposed on mobile platforms, and at the same time, battery life and energy efficiency remain critical concerns. Yet modern processors are typically designed to meet only one, not both, of these two conflicting goals of performance vs. efficiency. In response, chip vendors have adopted heterogeneous multicore processors (HMPs) as their platforms of choice, which consist of cores with different performance/power characteristics. Examples include Variable SMP from NVIDIA [1] and Big.LITTLE processing from ARM [6]. HMPs make it possible for different applications within a diverse mix of workloads to be run on the 'most appropriate' cores [9], [11], [12], [19]. For example, applications not time critical to the user can be run on low-power small cores, while applications with their outputs visible to the user can be allocated to high-performance big cores.

This paper presents the *HeteroMates* system, which uses heterogeneous cores to provide a wider *dynamic power range* for client devices, to meet both their high-performance and low-power demands. Specifically, HeteroMates forms execution units from *core groups*, where each group consists of a small number of (e.g., 2-4) heterogeneous cores. Cores within a core group are exposed to the system as multiple *heterogeneity (H) states*, similar to the P-states used for voltage and frequency scaling. An *H-state controller* module performs H-state transitions based upon workload behavior and user-defined policies. Depending on the selected H-state, the workload is transparently migrated to the appropriate core by a *core switcher*.

H-state abstraction decouples heterogeneity from scheduling such that the scheduler perceives only homogeneous cores. The performance/power differences among cores are transparently handled by a separate H-state driver. H-states can be implemented in hardware, firmware, or software, thereby providing a way to hide heterogeneity from the operating system to support legacy software for wider adoption. Further, core groups allow the system to easily accommodate a variable number of different heterogeneous cores, by adding an H-state for each core. Finally, core groups can also be useful in thermal-constrained scenarios (also known as *dark silicon* [4]) which allow only a fraction of the chip components to be active simultaneously.

HeteroMates is implemented on top of the Linux kernel. Experimental evaluations use a unique, experimental heterogeneous multicore platform comprised of both high and low power cores, along with client applications typically seen in modern end-user devices. Two different usage policies are compared: a performance-driven policy favors high performance for user-facing applications, whereas a power-driven policy favors reduced power consumption and longer battery life. Experimental results demonstrate that by opportunistically utilizing heterogeneous cores, HeteroMates can provide both improved performance and lowered energy consumption for various client applications when compared to homogeneous cores. They also highlight the need for a scalable uncore in order to fully realize the potential gains obtained from the use of heterogeneity.



(a) Intermittent (browse) (b) Sustained-low (openarena) (c) Sustained-high (x264) (d) Multi-threaded (mytube)

Figure 1. Diverse client workload profiles (IPC vs. Time)

II. Motivation

Users perform a wide variety of tasks on mobile devices, resulting in diverse platform demands. Since their battery capacities are severely restricted due to constraints on size and weight, energy efficiency is critical to their usability. To provide extended battery life and at the same time, meet the rapidly increasing demands of high performance mobile use cases, a client device must offer a wide *dynamic power range* – it must be able to operate both in high-performance and in power-savings modes. As explained in detail in Section III, heterogeneous cores can be used to extend the dynamic power range offered by homogeneous processor configurations. In that context, this section presents examples of client workloads (see the list in Table I) and the usage patterns of client devices that motivate the need for a wide dynamic power range and discusses opportunities for exploiting heterogeneous cores.

A. *Client Workloads*

Client applications exhibit highly diverse behavior in their processor usage and performance requirements. These applications can be categorized based on their behavior as described below.

1) Intermittent Workloads: Client devices like cell phones and tablets are typically powered-on for long periods of time, but often perform their heavy processing in short bursts. Web-browsing is an example of such usage, and workloads browse and palbum in Table I belong to this category. A timeline trace of IPC (instructions-per-cycle) for the browse workload is shown in Figure 1(a). Idle periods are marked by low IPC periods, while page-loads correspond to spikes in the graph. Since page-loads generate high IPC activity, a big core can be used for rendering the pages and improving page-load performance, while resorting to a small core during low activity periods to conserve power.

2) Sustained Workloads: These differ from intermittent workloads in that their behavior is uniform over a longer duration. They can be further classified into two sub-categories: sustained-high and sustained-low.

Sustained-low: Client applications like gaming and media playback typically run for a long duration (a few minutes to hours). Moreover, the wide adoption of accelerators allows them to offload significant portions of their computation to accelerators. Figure 1(b) shows the IPC trace of the openarena gaming benchmark. As the observed IPC is low for the application, it can be run on a small core without significant degradation in performance and at lower power.

Sustained-high: Mobile devices are also used for compute-intensive tasks such as media encoding, video editing etc. These applications typically have a high IPC (e.g., see x264 encoder in Figure 1(c)), and their performance scales well on a big core. This makes big cores suitable for these applications when they require high performance, e.g., when they are user-facing, while a small core may provide higher energy-efficiency when they run in background mode (e.g, virus-scan).

3) Multi-threaded Workloads: With increasing numbers of cores on mobile devices, parallelization of client applications is key to further performance enhancement. Such multi-threaded applications also present opportunities for exploiting heterogeneity. 7zip, gmagick, and eclipse are examples of parallel applications. The mytube workload also uses multiple threads for audio, video decoding, and rendering, for instance. Since such threads differ in behavior and needs, their performance will be affected by how they are mapped to different heterogeneous cores. For example, Figure 1(d) shows that various threads within the mytube workload differ significantly in their IPC, which can be leveraged by task mapping and scheduling methods.

