open proceedings

Micro Analysis to Enable Energy-Efficient Database Systems

Chen Yang[1], Yongjie Du[1], Zhihui Du[2], Xiaofeng Meng[1]
[1]School of Information, Renmin University, China
[2]Department of Computer Science and Technology, Tsinghua University, China

ABSTRACT

CPU has been identified as the energy bottleneck for database systems and existing approaches only allow database systems to trade the performance for energy. However, our work show that cutting down the energy cost of database systems without losing the CPU performance is feasible. We first develop a measurement methodology to accurately evaluate the energy cost of different CPU micro-operations. Then three popular database systems with different setups and data sizes are used as benchmarks to explore the energy distribution on CPU micro-operations. Our experimental results show that L1 data cache (L1D cache) consumes 39%-67% of total CPU energy and it is definitely the energy bottleneck of database systems. This finding inspires us a novel idea on building energy-efficient database systems with customized CPU architecture that features low L1D cache energy cost. A proof-of-concept system is developed to evaluate this idea and the experimental results show that our solution can not only achieve 60% of peak energy saving but also gain further performance improvement.

1 INTRODUCTION

As the infrastructure of the data center, database system has been limited by the energy wall. The energy cost of powering the database server is not only rapidly approaching the machine acquisition cost[24], but the energy wall also limits the scalability of database server. CPU[13, 19, 25], main memory[23] and disk[24] may become the energy bottleneck due to different computer architectures and database types. In this paper, we focus on the energy profiling of typical relational database systems on the x86_64 architecture with local disk, and many evidences have confirmed that CPU consumes more power than other major components in our scenario[13, 19, 25].

According to whether the workload is running or not, the CPU energy cost may be classified into Busy-CPU energy cost and Idle-CPU energy cost. The Idle-CPU energy cost has been reported to reduce from 50% to 18%[19]. Undoubtedly, the Busy-CPU energy cost as the dominant part attracts a lot of research work. For example, both academia and industry have expended a great deal of effort in energy-oriented query optimization[21, 28, 29] and the external energy knobs based approaches[13, 19, 21, 23]. The basic idea is building the cost model based on the Busy-CPU energy cost to choose the energy-optimized query plan or set the appropriate CPU voltage and frequency according to the database load status. These methods consider CPU as a black box and the energy cost is reduced by trading the performance, such as a 43%-80% performance loss[21, 28].

Actually, there are many different micro-operations inside the CPU and they expect different energy costs. Existing black-box optimization methods cannot take advantage of micro energy cost characteristics of database workloads so they often lead to significant performance loss to meet the energy saving demand.

Energy-efficient database systems expect that the CPU architecture can significantly cut down the Busy-CPU energy cost. To achieve this object, an in-depth breakdown of Busy-CPU energy cost is indispensable. It cannot only help us identify the energy bottleneck on CPU, but it is also the basic work to design a novel customized CPU architecture for energy-efficient database machine. Breakdown analysis of Busy-CPU energy cost will help to answer some important questions such as what is the microscopic distribution of Busy-CPU energy cost and how different database implementations and settings affect this distribution.

In this paper, we design micro-benchmarks to breakdown the Busy-CPU energy cost of typical read query workloads, and enable the energy-efficient database system on the customized CPU architecture. Unless otherwise specified, the query will refer to read query. It is very difficult to achieve an accurate breakdown of energy cost on a real database system. The micro-operations which can be monitored are so many e.g., about 514 events inside our CPU that we cannot evaluate the energy cost for every micro-operation. To isolate the energy cost of an individual micro-operation, we need to overcome many mutual related factors, such as the compiler optimization and architectural features. In addition, to enable energy-efficient database systems, we have investigated many CPU architectures and updated the kernels of both operator system and database system to make them support the customized CPU architecture well. Finally, we present a clear energy cost distribution pattern of database systems and give a proof-of-concept system to cut down the energy cost without losing the performance. The major contributions are as follows.

- A micro analysis based accurate energy breakdown method is proposed for the Busy-CPU energy cost with typical query workloads.
- The energy bottleneck L1D cache is identified based on extensive experiments on three typical database systems.
- The L1D energy-efficient CPU architecture design is proposed and the experimental results show that 60% of peak energy saving can be achieved with further performance improvement.

The rest of the paper is organized as follows. Section 2 presents our evaluation approach and the energy cost of micro-operations. Section 3 profiles the energy cost of database systems along with a detailed analysis. Section 4 presents our proof-of-concept system design, optimization approach and evaluation results. Section 5 analyzes the energy cost preference of some typical scenarios. Section 6 describes the related work. Section 7 summaries our work and presents directions for future work.

2 MICRO ANALYSIS METHOD FOR BUSY-CPU ENERGY

In this section, we will present our methodology on how to break down the Busy-CPU energy cost into the energy cost of different micro-operations for queries in relational database systems.

Corresponding author is Xiaofeng Meng.
© 2020 Copyright held by the owner/author(s). Published in Proceedings of the 23rd International Conference on Extending Database Technology (EDBT), March 30-April 2, 2020, Copenhagen, Denmark, ISBN 978-3-89318-083-7 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

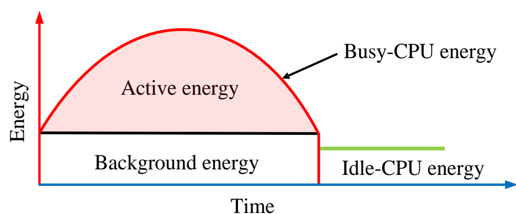

Figure 1: Example of energy cost along the workload running.

2.1 Key Idea

Depending on whether the CPU is on operational states or idle states, we name the total energy cost as Busy-CPU energy cost and Idle-CPU energy cost, shown in Figure 1. When fixing the CPU frequency and voltage, we further divide Busy-CPU energy cost into the Active energy cost and the Background energy cost, defined as

$$Busy\text{-}CPU\ \ energy = Active\ \ energy + Background\ \ energy.$$

When running a workload, the Active energy is the real cost used for the calculation and data movement and the Background energy is the fixed cost to activate the hardware. Obviously, the Active energy cost can reveal the power usage of a workload so that it is our profiling target.

We first formalize the Active energy cost as the sum of the energy cost of micro-operations. Next, we mainly solve three issues. (1) We have to identify which micro-operations to be profiled. The query workloads are typically data-intensive. This motivates us to pay an attention to the data movement related micro-operations, such as the cache load. When we know the executed counts of these micro-operations and the energy cost driving them once, we can actually evaluate their energy cost. (2) How to quantify the energy cost of an individual micro-operation. We construct many micro-benchmarks, each of which shows a simple performance behavior issued by the specific micro-operations. Then, we also build energy models to map the energy cost of micro-benchmarks into the energy cost of an individual micro-operation. (3) How to verify the accuracy of the energy cost of an individual micro-operation. We have to construct the other micro-benchmarks which have the clear and complex performance behaviors. We can identify the difference between the estimated energy cost and the measurement value to take the verification. We will give a detailed introduction focusing on the above three issues.

