Energy-Efficient Scheduling on Heterogeneous Multi-Core Architectures

Jason Cong
Computer Science Department
University of California, Los Angeles
cong@cs.ucla.edu

Bo Yuan
Computer Science Department
University of California, Los Angeles
boyuan@cs.ucla.edu

ABSTRACT

The use of heterogeneous multi-core architectures has increased because of their potential energy efficiency compared to the homogeneous multi-core architectures. The shift from homogeneous multi-core to heterogeneous multi-core architectures creates many challenges for scheduling applications on the heterogeneous multi-core system. This paper studies the energy-efficient scheduling on Intel's QuickIA heterogeneous prototype platform [6]. A regression model is developed to estimate the energy consumption on the real heterogeneous multi-core platform. Our scheduling approach maps the program to the most appropriate core, based on program phases, through a combination of static analysis and runtime scheduling. We demonstrate the energy efficiency of our phase-based scheduling method by comparing it against the statical mapping approach proposed in [5] and the periodic sampling based approach proposed in [11], The experimental results show that our scheduling scheme can achieve an average 10.20% reduction in the energy delay product compared to [5] and an average 19.81% reduction in energy delay product compared to [11].

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*

Keywords

Energy Efficiency, Scheduling, Heterogeneous Multi-core

1. INTRODUCTION

Due to the increase in transistor budgets enabled by Moore's law, more and more cores are now integrated on chips. Accordingly, the on-chip power consumption becomes a critical issue. Conventional multi-core processors consist of identical cores. An alternative design approach for multi-core processors is to implement heterogeneous cores on a chip, which is a promising solution for power-efficient computing [8, 11].

Heterogeneous cores provide different power/performance tradeoffs. In order to benefit from heterogeneous multi-core architectures, the scheduler will need to consider the power/performance asymmetry of heterogeneous multi-core architectures when making a scheduling decision. Because the program has different performances and energy consumptions on different cores, scheduling the program to the most appropriate core is a challenging problem on heterogeneous multi-core architectures. To address these challenges, recent research proposes several scheduling schemes for heterogeneous multi-core architectures. Single-ISA heterogeneous multi-core architectures with simultaneous multi-threading processors are explored by Kumar et al. [10]. Dynamic core assignment policies are proposed to support the scheduling of multiprogrammed workloads. By adapting to inter-thread and intrathread diversities, heterogeneous multi-core architectures outperform homogeneous platforms in terms of performance. Lakshminarayana et al. [12] proposed an age-based scheduling policy, which schedules a task with a longer remaining execution time to a faster core. Shelepov et al. [16] collected the caching behaviors of applications via offline profiling, and predicted the performance of different threads on different cores based on a thread's caching behavior and a core's cache size and frequency. The OS scheduler assigned the threads to the cores based on the predicted performance. Srinivasan et al. [18] use the platform performance counters along with a performance prediction model to predict an application's execution time on different types of cores. With this information, the OS scheduler can schedule the applications to the suitable cores to improve system performance. Becchi and Crowley [4] assigned threads to cores based on the instructions-per-cycle (IPC) ratio to maximize the overall IPC on heterogeneous multiprocessor architectures.

The aforementioned work focuses on maximizing system performance such as the overall IPC and the overall performance gain. In this paper we focus on energy-efficiency scheduling, since the introduction of the heterogeneous platform is motivated by its potential energy efficiency. Kumar et al. [11] proposed single-ISA heterogeneous multi-core architectures to reduce power consumption. In order to optimize the energy-delay product, sampling-based dynamic switching heuristics are proposed to allow heterogeneous multi-core architectures to adapt to differences between applications or phases in the same application. Chen and John [5] project the core's configuration and the program's resource demands to a multi-dimensional space, and schedule programs to cores based on the weighted Euclidean distance between the core's configurations and the program's resource demands. The proposed approach in [5] statically maps programs to cores based on a program's resource demands for the entire execution without considering program phases.