B. *Client Devices*

1) Mobility Constraints: Mobility means that client devices will either be powered via wall-power or by battery. Wall-power usage does not impose energy constraints, so that big cores can provide desired high levels of performance. During battery-driven operation, however, a user may be willing to accept lower performance at the benefit of higher battery life. Low-powered energy-efficient small cores may be more suitable under such conditions.

2) Thermal Constraints: Client devices like cell phones and tablets rely on natural cooling. Therefore, these devices are quite sensitive to platform thermal constraints that impose limits on the extent to which it is possible to use power-hungry big cores for sustained periods of time. A small core can be used for moving the execution away from a big core when thermal constraints are violated.

III. Dynamic Power Range

This section describes the use of heterogeneous cores to enable a wide dynamic power range, and the role of the uncore subsystem in achieving the same.

A. Heterogeneous Cores

Modern processors are typically designed to satisfy only one of the two conflicting requirements: high-performance and energy-efficiency. Current low-power cores (e.g., Intel's Atom processor) are energy efficient, but their performance is limited. More powerful big cores like Intel Core® processors provide high performance, but at the cost of higher levels of power consumption. The technological reasons for this is the fact that the power consumption of a processor core consists of static (leakage) power and dynamic (switching) power. During high activity periods, the total power consumption of the device is dominated by dynamic power consumption, while during low activity periods, leakage power becomes a significant component of the total power consumption. Current high performance cores are built from transistors on fast process technologies that have high leakage power and very fast switching times [1]. Such big cores, therefore, consume high leakage power under idle or near-idle conditions, but can provide high performance without significant increase in dynamic power, as shown in Figure 2. Conversely, low power small cores are built from low power process technologies with low leakage power but slower switching times [1]. Such processors consume small amounts of leakage power, but significantly increase dynamic power consumption to provide a high-performance mode (see Figure 2).

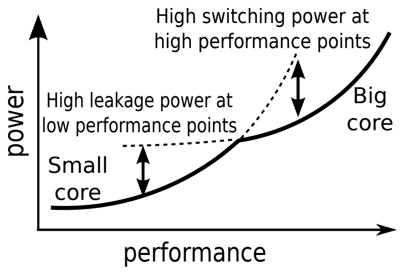


Figure 2. Big cores are less efficient at low activity points, while small cores are less efficient at high activity points. Using a heterogeneous processor provides a wide dynamic power range.

The intuitive outcome is that by using both types of cores, a single platform can be optimized for both high performance and low power consumption. The objective of such a system would be to always use its most efficient cores for the tasks at hand (shown by the solid line in Figure 2). Such a heterogeneous platform exhibits a higher power-performance range than individual big or small cores. This paper explores whether and to what extent the hardware-based arguments for heterogeneity stated above lead to realistically achievable gains on client devices.

B. Beyond Core: Uncore

The dynamic power range offered by a platform consisting of heterogeneous cores can be strongly affected by the *uncore* subsystem present on modern multicore processors. This subsystem consists of components like the last level cache, integrated memory controller, etc. With growing cache sizes, increasing complexity of the interconnection network, and integration of various SoC (system-on-a-chip) components on CPU die, the uncore is increasingly becoming a major power component in total SoC power [14].

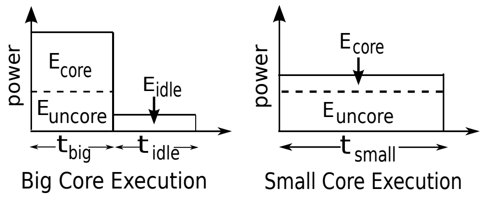


Figure 3. Effect of uncore power on the dynamic power range of heterogeneous cores

Figure 3 illustrates the impact of uncore power on the energy consumption of an application executing on heterogeneous cores. A big core running an application finishes its execution faster and enters a low-power idle state. The same application when executed on a small core takes longer (t_{small}) to finish, which also keeps the uncore active for a longer period of time. If uncore power is substantial in comparison to core power, then the energy gains from running on a small core are strongly affected by the uncore power. For such a system, energy-efficiency gains from small core execution may be offset by the increase in uncore energy consumption due to longer execution time [7]. This observation is in line with prior work that highlights the tradeoff between CPU and system-level power reduction in the context of frequency scaling [3], [15].

Energy consumption for the big core and small core execution for such platforms can be modeled using Equations 1 and 2, respectively. Here, E refers to the energy consumed, t denotes execution time, and P_{core} and P_{uncore} represent average core and uncore power, respectively. P_{idle} is the idle platform power, and t_{idle} is the corresponding idle time.

$$E_{big} = t_{big} * (P_{core}^{big} + P_{uncore}^{big}) + P_{idle} * t_{idle} \tag{1}$$

$$E_{small} = t_{small} * (P_{core}^{small} + P_{uncore}^{small}) \tag{2}$$

To understand the impact of uncore power, the evaluation in Section VII considers two uncore configurations: fixed and scalable. The fixed uncore configuration uses the same uncore subsystem when executing on either big or small cores. The scalable uncore scenario models an uncore where certain uncore components such as memory controllers or cache units are turned off or powered down as we move to the small core. Hence, in this case, the uncore power scales along with core power when a workload moves to a different core.

IV. HeteroMates Design

HeteroMates enables a wide dynamic power range using heterogeneous cores. This section describes its key components and concepts.

