Energy Characterization and Instruction-Level Energy Model of Intel's Xeon Phi Processor

Yakun Sophia Shao and David Brooks
Harvard University
{shao,dbrooks}@eecs.harvard.edu

***Abstract*—Intel's Xeon Phi is the first commercial many-core/multi-thread x86-based processor. Xeon Phi belongs to a new breed of high performance computing processors that seek high compute density as well as energy efficiency. However, no high-level energy model is available for Xeon Phi software developers to quickly evaluate and optimize energy efficiency. This work demonstrates an instruction-level energy model for the Xeon Phi processor to facilitate the development of energy-efficient software. In order to construct this model, we first characterize the energy consumption of the processor, identifying how energy per instruction scales with the number of cores, the number of active threads per core, and instruction types. Based on the energy characterization, we construct an instruction-level energy model and validate the accuracy of the model between 1% and 5% for real world benchmarks. We show that the energy model can be used to identify software inefficiencies for these benchmarks and find that *Linpack* code can be optimized to increase energy efficiency by as much as 10%.**

***Index Terms*—Xeon Phi, Energy Characterization, Instruction-Level Energy Model.**

I. Introduction

Current processors increasingly exploit thread-level parallelism (TLP) to improve performance. As a result, multi-core/multi-thread processors are becoming the dominant architectures for domains ranging from mobile platforms to high-performance computing. As technology scales, the number of transistors available will continue to grow every generation. To make full use of those transistors and further exploit the potential of TLP, architects will design chips with larger core counts. While software developers have been focusing on the use of many-core/multi-thread processor to boost throughput, performance per watt is crucial. As such, it is important to study the energy related characteristics of many-core/multi-thread processors to facilitate energy-efficient code design.

Xeon Phi, also known as Knights Corner (KNC), is the first product using Intel's Many Integrated Core (MIC) architecture, which is designed for high-performance computing (HPC) systems. The processor consists of 60 in-order x86-based cores, each of which is able to run at most 4 threads. By packing such a large number of hardware threads into a single processor, Xeon Phi provides much higher compute density than current multi-core processors. Xeon Phi also demonstrates impressive energy efficiency. A Xeon Phi based system tops the Green500 list as the world's most energy efficient supercomputer as of November 2012 [1].

To provide opportunities for software developers to optimize workloads towards energy efficiency, we develop an instruction-level energy model of the Xeon Phi processor based on detailed energy characterization. An instruction-level energy model links energy consumption directly to software code, which is intuitive for software developers to understand energy cost. In addtion, this model only needs performance counter statistics as input, which existing software profiling tools already provide.

This paper makes the following contributions:

1) We characterize the energy per instruction (EPI) of Xeon Phi using a set of specialized microbenchmarks exercising different categories of instructions with varying memory behavior, number of active cores, and number of active threads per core.
2) We build an instruction-level energy model for Xeon Phi using characterized EPIs along performance counter statistics to capture workload activity. The model accurately predicts dynamic energy consumption with an average error rate under 5%. To the best of our knowledge, this is the first instruction-level energy model for a many-core/multi-thread x86 processor.
3) This model provides software developers opportunities to improve energy efficiency. In particular, our model identifies that more than 10% of energy consumption is due to redundant software prefetch operations for a performance-tuned *Linpack* implementation.

II. Energy Modeling Taxonomy

We present a taxonomy of dynamic energy modeling approaches based on how to model processor intrinsic energy characteristics and how to capture runtime activity factors, shown in Table 1. This section explains each taxonomy category and where our instruction-level model for Xeon Phi fits in.

A. Intrinsic Energy Characteristics

There are two ways to model intrinsic energy characteristics of processor: architecture- and instruction-level. Architecture-level modeling requires capacitance information of major architectural blocks to compute per access energy. Instruction-level modeling uses characterized EPI to model processor energy.

Although architecture-level modeling provides detailed energy breakdown for each architectural block, which is useful for microarchitecture-level exploration, it is often difficult to obtain low-level capacitance information. At the same time, the microarchitecture-level analysis provides little intuition for software developers.

978-1-4799-1235-3/13/$31.00 ©2013 IEEE

Table 1: Taxonomy of energy modeling approaches.