However, less attention has been given to phase-based scheduling in previous work. Sondag and Rajan [17] identified the program phases through compiler analysis, and dynamically determined which type of core is most appropriate for each program phase by monitoring IPC at runtime. However, they identify program phases by clustering the basic blocks with similar properties, such as instruction types. Their approach considers the program phases at the granularity of basic block level, which might not capture program phases well without considering the program structure at the granularity of function calls and loops. Sawalha et al. [15] identified the program phases by tracking the program counters with a history table, which requires additional hardware support.

A previous temporal approach [11] sampled each type of core periodically and selected one type of core to schedule the program based on the measurements during sampling. The thread migration happens when one other core is sampled; sampling on both cores incurs certain overhead on both performance and energy. It is crucial to minimize the number of samplings to reduce the switching overhead and the energy consumed during samplings. Instead of periodic sampling, our approach explores the program phases based on program structures and proposes a phase-based sampling to guide scheduling to minimize the energy delay product. In contrast to prior research based on simulation, we evaluate our scheduling scheme using Intel's QuickIA heterogeneous prototype platform. The main contributions of our proposed energy-efficient scheduling method are as follows:

1. A regression model is developed to estimate the energy consumption on Intel's QuickIA heterogeneous prototype platform.

2. An energy-efficient scheduling approach is proposed to map the program to the most appropriate core based on program phases using a combination of static analysis and runtime scheduling.

The remainder of this paper is organized as follows. In Section 2 we characterize the Intel heterogeneous prototype platform in terms of microarchitectures and energy efficiency. Section 3 shows our regression model for energy consumption. Section 4 describes our energy-efficient scheduling approach. We show our evaluation methodology and experimental results in Section 5. Finally, we conclude this paper in Section 6.

2. HETEROGENEOUS MULTI-CORE ARCHITECTURES

2.1 Heterogeneous Prototype Platform

We conduct our experiments on the Intel QuickIA platform [6]. The heterogeneous platform consists of one Intel Atom Processor 330 and one Intel Xeon Processor E5450, as shown in Figure 1. This kind of experimental platform is considered to be a perfect heterogeneous system for evaluation [6]. The Intel Atom processor and Intel Xeon processor are representatives of two opposite types of microarchitecture. The Intel Xeon processor employs the high-performance server-class microarchitecture, while the Intel Atom processor employs the low-power microarchitecture targeted for mobile devices. We focused on these two types of microarchitectures because they capture most of the performance/power benefits from asymmetry [6].

2.2 Energy Efficiency

The heterogeneous cores lead to a variety of performance and energy consumptions. Here we propose the energy-efficient scheduling on heterogeneous multi-core architectures. The metric that we

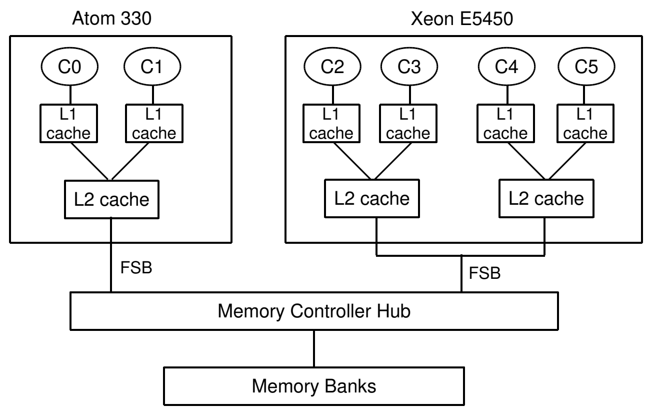


Figure 1: The configuration of the Intel QuickIA heterogeneous prototype platform

use to characterize energy efficiency is the energy delay product (EDP). Figure 2 shows the energy delay product over instruction intervals for the SPEC benchmark *473.astar*. Each instruction interval contains 50 million instructions.