A. Core Groups

A heterogeneous *core group* is a collection of a small number of (e.g., 2-4) heterogeneous cores that are grouped together to form a single execution unit. For example, Figure 4 shows a core group consisting of three heterogeneous cores: a big (B), a small (S), and a tiny (T) core. The core group appears as a single execution unit with multiple performance/power levels. Depending on application behavior and user-defined policies, an appropriate core is dynamically chosen to run the user task in question, by transparently moving the task's execution to that core, and by placing the other inactive cores into a low power idle state to conserve power. For example, the tiny core can be used for background tasks like email update checks, the small core for normal user operation, and the big core is reserved for performance-critical tasks.

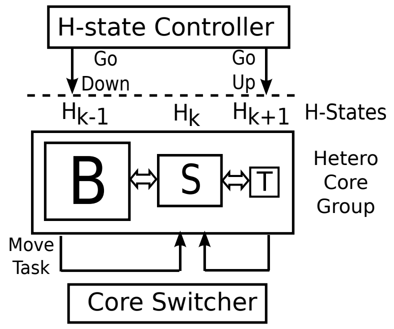


Figure 4. A core groups consisting of three heterogeneous cores: a big (B), a small (S), and a tiny (T) core exposed as three H-states.

Different cores within a core group are exposed using *heterogeneity-states* (H-states), an interface similar to the P-state (performance-states) interface defined by the ACPI standard and used by operating systems to scale the frequency and voltage of processors. Higher P-state numbers represent slower processor speeds. Thus, P0 is the highest-performance state, with P1 to Pn being successively lower-performance states. Similarly, an H-state is assigned to each type of core in the core group. A high-performance big core corresponds to a lower numbered H-state, while a low-power small core corresponds to a higher-numbered H-state. Thus, a core group exposes a set of H-states ($H_0 \dots H_n$) which are controlled by an *H-state controller* module. Depending on the state transition logic and the resultant H-state, a *core switcher* transparently migrates the execution to the appropriate core. In this manner, applications perceive only homogeneous cores with larger dynamic power range than any of the individual cores.

The design of HeteroMates offers multiple advantages. First, H-state interface decouples heterogeneity from scheduling such that the scheduler need not deal with performance/power differences among cores. Instead, a separate H-state driver handles this transparently to the scheduler. Second, H-states can be implemented either in hardware, firmware, operating system, or even hypervisors, allowing a broader applicability. As an architectural solution, it provides a way to completely hide heterogeneity from the operating system, which is critical to support legacy software and applications. Further, core groups provide a unified mechanism to easily accommodate a variable number of heterogeneous cores by adding an H-state for each type of core. Finally, core groups can also be useful when TDP (thermal-design-point) limits may constrain the number of cores that can be active simultaneously. As transistor density on modern processors keep increasing, such TDP limits are proving to be a critical design constraint in the form of *dark silicon* [4].

B. H-state Controller

H-states on a core group are controlled by the H-state controller, in a manner similar to frequency scaling operations performed by a CPU governor. A CPU governor is a kernel module that changes core P-states based on a policy. In a similar manner, the H-state controller performs H-state scaling operations. However, instead of changing voltage and frequency as in the case of P-states, a change in H-state causes the execution to move to a different core. The functions of the H-state controller and of the traditional P-state governor complement each other. For example, Figure 5 shows the combined P-state and H-state transition diagram for a two-core heterogeneous core group. Here, H_k corresponds to the small core, and H_{k-1} corresponds to the big core. P-state changes within a core are performed by the P-state governor, while cross-core migrations are governed by the H-state controller.

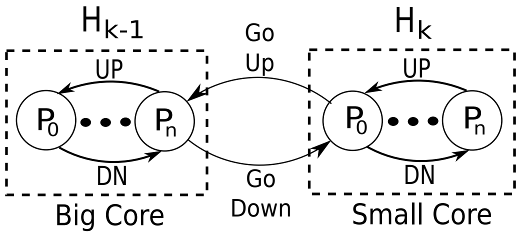


Figure 5. H-state and P-state transition state machines. H-state determine the core for execution, while P-states determine the frequency on that core.

CPU governors available in current operating systems (e.g., the ondemand governor in Linux [16]) dynamically change CPU frequency in response to CPU load (utilization). However, CPU load alone is not sufficient to drive H-state scaling operations, which also require determining whether a bigger or smaller core is more suitable for execution. Previous work on heterogeneous processor scheduling [11],

[12], [19] has identified application IPC (instructions-per-cycle) as a key metric to select the right core for execution. Therefore, HeteroMates uses a combination of CPU load and application IPC to form the H-state transition logic shown in Figure 6.

The intuition behind the scaling algorithm can be explained as follows. An application with high CPU load but low IPC is likely to perform equally well on both big and small cores due to its low IPC requirements, which can easily be met on a small core. Applications with high IPC but small CPU load under-utilize the big core. Moving such applications to a smaller core results in higher utilization of the small core, but without a significant penalty in application performance. When both of these conditions are violated, the application is likely to gain performance by executing on a bigger core.

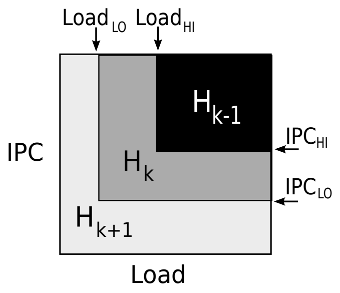


Figure 6. H-state scaling operations in response to application IPC and CPU load.