		Intrinsic Energy Characteristics	
		Architecture-Level	Instruction-Level
Activity Factor	Simulator/ Profiler	Wattch [2]/McPAT [7] (Out-of-Order)	Tiwari [8] (Embedded)
	Performance Counters	Isci [5] (Out-of-Order)	**This work** (Xeon Phi)
	Analytical Perf. Model	Karkhanis [6] (Out-of-Order)	Hong [4] (GPU)

On the other hand, instruction-level models provide more insights for software developers to find opportunities to optimize their code for energy efficiency. Traditionally these models work well for in-order core designs [8], which tends to be the dominant microarchitecture for current many-core/multi-thread processors, such as Intel's Xeon Phi and GPUs [4].

B. Activity Factor

Activity factors indicate how frequent an architectural block is accessed for architecture-level models or how many dynamic instructions are executed in each category of instructions for instruction-level models. The methods to collect activity factors fall into three categories:

1) Simulation or Profiling: The first approach obtains the activity factors through detailed simulation or profiling. Wattch [2] and McPAT [7] are examples of architecture-level energy models using simulators to collect per block access counts. Tiwari et al. use profiling to provide instruction breakdown for their instruction-level energy model of embedded systems [8]. Although simulation or profiling provides the most detailed information for accurate activity factors, it is relatively slow to simulate or profile workloads, making it unattractive for software developers.

2) Performance Counters: Performance counters can profile microarchitectural block activity and instruction breakdown. Isci et al. use performance counters to estimate activity factors for Pentium 4's architecture-level energy model [5].

3) Analytical Performance Model: Analytical performance model can quickly predict workload behaviors without the need for real hardware or detailed simulators. Previous work has demonstrated this approach for out-of-order CPUs [6] and GPUs [4].

C. Our Energy Model

We construct an energy model for Xeon Phi using EPI to model processor intrinsic energy characteristics and performance counters to profile activity factors. Our work is the first to construct an instruction-level energy model using performance counters on a x86-based many-core/multi-thread processor. The foundations for our model are characterized EPIs of representative instruction types with different number of cores and threads per core configuration. This energy model can be attached to a range of runtime performance counter tools and analytical models. In this work we have integrated our model with Intel's VTune performance profiling tool to predict workload dynamic energy.

Figure 1 illustrates the overall structure of our instruction-level energy model and the interface between workloads and the energy model. We develop microbenchmarks for each instruction type running with different number of cores and threads per core configurations to characterize EPI. Instruction is categorized based on its opcode type and operands location, since EPI heavily depends on where operands reside. Runtime performance counter statistics compute the breakdown of instruction and operand source combinations. In the end, we multiply the runtime instruction counts with the coresponding EPI to compute the total energy of the workload. We have validated our model against measurement and show software developers can use this model to identify energy inefficiencies of their code in Sec. V

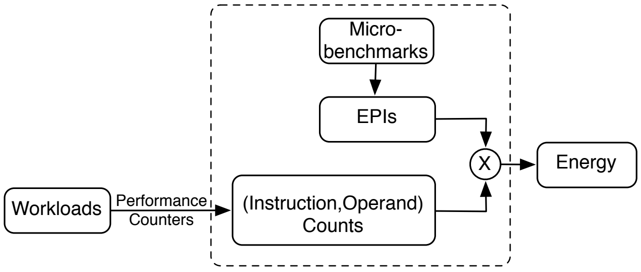

Fig. 1: Instruction-level energy model.

III. Methodology

This section discusses the background of the Xeon Phi processor, the measurement setup, microbenchmarks developed, and power and timing statistics collected for microbenchmark characterization.

A. Xeon Phi Processor

Xeon Phi is built in a 22 nm process and contains 60 cores running at 1.09 GHz, where each core can run 4 threads at the same time. Each core is in-order with a 512-bit vector-processing unit, a 32 KB L1 I-cache and D-cache and a 512 KB private L2 cache. Cores are connected together via a ring bus and follow the standard MESI coherency protocol for maintaining the shared state among cores.

B. Measurement Setup

We have instrumented a Xeon Phi card, which is part of the Xeon Phi Beta Software Development Platform (SDP), for power measurement. We measure voltages and currents inside the board to compute dynamic power. The power not only includes the power of the Xeon Phi processor, but also other components like memory and fan. A National Instruments Data Acquisition system collects the statistics sampled at 1 KHz, which is sufficient for our purpose.