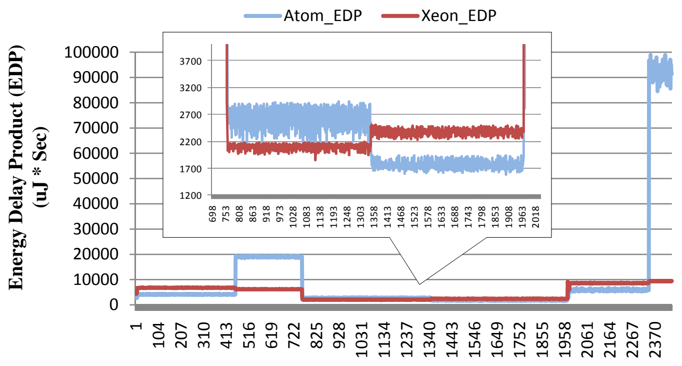


Figure 2: EDP on the Xeon processor and Atom processor vs. instruction intervals for the SPEC CPU2006 Benchmark *473.astar*

We can see that the program has different energy delay products on the Xeon core and Atom core in different program phases. The heterogeneous multi-core architectures have the potential to reduce the energy delay product by adaptively mapping the program to the most appropriate core. In order to make the correct scheduling decision, we need to know about the energy delay products on the Xeon processor and Atom processor. That means we need to obtain the execution time and energy consumption at runtime. We can obtain the execution time by using a system API such as *gettimeofday*. It is difficult to measure the energy consumption in real-time, although we can use thermal design power (TDP) to approximate the full-load energy consumption, CPU might not always work in the full-load situation during the entire execution. Therefore, we build a regression model to predict energy consumption at runtime.

3. REGRESSION MODELING FOR ENERGY

Regression modeling has been proposed to predict performance and power [13]. Here we use regression modeling to predict the energy consumption. To derive the regression model, we need to develop the training data to train the model. To generate the training data, we randomly select four benchmarks (*astar, bzip2, h264ref and hmmer*) from SPEC CPU2006 suite. For each benchmark, we collect fifteen pieces of hardware performance data (as shown in Table 1) on the Xeon core and Atom core for each instruction interval ranging from 50 million instructions to 25 billion instructions in 50-million instruction increments, 500 intervals in total. The hardware performance data is collected via hardware performance counters by using the Linux Perfctr library. Then, we feed this collected hardware performance data into McPAT [14] to calculate the energy consumption for each instruction interval. By using this approach, we can obtain 2000 training data samples.

Table 1: Collected hardware performance data

Category	Hardware performance data	
Performance	Cycles	
Instructions	Retired instructions	Floating point instructions
	Load instructions	Store instructions
Cache	L1 I-cache access	L1 I-cache miss
	L1 D-cache access	L1 D-cache miss
	L2 cache access	L2 cache miss
TLB	I-TLB miss	D-TLB miss
Branches	Branch instructions	Branch misprediction rate

With this training data, we build and evaluate our regression model by using statistical package R [2]. Given the fifteen hardware performance parameters shown in Table 1, we use variable clustering and correlation analysis [13] to identify the key hardware performance parameters that are most related to the energy consumption. The four selected key hardware performance parameters include cycles, the number of retired instructions, L1 D-cache access and L2 cache access. After identifying the four key hardware performance parameters, we specify a linear regression model as shown in equation (1) to predict the energy consumption with these four selected parameters. $\beta_1, \ldots, \beta_4$ denotes the corresponding regression coefficients, which can be interpreted as the expected change in energy per unit change in cycles, retired instructions, L1 D-cache access, or L2 cache access. The reason for using the linear regression model is that the static energy consumption is proportional to the execution time, and the dynamic energy consumption is proportional to the number of retired instructions, L1 D-cache access and L2 cache access.

$$\begin{aligned} Energy &= \beta_0 + \beta_1 \times Cycles + \beta_2 \times RetiredInsts \\ &\quad + \beta_3 \times L1DCacheAccess \\ &\quad + \beta_4 \times L2CacheAccess \end{aligned} \tag{1}$$

Based on the training data, we use the ordinary least-squares method to determine the regression coefficients. We derive the regression model for the Xeon processor and Atom processor respectively. Our regression model with four parameters that include cycles (C), the number of retired instructions (I), L1 D-cache access (L1DCA), and L2 cache access (L2CA) achieves an average error of 7.6% across all training data samples. We also tried to derive the linear regression model with three parameters. For the Atom processor, eliminating L2 cache access from the regression model increased the average error to 200.2%, and eliminating L1 D-cache access from the regression model increased the average error to 68%. The comparisons are shown in Figure 3. So, we concluded that three variables are not sufficient for the regression model.