The H-state controller monitors application IPC and CPU load at periodic intervals and compares them with predefined thresholds to determine the resultant state. If both the IPC and load are above thresholds IPC_{HI} and $Load_{HI}$ respectively, the core group is scaled up, i.e, moved to a higher-performance or lower numbered state. If either IPC or load are lower than thresholds IPC_{LO} and $Load_{LO}$, the H-state is scaled down to a lower-performance state. For values in between these thresholds, no H-state change is performed. These thresholds are defined for each type of core in the system. By setting different values for these thresholds, different policies can be enforced. For example, low values of thresholds force the execution to big cores, and thus prefer performance over power. Similarly, a policy having thresholds with high values picks smaller cores more often.

An H-state change operation causes the execution to switch to a different core. This switching overhead could be substantial due to migration latency and loss of private cached data if such changes are very frequent. In response, we use *history counters* to dampen core switching frequency. A switch is performed only after a certain number of consecutive identical H-state change requests are received. The history counter is a simple integer counter associated with each core group, which is incremented whenever consecutive intervals generate the same requests and reset otherwise.

C. Uncore-aware Operation

As discussed in Section III-B, the energy-efficiency of a platform is not only determined by the type of core used for execution, but also by the power consumption of the shared uncore subsystem. Workloads for which execution on a bigger core provides both higher performance and better energy-efficiency due to improved performance scaling, should always be run on big cores as small core degrades both performance and efficiency. HeteroMates addresses this issue by adding the *energy override* condition in Equation 3 to the heuristic described earlier. If the energy consumption of the current H-state (H_{cur}) is greater than the energy consumption of the next higher state (H_{cur-1}), a scale up operation is performed to move the execution to the bigger core.

$$\textbf{if } \frac{Energy(H_{cur-1})}{Energy(H_{cur})} < 1 \textbf{ then } H_{next} = H_{cur-1} \quad (3)$$

For energy-aware operation, Equation 3 requires the energy consumption of the application to be estimated on a different core (H-state). This task can be divided into two components: processor power prediction and application behavior (e.g., execution time, IPC) prediction. CPU power visibility to the operating system is becoming increasingly important, with multiple CPU vendors providing hardware counters to measure the power of different components on the platform [24]. Further, previous work has developed light-weight models to accurately predict per-core power using existing performance events [5]. Using a similar approach, this work also uses power models, described in Section VI-D, to obtain core and uncore power consumption.

In order to understand the impact of a core transition on application behavior, hardware assistance can be provided. For example, HeteroScouts [22] proposes hardware performance counters to predict workload behavior on a remote core (after-transition) from the parameters available on the local core (before-transition). Due to unavailability of such counters in current processors, simple prediction models are developed using experimental data. The following section provides details of the modeling methodology.

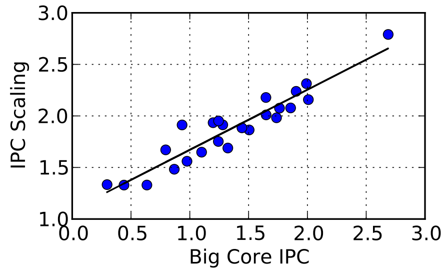


Figure 7. Modeling IPC scaling as a function of IPC

D. Remote Behavior Prediction

To model the relationship between application IPC on a big and a small core in our experimental platform (see Figure 8), the client workloads in Table I and SPEC CINT2006 benchmarks are executed on both types of cores. Figure 7 plots the obtained $IPC_{scaling}$ data, defined as the ratio of the big core IPC and the small core IPC, as a function of the IPC on the big core. We omit the corresponding scaling data for the small core due to lack of space. As evident from the figure, a linear curve fits the data well, with the resultant model given by the equations below.

$$IPC_{scaling} = 0.6 * IPC_{big} + 1.01 \quad (4)$$
$$IPC_{scaling} = 1.31 * IPC_{small} + 0.94 \quad (5)$$

The impact of IPC scaling on the execution time of an application is workload dependent. CPU-bound workloads show a proportional relationship between IPC scaling and execution-time scaling. However, this does not hold true for many client workloads with significant idle phases, e.g., media and graphics workloads. For such workloads, execution time is not affected by the core performance. Instead, a change in core performance translates into change in core idle state residency. These conditions are modeled by applying the scaling function to the product of core active state (R_{active}) residency and execution time (t), as shown in Equation 6. The equation was experimentally verified using all of the client workloads in Table I as majority of the workloads closely follow the modeled relationship. In the online model, sampling interval is substituted for the execution time.

$$(R^{small}_{active} * t_{small}) = IPC_{scaling} * (R^{big}_{active} * t_{big}) \quad (6)$$

Further, the change in core idle residency (R_{idle}) impacts package idle state (U_{idle}) residency in an application dependent manner. Applications for which the package becomes idle as soon as the core becomes idle, show a strong correlation between core and package idle states. On the other hand for multi-threaded applications and graphics-intensive applications, a core's idle state does not necessarily translate to the package idle state since the package can still be busy due to activity in another core or the graphics processor. Such applications show a weak or negligible correlation between core and package idle states. These two scenarios are modeled in Equation 7 where a difference of 20% between U_{idle} and R_{idle} is assumed as an indicator of weak correlation. For such cases, U_{idle} is assumed to be the same irrespective of the type of core used for execution.

$$U^{small}_{idle} = \begin{cases} U^{big}_{idle} & \text{if } U^{big}_{idle} \ll R^{big}_{idle}, \\ R^{small}_{idle} & \text{otherwise} \end{cases} \quad (7)$$

Using the models presented above and the power models described later in Section VI-D, an application's relative energy consumption on two different H-states can be obtained. These values are used to perform energy override operations as defined earlier by Equation 3.