2.2 Problem Formalization

We denote the analyzed micro-operation set as MS and then formalize the Active energy cost E_{active} for the workload w as

$$E_{active}(w) = E_{other}(w) + \sum_{m \in MS} E_m(w), \tag{1}$$

where $E_m(w) = N_m(w) \times \Delta E_m$ is the energy cost of the micro-operation m. ΔE_m is the energy cost driving the micro-operation m once and N_m is the count executing the micro-operation m and E_{other} is the unisolated energy cost, including the calculation, L1I cache and TLB, etc. According to Eq. (1), we have to solve the ΔE_m and N_m. Noting that our energy breakdown model is based on the stable voltage and CPU frequency. The dynamic voltage and frequency scaling (DVFS) will cause fluctuations on ΔE_m, so we disable DVFS during the testing. However, we can also evaluate the impact of different voltages and CPU frequencies. To solve Eq. (1), we must construct the set MS and evaluate the ΔE_m of every micro-operation in MS.

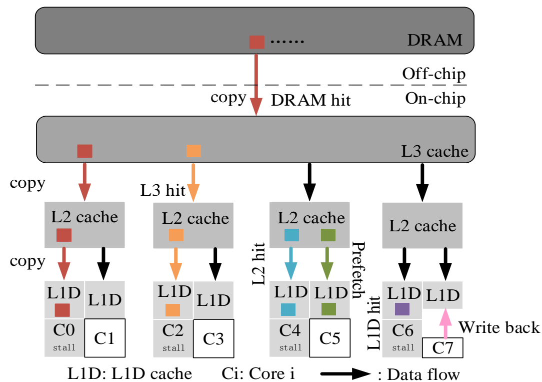

Figure 2: Example of data movement on the modern CPU architecture. The CPU core in the white box is busy and it in the gray box is stall.

2.3 Energy Breakdown Representation

Generally speaking, the query workload is usually data-intensive, so that we tend to choose data movement related micro-operations to characterize the energy cost. For the modern CPU architecture, the data is frequently moved between main memory and registers. In order to cross the memory wall, the CPU (e.g., Intel i7-4790 in our experiment) contains a three-level cache memory sub-system, shown in Figure 2. The closer the cache memory is to the CPU core, the smaller, faster and more energy-efficient it will be. We describe three ways of data movement that have the great impact on query workloads.

Regular data fetching. If the data is not in registers, the load instruction will fetch the data of the cache line size (e.g., 64 Bytes) and the CPU core could stall to wait for the data return. Noting that if instructions are uncorrelated with each other, speculation and out-of-order execution can disable the pipeline bubble. In addition, data fetching follows a step-by-step replication strategy. The CPU will first fetch data from L1D cache. If L1D cache hits (L1D hit), the data will be loaded into the register, like Core 6 in Figure 2. Otherwise, the CPU will go to the next level cache to search for data, called as L1D cache miss (L1D miss). Specifically, when L1D cache misses, L2 cache starts to search for data. There are also two cases: hit and miss. If L2 cache hits, the data will be copied to L1D cache first, and then copied to the register, like Core 4 in Figure 2. Similarly, L3 hit and DRAM hit are like Core 2 and Core 0 in Figure 2. Although the step-by-step replication strategy can provide the good data locality, the data movement leads to much energy cost.

Data prefetching. In order to improve the CPU performance, the data prefetching technique is used to predict the data usage under the background and fetch it without the pipeline bubble, like Core 5 in Figure 2. So, the data prefetching will also cause the data movement behavior. Four prefetching techniques are provided in Intel i7-4790[3]. Two are implemented through L1D hardware prefetcher to replicate the data into L1D cache in advance. Unfortunately, they cannot support the performance counter in Intel i7-4790. The two other types of prefetches that can be generated by the L2 hardware prefetcher – prefetches into

the L2 cache (L2 prefetching) or prefetches into the L3 cache (L3 prefetching). Their behaviors can be monitored by the performance counter. In our paper, the data prefetching only means the L2 hardware prefetcher.

Write-back. Although the query is the read-only workload, lots of temporary data, e.g., local variables, have to be created and updated. The store operation always updates the data into L1D cache within 1 cycle. Because the local variables do not need to be eventually persisted so that they are rarely written to the lower level memory due to the write-back strategy. For example, L1D cache store hit rate is 99.86% in our experiment. So, we also evaluate the store operation which writes the data into L1D cache.

Based on the above analysis, we define the micro-operation set MS as

$$MS = \{L1D, Reg2L1D, L2, L3, mem, pf, stall\}$$

for the query workload. $\forall m \in \{L1D, L2, L3, mem\}$ is that a load operation reads the data from m to the next higher level memory. For example, $m = L2$ means to load the data from L2 cache to L1D cache. $Reg2L1D$ means that the store operation writes the data from the register to L1D cache. We combine two different types of prefetches generated by the L2 hardware prefetcher together as a micro-operation pf meaning to prefetch data. The micro-operation $stall$ is the stall event due to memory access. Recalling the Eq. (1), we next evaluate N_m and ΔE_m.

The energy breakdown of update/write queries is a totally different problem from the read queries. We need to know how to write the data into main memory. It may involve more micro-operations about writing. We do not discuss it in depth in this paper.

2.4 Micro-Operation Counting

The modern processors have built-in performance monitor unit (PMU) to record the performance related events. We can use Linux Perf[5] or ocperf[6] to get them from PMU to evaluate the micro-operation count N_m. For $\forall m \in \{L1D, L2, L3, pf\}$, it is worth noting that N_m is the sum of both the hit count and the miss count due to the step-by-step replication strategy. N_{pf} involves the L2 prefetching count N_{pf}^{L2} and the L3 prefetching count N_{pf}^{L3}. Especially, N_{mem} is the miss count of L3 cache. $N_{Reg2L1D}$ is the hit count when writing the data into L1D cache. N_{stall} is the stall cycle count due to the data load.

2.5 Energy Evaluation of Micro-Operation

To quantify ΔE_m, we design a set of micro-benchmarks which can achieve the specific performance behavior, such as only accessing L1D cache, etc. Finally, we can use the Active energy cost of micro-benchmarks to evaluate the ΔE_m of different micro-operations.

2.5.1 Micro-Benchmark Design. Isolating memory access to only follow a specific performance behavior is a challenging task in modern processors. The design of the micro-benchmark methodology is inspired by the recent work[22]. The out-of-order execution, speculation execution and data prefetching have worked well in hiding memory latencies but at the same time make the energy cost benchmarking for an individual instruction difficult. In addition, the treading switching, DVFS and the compiler optimization could affect the evaluation accuracy. In order to accurately evaluate ΔE_m, we make a lot of effort on it. First, our micro-benchmarks should minimize the effect of CPU architectural optimization. Second, we must configure the

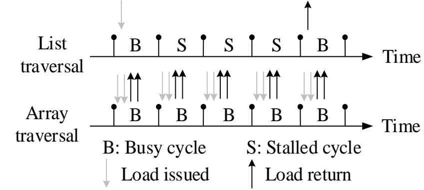

Figure 3: CPU execution behaviors when list traversal and array traversal only load data from L1D cache.

appropriate runtime environment to reduce the error, shown in Section 2.5.3. Third, we must review the assembly code which is generated by the compiler to disable some compiler flags who will change the performance behavior. Our micro-benchmarks follow two design frameworks to skip out of the architectural optimization.

