Scientific Programming 22 (2014) 93–108
DOI 10.3233/SPR-140380
IOS Press

Coordinated energy management in heterogeneous processors [1]

Indrani Paul [a,b,*], Vignesh Ravi [a], Srilatha Manne [a], Manish Arora [a,c] and Sudhakar Yalamanchili [b]
[a] *Advanced Micro Devices, Inc., USA*
E-mails: {indrani.paul, vignesh.ravi, srilatha.manne}@amd.com, marora@eng.ucsd.edu
[b] *Georgia Institute of Technology, Atlanta, GA, USA*
E-mail: sudha@gatech.edu
[c] *University of California, San Diego, CA, USA*

Abstract. This paper examines energy management in a heterogeneous processor consisting of an integrated CPU–GPU for high-performance computing (HPC) applications. Energy management for HPC applications is challenged by their uncompromising performance requirements and complicated by the need for coordinating energy management across distinct core types – a new and less understood problem.

We examine the intra-node CPU–GPU frequency sensitivity of HPC applications on tightly coupled CPU–GPU architectures as the first step in understanding power and performance optimization for a heterogeneous multi-node HPC system. The insights from this analysis form the basis of a coordinated energy management scheme, called DynaCo, for integrated CPU–GPU architectures. We implement DynaCo on a modern heterogeneous processor and compare its performance to a state-of-the-art power- and performance-management algorithm. DynaCo improves measured average energy-delay squared (ED^2) product by up to 30% with less than 2% average performance loss across several exascale and other HPC workloads.

Keywords: Energy management, high-performance computing

1. Introduction

Efficient energy management is central to the effective operation of modern processors in platforms from mobile to data centers and high-performance computing (HPC) machines. However, HPC systems are unique in their uncompromising emphasis on performance. For example, the national roadmap for HPC now has the goal of establishing systems capable of sustained exaflop (10^{18} flops/s) performance. However, the road to exascale is burdened by significant challenges in the power and energy costs incurred by such machines.

Many current HPC systems use general-purpose, multi-core processors such as Xeon from Intel and AMD Opteron™ that are equipped with several power-saving features, including dynamic voltage and frequency scaling (DVFS). More recently, driven in part by demand for energy efficiency, we have seen the emergence of such processors with attached graphics processing units (GPUs) acting as accelerators. As of November 2012, four of the top ten and 62 of the top 500 supercomputers on the Top500 list were powered by accelerators [19,20].

This trend towards heterogeneous processors is continuing with tightly coupled accelerated processing unit (APU) designs in which the CPU and the GPU are integrated on the die and share on-die resources such as the memory hierarchy and interconnect. The companion emergence of programming models such as CUDA, OpenACC and OpenCL is making such processors viable for HPC. However, the tighter integration of CPUs and GPUs results in greater performance dependencies between the CPU and the GPU. For example, CPU and GPU memory accesses interact in the memory hierarchy, and may interfere, while they share a chip-level power budget and thermal capacity. Therefore, effective performance management and energy management must be coordinated carefully between the CPU and the GPU [34].

Figure 1 illustrates an HPC application running on an AMD A-Series APU heterogeneous processor, for-

[1] This paper received a nomination for the Best Paper Award at the SC2013 conference and is published here with permission of ACM.

[*] Corresponding author. E-mail: indrani.paul@amd.com.

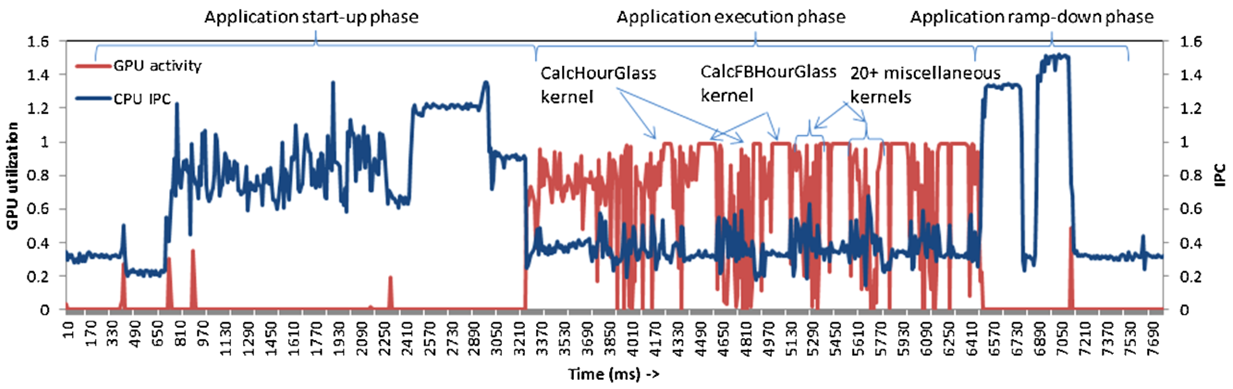

Fig. 1. Example phase behavior in an exascale proxy application (Lulesh). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

merly code-named "Trinity". The figure shows fine-grain communication between the CPU and the GPU on an OpenCL variant of Lulesh with 100 node elements per dimension [24]. The x-axis shows time (in milliseconds) and the y-axis shows the CPU utilization as measured by IPC for the multi-threaded CPU, and the GPU utilization as measured by active clock cycles for the data-parallel GPU. The application is in the start-up phase up to 3200 ms, and the CPU is the primary active component. Subsequently, the CPU primarily plays an assist role delivering data to the GPU for computation leading to low CPU activity (IPC) and high GPU activity. However, there is constant communication between the CPU and the GPU and the performance required from each is a function of the kernel being run. For instance, the CalcFBHourGlass kernel has a higher GPU utilization than the other 20+ miscellaneous kernels in the application. The computational demands of the CPU and the GPU vary across program phases, as does the intensity of their interactions. Power and energy management techniques must be made cognizant of these interactions to minimize performance degradation with improvements in energy efficiency.

While there has been a significant body of work in dynamic voltage frequency scaling (DVFS) for energy management in single- and multi-core homogeneous architectures, heterogeneous architectures embody several characteristics that render direct application of these techniques ineffective. Performance-coupling between the CPU and the GPU produce dependencies between their respective DVFS states. However, unlike multi-core homogenous architectures in which all cores are identical and the majority of threads are identical, the CPU and GPU differ in both architecture and execution model. While the former supports asynchronous execution of (relatively) coarse-grain threads, the latter implements a model orchestrating the synchronous execution of thousands of thread blocks or wavefronts, comprising tens to hundreds of fine-grain threads. Consequently, their energy and power behaviors are quite distinct. Further, while the CPU–GPU behaviors are *directly* coupled through the programming model, their executions *indirectly* interact via interference and competition for shared on-chip resources. To be effective, algorithms that determine the DVFS states of the CPU and the GPU must be cognizant of these effects, their interrelationships, and their combined effect on performance.

Our ultimate goal is to optimize energy efficiency and performance in a multi-node HPC system consisting of tightly coupled heterogeneous node architectures. We view the path to this goal as a two-step process: The first step analyzes and optimizes intra-node power and performance, and the second step optimizes these metrics in a multi-node system. This work focuses on maximizing energy efficiency for HPC applications with minimal to no compromise in performance in a tightly coupled heterogeneous node architecture. Specifically, this paper makes the following contributions:

- We empirically characterize the frequency sensitivity of proxy applications developed to represent exascale applications. The analysis exposes several opportunities for improving energy efficiency without degrading the performance of the application.
- We identify a key set of CPU and GPU run-time parameters that reflects the frequency sensitivity of the application and use regression techniques to construct an analytic model of frequency sensitivity.

- We propose DynaCo – a coordinated, dynamic energy-management algorithm using online frequency-sensitivity analysis to coordinate the DVFS states of the CPU and the GPU. DynaCo is implemented on a state-of-the-art heterogeneous processor.
- Using measurements on real hardware, we compare DynaCo to a commercial, state-of-the-practice power- and performance-management algorithm for several OpenCL exascale proxy applications and other HPC applications, demonstrating that significant improvements in energy efficiency are feasible without sacrificing performance.

The following section provides background information. Section 3 presents an analysis of the frequency sensitivity of HPC applications. We use the insights from that analysis to develop a model of frequency sensitivity that forms the basis of the energy-management algorithm described in Section 4. Sections 5 and 6 describe the implementation and experimental results. Sections 7 and 8 present related work and our conclusions.