In order to evaluate our regression model, we use the derived regression model to predict the energy consumption of *lbm* (note that *lbm* is not used to generate the training data). We collect fifteen pieces of hardware performance data and calculate the energy consumption using McPAT to generate the observed energy consumption. We use our derived regression model with four parameters (C+I+L1DCA+L2CA) to generate the predicted energy consumption. Figure 4 shows the comparisons between the observed and predicted energy consumption of *lbm* over instruction intervals on both the Xeon processor and Atom processor. Each instruction interval contains 50 million instructions. The average error on the Xeon processor is 2.6%, while the average error on the Atom processor is 2.25%. These evaluations show that our regression model with four parameters (C+I+L1DCA+L2CA) can accurately predict energy consumption.

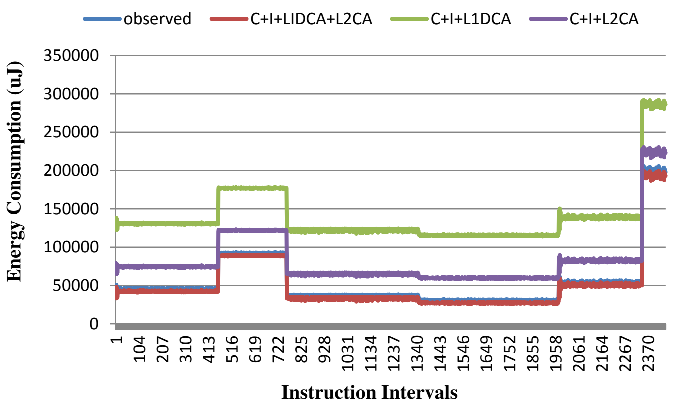
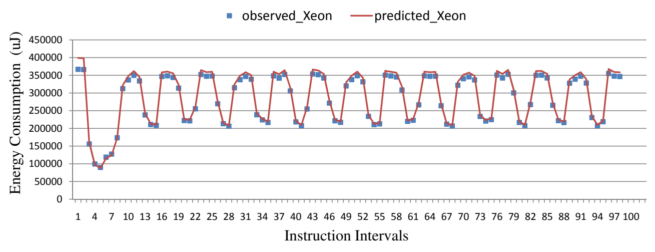
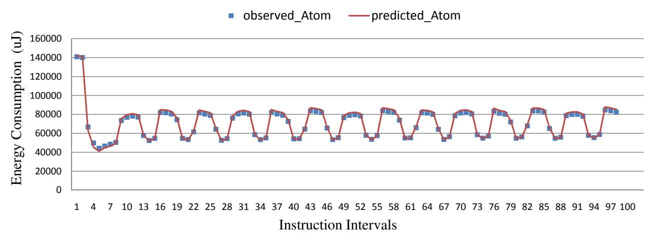


Figure 3: Comparisons between the observed and predicted energy consumption with training data on the Atom processor



(a)



(b)

Figure 4: Comparisons between the observed and predicted energy consumption of *lbm* on both the Xeon processor (a) and Atom processor (b)

4. ENERGY-EFFICIENT SCHEDULING

Figure 5 shows *astar*'s EDP on the Atom processor and most executed function/loop ID over instruction intervals. Each instruction interval contains 50 million instructions. A unique function or loop

ID is assigned to each function and loop at the stage of static instrumentation (to be explained in Section 4.1). We can see that the program has a similar energy delay product across different invocations of the same functions and loops. Based on this observation, we identify the program phases based on the program structures

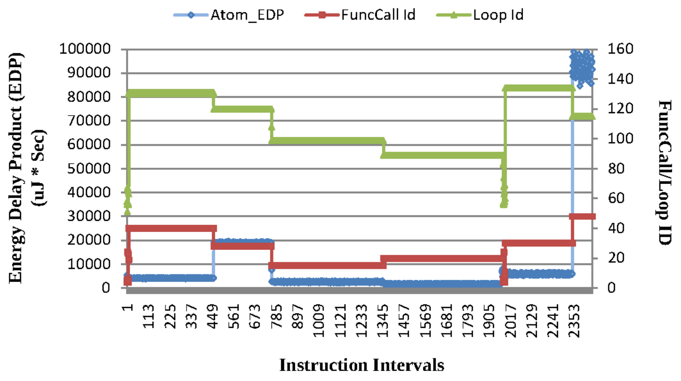


Figure 5: EDP and function/loop IDs vs. instruction intervals for SPEC CPU2006 Benchmark ***473.astar***