List traversal. We allocate a size of memory as an array Arr of pointers. The size of each item is 64 Bytes (i.e., cache line size) which can be processed by a load operation. We link every item as the list. Then, we traverse the list many times. Through the correct implementation, we can ensure that micro-benchmarks only load data from the specific memory layer. In this way, the energy cost of each micro-benchmark can be broken down into the energy cost of the specific micro-operations and stall cycle.

The list structure can make sure the data items have the back-and-forth dependency. Due to the access dependency, in the premise of disabling the perfetching the CPU does not know the address of the next data item until the previous item is finished. An example is shown in Figure 3. We assume the data in L1D cache and a L1D load requires 4 cycles from issue to return. Due to the unknown address of next data item, the pipeline is forced to break, leading to 1 load cycle and 3 stall cycles, i.e., L1D load energy and stall energy. The link traversal can disable the out-of-order execution and the pipeline to minimize the energy cost measurement error incurred by CPU architectural optimization.

Array traversal. The energy cost of stall cycle cannot be separated by list traversal. So, we design the array traversal framework. In it, the micro-benchmark allocates a size of memory as an array Arr (also 64 Bytes per data item), and then sequentially traverses it many times. Under some special conditions, CPU architectural optimization can make micro-benchmarks only load data from the specific memory layer without stall cycles.

As well known, the data item in array is completely independent and supports random access because the CPU knows the address of all data items before traversing. The load of next data does not have to wait for the finish of the previous data item. As shown in Figure 3, although a L1D load still requires 4 cycles, 3 stall cycles can be hidden due to the continuous pipeline. In addition, the Dual-Issue technique of Intel i7-4790 can issue two load instructions per cycle without stall.

2.5.2 Micro-Benchmark Set. For evaluating ΔE_m ($\forall m \in MS$), we build a micro-benchmark set MBS including 6 micro-benchmarks to show the specific performance behaviors as follows.

B_L1D_array. This micro-benchmark is used to evaluate ΔE_{L1D}, only accessing L1D cache without stall cycles. As shown in Algorithm 1, it follows the array traversal framework and allocates memory size smaller than or equal to the size of L1D cache. As shown in Figure 4a, if $S_{mem} = 32KB$ (i.e., L1D cache size in i7-4790), the Arr size is 500 and the size of each item is 64 Bytes. T is usually set into a huge value, such as 1 billion, to maintain

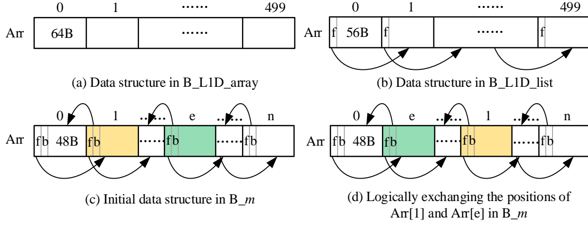

Figure 4: Examples of data structures in different micro-benchmarks. B_m represents B_L2, B_L3 and B_mem. Pointer f points the next data item and pointer b points the previous data item.

a stable memory access pattern. A large T also applies to Algorithm 2-4. For balancing the performance, it can also be reduced moderately. So, all the data easily fit the L1D cache, so there will not be any miss after initial set of loads. Noting that we unroll the *Arr* and traverse it instead of loop traversal, it will avoid the effects of loop control statements as much as possible. By this programming optimization, 98.6% instructions are the desired load instructions.

Algorithm 1 B_L1D_array

Input: S_{mem}: allocated memory size; T: loop times;
1: allocate S_{mem} memory size as an array Arr with $\frac{S_{mem}}{64}$ items;
2: **for** iter i in range T **do**
3: traverse Arr through unrolling $\frac{S_{mem}}{64}$ times;
4: **end for**

B_L1D_list. The energy cost of this micro-benchmark mainly involves the load operation from L1D cache and the CPU stall. As shown in Algorithm 2, it follows the list traversal framework and allocates memory size smaller than or equal to the size of L1D cache, such as still 32KB. We also give an example to show such data structure in Figure 4b.

Algorithm 2 B_L1D_list

Input: S_{mem}: allocated memory size; T: loop times;
1: allocate S_{mem} memory size as an array Arr with $\frac{S_{mem}}{64}$ items;
2: partition each item into a pointer f (the first 8 Bytes) and the last 56 Bytes of data;
3: **for** iter j in range $\frac{S_{mem}}{64} - 1$ **do**
4: $Arr[j].f = \&Arr[j+1]$;
5: **end for**
6: **for** iter i in range T **do**
7: use pointer f to traverse Arr and unroll $\frac{S_{mem}}{64}$ times;
8: **end for**

B_L2, B_L3 and B_mem. They will be used to evaluate ΔE_m ($\forall m \in \{L2, L3, mem\}$). As shown in Algorithm 3, they follow the list traversal framework and only access the specific memory layers. When only accessing the memory layer *m*, the allocated memory size should be as close as possible to the sum of the *m* size and sizes of its higher caches to ensure that the majority of data is in *m*. Using B_L2 as an example, the allocated memory size should be close to 288KB (32KB L1D cache and 256KB L2 cache). Similar setup methods can be extended to B_L3 and B_mem. Noting that we do not use the linked list structure similar to B_L1D_list, the sequential load along physical position means the simple access pattern, which is easily employed by the modern CPU to improve the performance. However, it has the serious impact on profiling. For example, when setting $S_{mem} = 260KB$ for Algorithm 2, the L1D hit rate is 55%, so that there is no guarantee that only L2 cache is accessed. Thus, we randomize the access order (logical position) and generate jump access on a large span to break the data locality, as shown in Figure 4d. Because the low memory layer is far larger than the high memory layer, the data will usually miss in the high memory layer.