2. Background

Figure 2 shows the floor plan of the AMD A-Series heterogeneous APU used in the rest of the paper. It contains two out-of-order dual-core CPU compute units (CUs, also referred to as Piledriver modules) and a GPU. The cores in a CU share the front-end and floating-point units and a 2 MB L2 cache. The

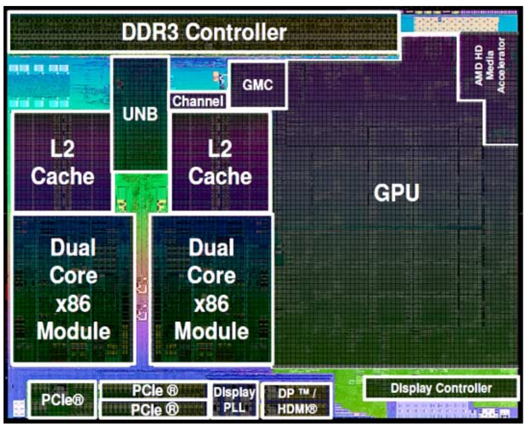

Fig. 2. Die shot of AMD A-Series APU [32]. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

CPUs share a power plane and the GPU is on a separate power plane. The GPU consists of 384 AMD Radeon™ cores, each capable of one single-precision fused multiply-add computation (FMAC) operation per cycle (the methodology and techniques in this paper are equally applicable to processors that support double-precision). The GPU is organized as six SIMD units, each containing 16 processing units that are each four-way VLIW. The memory controller is shared between the CPU and the GPU. More details on the AMD A-Series processor can be found in [32].

Table 1 shows all possible DVFS states for the CPU cores in the AMD A-Series A10-5800k. Here, DVFS states can be assigned per CU; however, because the CUs share a voltage plane, the voltage across all CUs is set by the maximum-frequency CU. P0 through P5 are software-visible DVFS states that are referred to as performance states, or P-states, and are managed either by the OS through the Advanced Configuration and Power Interface (ACPI) specification [1] or by the hardware. Pb0 and Pb1 are called the boost states and are visible only to, and managed by, the hardware. Entrance to and exit from the boost states are managed exclusively by hardware when the CPU is at P0; hence, P0 is usually called the base state. P1 through P5 are increasingly lower-power P-states. The GPU has an independent power plane whose voltage and frequency are controlled independently. Unlike the CPU, the GPU does not have architecturally visible P-states. Throughout the rest of the paper, we will refer to the GPU DVFS states as GPU-high (highest frequency), GPU-med (medium frequency) and GPU-low (lowest frequency).

The AMD A-Series APU uses a sophisticated power-monitoring and -management technology referred to as AMD Turbo CORE to optimize performance for a given power and thermal constraint. This technology uses approximated power and temperature values to monitor and guide the power-management

Table 1

CPU DVFS states for AMD A-Series APU

	P-state	Volt (V)	Freq (MHz)
HW-only	Pb0	1.475	4200
	Pb1	1.45	4000
SW-visible	P0	1.363	3800
	P1	1.288	3400
	P2	1.2	2900
	P3	1.075	2400
	P4	0.963	1900
	P5	0.925	1400

algorithms. AMD Turbo CORE uses the bidirectional application power management (BAPM) algorithm to control the power allocated to each compute entity in the processor [32]. Each compute entity interfaces with BAPM to report its power consumption, and BAPM determines its power limits based on the available thermal headroom. At regular time intervals, the BAPM algorithm does the following:

(1) Calculates a digital estimate of power consumption for each CU and GPU.
(2) Converts the power estimates into temperature estimates for each component.
(3) Assigns new power limits to each entity.

Once BAPM has assigned power limits, each CU and GPU manages its own frequencies and voltages to fit in the assigned limit (i.e., local to a unit, the hardware will employ DVFS to keep the power dissipation in the assigned limit). The BAPM algorithm sets power limits based on thermal constraints and greedily boosts the power states to maximize use of the thermal capacity. If the processor never reaches maximum temperature, then power is allocated to the processor until the maximum CPU and GPU frequencies are reached.

The BAPM algorithm is optimized to maximize performance with a fair and balanced sharing of power between on-chip entities. BAPM allocates power to each entity using a pre-set static distribution weight that is derived using empirical analysis. Such static allocation is the best choice in the absence of dynamic feedback from the application. As a general-purpose state-of-the-practice controller, BAPM is designed to provide reasonable performance improvements without any significant outliers.

3. Motivation and opportunities

Figure 3 shows the peak temperature normalized to the maximum junction temperature allowed for each CU and the GPU for miniMD as the application runs on a 100 W TDP processor. Processors with such a thermal design power package are commonly found in HPC clusters [17]. Although temperature tracks power and inversely tracks performance, it never reaches the peak thermal limits. This means that the performance of the CUs and the GPU are not constrained by temperature, and therefore they generally run at their maximum frequency. However, just because they can run at their maximum frequency does not mean that they should; there has to be a reasonable return in performance for the increase in frequency and higher power.

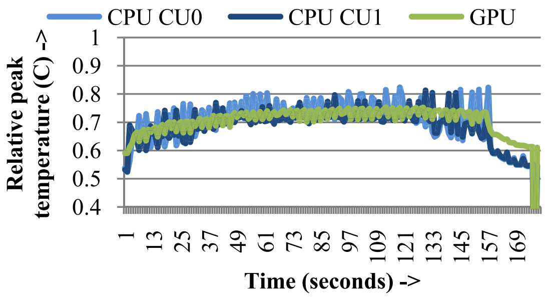

Fig. 3. Thermal profile of miniMD running on GPU. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

We characterize this return on performance with the notion of frequency sensitivity – a measure of the improvement in performance for a unit increase in frequency. Frequency sensitivity is a time-varying function of the workload on a target processor. In general, the frequency–performance function is unknown. Thus, the idea is to measure the frequency sensitivity of an application periodically and determine whether it is productive (efficient) to change the frequency. While Rountree et al. [39] developed a frequency-sensitivity predictor for homogeneous CPUs, the problem in APUs is more complex due to shared resources and subtle CPU–GPU interactions.

The rest of this section identifies and categorizes behaviors that have a substantive impact on frequency sensitivity of the components. All results are based on hardware measurements on an AMD A-Series APU (experimental set-up described in Section 5). This understanding is used in Section 4 to develop a model of frequency sensitivity for tightly coupled heterogeneous processors and to use the model to guide DVFS decisions.

3.1. Shared resource interference

The memory hierarchy is a key determinant of performance, and the CPU and the GPU share the Northbridge and memory controllers. The extent of interference at these points (which is time-varying) has a significant impact on the effectiveness of DVFS for the CPU or the GPU.

Figure 4 (left bar) breaks down the CPU and GPU memory access rates, normalized to peak-DDR bandwidth with 75% bus efficiency, of one of the main computation kernels (neighbor) in miniMD [8]. The kernel is run iteratively in the application for the entire

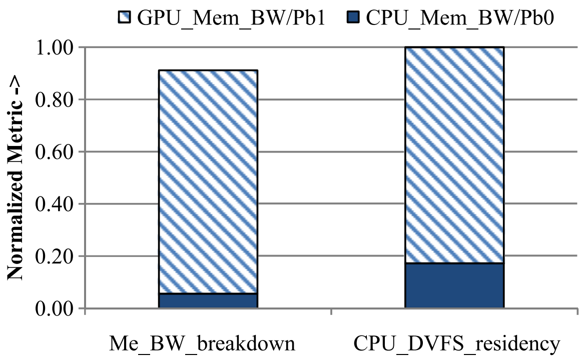

Fig. 4. Break-down of memory interference between CPU and GPU and corresponding CPU DVFS residency. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

steady-state duration. Figure 4 (right bar) breaks down the average CPU DVFS state residency for the active CPU time under BAPM, which shows that the kernel DVFS residency is entirely in the hardware managed CPU boost states.

We observe that this kernel saturates the overall shared-memory bandwidth primarily due to the high rate of memory references from the GPU. The CPU portion of memory demand, which is captured by looking at last-level cache L2 miss rates, is relatively insignificant. Further (not shown), the CPU IPC of this kernel is higher than a typical memory-bound application.