Our scheduling method consists of four steps. Step one identifies the boundaries of the function calls and loops through static analysis. Step two profiles the program to construct the call graph that captures the caller-callee relationship among function calls and loops. Step three identifies the major program phases and instruments the selected function calls and loops. Step four estimates the energy consumption with the regression model built in Section 3 and predicts the energy delay product to make scheduling decisions.

4.1 Static Analysis and Instrumentation

We implement the static instrumentation in the LLVM compiler infrastructure [1]. LLVM takes the program source codes as inputs to generate LLVM intermediate representations. The LLVM intermediate representation (IR) is a static single-assignment (SSA)-based representation that essentially models a RISC processor with infinite registers. Starting from the LLVM IR, we identify the function calls and their returns through 'call' instruction and 'ret' instruction. We identify the loops and loop boundaries based on the LLVM loop analysis passes. During the static instrumentation stage, we insert calls to instrumentation functions at the boundaries of functions and loops. These instrumentation functions are used to calculate invocation times and the total number of instructions for each function call and loop, capture the caller and call sites, and track the caller-callee relationships among functions and loops. The reason for using static analysis and instrumentation is that it is difficult to identify the function boundaries and loop boundaries by using dynamic binary instrumentation tools such as Intel Pin tools.

4.2 Creating Call Graph

Our call graph is based on the calling context tree of Ammons et al. [3]. Each node of the call graph is a function call or loop. We label each node with the name of the function or loop. The path from the root of the tree to a node captures the callers and call sites. To differentiate the invocations of a function or loop in different paths, we extend the node name with the path ID. If a function is called from the inside of a loop, it will be considered a single node in the call graph, even if it might have different program behavior among different invocations. We profile the instrumented binary to count the number of invocations and the total number of executed instructions for each function call and loop. We annotate the node with the total number of instructions executed and annotate the edge with the number of invocations by callers.

Here we use the sample code shown in Figure 6 to illustrate how to create the call graph. Assume the number of instructions executed in *func2* is 1 million. The corresponding call graph of the sample code is shown in Figure 7. In the call graph, *main* has two children *func1*. Each *func1* has its own *Loop1*, *Loop2*, *Loop3*, and *func2*. Since *func2_1* is called 1000 times in path 1, *func1_1*, *Loop1_1*, *Loop2_1*, *Loop3_1*, and *func2_1* each have 1 billion instructions executed. In this way, we also can infer the average number of executed instructions in each invocation for other functions and loops, as shown in Figure 7.

```
void func1(float * a, int m, int n, int p)
{
Loop1:  for (i=0; i<m; i++)
Loop2:    for (j=0; j<n; j++)
Loop3:      for (k=0; k<p; k++)
              a[i] += func2(i+j+k);
}
int main()
{
  func1(A, 10, 10, 10);
  func1(B, 100, 100, 100);
}
```