Algorithm 3 B_L2, B_L3 and B_mem

Input: S_{mem}: allocated memory size; T: loop times; ε_{span}: span threshold of two given data items;
1: allocate S_{mem} memory size as an array Arr with $\frac{S_{mem}}{64}$ items;
2: partition each item into a pointer f (the first 8 Bytes), and pointer b (the second 8 Bytes) and the last 48 Bytes of data;
3: **for** iter j in range $\frac{S_{mem}}{64} - 1$ **do**
4: $Arr[j].f = \&Arr[j+1]$;
5: $Arr[j].b = \&Arr[j-1]$ $(j-1>0)$;
6: **end for**
7: **for** iter z in range $\frac{S_{mem}}{64} - 1$ **do**
8: //avoid frequent exchange of logical neighbors when e is always the same value.
9: randomly pick $e \in [1, \frac{S_{mem}}{64} - 2]$ to satisfy $|z-e| > \varepsilon_{span}$ and $Arr[e]$ is not the logical neighbor of $Arr[z]$;
10: exchange the logical positions of $Arr[z]$ and $Arr[e]$;
11: **end for**
12: **for** iter i in range T **do**
13: use pointer f to traverse Arr and unroll $\frac{S_{mem}}{64}$ times;
14: **end for**

B_Reg2L1D. The energy cost of this micro-benchmark mainly involves the store operation from registers and the L1D cache. As shown in Algorithm 4, it only accesses the same variable repeatedly, but it is effective to ensure that the CPU only execute the store operation. This benchmark always access the same variable, the CPU can find it in registers, instead of reading it from L1D cache every time. In addition, the allocated memory size is large enough, so that the CPU has to perform multiple store operations to complete the assignment. Due to temporary variable assignment, the vast majority of store operations only involve L1D cache.

Algorithm 4 B_Reg2L1D

Input: T: loop times; ut: unrolling times;
1: allocate 64 Bytes of memory size as a variable A;
2: **for** iter i in range T **do**
3: execute $p = A$ through unrolling ut times; //variable assignment
4: **end for**

Our micro-benchmark set can achieve the specific performance behavior. Noting that it may be not the only way, but it has been enough accurate to help us profile the energy cost of database systems.

2.5.3 Runtime Configuration. The accurate execution of our micro-benchmarks depends on some runtime configurations to overcome the measurement error.

Compiler optimization. In order to minimize the impact of unnecessary instructions, our micro-benchmark set is compiled

with an optimization level of -O3. The necessary temporary variables are added with a *volatile* modifier, such as a temporary pointer variable for linked list pointer tracking, which can cancel the compiler's active optimization to avoid the microscopic behavior changing.

Thread switching. In order to prevent the thread attached by the micro-benchmark from being switched between different idle CPUs during execution, the micro-benchmark will be fixed on a specific logic core to run.

DVFS knobs. EIST (Enhanced Intel SpeedStep Technology), an Intel DVFS implementation, changes CPU frequency and voltage to balance energy cost and performance. The energy cost of micro-components will increase with the improvement of CPU frequency and voltage. Because our micro-benchmarks are effective under the stable frequency and voltage, EIST technique will incur the measurement error of energy cost. We turn off these options and execute our micro-benchmarks under the given frequency and voltage according to our experimental requirement.

Prefetcher. The data prefetching could lead to the unexpected load instructions into our micro-benchmarks. So, the hardware prefetcher will be turned off by modifying MSR registers when running our micro-benchmarks. It will be turned on when evaluating the energy cost of query workloads.

Through running our micro-benchmarks, we can isolate the specific micro-operations and reduce most of measuring errors. It will lead to an easy solution to ΔE_m in next section.

2.5.4 Energy Model to Evaluate ΔE_m. We construct a series of energy models to map the energy cost of the micro-benchmark into ΔE_m. For any micro-benchmark $mb \in MBS$, we define the Active energy cost of mb as $E(mb)$ which will be given in next section. Here, we assume that $E(mb)$ is known to give the solution of ΔE_m.

Through running B_L1D_array which only loads data from L1D cache, we can solve the ΔE_{L1D} as

$$\Delta E_{L1D} = \frac{E(B_L1D_array)}{N_{L1D}}.$$

Recalling that N_{L1D} is the count loading data from L1D cache in Section 2.4. Similarly, when running B_L1D_list, we can solve the ΔE_{stall} as

$$\Delta E_{stall} = \frac{E(B_L1D_list) - E_{L1D}}{N_{stall}}.$$

For the micro-operations $x, y \in C = \{L1D, L2, L3, mem\}$, if the micro-operation x loads data from the higher memory layer than y, we denote $x > y$. When running the micro-benchmark B_L2, B_L3 and B_mem, respectively, we can solve the ΔE_m as

$$\Delta E_m = \frac{E(B_m) - \sum_{i>m}^{i \in C} \Delta E_i N_i - E_{stall}}{N_m}. \quad (2)$$

Due to the step-by-step replication strategy, loading data from the low memory layer must also lead to a load operation from the higher memory layer. So, the load energy cost of the higher memory layer needs to be eliminated in Eq. (2). When running B_Reg2L1D, we can solve the $\Delta E_{Reg2L1D}$ as

$$\Delta E_{Reg2L1D} = \frac{E(B_Reg2L1D)}{N_{Reg2L1D}}.$$

Prefetching energy. In term of the energy cost, the data prefetching and the regular data fetching are similar. we follow the assumption that the energy is mainly consumed in moving data from a specific memory layer to a higher layer [18] and set $\Delta E_{pf}^{L2} = \Delta E_{L3}$ and $\Delta E_{pf}^{L3} = \Delta E_{mem}$. ΔE_{pf}^{L2} is the energy cost of an individual L2 prefetching from L3 cache and ΔE_{pf}^{L3} is the energy cost of an individual L3 prefetching from main memory[4].

2.5.5 Verification Method of ΔE_m. For verifying the accuracy of ΔE_m, we propose a verification micro-benchmark set $VMBS$ derived from MBS, where each micro-benchmark shows a more complex performance behavior.

Micro-benchmarks in $VMBS$ essentially perform a series of data movement operations and data calculation operations to simulate the real workload. We first construct two micro-benchmarks B_add and B_nop. They only loop the known number of `add` and `nop` instructions to evaluate the ΔE_{add} and ΔE_{nop}. Next, we add the *add* and *nop* instructions into the micro-benchmarks in MBS to finally construct $VMBS$ including 7 micro-benchmarks shown in Table 3. For example, B_L1D_list_nop is to add the *nop* instruction into B_L1D_list.

When running a micro-benchmark $v \in VMBS$, Eq. (1) is used to solve the estimated Active energy cost $\overline{E}_{active}(v)$ where we set the $E_{other}(v) = \Delta E_{add} N_{add}(v) + \Delta E_{nop} N_{nop}(v)$. We measure the real $E_{active}(v)$ and define the accuracy as

$$acc(v) = 1 - \frac{|\overline{E}_{active}(v) - E_{active}(v)|}{E_{active}(v)},$$

where $acc < 0$, set $acc = 0$. If the acc is closer to 1, the estimated energy cost is closer to the real energy cost, showing that the ΔE_m got by our approach is accurate.

Through the above effort, we have got all of ΔE_m ($m \in MS$). Next, we introduce how to measure the Active energy.

2.6 Active Energy Evaluation

Our experiments run on the server with an Intel i7-4790 processor (L1D cache size is 32KB, L2 cache size is 256KB and L3 cache size is 8MB), 32GB DDR3-1600 main memory and a 500GB SATA hard drive. Our processor is popular to enable that our results are representative. The operator system is Ubuntu 14.04 including Linux Perf, ocperf and RAPL (Running Average Power Limit)[10].