Power- and performance-management schemes that determine the CPU DVFS state in isolation of shared resources might conclude that the CPU voltage-frequency can be boosted within thermal limits to improve performance. This is, in fact, what the BAPM algorithm does. However, the application performance is memory bandwidth-limited due to the GPU memory demands, so scaling up the CPU voltage-frequency has little performance benefit and will degrade energy efficiency (discussed in Section 3.3). The lesson here is that we need online measurements of chip-scale global interactions to make good decisions regarding the CPU or the GPU DVFS state.

3.2. Computation and control divergence

GPUs are exceptional execution engines for data-parallel workloads with little control divergence. However, performance efficiency degrades significantly with increasing control divergence. That does not imply that lower-frequency states should be used for control divergent applications. Consider the Breadth-first Search (BFS) graph application from the Rodinia

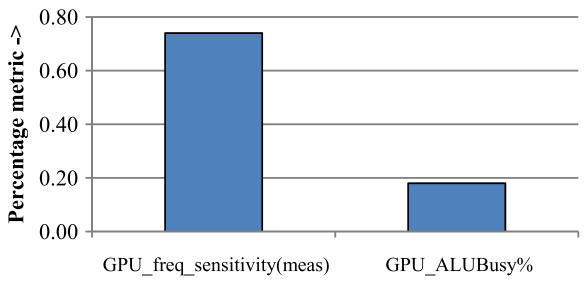

Fig. 5. GPU frequency sensitivity to control divergence. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

benchmark suite [9]. Figure 5 illustrates GPU frequency sensitivity for BFS (left bar). Execution times are measured at the lowest and highest frequencies. We compute frequency sensitivity as the ratio of the difference in execution times to the difference in frequencies. The figure also shows the GPU ALU compute utilization (right bar). While GPU ALU utilization and computation are fairly low, GPU frequency sensitivity is quite high. This is due to the high control flow-divergent behavior of the kernels in BFS, which leads to low utilization. However, higher-frequency operation leads to faster re-convergence, and thus shorter execution time.

Conventional cores that extract instruction-level parallelism from a single thread correctly associate low IPC with low frequency sensitivity. The converse is true here due to the bulk-synchronous parallel-processing nature of GPU kernels. Control flow serializes the execution of threads in a thread block. The correct analogy with traditional core execution is the observation that higher-frequency operation will speed-up the serial sections of code and, therefore, the application as a whole. In this case, the greater the serial fraction or divergence, the greater the speed-up. The lesson here is that control flow-divergence measures should be captured in the compute behavior when determining frequency sensitivity.

3.3. Performance-coupling and kernel sensitivity

Each application has phases that vary in their frequency sensitivity due to the type of their activity rates and the degree of performance-coupling between CPU and GPU. This is true also of HPC applications. While computations are offloaded to the GPU, there are control and data dependencies between computations executing on the CPU and the GPU cores. For example, for peak GPU utilization, the CPU must deliver data to the

GPU at a certain rate; otherwise, the GPU will starve, resulting in a reduction in overall performance. Such performance-coupling between the CPU and the GPU cores is accentuated by the tighter physical coupling due to on-die integration and the emergence of applications that attempt a more balanced use of the CPU and the GPU. Hence, any cooperative energy-management technique must balance such interactions against energy/power savings.

Here we evaluate the opportunities to save energy of an exascale proxy application from the Mantevo suite called miniMD [8]. In particular, we characterize the frequency and resource sensitivity at the kernel granularity for both the CPU and the GPU. We have observed this behavior in other HPC applications as well; however, due to space limitations, we present only miniMD results here. Figure 6 illustrates the GPU frequency sensitivity for the main miniMD kernels by measuring the impact of frequency on the speed-up of each kernel. The x-axis records the GPU DVFS states for each kernel. The y-axis shows the increase in run-time from the baseline BAPM case as GPU frequency is reduced. Because we are not thermally limited, the baseline algorithm runs the GPU at the highest frequency.

We can observe many interesting behaviors in the Fig. 6 graph, with the key insight being that different kernels in miniMD have different resource requirements and their relative sensitivities to GPU frequency reflect those needs. One of the main computation kernels, Force, scales very well with GPU frequency and performs the best at the highest-frequency GPU DVFS state. This is because of the heavy compute-bound nature of the kernel. The Neighbor kernel shows high sensitivity to GPU frequency when going from low to medium frequency; however, Neighbor sees little

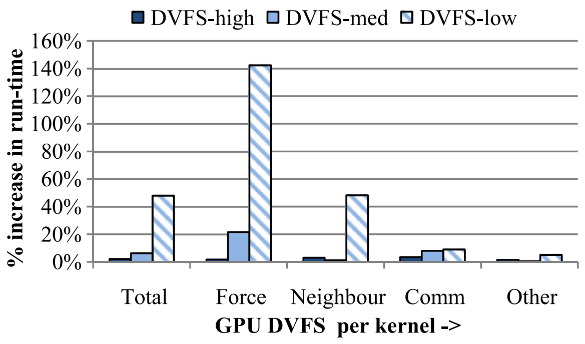

Fig. 6. Percent increase in kernel run-time due GPU DVFS changes relative to the baseline (BAPM). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

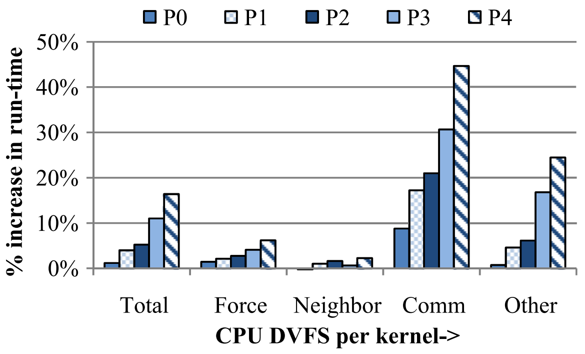

Fig. 7. Percent increase in kernel run-time due CPU DVFS changes relative to the baseline (BAPM). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

to no performance benefit at the highest GPU frequency because the Neighbor kernel becomes memory bandwidth-limited at the highest GPU frequency. Communication and other fine-grained, relatively short kernels labeled Other seem to be less sensitive to GPU frequency. There is a 6% increase in total run-time at the medium GPU DVFS state, with the Force kernel being the main contributor to the slow-down.

Consider the frequency sensitivity of the CPU for each of the miniMD kernels (recall the performance-coupling between the CPU and the GPU) illustrated in Fig. 7. The Force and Neighbor kernels do not scale well with CPU frequency. The memory-bounded behavior of Neighbor makes it insensitive to CPU frequency with minimal performance loss at the lower CPU DVFS state of P4. The GPU compute-intensive nature of Force makes it less dependent on CPU frequency; however, decreasing CPU frequency beyond P2 starts starving the GPU. On the other hand, fine-grained, shorter kernels such as Communication and others have higher data dependencies on the CPU and are tightly performance-coupled. Launch overhead, combined with the relatively small kernel timings compared to the actual execution time, make these kernels more tightly performance-coupled to CPU frequency and less GPU frequency-sensitive. The lesson here is that the frequency-sensitivity metric in an APU needs to account for performance-coupling effects.

3.4. Summary

The preceding analysis shows that HPC applications exhibit varying degrees of CPU and GPU frequency sensitivity for a variety of subtle and non-obvious reasons. Overall, the results in this section clearly point towards the need for a set of metrics for energy management that can predict CPU–GPU frequency sensitivity in a tightly coupled heterogeneous architecture. Using these metrics, we envision extending BAPM with frequency-sensitivity information to augment its functionality. We describe the model, its application, and results with measurements on real hardware in the following sections.

4. Run-time system for energy management

The first step is to develop a predictor for the frequency sensitivity of an application. Specifically, at any point in time we need to be able to predict the performance sensitivities of the application to the frequency of the CPU and the GPU, which may be different. As we observed in Section 3, this sensitivity analysis must account for indirect interactions between the CPUs and the GPU in the memory system and their coupled performance.

The second step is to encapsulate this into an energy-management algorithm that periodically computes the frequency sensitivity and, in response, adjusts the DVFS states of the CPU cores and the GPU. In this section, we derive a frequency-sensitivity predictor in heterogeneous processors and use it to construct a run-time energy-management scheme. Our goal is to develop a simple and practical predictor that can be implemented efficiently in a dynamic run-time algorithm with minimal hardware overhead and complexity.

4.1. Frequency sensitivity correlation

We developed frequency-sensitivity predictors to capture the dominant behaviors described in Section 3 for the GPU and the CPU.