V. Implementation

HeteroMates is implemented for the Linux kernel. The current implementation considers systems involving pairs of heterogeneous cores. H-states are implemented by customizing the P-state tables on each core to expose two P-states corresponding to each core in a pair. H-state changes work in lock-step on both of these cores to avoid conflicting operations. An H-state change causes execution to switch cores instead of performing DVFS. Our current implementation does not consider traditional voltage and frequency scaling. This is because there is substantial previous work on DVFS [15], [17], [20], [25], which can be used to perform P-state scaling in addition to H-state transitions.

The H-state controller is implemented as a kernel module which runs on each active core as a kernel thread. It periodically (40ms) reads various hardware performance monitoring counters (PMCs), applies models, and performs any H-state changes depending on the policy and thresholds chosen. The overhead of running models is measured to be small (approximately 2% increase in core active and 5% increase in package active residency). The core switcher is implemented in the OS kernel by changing the runqueue pointer for the tasks in the source runqueue to point to the destination runqueue. The overhead of this operation is minimal when run-queue length is not large, which we have observed as being the case for the typical client workloads used in our experiments. We note that similar functionality can be provided by hardware, to further reduce overheads. Also, only active cores are made available for scheduling to the Linux CFS scheduler. Inactive cores are put into an offline mode using a lightweight mechanism. A value of three is used for history counters.

VI. Experimental Evaluation

A. Experimental Platform

Our experimental platform consists of a quad-core Intel i7-2600 client processor. To create heterogeneity, we use an Intel proprietary tool to defeature and emulate the performance of low-powered small cores for a subset of the cores [11]. A block diagram of the platform configuration is shown in Figure 8. An on-die graphics processor is used to accelerate graphics workloads. All of the cores operate at a frequency of 2.6 GHz and share an LLC of size 8 MB. All the workloads are run using Linux kernel 3.0 and automated.

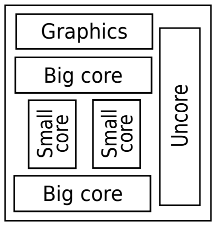


Figure 8. Experimental heterogeneous platform

Workload	Category	Description	Metric
7zip	utility	a parallelized version of 7zip is used to compress a text file	Time
applaunch	operation	launches and executes a series of graphics-intensive applications	Load time
browse	web	loads a set of web-pages at an interval of 3s to emulate user's think time	Load time
canvas	web	HTML5 benchmark performs browser canvasing tests	FPS
eclipse	utility	Java based benchmark runs performance tests for the Eclipse IDE	Time
gmagick	media	GraphicsMagick image editing application is used to resize a set of images	Time
javascript	web	Javascript benchmark performs a series of standard browser operations	Load time
lightsmark	graphics	renders scenes from a 3D game and measures graphics performance	FPS
mplayer	media	a H/W accelerated version of mplayer plays an HD movie clip (60s)	FPS
mytube	web-media	plays an H.264 streaming video inside the browser for 60s	FPS
openarena	gaming	plays a benchmarking demo from a 3D first-person-shooter game	FPS
palbum	web	photo album application that flips through photographs at 0.5s interval	Load time
strike	web-gaming	replays a demo session of a web-based 2D game (60s)	FPS
x264	media	x264 media encoder is used to encode a media file	Time

Table I
CLIENT WORKLOAD SUMMARY

B. Client Workloads

To assess the viability of using heterogeneity for client systems, a diverse set of real-world applications are chosen which are typical of modern end-user devices since prior server-centric research on heterogeneous processors [11], [12], [19] does not directly address the needs and processor usage models seen on client devices. Table I provides a summary of the applications used in our analysis which include browsing, gaming, media, etc. and relevant performance metrics which are different from server workloads.

C. Methodology

Two different policies are used, one performance-driven, the other power-driven. This is done by choosing different threshold values, obtained after experimenting with several combinations of thresholds. Table II summarizes the various thresholds used to cater to these policies. For a paired-core system, small cores can only perform scale up operations and not scale down, therefore, only HI thresholds are relevant for small cores. Similarly, only LO thresholds are relevant for the big cores. The first performance-driven policy favors performance over power by using big cores for execution in an aggressive manner. This is achieved by choosing smaller thresholds in the table. The power-driven policy, on the other hand, focuses on power by choosing bigger thresholds and forcing the execution to small cores more often. The evaluation is carried out by comparing the performance and energy consumption of the performance-driven policy with only big core execution and of the power-driven policy with just small core execution. These two comparison points provide us a perspective of the advantage of using heterogeneous cores over homogeneous configurations.

	Small Core		Big Core	
	IPC_{HI}	$Load_{HI}$	IPC_{LO}	$Load_{LO}$
Performance-driven	0.5	70%	0.8	40%
Power-driven	0.7	80%	1.25	50%

Table II
THRESHOLDS FOR PERFORMANCE- AND POWER-DRIVEN POLICIES

D. Power Model

The emulated heterogeneous platform mimics the performance of small cores. However, it does not match the power characteristics of an actual small core built using a different process technology for low power consumption. We therefore, rely on power models to obtain core and uncore power consumption.