We leverage RAPL to measure the energy cost. RAPL counters are highly accurate on x86_64 and allow us to separately measure the energy cost of the core domain $E(core)$, the package domain $E(package)$ (including the core, L3 cache and memory controller) and the main memory domain $E(memory)$. We can run an only-blocked program (e.g., sleep 1) to use its RAPL's measurement values as the Background energy cost of three different domains per second when disabling idle states (i.e., C-states[7]). For workloads which do not access L3 cache and main memory, we observe the $E(core)$ as the Busy-CPU energy cost, such as B_L1D_list and B_L1D_array. For the workloads which do not access main memory, we observe the $E(package)$ as the Busy-CPU energy cost. For other workloads, we observe $E(package) + E(memory)$ as the Busy-CPU energy cost. E_{active} is Busy-CPU energy cost minus the corresponding Background energy cost.

2.7 Selection of CPU Frequency and Voltage

Our micro-benchmarks set is usually used to evaluate ΔE_m under the fixed CPU frequency and voltage. However, the real workloads widely run with the EIST knob turned on to balance the performance and energy. So, in this section we will profile the performance of TPCH query workloads when turning on EIST

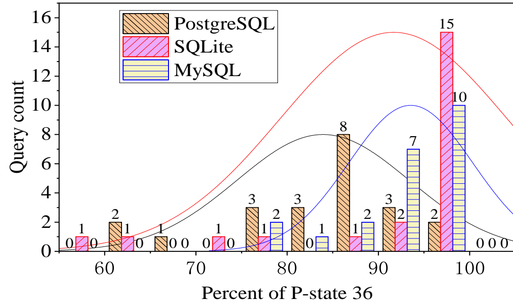

Figure 5: Query count distribution bars and the fitting distribution curves based on the percent of P-state 36.

Table 1: Runtime behaviors of micro-benchmarks

Micro-benchmarks	BLI	L1D miss%	L2 miss%	L3 miss%	IPC
B_L1D_list	98.9	0.01	-	-	0.26
B_L1D_array	99.5	0.01	-	-	2.02
B_L2	98.5	99.93	0.02	-	0.09
B_L3	98.6	98.68	99.98	0.01	0.03
B_mem	97.8	98.86	99.88	97.45	0.005
B_Reg2L1D	99.98	0.02	-	-	1.01
B_add	98.4	-	-	-	2.01
B_nop	99.87	-	-	-	3.99

Table 2: Energy cost of micro-operations at different CPU frequencies and voltages

P-state	36 (3.6GHz)	24 (2.4GHz)	12 (1.2GHz)
Micro-operations	Energy cost (nJ)		
ΔE_{L1D}	1.30	0.90 ↓	0.60 ↓
ΔE_{L2}	4.37	3.25 ↓	1.64 ↓
ΔE_{L3}, ΔE_{pf}^{L2}	6.64	5.91 ↓	5.33 ↓
ΔE_{mem}, ΔE_{pf}^{L3}	103.1	99.1 ↓	99.04 ↓
$\Delta E_{Reg2L1D}$	2.42	1.60 ↓	1.10 ↓
ΔE_{stall}	1.72	1.07 ↓	0.80 ↓
ΔE_{add}	1.03	-	-
ΔE_{nop}	0.65	-	-

to identify their preference for CPU frequency and voltage. That can help us evaluate a more reasonable ΔE_m.

EIST usually sets the CPU into different states to save energy[1], including C-states and P-states[7]. C-states are idle states {C0, C1, C2, ...}. C0 means the CPU non-idle and others mean that the CPU enters an idle state with different energy-saving levels. C0 can be further subdivided into different P-states. So, P-states can be called as operational states. Each P-state also has a different energy-saving level. We focus on P-states due to the profiling of Active energy cost. In truth, a P-state is both a frequency and voltage operating point. For the high CPU load, a high-performance P-state might be set, and vice versa. Intel i7-4790 includes 29 candidate P-states. CPU frequency of each P-state differs by 100MHz.

[1]The modern processor frequency can be separated into (1) core frequency involving ALU, L1 cache, L2 cache and etc, and (2) uncore frequency involving L3 cache, memory controller and etc. In this paper, the CPU frequency means core frequency. The uncore frequency in Intel i7-4790 will dynamically match the CPU frequencies.

Table 3: Energy cost of verification micro-benchmarks and the accuracy

Verification micro-benchmarks	$\bar{E}_{active}$ (J)	E_{active} (J)	acc%
B_L1D_list_nop	129.34	122.04	94.36
B_L1D_array_add	169.85	150.71	88.73
B_L2_nop	122.01	125.57	97.08
B_L3_add	215.37	224.16	96.07
B_mem_nop	396	345.37	87.22
B_L1D_list_L2	168.29	158.26	94.01
B_L1D_list_nop_add	193.06	186.94	96.83

The highest P-state is 36 (3.6GHz CPU frequency) and the lowest is 8 (800MHz CPU frequency).

In this experiment, we turn on the EIST knob and set the P-state range from 8 to 36. Our purpose is to analyze the P-state preference of query workloads. We use 22 TPCH queries [8] to benchmark PostgreSQL, SQLite and MySQL with baseline configuration and baseline data size (more detailed instructions in Section 3) and sample the runtime P-state per 100 milliseconds. According to the percent of P-state 36, Figure 5 shows query count distributions of three database systems. We find that most of queries tend to run at P-state 36, due to the high CPU load (average 96% CPU usage). So, we will fix the CPU at P-state 36 in the following trunk experiment. We also evaluate the impact of other P-states on our results in Section 3.5.

2.8 Results of Micro-Benchmarks

We turn off these knobs which will lead to measurement errors shown in Section 2.5.3, fix the CPU at P-state 36 and specify all workloads to run on the core CPU0. For B_L1D_list, B_L1D_array and B_Reg2L1D, we allocate the memory size as 31KB; for B_L2, allocate 260KB; for B_L3, allocate 6MB and for B_mem allocate 60MB. These setups ensure that our micro-benchmarks only fetch data from the single memory layer under an acceptable latency.

Performance behaviors of micro-benchmarks. As shown in Table 1, BLI (Body-Loop Instruction%) is the percentage of the desired instructions in the main loop and IPC (Instruction Per Clock) is the number of instructions per cycle. For BLI, 98.9% instructions of B_L1D_list is to load data from L1D cache. For other micro-benchmarks, this metric also has a good performance, showing that our micro-benchmarks have little noise instructions. In addition, our micro-benchmarks can provide the specific performance behavior. For B_L1D_list, L1D miss rate is only 0.01%, showing that it always only accesses the L1D cache. Even for B_mem, it can still skip out of cache memory and load data from main memory with a hit rate 97.45%. Especially, IPC shows the CPU stall status. For B_L1D_array, IPC is 2.02 showing that CPU is always busy and no stall. However, IPC=0.26 for B_L1D_list shows that 4 cycles are required to execute a load operation. For B_Reg2L1D, IPC is 1.01 and the number of L1D store instructions are 98.37% of all instructions, showing that CPU always executes 1 store instruction per cycle. These results reveal that our benchmarks can work properly with specific performance behaviors.