First, we selected performance counters that are indicators of frequency sensitivity. Modern processors provide hundreds of detectable performance counters, which makes the selection quite challenging [5]. We used three exascale proxy applications (miniMD, miniFE and Lulesh), consisting of many different kernels [8,14,24]. We also utilized six scientific applications from the Rodinia benchmark suite: Needleman–Wunsch, HotSpot, LU Decomposition (LUD), Speckle-reducing Anisotropic Diffusion (SRAD), Computational Fluid Dynamics (CFD) and BFS [9,10]. The chosen applications have a wide range of characteristics such as coarse- and fine-grained kernels, compute- and memory-boundedness, different degrees of CPU–GPU performance-coupling, and divergent control flow.

Using an application analysis and profiling tool called CodeXL, we measured the execution times and the corresponding values of a set of performance counters/metrics at kernel boundaries over a range of CPU and GPU frequencies [18]. We initially attempted to find correlation across multiple sample points in a single application trace but found that minor discrepancies in phase alignment with performance metric traces can cause large variations in correlation. Hence, we looked for alignment only at the kernel granularity in an application. We performed a correlation analysis between each performance counter/metric and the CPU or GPU frequency sensitivity, measured as the ratio of the difference in execution times to the corresponding differences in frequency. We computed the correlation coefficients using linear regression (shown in Table 2). These performance counters/metrics were derived from a set of more than 40 hardware performance counters in the CPU, GPU, and Northbridge selected based on the insights gained from Section 3. Coefficient values greater than 0.5 or less than -0.5 are considered a strong positive or negative correlation, respectively [7]. These values are highlighted in Table 2.

Second, we calculated overall GPU or CPU frequency sensitivity based on the following analysis. As expected, ClockWeightedUPC shows high correlation for CPU frequency sensitivity, as does GPU ALU activity and ALUBusy for the GPU. This captures the compute behavior of an application in either type of core. However, to capture the compute behavior for normal operations as well as control-divergent applications, we weighed the ALUBusy metric with GPUClockBusy (note the improvement in correlation between line 3 and line 1 in Table 2).

As Fig. 5 shows, graph algorithms have a high degree of control-flow divergence; thus, some SIMD engines are idle and waiting for a thread to finish executing before all threads can re-converge and proceed. This produces poor ALU throughput, making it appear that the GPU is lightly utilized. However, when ALUBusy is weighted with the actual GPU clock activity, we get a higher rate of ALU activity for the active period and better correlation. Similar accounting has been done for CPUs; however, unlike the CPU, which is latency sensitive, the GPU's massively parallel bulk-synchronous computation creates a complex inter-relationship between control behavior and power [7].

GPU frequency sensitivity shows a strong negative correlation to CPU UPC. Similarly, CPU frequency

Table 2

Sensitivity analysis of various performance metrics

Metric	Description	Correlation coefficient to GPU FS (meas)	Correlation coefficient to CPU FS (meas)
WeightedALUBusy	ALUBusy weighted by GPUClockBusy	0.85	−0.62
ALUInsts PTI	Compute instructions per thousand instructions	0.78	−0.54
ALUBusy	The percentage of GPUTime ALU instructions are processed	0.76	−0.54
ALUFetchRatio	The ratio of ALU to fetch instructions. If the number of fetch instructions is 0, then 1 will be used instead	0.57	−0.31
L2 cache miss/cycle	Level 2 cache miss rate to main memory for CPU	0.13	−0.41
ALUPacking	The ALU vector packing efficiency (in percentage)	0.11	−0.22
GPUClockBusy	GPU utilization: Ratio of time when at least one of the SIMD units in the GPU is active compared to total execution time	0.06	−0.13
FetchUnitBusy	The percentage of GPUTime the fetch unit is active	−0.28	−0.01
FetchUnitStalled	The % of GPUTime main memory fetch/load unit is stalled	−0.49	−0.15
WriteUnitStalled	The % of GPUTime main memory write/store unit is stalled	−0.51	0.12
Writes to memory PTI	Main memory writes per thousand instructions	−0.60	−0.28
Fetch from memory PTI	Main memory reads per thousand instructions	−0.62	−0.23
Global_MemUtil	Aggregated CPU–GPU memory bandwidth consumed during theoretical peak bandwidth	−0.63	−0.56
ClockWeightedUPC	Retired micro-operations (includes all processor activity) per cycle weighted by each core's active clocks	−0.83	0.70

sensitivity shows a strong negative correlation to GPU ALUBusy. This is because of the data and execution dependencies between the GPU and CPU. As the computation becomes more balanced and distributed between the CPU and GPU, we expect the correlation coefficients to change. However, CPU and GPU performance still will be closely coupled in their interactions and dependencies. Therefore, a GPU frequency-sensitivity predictor needs to account for CPU UPC as a way to measure its performance-coupling. Similarly, CPU frequency sensitivity in a heterogeneous architecture needs to account for GPU ALU activity.

We found a better correlation between frequency sensitivity and aggregated memory bandwidth (Global_MemUtil) compared to the localized memory access metrics such as L2 cache miss in the CPU or memory fetch/write stalls in the GPU. This is largely because of the disparity in memory-bandwidth demand between the CPU and the GPU while accessing a shared resource, as shown in Fig. 4.

Based on the preceding analysis, we summarized a key set of performance metrics below to use in our run-time energy-management scheme to determine frequency sensitivities in a performance-coupled heterogeneous architecture. We determined CPU and GPU frequency sensitivities as weighted linear regression functions of these combined metrics to capture performance-coupling, core compute behavior, and global memory interference. The correlation coefficient using this combination of metrics improved to 0.97:

$$\text{WeightedALUBusy} = \frac{\text{ALUBusy}}{\text{GPUClockBusy}},$$

$$\text{Global\_MemUtil} = \frac{\text{AggregatedMemBW}}{\text{TheoreticalPeakMemBW}},$$

where

$$\begin{aligned}&\text{TheoreticalPeakMemBW}\\&\quad = (\text{DDRClockSpeed}) * (\text{8 bytes per clock})\\&\qquad * (\text{Total DDR channels}),\\&\text{ClockWeightedUPC}\\&\quad = \sum\bigl(\text{Total no. of retired uops}[i]\\&\qquad\quad * \text{UnhaltedCoreClocks}[i]\bigr)\\&\qquad \Big/ \sum \text{UnhaltedCoreClocks}[i].\end{aligned}$$

Although the set of applications analyzed here uses an offload model for computation, in which kernels run on the GPU with periodic synchronization points between CPU and GPU, we do not expect the performance metrics (WeightedALUBusy, Global_MemUtil, and ClockWeightedUPC) to change with more concurrent computation across CPU–GPU; however, the weights associated with the metrics in the linear regression equation may change to reflect even tighter performance-coupling between CPU and GPU. In future we plan on examining the impact of concurrent CPU–GPU execution on power-management algorithms.

4.2. *DynaCo*: *Coordinated dynamic energy management scheme*

We propose a run-time energy-management scheme called DynaCo based on the online measurement of the frequency sensitivity described in Section 4.1. DynaCo is implemented as a system software policy layered on top of the baseline AMD A-Series power-management system (BAPM).

The energy-management algorithm is partitioned into a monitoring block that samples the performance counters every 10 ms to coincide with the operating system timer tick for minimizing overheads, and a decision block that computes frequency sensitivities using measurements described at the end of Section 4.1. The CPU and GPU DVFS states are then configured. In general, DynaCo periodically determines whether the CPU and the GPU frequencies are high or low. In each case, the energy management algorithm embodies the following logic:

(1) *High GPU sensitivity, Low CPU sensitivity*: Shift power to the GPU (i.e., boost the GPU to maximize performance).
(2) *High GPU sensitivity, High CPU sensitivity*: Distribute power proportionally based on their relative sensitivities.
(3) *Low GPU sensitivity, High CPU sensitivity*: Shift power to the CPU (i.e., boost the CPU to maximize performance).
(4) *Low GPU sensitivity, low CPU sensitivity*: Reduce power of both the CPU and the GPU by using low-power states.