1) Core Power: The average power consumption of a CPU core can be modeled using the following equations:

$$P_{core} = R_{active} * P^{core}_{active} + R_{idle} * P^{core}_{idle} \tag{8}$$
$$P^{core}_{active} = C_{dyn} * V^2 * f \tag{9}$$

Here, R_{active} and R_{idle} denote core active and idle state residencies (%), and P^{core}_{active} and P_{idle} are the corresponding power values. C_{dyn} is the dynamic capacitance, V denotes the operating voltage, and f represents the switching frequency. Big core C_{dyn} is modeled as a function of IPC in Equation 10, as shown and validated by other researchers [21]. Similarly, Equation 11 models the capacitance for a small core having three-times smaller area than the big core.

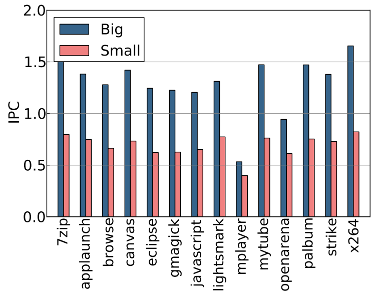
$$C_{big} = 0.499 * ipc_{big} + 0.841 \tag{10}$$
$$C_{small} = 0.472 * ipc_{small} + 0.176 \tag{11}$$

2) Uncore Power: Similar to core power, uncore power can be modeled using package idle state residencies (U_x) as shown in Equation 12.

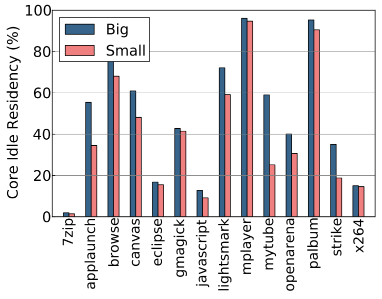
$$P_{uncore} = U_{active} * P^{uncore}_{active} + U_{idle} * P^{uncore}_{idle} \tag{12}$$
$$P^{uncore}_{active} = P_{wake} + P_{activity} * LLC_{rate} \tag{13}$$

Further, uncore active power (P^{uncore}_{active}) is modeled as a function of the LLC activity in Equation 13 where P_{wake} is the fixed power cost of waking up various uncore components, while the $P_{activity}$ component scales with the LLC access rate LLC_{rate} (relative to peak access rate including both cache hits and misses).

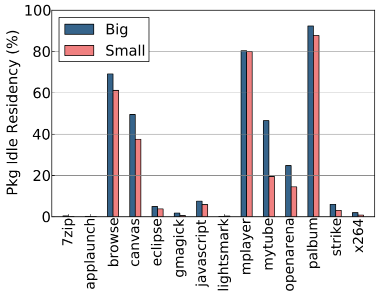
The analysis uses a value of 0.9 V for the voltage (V), and frequency (f) is kept at 2.6 GHz. For this platform,



(a) Instructions-per-cycle



(b) Core idle residency



(c) Package idle residency

Figure 9. A comparison of the behavior of several client workloads on big vs. small cores

the average big core and small core power for all our workloads is obtained to be 2.37 W and 0.95 W respectively. A comparable uncore is modeled using a value of 1.2 W for P_{wake} and $P_{activity}$ in case of a fixed uncore and scaled down to half for a scalable uncore. Core and uncore idle power are assumed to be 0.1 W and a 1.5 W power component is attributed to the on-die graphics processor which also scales with the LLC activity.

VII. Experimentation Results

A. Client Workload Characterization

The results shown in Figure 9 provide a comparison of the behavior of various client applications on heterogeneous cores. Specifically, they compare average IPC (instructions-per-cycle), core idle residency, and package idle state residency for all of the workloads in Table I for big and small core execution. As evident from Figure 9(a), most of the applications observe a significant decrease in their IPC when running on the small core as compared to the big core. This reduction in IPC results in the small core being active for longer durations, thereby causing a decrease in core and package idle residency (see Figures 9(b) and 9(c)). Further, many applications are seen to have almost negligible package idle residency. These applications either heavily use the graphics processor (e.g., openarena, lightsmark), or they always keep one of the CPU cores busy (e.g., 7zip, gmagick, x264), and do not allow the uncore to enter into an idle state.

B. Performance-driven Policy

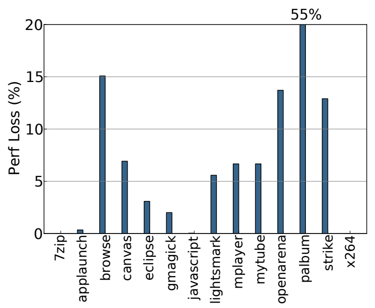
Figure 10 provides results comparing the performance and energy consumption of the performance-driven policy with execution on big cores. Specifically, Figure 10(a) shows performance loss (%) with respect to the maximum performance achievable by using big cores for the entire execution, and Figure 10(b) shows corresponding energy savings by using small cores for partial execution when big core is not energy-efficient. Performance is measured based upon the metrics in Table I, with inverse of latency as the metric for latency-oriented workloads. As evident from the figures, this policy is able to achieve performance within 15% of the big core performance for all the workloads except browse and palbum. This high performance loss for these two workloads is due to their bursty nature, i.e., these applications exhibit sudden bursts of high activity during page-rendering. HeteroMates uses history counters to dampen core switching frequency, which requires multiple consecutive state change requests to be received before actually making the change. Due to this reason, these bursty applications observe a short delay before they are moved to the big core which incurs a higher performance degradation. However, the absolute increase in the latency for these applications may not be user-perceivable.