Figure 6: Sample code

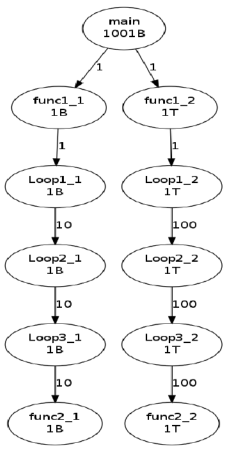


Figure 7: Call graph of the sample code

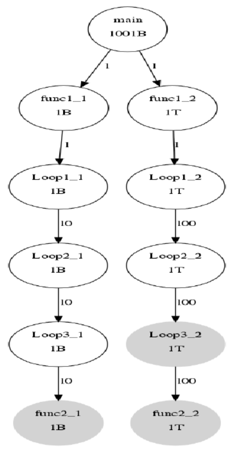


Figure 8: Major program phases on the call graph

4.3 Identifying Major Program Phases

The created call graph is used to identify the major program phases that run long enough. For function calls and loops to qualify as major program phases, the total number of instructions executed has to be larger than or equal to th_{inst}, and the number of invocations has to be larger than or equal to th_{invoke}. The reason for th_{inst} is that the thread migration will incur a certain overhead on performance and energy consumption. In order to obtain the benefits from scheduling, the function calls and loops need to have long execution time. The reason for th_{invoke} is that our sampling-based approach makes decisions based on the measurements during the first several invocations. In order to obtain the benefits from scheduling, the function calls and loops need to be invoked many times. The minor function calls and loops will be included as part of the closest major function calls and loops in the call graph. We insert calls to instrumentation functions at entry and exit points in the identified major function calls and loops by using the LLVM compiler infrastructure. These instrumentation functions are used to sample the hardware performance data, predict the energy delay products, and make the scheduling decisions.

Here we use the call graph shown in Figure 8 to illustrate how to identify the major program phases. Assume th_{inst} is equal to 1 billion and th_{invoke} is equal to 1000. We can obtain the number of invocations of a node by cumulative multiplication of all the edge values on the path from the root of the tree to the node. In Figure 8, the total number of instructions executed for every node is larger than or equal to th_{inst}. However, only node *func2_1*, node *Loop3_2*, and node *func2_2* are called more than or equal to th_{invoke}. So, *func2_1*, *Loop3_2*, and *func2_2* are identified as major function calls and loops. Since *func2_2* is called from the inside of *Loop3_2*, we can instrument either *func2_2* or *Loop3_2*.

4.4 Runtime Scheduling

The scheduling decision is made at runtime using the instrumented codes. In order to make the correct scheduling decision, we need to predict the energy delay products on the Xeon processor and Atom processor. That means we need to obtain the execution time and energy consumption at runtime. The regression model in Section 3 needs four key hardware performance parameters to accurately predict the energy consumption. But, there are only two hardware performance counters on the Xeon processor and on the Atom processor. During each invocation, we can only collect two pieces of hardware performance data by using the Linux Perfctr library. Therefore, we need two invocation to estimate the energy consumptions. The number of cycles is one parameter of the regression model; from this we can obtain the execution time. Then, we can multiply the estimated energy consumption by execution time to estimate the energy delay product.

Our heterogeneous prototype platform has two types of cores: the Xeon processor and Atom processor. In the first two invocations of each identified major function or loop, we run the invocations on the Xeon processor and estimate the energy delay product by using the above method. In the next two invocations, we run the invocations on the Atom processor and estimate the energy delay product. When an identified major function or loop finishes these runs, we select the core that has a lower energy delay product to execute the remaining invocations of this function or loop. The core switching is done using the Linux process affinity API.

4.5 Complexity and Scalability

The complexity of our scheduling scheme is $\mathcal{O}(P \times N)$, where P denotes the number of major program phases detected in the program, and N denotes the number of different types of processors on a chip. As the number of cores increases, scalability might become an issue for the sampling-based approach. In the future, the number of different types of processors on a chip might be much more than two. In that case, we need to do hierarchically sampling to reduce the sampling overhead. We can cluster the processor types into several categories based on the processors' characteristics, with each category having one representative processor type. Based on these representative processors, our scheduling scheme could determine which category the application or the program phase will be mapped into. Among the processors in that category, we can apply our scheduling method to determine onto which processor type the application or the program phases will be mapped.

5. EVALUATION

5.1 Evaluation Methodology

In the evaluation we use seven benchmarks (*astar, bzip2, h264ref, hmmer, lbm, libquantum and named*) from the SPEC CPU2006 suite and four benchmarks (*denoise, deblure, registration (reg) and segmentation (seg)*) from the medical imaging domain [7]. We evaluate our scheduling scheme on the Intel QuickIA heterogeneous platform [6]. To demonstrate the advantages of our scheduling scheme, we implement the following scheduling schemes for comparisons.