Because HPC applications are mostly uncompromising with respect to performance loss, we propose two energy-management algorithms – one more aggressive than the other in attempting to reduce power but with potentially higher performance degradation. In the less aggressive variant, DynaCo-1levelTh (Fig. 8), we limit the lowest-frequency P-state to P2; the CPU is not permitted to go to a lower-frequency state. Thus, in this case, there is potential to lose some power-saving opportunity. In the more aggressive version, DynaCo-multilevelTh (Fig. 9), the CPU is allowed to use all of the low-power P-states during low-sensitivity phases by analyzing gradients in memory access rates. In both versions, the GPU is handled similarly and allowed to use all DVFS states. In Fig. 9, we show DynaCo-multilevelTh for only the portions in which it is different from DynaCo-11evelTh. For our analysis, the GPU-high and -med thresholds for GPU WeightedALUBusy were set to 80% and 30%, respectively, based on GPU utilization and variations in workload intensity of graphics and HPC benchmarks;

Algorithm 1. Dynamic scheme (DynaCo-1levelTh)

```
while TRUE do
  if (Global_MemUtil $>=$ DDR_bus_efficiency) then
    /* Case: Memory is bottleneck */
    SetGPUFreqState(GPU-med);
  SetCPUFreqState(CPU-low-power_P2);
  end if
  else  /* Case: Memory is not bottleneck */
    if(ClockWeightedUPC $>=$ UPC_Threshold) then
/* CPU frequency sensitive, consider GPU sensitivity */
      if (WeightedALUBusy $>=$ HIGH) then
          SetGPUFreqState(GPU-high);
        SetCPUFreqState(CPU−base);
      else
      if (MEDIUM $<=$ WeightedALUBusy $<$ HIGH) then
          SetGPUFreqState(GPU-med);
        SetCPUFreqState(CPU-boost);
      else
          SetGPUFreqState(GPU-low);
        SetCPUFreqState(CPU-boost);
      end if
    else
    if(ClockWeightedUPC $<$ IPC_Threshold) then
/* CPU frequency insensitive, consider GPU sensitivity */
          SetCPUFreqState(CPU-low-power_P2);
      if (WeightedALUBusy $>=$ HIGH) then
          SetGPUFreqState(GPU-high);
      else
      if (MEDIUM $<=$ WeightedALUBusy $<$ HIGH) then
          SetGPUFreqState(GPU-med);
      else
          SetGPUFreqState(GPU-low);
      end if
    end if
  end if
    Sleep.time(SAMPLING_INTERVAL);
end while
```

Fig. 8. DynaCo-1levelTh pseudo-code.

Algorithm 2. Dynamic scheme (DynaCo-multilevelTh)

```
1:  while TRUE do
----lines 2 through 21 in Algorithm 1--------------
22:   if (ClockWeightedUPC < UPC_Threshold) then
23: /* CPU frequency insensitive, consider GPU sensitivity */
24:     if (WeightedALUBusy >= HIGH) then
25:         SetGPUFreqState(GPU-high);
26:     else
27:       if (MEDIUM <= WeightedALUBusy < HIGH) then
28:           SetGPUFreqState(GPU-med);
29:       else
30:           SetGPUFreqState(GPU-low);
31:       end if
32:       SetCPUFreqState(CPU-low-power_Pstate);
33:       Compute_ MemAccessRate_gradient();
34:       if (gradient>=Mem_threshold) then
35:         if (CPU-low-power_Pstate<= Pmin) then
36:             CPU-low-power _Pstate++;
37:         end if
38:         else
39:         if (CPU-low-power > CPU − base + 1) then
40:             CPU-low-power _Pstate–;
41:         end if
42:       end if
43:     end if
44:   end if
45:   Sleep.time(SAMPLING_INTERVAL);
46: end while
```

Fig. 9. DynaCo-multilevelTh pseudo-code.

UPC_threshold was set to 0.4 based on empirical observations across a wide range of workload characteristics in this architecture. The CPU and GPU DVFS settings are described in Section 2. Pmin is the lowest available CPU P-state.

The key observation is that when there is significant coupling/interaction between the CPU and the GPU, having the lowest CPU P-states can lead to significant power savings but significant performance degradation. At lower levels of coupling, significant power savings can occur with little performance degradation. The choice of algorithm depends on the degree of coupling, which can be time-varying. For example, if an HPC application has little communication overhead between the CPU and GPU, such as a compute-offload application in which the serial fraction of the code is insignificant compared to the total execution time, both DynaCo schemes may provide similar performance but DynaCo-multilevelTh will provide better power and energy savings.

5. Experimental set-up

We used the AMD[2] A10-5800 desktop APU with 100 W TDP as the baseline for all our experiments and analysis. Base CPU frequency is 3.8 GHz, with boost frequency up to 4.2 GHz. The GPU frequency is 800 MHz for the highest DVFS boost state [16]. We used four 2-GB DDR3-1600 DIMMs with two DIMMs per channel. Hardware performance counters for CPU and GPU were monitored using CPU and GPU performance counter libraries running in Red Hat Linux[3] OS. We set specific CPU DVFS states using model-specific registers as described in [5]; to set a specific GPU DVFS state, we send memory-mapped messages through the GPU driver layer to the power-management firmware.

Although our DynaCo scheme can be implemented in any layer such as hardware, power-management firmware, or system software, we implemented it as a run-time system software policy by layering it on top of the baseline AMD A-Series power-management system. For CPU and GPU power and temperature, we used the digital estimates provided by the power-management firmware running in the AMD A-Series processor, the accuracies for which are detailed in [32]. For all schemes, we ran the applications for several iterations to reach a thermally stable steady state. We took an average across those multiple iterations to eliminate run-to-run variance in our hardware measurements.

We selected the applications used in our experiments based on their relevance to future high-performance scientific computing. We evaluated seven OpenCL applications in this paper: miniMD, miniFE, Lulesh, S3D, Sort, Stencil2D and BFS. MiniMD, miniFE and Lulesh are proxy applications representative of HPC scientific application characteristics in the exascale time-frame. We also evaluated a sub-set of benchmarks (S3D, Sort, Stencil2D, BFS) from the Scalable Heterogeneous Computing (SHOC) benchmark suite [13] that represents a large portion of scientific code found in HPC applications. We analyzed all applications on a single node to explore energy-saving opportunities using our run-time schemes. These applications and the associated datasets are described in Table 3.

MiniMD is a molecular dynamics code derived from its parent code, LAMMPS [8]. It has two main compu-

[2]AMD, the AMD Arrow logo, AMD Opteron, AMD Radeon and combinations thereof are trademarks of Advanced Micro Devices, Inc.

[3]Linux is the registered trademark of Linus Torvalds.

Table 3

Application datasets

Application	Problem size
miniMD	$32 \times 32 \times 32$ elements
miniFE	$100 \times 100 \times 100$ elements
Lulesh	$100 \times 100 \times 100$ elements
Sort	2,097,152 elements
Stencil2D	$4{,}096 \times 4{,}096$ elements
S3D	SHOC default for integrated GPU
BFS	1,000,000 nodes

tational kernels. The first is the L–J potential function, or force kernel, and the second is the neighbor-binning algorithm, or neighbor kernel. Other kernels include communication kernel atom_comm and miscellaneous small kernels to integrate the atom forces and build the neighbor's list for each atom based on proximity and other variables.

MiniFE provides an implementation of a finite-element method [14]. It provides a conjugate gradient (CG) linear system solver with Jacobi preconditioning. The three main kernels in the CG solver are matvec, which does matrix vector operations; dot, which performs the dot product of two matrices; and waxpy, which does the weighted sum of two vectors.

Lulesh [24] approaches the hydrodynamics problem using Lagrangian numerical methods. The two main computation kernels in Lulesh are CalcHourGlassForces and CalcFBHourGlassForces.

SHOC consists of a collection of complex scientific applications and common kernels encapsulated into benchmarks that represent a majority of the numerical operations found in HPC. We use Sort; which sorts an array of key-value pairs using a radix sort algorithm; Stencil2D, which uses a nine-point stencil operation applied to a 2D dataset; S3D, which is a turbulent combustion simulation; and BFS, which is a graph traversal problem.

We report performance, power, and energy efficiency (energy-delay^2 product) for the two variants of DynaCo algorithm. We picked ED^2 because it has been widely used in HPC analysis [26] and it captures the importance of both power and performance, the latter being critical for HPC. The power and energy results include CPU, GPU, memory controller power, and a fixed IO-phy power budget. All results were obtained from real hardware and are normalized to the baseline BAPM discussed in Section 2. All averages represent geometric mean across the applications.

6. Results

This section describes the results from the two DynaCo schemes in the AMD A-Series APU and compares them with the state-of-the-practice power-management algorithm BAPM. We also compare our DynaCo schemes with an ideal static scheme that picks the best DVFS state for each kernel as determined through offline profiling and analysis by performing an entire state-space search. Offline techniques provide a good basis for comparison to evaluate the effectiveness of run-time techniques but are impractical as power-management strategies.