C. Microbenchmarks developed

Each microbenchmark is a loop that iterates a target instruction type. We cover all major instruction types, as shown in Table 2. Each column presents different instruction opcodes, including scalar, vector, `vprefetch0` (prefetch to L1 cache) and `vprefetch1` (prefetch to L2 cache). Each row shows the mode of data operand access, including register, L1, L2, prefetched from memory (both hardware and software prefetch) and memory without prefetch. Microbenchmarks are deployed as one copy per hardware thread with different number of cores and threads per core configuration. We sweep

Table 2: EPIs (nJs) of instruction types and modes of data operand access for single-core, single-thread.

	Scalar Op	Vector Op	vprefetch0 (to L1)	vprefetch1 (to L2)
Register	0.45	1.00	N/A	N/A
L1	0.88	1.43	1.19	1.19
L2	7.72	8.27	1.81	1.19
Mem w/ Prefetch	52.14	52.69	50.00	25.00
Mem w/o Prefetch	232.62	233.17	N/A	N/A
Write to Mem	62.14	62.69	N/A	N/A

the number of cores from 1 to 60 and the number of threads per core from 1 to 4.

D. Timing and Power Statistics Collection

To compute EPI, Figure 2 shows timing and power statistics we collect. *p0* is the average idle power before the microbenchmark starts, including power for fan, memory, operating system and leakage. Instruction RDTSC reads the cycle before the microbenchmark starts (*c0*) and right after the microbenchmark ends (*c1*). The difference between *c0* and *c1* is the total number of cycles executed by the microbenchmark. The measurement setup records the dynamic power sampled at 1 KHz as the microbenchmark runs. We use the average dynamic power (*p1*) subtracting the initial idle power (*p0*), the result of which is the power consumed by the microbenchmark. Equation 1 computes EPI for each instruction, where N is the total number of dynamic instructions in the microbenchmark.

$$EPI = \frac{(p1 - p0) * (c1 - c0)/Freq}{N} \tag{1}$$

IV. Xeon Phi Characterization

Energy of instructions mostly depends on instruction types, including both opcode types and operand locations in cache hierarchy, the number of active threads per core, and the number of active cores. Instruction type determines which functional units to stress, and different active thread-core configurations exhibit different resource contention, both of which impact the dynamic energy of a instruction. Since each core of Xeon Phi is a simple in-order core, the inter-instruction effect within a core is negligible, especially when running on many-core/multi-thread cases. In the following sections, we present EPI characterization results of Xeon Phi for different instruction types with different number of cores and threads per core configurations.

A. Single-Core, Single-Thread Characterization

Table 2 shows the EPIs for instructions with different instruction types and data operand access modes for a single thread running on one core. We observe several interesting characteristics. First, the EPI of vector instructions with register operands is about 2X compared to scalar instructions. However, because the VPU is 512-bit wide, fully-utilized vector instructions are actually 8X more energy efficient than scalar instructions. We find little EPI variation across different scalar or vector instruction subtypes with register operands. The second major finding is that the energy cost of data movement is significant compared to the energy of computation. Using vector instructions as an example, the EPI with register operands is 1.00 nJ while the EPI of moving data from memory to the ALU without prefetching is 233.17 nJ. However, the energy cost of data movement can be significantly reduced through intelligent prefetching. We see, for example, that moving data from L2 to L1 using vprefetch0 (1.81 nJ) followed by a vector compute instruction (1.43 nJ) is more energy efficient than a vector instruction generating a L1 miss and incurring a demand fetch from L2 (8.27 nJ). Prefetch instructions do not stall the pipeline as occurs for other instructions that generate cache misses, and the idle energy of the pipeline stalls significantly adds to the EPI of non-prefetch instructions. However, prefetch instructions must be used judiciously. For example, issuing a vprefetch1 when the data is already in L2 results a 1.19 nJ EPI overhead with no performance gain.

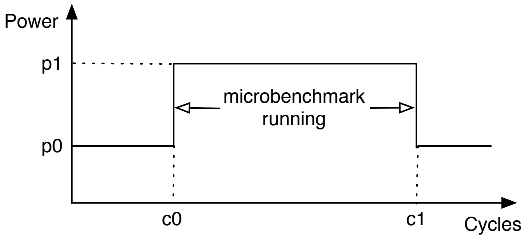

Fig. 2: Statistics collected for EPI characterization.