*The statical mapping approach **(SMap)***: This scheduling scheme proposed in [5] statically maps the programs to the cores, and does not switch the program among heterogeneous cores during program execution. We measure a set of program's inherent characteristics including instruction-level parallelism (ILP), branch predictability, and data locality by using the Pin tool, MICA [9]. Then, we use the projection functions [5] to identify the desired resource demands including issue width, branch predictor size, and data cache size. In the Intel QuickIA heterogeneous platform, two types of hardware resource are provided: one is the Intel Atom processor, the other is the Intel Xeon processor. We calculate the weighted Euclidean distances between the program's resource demands and the hardware resources of these two types of processors, respectively. The processor type with the smaller distance will be chosen as the preferred processor onto which the application will be mapped.

*The periodic sampling approach **(PS)***: This scheduling scheme proposed in [11] samples one or more cores for five intervals every 100 intervals. Each instruction interval contains 1 million instructions, and their scheduling scheme is implemented in the simulator. However, our evaluation is done on a heterogeneous prototype platform. Therefore, we implemented their method by using prior knowledge of the program execution. We first collect all hardware performance data (as shown in Table 1) using the Linux Perfctr library for each instruction interval. Then, we feed these hardware performance data into McPAT to calculate the energy consumption, and multiply the energy consumption by the execution time to obtain the energy delay product for each instruction interval. With this prior knowledge, we can use the scheduling method proposed in [11] to make the scheduling decision for each instruction interval. Given the scheduling decisions, we calculate the energy delay product by multiplying the total energy consumption with the total execution time.

*The phase-based sampling approach **(PhaseSamp)***: This is our scheduling scheme. Our approach first statically analyzes the program, detects the major program phases based on LLVM infrastructure, and inserts the instrumented codes at the boundaries of the program phases. We run the instrumented program on the Intel QuickIA prototype platform. Based on the instrumented codes, the program will dynamically select the most appropriate core in terms of energy delay product during the runtime.

5.2 Energy Delay Product Comparison

The comparisons of the energy delay product in our approach to the energy delay products in statical mapping and periodical sampling with different instruction intervals are shown in Figure 9. The results are normalized to our approach (PhaseSamp). In [11], one interval consists of 1 million instructions. In our evaluations, we also evaluated the impact of varying interval length on the scheduling scheme proposed in [11]. The interval lengths we evaluated contain 1 million instructions, 10 million instructions, and 50 million instructions.

Compared to the statical mapping approach, our approach can

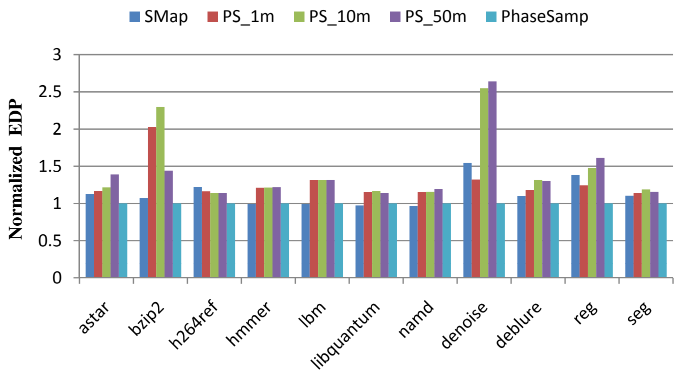


Figure 9: Comparison of the energy delay product in our approach to the energy delay products in statical mapping and periodical sampling with different instruction intervals

Table 2: Comparison of the energy delay product and the number of core switchings over benchmark sets

Benchmarks	PS_1m		PhaseSamp (Our Approach)	
	EDP (J * sec)	Total switches	EDP (J * sec)	Total switches
astar	27156.47	2452 (204X)	23346.32 (14.03%)	12
bzip2	489.13	468 (59X)	241.49 (50.63%)	8
h264ref	107048.22	9834 (307X)	92117.31 (13.95%)	32
hmmer	34060.29	5712 (238X)	28078.4 (17.56%)	24
lbm	65041.56	2062 (257X)	49550.21 (23.82%)	8
libquantum	90.43	246 (41X)	78.16 (13.57%)	6
namd	2988.97	1254 (313X)	2592.02 (13.28%)	4
denoise	1485.98	380 (47X)	1123.92 (24.37%)	8
deblure	1916.02	620 (78X)	1627.09 (15.08%)	8
reg	7716.94	1502 (57X)	6208.08 (19.55%)	26
seg	8642.92	1718 (286X)	7595.35 (12.12%)	6
average		171X	19.81%	