6.1. *Performance, power and energy*

Figure 10 shows the performance impact of DynaCo-1levelTh, DynaCo-multilevelTh, and ideal static schemes compared to the baseline for all six HPC applications. The y-axis represents the increase in run-time compared to a baseline value of 1.0, and lower is better. We see an average run-time increase of 0.78% across all the applications using DynaCo-1levelTh, with up to 2.58% maximum slow-down in the case of miniMD.

DynaCo-multilevelTh sees an average run-time increase of 1.61% across the same set of applications, with a worst-case slow-down of 4.19%. The ideal static scheme measures an average slow-down of 1.65%, with the worst case being 5.2% in miniMD. This illustrates the efficacy of the run-time schemes in optimizing energy efficiency under strict performance constraints. Ideal-static picks the best CPU and GPU DVFS states at a kernel-level granularity, and it is unable to detect fine-grained phase changes in a ker-

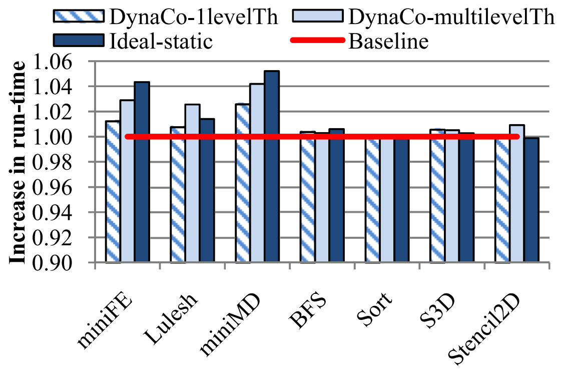

Fig. 10. Performance impact of DynaCo. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

nel. Hence, it penalizes short, high-frequency sensitive phases in a kernel that overall has low sensitivity.

As expected, we see much tighter performance control with DynaCo-1levelTh compared to DynaCo-multilevelTh and ideal static because it does not utilize the lowest-frequency states of the CPU. Because it always fixes the low-power P-state for CPU to P2 during phases of low CPU frequency sensitivity, it also removes the slight variability in performance over time when the algorithm is adapting dynamically to find the best low-power P-state. On the other hand, DynaCo-multilevelTh provides better energy efficiency gains, as we will see next, with slightly more performance degradation but still within reasonable bounds of HPC constraints [26]. We attribute the relatively higher performance loss in miniMD to the impact of variability in kernel phases shorter than our 10 ms sampling interval limitation.

The more aggressive DynaCo-multilevelTh outperforms ideal static in miniFE and miniMD because a run-time adaptive scheme is able to take advantage of the phase behavior in a kernel, whereas the static scheme based on profiling makes power-state decisions only at kernel-level granularity. Figure 11 shows an example phase behavior of the matvec kernel in miniFE for a single iteration. The y-axis shows GPU utilization and normalized memory-bandwidth utilization compared to the practical peak-DDR bandwidth. Matvec does sparse matrix-vector product and, in general, is heavily memory bandwidth-limited due to the large number of indirect memory references and register spills to global memory in the code. However, about 19% of the time it is compute-intensive without saturating memory bandwidth. This behavior is observed in every invocation of matvec in miniFE, a significant fraction of the application's total run-time. During this 19% compute-intensive phase, DynaCo boosts the GPU to its highest DVFS state while the profiling-based ideal static scheme fixes the GPU frequency to GPU-med due to this kernel's overall low GPU frequency sensitivity.

In Fig. 12 we evaluate the ED^2 gains using DynaCo during the entire run-time of the application. All data are normalized to a baseline of 1.0 and lower is better. Average energy efficiency improves by 24% using DynaCo-1levelTh compared to the baseline, with up to 32% savings in Sort and S3D. DynaCo-multilevelTh sees an average improvement of 30%, with up to 47% savings in S3D. Ideal-static achieves an energy-efficiency gain of 35%. We observe that 70–80% of the savings came from CPU scaling and the remainder came from GPU scaling.

The amount of energy-efficiency gain in S3D is slightly higher than the rest of the benchmarks. S3D is a compute-intensive application. However, when we run multiple iterations of this benchmark from the SHOC suite, the compute-intensive active phases appear to last for a small fraction of the total time it takes to compile and launch the application kernels. This causes only small periods of activity on the GPU followed by long idle periods. During this idle period, the GPU is power-gated for all three schemes as well as the baseline. However, the CPU is busy compiling and preparing the work to launch the kernels. Portions of this phase do not contribute to the overall performance of the application. Boost algorithms, such as the BAPM algorithm used for the baseline, allocate the highest CPU frequencies during this phase when power and thermal headroom is available. However, in our run-time and ideal-static schemes we are able to utilize the low frequency P-states during the frequency-insensitive phase. We also notice that DynaCo-multilevelTh provides better energy effi-

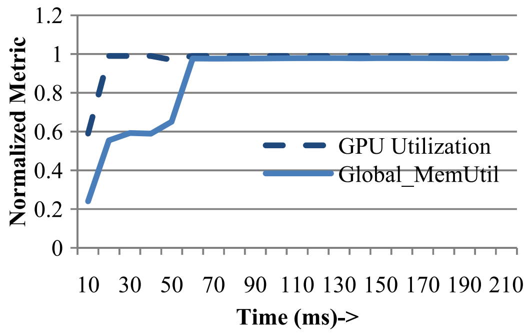

Fig. 11. Phase variation within MATVEC. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

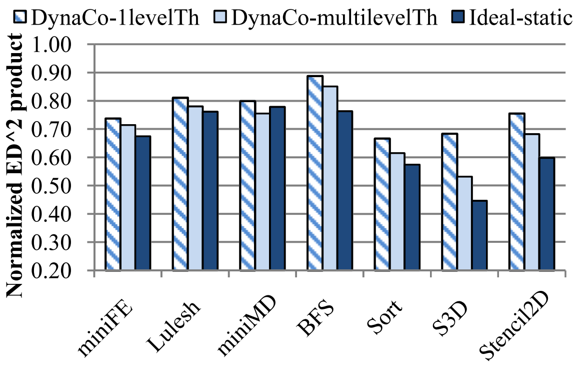

Fig. 12. Energy efficiency with DynaCo. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

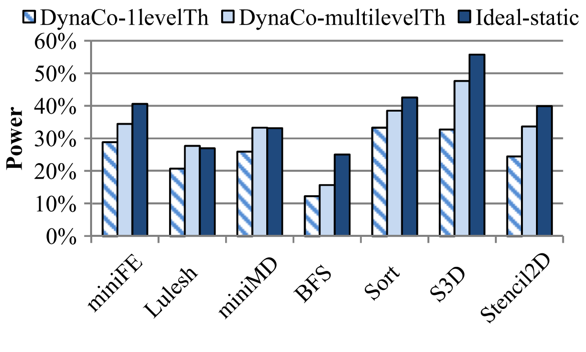

Fig. 13. Power savings with DynaCo. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

ciency than ideal static for miniMD due to the higher-performance slow-down observed with the profile-based scheme.

The power savings achieved with DynaCo are illustrated in Fig. 13. The average power savings are 24% with DynaCo-1levelTh, 31% with DynaCo-multilevelTh, and 36% with ideal-static. We see that DynaCo-multilevelTh provides greater power savings compared to DynaCo-1levelTh due to utilization of the very-low-frequency CPU P-states. While ideal static provides greater power savings by picking the best DVFS state for each kernel, it does not provide the same tight performance bounds as the other two schemes, as shown in Fig. 10. In addition, it requires user intervention and prior offline profiling of all the kernels in an application across multiple CPU and GPU frequencies to determine the best state.

6.2. *Performance analysis and power shifting*

We now analyze the case of power-shifting and power-reduction scenarios with the two DynaCo schemes for every application. We present a sub-set of those results here. Figure 14 shows the percentage GPU DVFS residency for each of the three main kernels of Lulesh as well as the overall application. Because the GPU DVFS decision between the two DynaCo schemes is handled similarly from an algorithmic perspective, we show GPU DVFS residency results for only DynaCo-1levelTh.