Evaluation of ΔE_m. According to E_{active} of micro benchmarks and energy models in Section 2.5.4, we give the energy cost of micro-operations in Table 2. The unit of energy cost is Nanojoule. For the load operation the data is closer to the CPU causing the energy cost to be lower. Especially for ΔE_{L1D}, it is

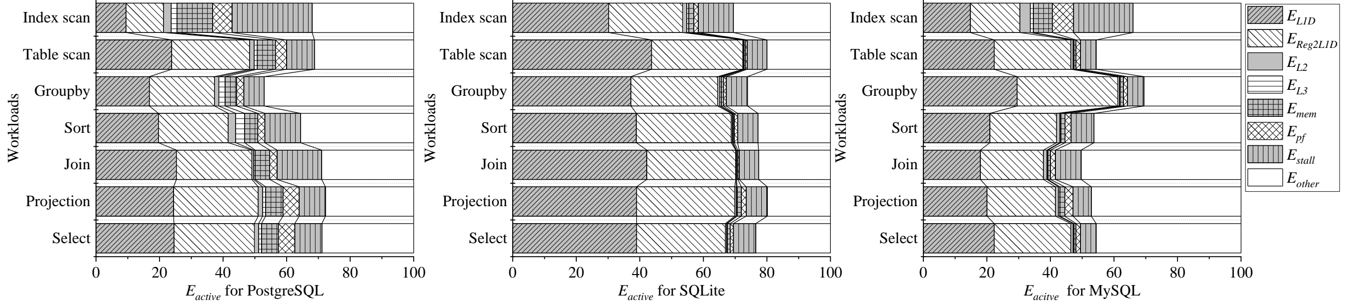

Figure 6: Active energy cost breakdown of the basic query operations for three different database systems.

the lowest than other load operations. Oppositely, loading from the main memory will get a high energy cost penalty. Actually, it explains that why improving cache hit rate can improve both the performance and energy-efficiency.

Verification of ΔE_m. We use the verification method in Section 2.5.5 to evaluate the accuracy of ΔE_m. In Table 3, we give the real Active energy cost and the estimated Active energy cost and the accuracy. The unit of energy cost is Joule. The average accuracy is 93.47%. Even for the most complex B_mem_nop, the accuracy is still high. It shows that ΔE_m proposed by us is accurate enough to break down the energy cost of real workloads.

3 ENERGY COST DISTRIBUTION OF QUERIES

In this section, we use ΔE_m to break down the energy cost of query workloads implemented on the real database systems. We use the PostgreSQL-9.5.2, SQLite-3.14.2 and MySQL-8.0.13 as our analysis targets. We will deeply analyze the energy cost of 7 basic query operations and 22 queries in TPCH[8] with different data sizes and knob settings. We will compare the energy cost breakdown of queries with the typical CPU-bound workloads. In addition, we also show the impact of different P-states on the energy cost breakdown.

To avoid the random error, we disable the result display by updating the database kernels and run the workloads 100 times (10 times for long-running workloads). Finally, the average energy cost is got. We turn on the hardware prefetchers, specify all workloads to run on the core CPU0 and fix the CPU into P-state 36. In addition, the percent of the Background energy cost is 47.2%-51.7% of Busy-CPU energy cost in our experiments. Our main findings are summarized as follows.

- 77.7%-89.2% of Busy-CPU energy cost can be broken down into the energy cost of data movement and the Background energy cost. The energy cost of data movement (7 micro-operations in *MS*) is 55%-76.4% of Active energy cost.
- $E_{L1D} + E_{Reg2L1D}$ is 39%-67% of Active energy cost, identified as the energy bottleneck. This phenomenon does not appear in the typical CPU-bound workloads. In addition, it is little affected by the data size, the database setting and CPU frequency and voltage.
- The sequential scanning in query workloads is the major reason that leads to this energy cost pattern.

3.1 Experimental setup

We take our experiments with 100MB (baseline), 500MB and 1GB data. In addition, each database system has many configurable knobs. We investigate them and tune two important knobs that also have similar roles in three database systems. In Table 4, we give three kinds of database settings to limit the memory usage. The resource size provided to three database systems at each setting is approximate. The `small` setting looks stringent and the `large` setting is relaxed.

Table 4: Knob settings for three database systems

Database systems	Knobs	Small	Baseline	Large
PostgreSQL	Shared_buffers	8MB	128MB	1024MB
	Work_mem	4MB	64MB	512MB
SQLite	Cache_size	2000	16000	65000
	Page_size	4KB	8KB	16KB
MySQL[1]	Inbuffer_size	8MB	128MB	1024MB
	Inpage_size	4KB	8KB	16KB

3.2 Energy Cost of Basic Query Operations

We profile the energy cost of 7 basic query operations with the baseline data size on the baseline setting. 77.7%-89.2% Busy-CPU energy cost can be broken down into the energy cost of data movement and the Background energy cost, showing that our energy breakdown approach can work well on database systems. Figure 6 gives the breakdown of E_{active}. The energy cost of data movement is 68.1% for PostgreSQL, 76.4% for SQLite and 56.8% for MySQL, becoming dominant. For the three database systems, the energy cost distribution is similar, and much energy is consumed in L1D cache load/store. $E_{L1D} + E_{Reg2L1D}$ is 41.6% for PostgreSQL, 66.6% for SQLite and 43.4% for MySQL. So, we can think that this phenomenon could be general for most of relational database systems.

L1D cache load. Actually, we can easily explain why the percent of E_{L1D} is high. In the database systems, almost all of the query operations is based on sequential scan. Even if segmentation or paging strategies are used to manage data, each data block is big enough to fit the CPU cache to ensure a good data locality. So, the modern CPU architecture can ensure most of data to be loaded at L1D cache when sequentially scanning. For example, L1D hit rate of 7 basic query operations is 97.74% and IPC of the complex `join` operator is 1.85, showing good data locality.

L1D cache store. In addition, the reason why the energy cost of L1D cache store is high is because the query workload will generate many temporary data, such as temporary storage of intermediate data and output stream. These temporary data are

[1]For MySQL knobs, `inbuffer_size` is short for `innodb_buffer_pool_size` and `inpage_size` is short for `innodb_page_size`.