B. Single-Core, Multi-Thread Characterization

We also evaluate EPI by running microbenchmarks on a single core with one (1T), two (2T), and four (4T) active threads. Figure 3 plots EPIs for different instruction types while varying the number of threads per core. We observe that for both scalar and vector instructions, the EPI of the 1T configuration is 67% higher than the 2T case. This is mainly due to Xeon Phi's core microarchitecture that prohibits instruction issue from the same thread in back-to-back cycles. This means that running one thread per core only utilizes half of the core throughput. However, due to non-ideal clock gating within the core, the power dissipation of

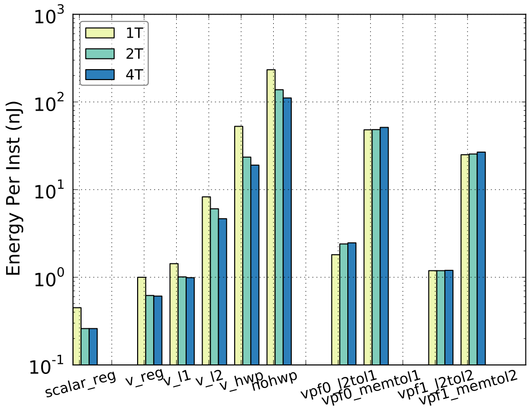

Fig. 3: EPI characterization for single-core, multi-thread.