The CalcHourGlass kernel spends 21% less time in GPU-high, 14% more time in GPU-med, and 8% more time in GPU-low than the baseline. On further examination, this kernel is memory-bounded 30% of its run-time; during those times, power is shifted away from the GPU. Similarly, the entire Lulesh application spends 9% less time in GPU-high with DynaCo. For the CalcFBHourGlass kernel, DynaCo performs simi-

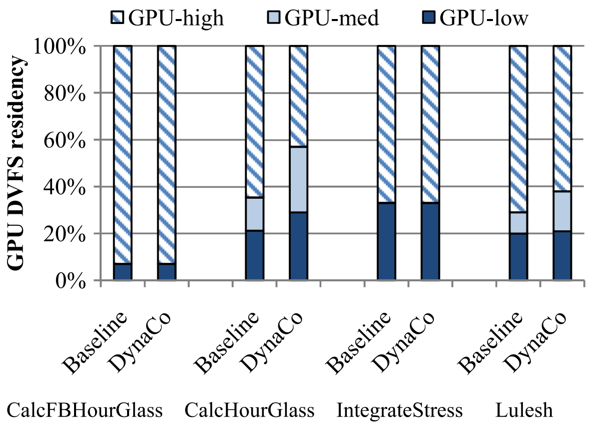

Fig. 14. GPU DVFS residency for DynaCo and baseline. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

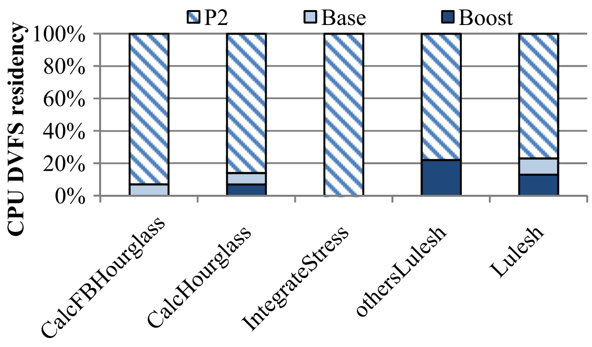

Fig. 15. CPU DVFS residency with DynaCo-1levelTh. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

larly to the baseline. This kernel is heavily compute-bound on GPU with high WeightedALUBusy; hence, DynaCo boosts performance by selecting the highest-frequency state.

Further, in Figs 15 and 16, we see that for all three kernels we are able to shift power away from CPU significantly due to poor CPU frequency sensitivity, while the baseline runs the CPU at the high-frequency boost states due to availability of power and thermal headroom. Specifically, during the more than 20 Other kernel phases, DynaCo correctly boosts CPU to the high-frequency P-states as needed due to the high CPU dependency observed for these miscellaneous fine-grained kernels, as depicted in the phase behavior shown in Fig. 1. Further, DynaCo-multilevelTh is able to utilize the lower-frequency CPU P-states P3 and P4 59% of the time.

We also observe that the fine-grain, relatively small kernels such as IntegrateStress become performance-

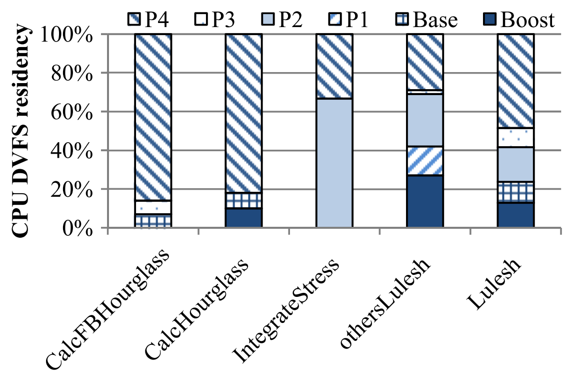

Fig. 16. CPU DVFS residency with DynaCo-multilevelTh. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-140380.)

coupled to the CPU more quickly than the main Hourglass computation kernels. Hence, IntegrateStress does not utilize the low-frequency P-states of P3 and P4 that can cause significant performance loss. Lulesh provides an example of a case when power can be saved from both CPU and GPU and boosting to higher frequencies is utilized when the application phase needs it.

Similarly, for miniMD (figure not shown due to space constraints), DynaCo correctly estimates the frequency sensitivity of the different kernels. The heavily compute-intensive nature of the force kernel causes it to boost to the highest GPU frequency 100% of its run-time, similar to the baseline. On the other hand, the neighbor kernel has aggregated CPU–GPU memory bandwidth that is close to the peak bandwidth that the DDR bus can sustain after accounting for bus efficiency. Hence, we are able to run the GPU at the medium DVFS frequency without noticeable performance degradation. Moreover, small kernels in miniMD such as atom_comm, which rely on the CPU for data transfer and launch frequently, spend almost 70% of their time in the medium- and low-frequency GPU DVFS states using DynaCo. During much of this time, CPU is closely coupled to the GPU and runs at high-frequency P-states. Contrary to this, the baseline algorithm runs at maximum CPU and GPU frequencies for all miniMD kernels due to temperature headroom.

In the graph algorithm BFS, we see that due to control divergence the GPU has short bursts of computation followed by phases of low utilization on the GPU. About 25% of the time, threads are waiting for reconvergence. DynaCo correctly assigns high GPU frequency to avoid slowing the critical path, but it saves power from the CPU due to low UPC. The baseline always runs BFS at the maximum CPU and GPU frequencies due to the available thermal headroom, causing energy inefficiency.

Due to the lower power consumption, we also see a reduction in the peak die temperatures using DynaCo. This is due to a combination of leakage power savings from reduced voltage operations and dynamic power savings from reduced frequency. With DynaCo, peak die temperature is, on average, up to 2°C lower across all the applications. Lower temperatures result in lower cooling costs, better energy efficiency, and better heat management.

In summary, we have shown that DynaCo successfully leverages the metrics discussed in Section 4 to improve the energy efficiency of HPC application on a heterogeneous processor. DynaCo is able to produce significant power savings with a small reduction in performance, resulting in energy efficiencies comparable to an ideal static management scheme without the additional overhead of profiling required for the static scheme.

7. Related work

There is a considerable amount of research in power and energy management in homogeneous uni- and multi-core processors using dynamic voltage and frequency scaling. Several research works have proposed analytical models for DVFS [12,21,25,42], compiler-driven techniques [43], and control-theoretic approaches [40]. Li et al. [31] proposed a run-time voltage/frequency and core-scaling scheduling algorithm that minimizes the power consumption of general-purpose chip multi-processors within a performance constraint. Lee et al. [29] analyzed throughput improvement of power-constrained multi-core processors by using power-gating and DVFS techniques.

In the HPC community, Pakin et al. [33] characterized power usage on production supercomputers using production workloads. Laros et al. [27] performed extensive large-scale analysis of power and performance requirements for scientific applications in supercomputers based on static tuning of applications through DVFS, core, and bandwidth scaling. Rountree et al. [38] explored energy-performance trade-offs for HPC applications bottlenecked by memory and communication. In [37] and [39], Rountree et al. investigated speeding up the critical path of an application in a multi-processor cluster using slack-prediction and leading-load techniques, respectively. In [4], Balaprakash et al. described exascale workload characteristics and created a statistical model to extrapolate application characteristics as a function of problem size. All these efforts focused only on CPU architectures; this paper focuses on the integrated CPU–GPU architectures that bring new challenges.

There has been a renewed interest in using machine learning to construct behavior models for use in runtimes, compilers, and even hardware to make scheduling decisions. Techniques to automate the construction of models of execution time for GPUs using basic machine learning are described in [23] and [26]. Other efforts have constructed models for predicting power and thermal behaviors from measurements made with performance counters [6,15]. Such techniques focus on model construction and are distinct from model application (e.g., in making power-management decisions). They certainly could be investigated to improve the models presented here further with the requirement that simplicity of implementation be met.

Recently, there has been a significant interest in the power management of GPUs. Lee et al. [30] proposed DVFS techniques to maximize performance within a power budget for discrete GPUs. In [15], Hong et al. developed a power and performance model for a discrete GPU processor. Recent studies [2,3] have identified throughput-computing performance-coupled applications as an emergent class of future applications. However, none of this work focuses on managing energy in tightly coupled architectures.

A number of papers have examined CPU–GPU heterogeneous architectures. Research in [11,22,35,36] proposed run-time systems with scheduling schemes for applications like generalized reductions, irregular reductions, and MapReduce to improve performance in CPU–GPU architectures. In [28], Lee et al. proposed thread-level-parallelism-aware cache management policies in CPU–GPU processors. Wang et al. [41] proposed workload-partitioning mechanisms between the CPU and GPU to utilize the overall chip power budget to improve throughput. In [34], Paul et al. characterized thermal coupling effects between CPU and GPU and proposed a solution to balance thermal and performance-coupling effects dynamically.