achieve an average 10.20% reduction up to 35.25% reduction in the energy delay product. We observe that the statical mapping approach works better than our approach for the benchmarks: *hmmer, lbm, libquantum, and namd*. We take a close look at these four benchmarks, and we find that they do not have obvious program phase changes in terms of energy-delay product. These four benchmarks all have a lower energy delay product on a specific processor (either the Xeon processor or the Atom processor) during the entire execution. The core switchings of our approach bring a certain overhead into the energy delay product. For other benchmarks, our approach outperforms the statical mapping approach. The other benchmarks have clear program phase changes in terms of energy-delay product. Some program phases have lower energy delay products on the Xeon processor, while other program phases have lower energy delay products on the Atom processor. And our approach can capture the program phases much better.

Compared to the periodical sampling approach, our approach can achieve an average 19.81% reduction in energy delay product when the interval length is 1 million instructions; it can achieve an average 26.25% reduction when the interval length is 10 million instructions; it can achieve an average 25.31% reduction when the interval length is 50 million instructions. The reduction in energy delay product comes from the improved capturing of program phases and the fewer number of switches needed (as shown in Table 2). The core switching will incur a certain overhead on performance and energy consumption. We measure the cost of core switching on the real platform, and the overhead of core switching on performance is around 20 μs per switching; the overhead of core switching on energy consumption is around 40 μJ per switching.

6. CONCLUSION

In this paper we propose an energy-efficient scheduling method for heterogeneous multi-core architectures. We develop a regression model to estimate the energy consumption. Our scheduling approach maps the program to the most appropriate core based on program phases using a combination of static analysis and runtime scheduling. We demonstrate the efficiency of our scheduling approach on the Intel QuickIA heterogeneous prototype platform [6]. Our approach achieves an average 10.20% reduction in energy delay product over the static mapping approach proposed in [5] and an average 19.81% reduction in energy delay product over the periodic-sampling approach proposed in [11].

7. ACKNOWLEDGMENTS

This research is partially supported by SRC contract 2009-TJ-1984, NSF grant CCF-0903541, and the NSF Expedition in Computing grant CCF-0926127.

We greatly appreciate research gifts and equipment donations from Intel, and helpful discussions with Ganapati Srinivasa, Ravishankar Iyer, and Mishali Naik from Intel concerning the Intel QuickIA system.

8. REFERENCES

[1] The llvm compiler infrastructure. `http://llvm.org/`.
[2] Statistical package R. `http://www.r-project.org/`.
[3] G. Ammons et al. Exploiting hardware performance counters with flow and context sensitive profiling. PLDI '97, pages 85–96.
[4] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. CF '06, pages 29–40.
[5] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. DAC '09, pages 927–930.
[6] N. Chitlur et al. QuickIA: Exploring heterogeneous architectures on real prototypes. HPCA '12, pages 1–8.
[7] J. Cong et al. Customizable domain-specific computing. *IEEE Design Test of Computers*, 28(2):6–15, 2011.
[8] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33 –38, 2008.
[9] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63 –72, 2007.
[10] R. Kumar et al. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. ISCA '04, pages 64–75.
[11] R. Kumar et al. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. MICRO '03, pages 81 – 92.
[12] N. B. Lakshminarayana et al. Age based scheduling for asymmetric multiprocessors. SC '09, pages 25:1–25:12.
[13] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. ASPLOS '06, pages 185–194.
[14] S. Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. MICRO '09, pages 469–480.
[15] L. Sawalha et al. Phase-guided scheduling on single-ISA heterogeneous multicore processors. pages 736 –745, DSD '11.
[16] D. Shelepov et al. HASS: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43:66–75, 2009.
[17] T. Sondag and H. Rajan. Phase-based tuning for better utilization of performance-asymmetric multicore processors. CGO '11, pages 11–20.
[18] S. Srinivasan et al. Heteroscouts: hardware assist for os scheduling in heterogeneous cmps. *SIGMETRICS Perform. Eval. Rev.*, 39:341–342, 2011.