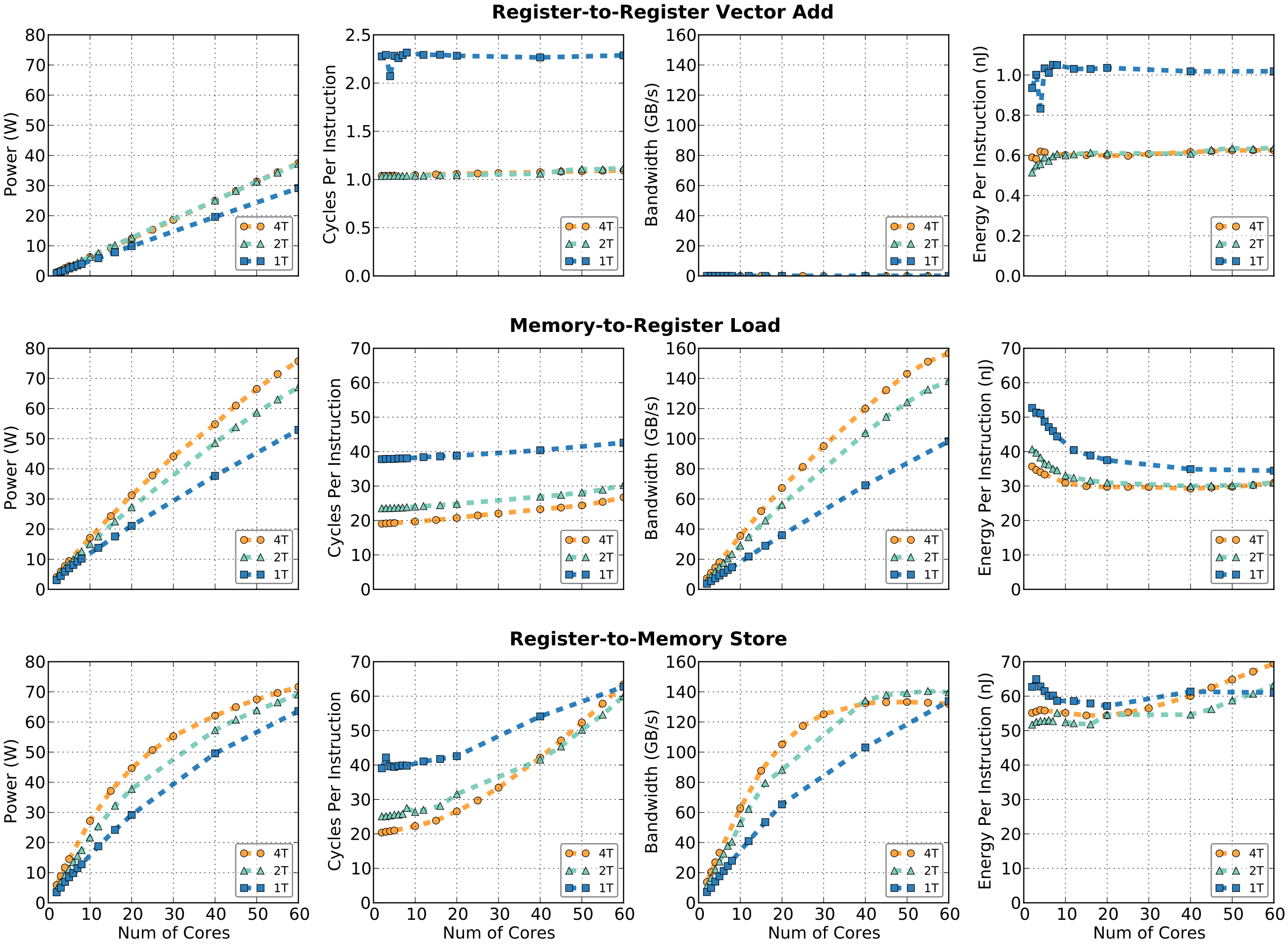

Fig. 4: Total power, per-core CPI, chip bandwidth, and per-core EPI sweeping the number of active cores and threads.

the core running a single thread remains at 83% of the fully-utilized core, explaining why the 1T configuration is energy inefficient. The second effect we observe is that for scalar and vector instructions with register operands or L1 hit cache access, the 2T configuration already fully utilizes the core and hence increasing to 4 threads does not change the EPI. For instructions with cache accesses that have longer delay due to L2 hits (v_l2), hardware prefetch (v_hwp), or L2 misses (nohwp), more threads allow overlap with memory access idle time and we observe a decrease in EPI. Finally, the EPI of prefetch instructions is largely independent of the number of threads per core because the throughput is mainly dependent on the number of entries in the prefetch buffer, which is independent of the thread configuration.

C. Multi-Core, Multi-Thread Characterization

As a many-core processor, Xeon Phi is commonly expected to utilize a large number of cores and active thread contexts. The energy characteristics of the processor are different with many active cores because of contention on shared, un-core resources such as the ring interconnect, memory controllers, and DRAMs. In this section, we study how EPI scales with the number of cores ranging from one to sixty. In order to study contention effect, we analyze three microbenchmarks with distinct properties. The first microbenchmark, *Register-to-Register Vector Add*, performs a simple vector arithmetic operation with register source and destination operands, incurring no cache misses. The second microbenchmark, *Memory-to-Register Load*, is designed to load a full cache line from memory into the local cache and deliver a vector of data into the register file. Finally, *Register-to-Memory Store*, fetches the cache line from memory to the local cache and writes the value of the vector register to the memory location. This microbenchmark requires an equal amount of read and write bandwidth to the memory system. Each row in Figure 4 presents results for the three microbenchmarks. We show total power consumption, per-core CPI, total chip memory bandwidth and per-core EPI.

Register-to-Register Vector Add: Starting from the chip power consumption on the upper left of Figure 4, we observe that for all threads configurations (1T, 2T, and 4T) the total chip power increases linearly with the number of cores, reflecting nearly ideal clock gating when all threads in the core are completely idle. We also notice that from 1T to 2T power increases by 1.2X but power does not change from 2T to 4T, as observed in our previous single-core multi-thread characterization. The CPI characterization plot also illustrates that 1T performance is more than 2X worse

Table 3: Performance counter equations to compute combinations of instruction types and operand source modes.

Instruction Types	Operand Sources	(Instruction, Operand) Counts
Scalar Op	Register	INSTRUCTIONS_EXECUTED − VPU_INSTRUCTIONS_EXECUTED
Vector Op	Register	VPU_INSTRUCTIONS_EXECUTED
Scalar & Vector Ops	L1	DATA_READ_OR_WRITE − DATA_READ_MISS_OR_WRITE_MISS
	L2	DATA_READ_MISS_OR_WRITE_MISS − (L2_DATA_READ_MISS_MEM_FILL + L2_DATA_READ_MISS_CACHE_FILL + L2_DATA_WRITE_MISS_MEM_FILL + L2_DATA_WRITE_MISS_CACHE_FILL)
	Mem w/ Hardware Prefetch	HWP_L2MISS
	Write to Mem	L2_WRITE_HIT
vprefetch0	L1	L1_DATA_PF1 − L1_DATA_PF1_MISS
vprefetch0	L2	L1_DATA_PF1_MISS
vprefetch1	L2	L2_DATA_PF2 − L2_DATA_PF2_MISS
vprefetch1	MEM	L2_DATA_PF2_MISS
All Ops	Core-to-Core	L2_DATA_READ_MISS_CACHE_FILL + L2_DATA_WRITE_MISS_CACHE_FILL

than 2T and 4T due to the pipeline bubble effect discussed previously. The third plot in the first row of the figure shows that bandwidth is zero, since vector instructions with register operands do not generate any memory activity. We see that EPI is constant while scaling the number of active cores since register operations are local to each core and independent of other core activity.