Unlike many of the previous studies, our work is to our knowledge the first to address energy management in integrated CPU–GPU processors for HPC applications. Furthermore, unlike much past work in this area, we have implemented our algorithms on commodity hardware and show measurable performance, power, and energy benefits compared to a state-of-the-practice power-management algorithm.

8. Conclusions

This paper proposed and implemented a set of techniques to improve the energy efficiency of integrated CPU–GPU processors. To the best of our knowledge, this is the first such implementation. We described the unique characteristics of HPC applications and the opportunities they present to save energy. We proposed a model to capture the application's frequency sensitivity in such architectures and used this model as the basis for a dynamic, coordinated energy-management scheme to improve energy efficiency at negligible performance loss. The proposed scheme achieves an average ED^2 benefit of up to 30% compared to the baseline with less than 2% average performance loss across a variety of exascale and other HPC applications.

In the future, we plan to expand this work to manage memory sub-systems directly, explore techniques to balance computation on both CPU and GPU efficiently for energy, and extend this node-level analysis to the level of an HPC cluster.

Acknowledgements

The authors gratefully acknowledge the efforts and detailed comments of the reviewers, which substantially improved the final manuscript. This research was supported in part by the Semiconductor Research Corporation under contract 2012-HJ-2318.

References

[1] Advanced Configuration and Power Interface (ACPI), Specification, available at: http://www.acpi.info/spec.htm.

[2] M. Arora, S. Nath, S. Mazumdar, S. Baden and D. Tullsen, Redefining the role of the CPU in the era of CPU–GPU integration, in: *IEEE Micro*, 2012.

[3] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams and K.A. Yelick, The landscape of parallel computing research: A view from Berkeley, Technical Report UCB/EECS-183, 2006.

[4] P. Balaprakash, D. Buntinas, A. Chan, A. Guha, R. Gupta, S. Narayanan, A. Chieny, P. Hovland and B. Norris, An exascale workload study, in: *SCC*, 2012.

[5] BIOS and Kernel Developer's Guide: http://support.amd.com/us/Processor_TechDocs/42300_15h_Mod_10h-1Fh_BKDG.pdf.

[6] W.L. Bircher and L.K. John, Complete system power estimation: A trickle-down approach based on performance events, in: *ISPASS*, 2007.

[7] W.L. Bircher, M. Valluri, J. Law and L.K. John, Runtime identification of microprocessor energy saving opportunities, in: *ISLPED*, 2005.

[8] W.M. Brown, P. Wang, S.J. Plimpton and A.N. Tharrington, Implementing molecular dynamics on hybrid high performance computers-short range forces, in: *Compute Physics Communications*, 2011.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee and K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: *IISWC*, 2009.

[10] S. Che, J.W. Sheaffer, M. Boyer, L. Szafaryn and K. Skadron, A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads, in: *IISWC*, 2010.

[11] L. Chen, X. Huo and G. Agrawal, Accelerating map-reduce on a coupled CPU–GPU architecture, in: *SC*, 2012.

[12] M. Curtis-Maury, A. Shah, F. Blagojevic, D. Nikolopoulos, B.R. de Supinski and M. Schulz, Prediction models for multidimensional power-performance optimization on many cores, in: *PACT*, 2008.

[13] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju and J.S. Vetter, The scalable heterogeneous computing (SHOC) benchmarking suite, in: *GPGPU*, 2010.

[14] M. Heroux, D. Doerfler, P. Crozier, J. Willenbring, H.C. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thornquist and R. Numrich, Improving performance via mini-applications, in: *SAND2009-5574*, 2009.

[15] S. Hong and H. Kim, An integrated GPU power and performance model, in: *ISCA*, 2010.

[16] http://www.amd.com/us/products/desktop/processors/a-series/Pages/a-series-model-number-comparison.aspx.

[17] http://www.xbitlabs.com/news/other/display/20111102214137_AMD_and_Penguin_Build_World_s_First_HPC_Cluster_Based_on_Fusion_APUs.html.

[18] http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/.

[19] http://www.green500.org.

[20] http://www.top500.org.

[21] Z. Hu, D. Brooks, V. Zyuban and P. Bose, Microarchitecture-level power-performance simulators: modeling, validation and impact on design, in: *MICRO*, 2003.

[22] X. Huo, V.T. Ravi and G. Agrawal, Porting irregular reductions on heterogeneous CPU–GPU configurations, in: *HiPC*, 2011.

[23] W. Jia, K. Shaw and M. Martonosi, Stargazer: Automated regression-based GPU design space exploration, in: *IEEE ISPASS*, 2012.

[24] I. Karlin, LULESH programming model and performance ports overview, LLNL-TR-608824, 2012.

[25] S. Kaxiras and M. Martonosi, Computer architecture techniques for power efficiency, in: *Synthesis Lectures on Computer Architecture*, 2008.

[26] A. Kerr, E. Anger, G. Hendry and S. Yalamanchili, Eiger: A framework for the automated synthesis of statistical performance models, in: *1st Workshop on Performance Engineering and Applications* (*WPEA*), 2012, held with *HiPC*.

[27] J.H. Laros, III, K.T. Pedretti, S.M. Kelly, W. Shu and C.T. Vaughan, Energy based performance tuning for large scale high performance computing systems, in: *HPC*, 2012.

[28] J. Lee and H. Kim, TAP: A TLP-aware cache management policy for a CPU–GPU heterogeneous architecture, in: *HPCA*, 2012.

[29] J. Lee and N. Kim, Optimizing throughput of power- and thermal-constrained multicore processors using DVFS and per-core power-gating, in: *DAC*, 2009.

[30] J. Lee, V. Sathish, M. Schulte, K. Compton and N. Kim, Improving throughput of power-constrained GPUs using dynamic voltage/frequency and core scaling, in: *PACT*, 2011.

[31] J. Li and J. Martinez, Dynamic power-performance adaptation of parallel computation on chip multiprocessors, in: *HPCA*, 2006.

[32] S. Nussabaum, AMD, Trinity, APU, in: *Hotchips*, 2012.

[33] S. Pakin, C. Storlie, M. Lang, R. Fields, III, E. Romero, C. Idler, S. Michalak, H. Greenberg, J. Loncaric, R. Rheinheimer, G. Grider and J. Wendelberger, Power usage of production supercomputers and production workloads, in: *SC*, 2012.

[34] I. Paul, S. Manne, M. Arora, W.L. Bircher and S. Yalamanchili, Cooperative boosting: needy versus greedy power management, in: *ISCA*, 2013.

[35] V.T. Ravi and G. Agrawal, A dynamic scheduling framework for emerging heterogeneous systems, in: *HiPC*, 2011.

[36] V.T. Ravi, W. Ma, D. Chiu and G. Agrawal, Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations, in: *ICS*, 2010.

[37] B. Rountree, D.K. Lowenthal, B.R. de Supinski, M. Schulz, V. Freeh and T. Bletsch, Adagio: Making DVS practical for complex HPC applications, in: *ICS*, 2009.

[38] B. Rountree, D.K. Lowenthal, S. Funk, V. Freeh, B.R. de Supinski and M. Schulz, Bounding energy consumption in large-scale MPI programs, in: *SC*, 2007.

[39] B. Rountree, D.K. Lowenthal, M. Schulz and B.R. de Supinski, Practical performance prediction under dynamic voltage frequency scaling, in: *IGCC*, 2011.

[40] A. Varma, B. Ganesh, M. Sen, S.R. Choudhury, L. Srinivasan and B.L. Jacob, A control-theoretic approach to dynamic voltage, in: *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2003.

[41] H. Wang, V. Sathish, R. Singh, M. Schulte and N. Kim, Workload and power budget partitioning for single chip heterogeneous processors, in: *PACT*, 2012.

[42] Q. Wu, P. Juang, M. Martonosi and D.W. Clark, Formal online methods for voltage/frequency control in multiple clock domain microprocessors, in: *ASPLOS*, 2004.

[43] Q. Wu, M. Martonosi, D. Clark, V. Reddi, D. Connors, Y. Wu, J. Lee and D. Brooks, Dynamic compiler-driven control for microprocessor energy and performance, in: *IEEE Micro*, 2006.