Figure 10(b) shows corresponding energy savings results for three scenarios: core-only savings (C), SoC-wide savings (C+UC) with a fixed uncore, and SoC-wide savings with a scalable uncore. As seen from the figure, it is able to save significant energy for several applications with a small performance degradation. Workload openarena achieves highest gains with 39% core energy savings. However, these savings are strongly affected when the power consumption of the uncore is taken into account. On the other hand, when a scalable uncore is used, these savings increase and become comparable (25%) to core-only energy savings.

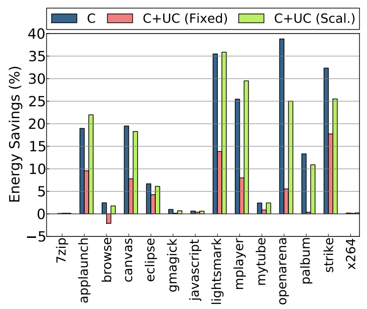
To elaborate on the importance of uncore power in total SoC power, Figure 10(c) shows the distribution of core and uncore energy consumption for various applications. Core energy component dominates for CPU-intensive applications like 7zip, eclipse, gmagick, and x264, while uncore component is significant for other applications including lightsmark, mplayer, and openarena. These results highlight the growing importance of uncore power in the processor power consumption and motivate the need for a scalable uncore design when seeking to obtain large gains from heterogeneous multicores.

C. Power-driven Policy

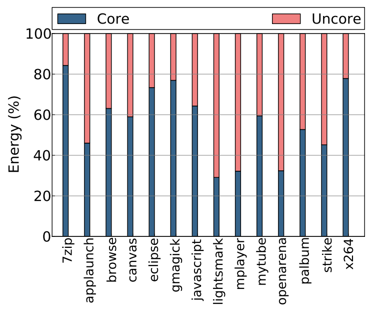
Results for the power-driven policy are presented in Figure 11, where Figures 11(a) and 11(b) respectively, show performance gain and energy loss (SoC-wide) in comparison to small-core-only execution. As results show, this policy is able to achieve significant performance gains for many applications by selectively using big cores. Further, it is able



(a) Performance loss wrt. big cores

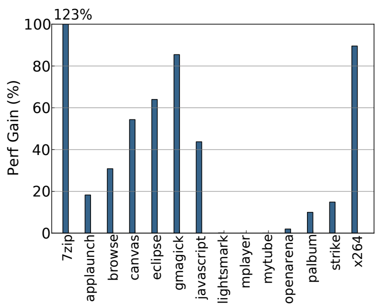


(b) Energy savings wrt. big cores

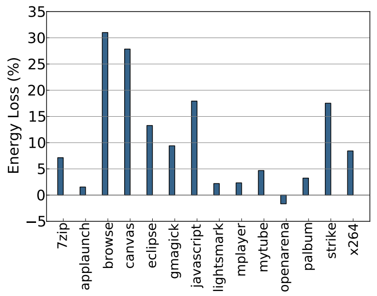


(c) Core and uncore energy distribution

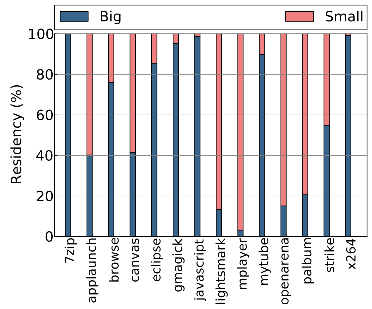
Figure 10. Comparison of performance-driven policy with big core execution



(a) Performance gain wrt. small cores



(b) Energy loss wrt. small cores

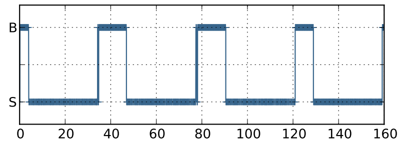


(c) Core residency

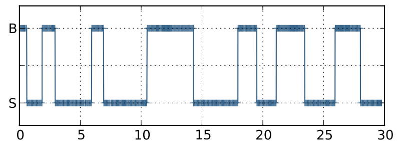
Figure 11. Comparison of power-driven policy with small core execution

to do so with only a small to moderate increase in energy consumption. For example, the browse and canvas workloads observe the highest increases in energy consumption of 31% and 28% respectively, while most of the other applications show a smaller increase. However, these two applications also show a 31% and 54% performance gain for the increased energy consumption due to their usage of big cores. We note that some applications like lightsmark, mplayer, and openarena exhibit negligible performance improvement due to poor scalability.

Results in Figure 11(c) show the percentage residency on big and small cores for all of the applications. Different applications exhibit different degrees of big and small core usage. For example, applications like 7zip, eclipse, and x264 with good performance scalability spend the majority of their execution on big cores. On the other hand, applications like lightsmark, mplayer, and palbum remain on small cores for a significant portion of their execution time. Other applications like applaunch, canvas, and strike make use of both types of cores during their execution. To illustrate this further, the big and small core usage profiles of the applaunch and strike workloads are shown in Figure 12. The applaunch workload launches and executes a series of graphics-intensive applications. The launch operation is CPU-intensive and performs better on a big-core, while the execution phase is accelerated using the on-die graphics processor and a small core provides comparable performance to the big core at a lower power. Therefore, this workload transits between big and small cores during launch and execution phases (see Figure 12(a)). Similarly, Figure 12(b) shows the execution profile for the strike gaming workload. This workload exhibit several phases with high activity (e.g., bots shooting) when big cores are used and phases with low activity (e.g., bots aiming and moving) when small cores may suffice. In this manner, the appropriate core is used depending on the activity.