Memory-to-Register Load: The second row in Figure 4 characterizes the behavior of the memory read intensive microbenchmark. The total power more than doubles compared to the core-bound microbenchmark, reflecting the increased activity of the ring interconnect, the memory controllers, and DRAMs. Additional active threads per core lead to higher power consumption due to the additional memory requests initiated by those threads. We see that the slope of the power curve gradually decreases after about 10 active cores due to the amortization of the power consumption of the un-core resources. The CPI characterization demonstrates that scaling from 1 core to 60 cores slowly increases the CPI for all thread configurations due to increased effect of shared resource contention. Multiple threads per core (4T) exaggerate this effect but still offer CPI benefit due to overlapping memory requests. These issues are better understood by observing the bandwidth characterization results. For 1T there is a linear increase in bandwidth with number of active cores, but for the 4T case, the processor gradually becomes bandwidth starved beyond 40 cores, explaining the CPI results. The EPI calculation, shown in Equation 1, illustrates a proportional relationship to both power and CPI. The initial steep drop in EPI is due to the decreased slope of power while increasing the number of cores. After about 20 active cores EPI is relatively flat with increased active core counts due to the counteracting trends for power and CPI.

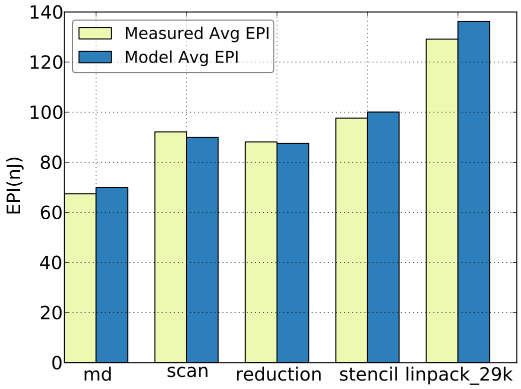

Fig. 5: EPI validation of energy model.

Register-to-Memory Store: The third microbenchmark combines the memory read characteristic of the previous microbenchmark with an equal amount of write bandwidth. Starting from the power and CPI characterization, we see an obvious decrease in the power slope and a significant increase in CPI, both of which can be explained by the bandwidth characterization. Bandwidth utilization increases very quickly from 1 to 20 cores and almost doubles the bandwidth requirement for the previous microbenchmark at this point. Bandwidth quickly saturates after 30 cores for the 2T and 4T configurations, and at 60 cores all thread configurations consume nearly the same bandwidth. Combing the effects of power and CPI scaling, the EPI characterization shows that a large number of active cores is detrimental and a minimum EPI exists for fewer active cores due to the bandwidth saturation of such a large number of memory requests.

V. Xeon Phi Energy Model

Using characterized EPIs from previous section as internal parameters, our model takes runtime performance counter statistics as input to predict energy consumption of workloads. This section describes dynamic performance counter statistics collected, model validation against measurement, and usecases for software developers to identify energy saving opportunities.

A. *Collecting Instruction Information*

We use performance counters to collect the instruction mix and operand source behavior of the workloads. These statistics are collected using standard performance counters with Intel's VTune tool. Table 3 shows how performance counters are used to compute the relevant combinations of instruction types and operand sources.

B. *Model Validation*

The energy model provides detailed instruction-level EPI breakdowns for complex, real-world applications using only performance counter instrumentation. We validate the energy model using the SHOC benchmark suite [3] and *Linpack* against energy measurement from our instrumented Xeon Phi card. All benchmarks are run using 60 cores and four threads per core. The results, shown in Figure 5, demonstrates error rates between 1% and 5%.

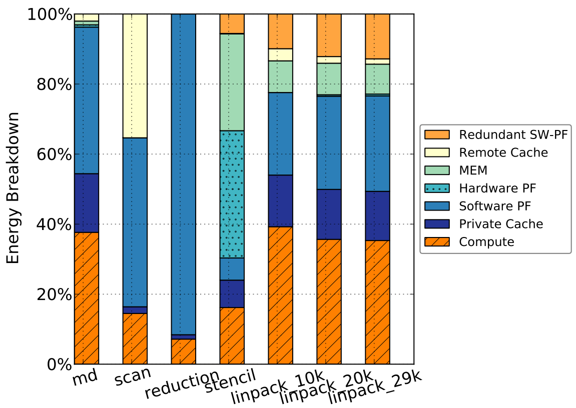

Fig. 6: Energy breakdown of the SHOC workloads and *Linpack*.