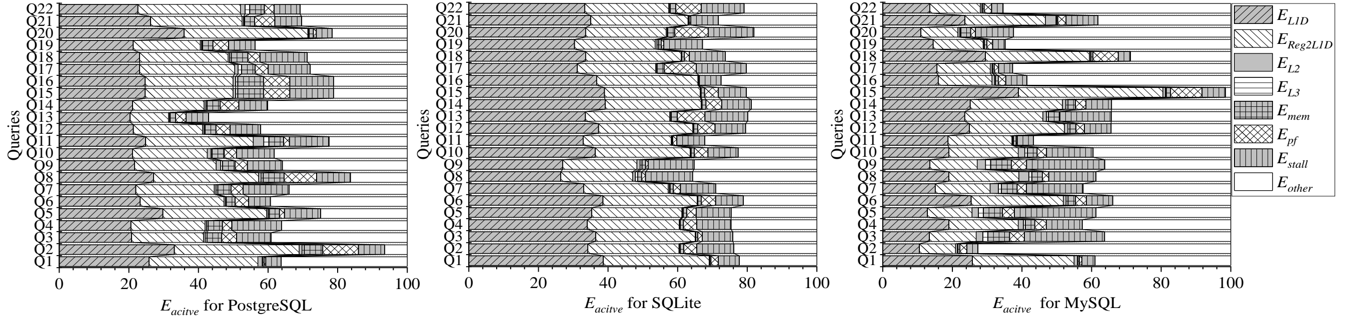

Figure 7: Active energy cost breakdown of TPCH for three different database systems.

written in L1D cache but they little have to be persisted due to the write-back strategy. In our experiment, the store operations are frequently issued by 7 basic query operations, being about 66% of the load operations but 99.86% of them occur at L1D cache.

In summary, although the implementation of three database systems has impact on the energy cost distribution, L1D cache load/store is still their energy bottleneck. In addition, the query workload is read-only operation, but the energy cost of temporary data write is still high.

3.3 Energy Cost of TPCH

We profile the energy cost of TPCH with the baseline data size on the baseline setting. 79.2%-88.7% Busy-CPU energy cost can be broken down. As shown in Figure 7, the energy cost of data movement is 65% for PostgreSQL, 75% for SQLite and 55% for MySQL. In addition, the energy cost distributions of three database systems are similar. The percent of E_{L1D} + $E_{Reg2L1D}$ is so attractive, 46.8% for PostgreSQL, 60% for SQLite and 38.6% for MySQL. The phenomenon is like it in basic query operations because complex queries are the combination of basic operations.

Sequential scanning. For SQLite, the percent of E_{L1D} + $E_{Reg2L1D}$ is higher than that of two other database systems either in the TPCH or the basic query operations because SQLite tends to the sequential scanning. Actually, the energy cost of sequential scanning prefers to L1D cache. We take an example to illustrate the relationship between sequential scanning and the energy cost of L1D cache. As shown in Figure 6, the difference of both index scan and table scan is scan table using the index (B tree) or not. Obviously, index scan utilizes the pointer chasing to reorganize data causing the relatively weak data locality. Table scan tends to the sequential scanning. Without exception, the percent of E_{L1D} + $E_{Reg2L1D}$ reduces and the percent of E_{stall} increases for index scan compared with table scan.

Similarly, SQlite as the mobile database is usually used to manage the small-scale data so that it does not involve many complex optimization strategies, such as the hash join. The main data access method is sequential scanning. It is reasonable because the hardware optimization is more important than software optimization for the small-scale data. The sequential scanning is easily sped up by the hardware optimization, such as speculation and out-of-order execution. It will lead to the less stall cycles. For SQLite, the present of E_{stall} is 12% lower than two other database systems, showing the good performance of sequential scanning. For optimizing performance on large-scale data, we think that both PostgreSQL and MySQL may construct the complex data structure and reorganize the data, such as the compact buffer management. These optimizations can improve the performance, e.g., average 3.31× faster than SQlite in our experiment. However, they will introduce the extra calculations, and they also hinder hardware optimization due to the weak data locality, leading to the low percent of E_{L1D} + $E_{Reg2L1D}$.

Impact of data size. We evaluate the energy cost distributions of three database systems with different data sizes (100MB, 500MB and 1GB) on the baseline setup. As shown Figure 8, we only illustrate the average energy cost result of 22 queries in TPCH as a vector due to the space limitation and PG is short for PostgreSQL. As the data size increases, the energy cost distributions of three database systems have not changed significantly. We also analyze the energy cost change of every query and find that E_{stall} of 14% queries is improved by 2× when increasing the data size. The large data size may lead to frequent swapping in and out of data pages and CPU stall. In general, the L1D cache load/store is still the energy bottleneck which is hardly affected by the data size.

Impact of different settings. As shown in Figure 9, we also compare the impact of different database settings being from Table 4. For MySQL, E_{stall} is reduced when the *large* setting provides more memory to fit data pages but PostgreSQL and SQLite are not sensitive to different settings. It still suggests that different settings have little impact on the energy cost distribution.

In summary, sequential scanning is the basic behavior of query workloads. For different database system implementations, the dependency of sequential scanning affects the energy allocation for L1D cache. Data size and database settings do not lead to substantial changes in this energy cost distribution. So, our findings could be general for query workloads of database systems.

3.4 Comparison to CPU-Bound Workloads

With the optimization of database systems and the performance improvement of the disk, the database system has tended to be CPU-bound from disk-bound. For example, CPU usage is 96% and IPC=1.9 showing a busy CPU, when running TPCH in our experiment. In this section, we compare the energy cost distributions of query workloads with the energy cost distributions of typical CPU-bound workloads. As shown in Figure 10, we evaluate the energy cost of 9 workloads in the classic CPU benchmark CPU2006-v99[2], involving the compression, compiling and simulation workloads, etc. Not like the query workloads, the energy cost distributions of typical CPU-bound workloads are not similar to each other. For three database systems, the percent of E_{L1D} + $E_{Reg2L1D}$ of 76% queries is greater than 40%, but the percent is only 11% for CPU2006. For some extreme CPU2006 workloads (Mcf and Libquantum), E_{L1D} + $E_{Reg2L1D}$ is so low at only 5.6%, but this behavior does not occur in query workloads.

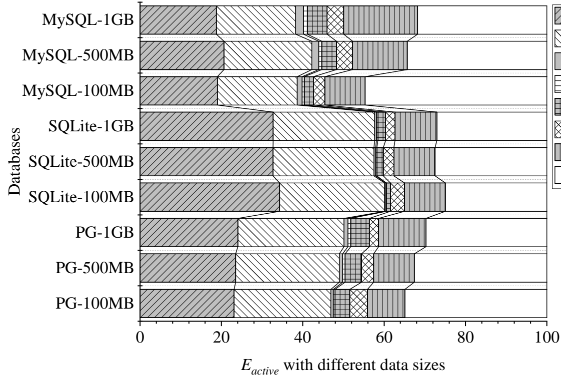

Figure 8: Impact of data size.

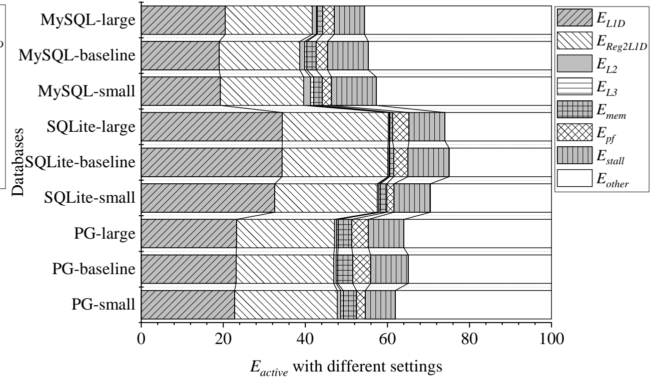

Figure 9: Impact of database setting.