(a) applaunch



(b) strike

Figure 12. Big (B) and small (S) core usage profile (x-axis: time(s))

VIII. Related Work

Heterogenous chip multiprocessors (CMPs) have been proposed to achieve higher energy-efficiency than symmetric multicore processors. Using a mix of big and small cores, different phases within an application can be mapped to the core which can run them most efficiently [12]. Similarly, heterogeneous cores can be used to improve the performance of parallel applications by speeding up sequential phases within the application [9], [23]. Researchers have also developed appropriate scheduling algorithms to efficiently run applications on heterogeneous cores [10], [11], [13], [19]. In addition, previous work has proposed mechanisms to effectively manage functionally heterogeneous multicores [18] and virtualize accelerator-based systems [8].

There is also substantial previous work on dynamic voltage and frequency scaling (DVFS). Several techniques have been developed to dynamically select appropriate voltage and frequency for maximum efficiency [15], [17], [20], [25]. However, others have questioned the effectiveness of DVFS on modern processors [2], [3].

In comparison, our work targets client devices where energy is a premium resource, with diverse application behavior and performance metrics. In that context, we extend the existing DVFS mechanisms to go beyond homogeneous cores and support core heterogeneity to enable a wide dynamic power range on these client devices. In addition, we highlight the significance of uncore power in total SoC power and motivate the need for a scalable uncore for exploiting maximum gains from heterogeneous CMPs.

IX. Conclusions & Future Work

This paper presents the *HeteroMates* solution, which utilizes heterogeneous multicores in order to provide a wide dynamic power range on client devices. It proposes *core groups*, an abstraction that groups together a small number of heterogeneous cores to form a single execution unit. Cores within a core group are exposed as multiple heterogeneity (H) states. H-state transitions are governed by an H-state controller, while a core switcher transparently migrates the task to the appropriate core depending on the resultant H-state. Using a diverse mix of client applications and an experimental heterogeneous platform, we show that heterogeneous CMPs can be used to provide a superior solution for client devices. We also highlight the growing importance of uncore power in total SoC power consumption and the need for a scalable uncore design to completely realize the intended gains.

As part of future work, we plan to investigate scenarios when cores are shared across core groups. Another interesting venue for research would be to investigate the ideal ratio of the number of big and small cores for client systems.

References

[1] Variable SMP: A multi-core CPU architecture for low power and high performance. White paper, NVIDIA, 2011.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. In *Proceedings of the SOSP*. ACM, 2009.

[3] G. Dhiman, K. K. Pusukuri, and T. Rosing. Analysis of dynamic voltage scaling for system level energy management. In *HotPower*. USENIX Association, 2008.

[4] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *38th ISCA*, San Jose, CA, USA, 2011. ACM.

[5] B. Goel, S. A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati. Portable, scalable, per-core power estimation for intelligent resource management. In *IGCC*. IEEE, 2010.

[6] P. Greenhalgh. Big.LITTLE processing with ARM CortexTM-A15 & Cortex-A7. White paper, ARM, Sept 2011.

[7] V. Gupta, P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan, and G. Srinivasa. The forgotten 'uncore': On the energy-efficiency of heterogeneous cores. In *USENIX ATC*, 2012.

[8] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: coordinated scheduling for virtualized accelerator-based systems. In *USENIX ATC*, Portland, OR, USA, 2011.

[9] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[10] V. Kazempour, A. Kamali, and A. Fedorova. AASH: an asymmetry-aware scheduler for hypervisors. In *VEE '10*, pages 85–96, Pittsburgh, PA, USA, 2010. ACM.

[11] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *5th EuroSys*, pages 125–138, Paris, France, 2010. ACM.

[12] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *36th MICRO*, pages 81—-, San Diego, CA, USA, 2003.

[13] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *IEEE HPCA*, 2010.

[14] G. H. Loh. The cost of uncore in throughput-oriented many-core processors. In *In Proc. of Workshop on Architectures and Languages for Throughput Applications (ALTA)*, 2008.

[15] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *ICS*, 2002.

[16] V. Pallipadi and A. Starikovskiy. The ondemand governor: Past, present and future. *Linux Symposium*, 2:223–238, 2006.

[17] K. Rajamani, H. Hanson, J. Rubio, S. Ghiasi, and F. Rawson. Application-aware power management. In *IISWC*, 2006.

[18] D. Reddy, D. Koufaty, P. Brett, and S. Hahn. Bridging functional heterogeneity in multicore architectures. *SIGOPS Oper. Syst. Rev.*, 45(1):21–33, Feb. 2011.

[19] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *5th EuroSys*, pages 139–152, Paris, France, 2010.

[20] D. Snowdon, E. Le Sueur, S. Petters, and G. Heiser. Koala: A platform for OS-level power management. In *4th Eurosys*, pages 289–302, Nuremberg, Germany, 2009. ACM.

[21] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive DVFS. In *Green Computing Conference (IGCC)*, July 2011.

[22] S. Srinivasan, R. Iyer, L. Zhao, and R. Illikkal. HeteroScouts: hardware assist for OS scheduling in heterogeneous CMPs. *SIGMETRICS Perform. Eval. Rev.*, 39:341–342, June 2011.

[23] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *14th ASPLOS*. ACM, 2009.

[24] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. Rubio, F. Rawson, and J. Carter. Architecting for power management: The IBM POWER7 approach. In *IEEE HPCA*, jan. 2010.

[25] A. Weissel and F. Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *CASES '02*, Greenoble, France, 2002. ACM.