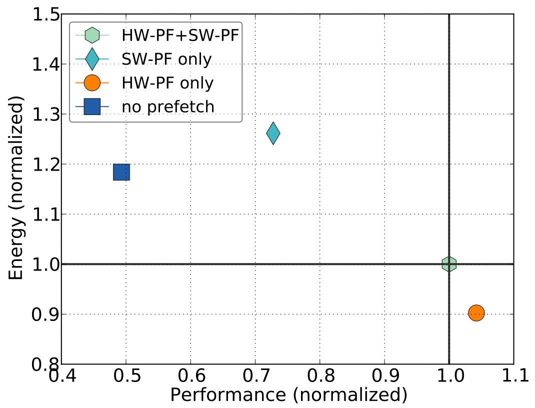

Fig. 7: Energy-delay tradeoffs for different prefetch configurations for the *stencil* benchmark.

C. Find Potential Opportunities for Energy Efficiency

The primary use of instruction-level energy model is to provide opportunities for software developers to optimize their code for energy efficiency. We identify two such opportunities for our benchmarks.

The first use case identifies opportunities to eliminate wasteful software prefetch operations. The energy breakdown in Figure 6 shows that for *Linpack*, across different input sizes, around 10% of the energy is spent on redundant software prefetch operations that fetch data already in the target cache. The reason for this inefficiency is because software developers insert prefetch instructions to fetch data early hoping to avoid memory stalls. Software developers tend to over provision prefetch operations because they generally do not hurt performance if the data is already in the target cache. However, redundant prefetch operations increase energy consumption because the hardware still needs to execute the prefetch instructions and compare tags to see whether the data is in the cache or not. Our model identifies that this inefficiency can be as high as 10% for complex workloads, providing opportunities for software developers to write more energy efficient code.

We also find that the instruction-level energy model can be used to find the best version of software implementation for a particular algorithm. To illustrate this situation, we implement four versions of the *stencil* algorithm with four different prefetch strategies. The results, shown in Figure 7, present the performance and energy consumption of the four versions of *stencil* normalized to the original implementation which includes both software and hardware prefetching (HW-PF+SW-PF). The most optimized version occupy the lower right corner of the plot with lower energy and better performance. We see that the version with hardware prefetch only (HW-PF only) is 10% more energy efficient than the baseline although performance is very similar. These two versions also perform significantly better than disabling the hardware prefetcher (SW-PF only) and disabling all prefetching. For *stencil*, the hardware prefetcher performs quite well and the instruction energy overhead of software prefetching is not justified. The other benchmarks do not display this behavior because the hardware prefetcher is less effective.

VI. Conclusion

We present detailed energy characterization of the Xeon Phi chip identifying energy behavior trends as a function of instruction types and the number of active threads and cores. Using this characterization data, we build a highly accurate instruction-level energy model for the processor. We show that this energy model can be used to identify opportunities to improve energy efficiency. This is the first work to characterize such a large many-core/multi-thread x86-based system and build a high-level energy model for software developers.

Acknowledgments

We thank the Intel Parallel Computing Lab for their support and feedback during this work. Special thanks are due to Victor Lee for hosting this research and providing guidance throughout the project. We also thank Mikhail Smelyanskiy for providing the Linpack implementation on Xeon Phi. This work was also partially supported by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. This work was also partially supported by the National Science Foundation (NSF) Expeditions in Computing Award #: CCF-0926148. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1]http://www.green500.org/lists/green201211.

[2]D. Brooks, V.Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

[3]A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite . In *GPGPU*, 2010.

[4]S. Hong and H. Kim. An integrated gpu power and performance model. In *ISCA*, 2010.

[5]C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and Empirical data. In *MICRO*, 2003.

[6]T. Karkhanis and J. Smith. Automated design of application specific superscalar processors: an analytical approach. In *ISCA*, 2007.

[7]S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[8]V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embeded software: a first step towards software power minimization. In *IEEE Transactions on VLSI Systems*, 1994.