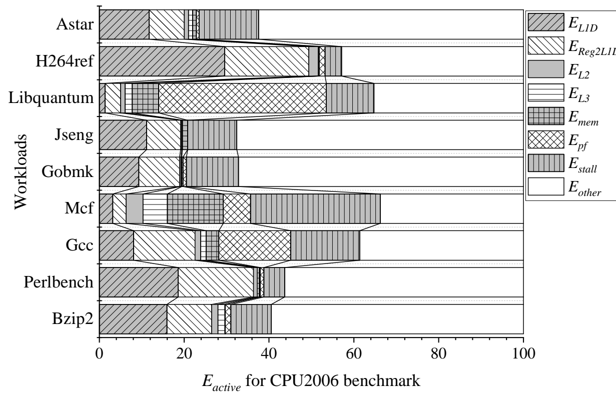

Figure 10: Energy cost breakdown of CPU2006.

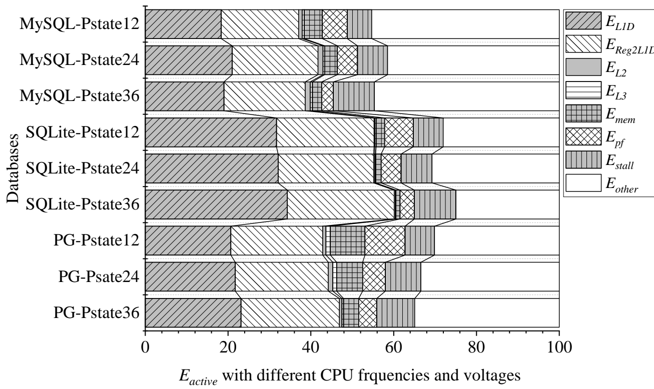

Figure 11: Impact of CPU frequencies and voltages.

In summary, we think that the energy cost pattern of query workloads is totally different from the typical CPU-bound workloads and it could be unique to query workloads.

3.5 Impact of CPU Frequency and Voltage

In the real scenario, the EIST knob is usually turned on, so that we in this section explore the impact of different CPU frequencies and voltages on the energy cost breakdown. We select two other P-states: P-state 24 and P-state 12 to first evaluate energy cost of micro-operations and then break down energy cost of three databases under baseline knob settings and baseline data size.

As shown in Table 2, the energy cost of micro-operations under low P-states will definitely reduce. The closer the micro-operation is to the CPU core, the more significantly the energy cost decreases. For example, ΔE_{L1D} reduces by 53.8% from P-state 36 to P-state 12, but 3.9% for ΔE_{mem}.

As shown in Figure 11, we compare the energy cost breakdown of three databases at different P-states. In our experiment, E_{active} decreases by 32%±2% at P-state 24 and 51%±1% at P-state 12, but the energy cost breakdown has little impact due to the lower energy cost of micro-operations. In detail, because ΔE_{mem} has little change at different P-states, the percent of both E_{mem} and E_{pf} (involving main memory accessing) have a significant improvement at P-state 12, about 2× and 2.2 × compared with P-state 36. However, the absolute impact is still little. Actually, different P-states cannot change the query runtime characteristics, so that the L1D cache load/store hit rate is still high, only leading to the slight reduction of the percent of $E_{L1D} + E_{Reg2L1D}$ at low P-states. For example, the percent of $E_{L1D} + E_{Reg2L1D}$ at P-state 12 only decreases by 4%-8.6% for three databases, compared with it at P-state 36. Our result shows that the L1D cache load/store operations are still the energy cost bottleneck at different CPU frequencies and voltages.

In essence, this experiment reveals the energy cost profiling for typical query runtime characteristics at different CPU frequencies and voltages. For other query scenarios, the CPU could not always be at the high P-state when turning on the EIST knob, such as real-time query workloads. However, their runtime characteristics should be similar to those shown in this paper. So, we think that this energy cost bottleneck could be general for many query scenarios even if turning on the EIST knob.

In summary, the low CPU frequency and voltage will lead to the low energy cost of micro-operations, but the L1D cache load/store operations are still the energy cost bottleneck.

4 PROOF-OF-CONCEPT SYSTEM

In the section, we will discuss the customized CPU architecture design which can enable the energy-efficient database systems. The optimization and evaluation on SQLite will be given.

4.1 L1D Energy-Efficient CPU Architecture

L1D cache load/store is the energy bottleneck, but their optimization is difficult on typical x86_64 architecture because L1D cache has the lowest energy cost than other memory layers.

For database systems, a good energy-efficient CPU architecture should provide the lower energy cost L1D cache (i.e., Arch

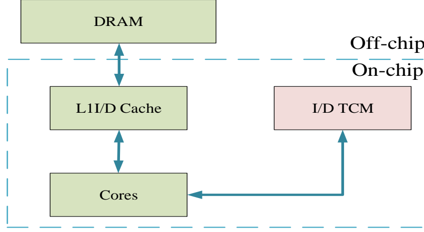

Figure 12: ARM1176JZF-S architecture as an L1D energy-efficient CPU architecture.

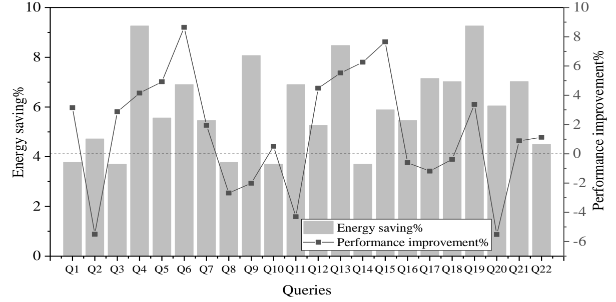

Figure 13: Energy saving and performance improvement for SQLite using DTCM or not on ARM1176JZF-S.

1) or provide a piece of the low energy cost memory at the same speed as L1D cache (i.e., Arch 2). For Arch 1, users can transparently migrate the database systems on it. For Arch 2, users need to update the kernel of database systems to decide what data to put into the low energy cost memory. We investigate many CPU architectures and only find the architecture similar to Arch 2. As shown in Figure 12, ARM1176JZF-S[1] supports 16KB L1D cache, 32KB DTCM (Data Tightly Coupled Memory) and 256MB main memory. TCM is the programmable on-chip memory which is as fast as L1 cache but its energy cost is lower than L1 cache. So, ARM1176JZF-S could be as an L1D energy-efficient CPU architecture. In this section, we will use the DTCM load instead of the L1D cache load to reduce Busy-CPU energy cost.

4.2 System-Level Co-Design

We will optimize SQLite because, as a mobile database it can work well on ARM architecture, but the optimization is still difficult. First, although Linux supports many hardware environments, it cannot be directly compiled in this ARM environment. The ARM hardware environments are so diverse that Linux can only identify some mainstream architectures and it does not support ARM1176JZF-S well. We have to modify and update Linux kernel to make it support TCM, Linux perf and cross compiling. Second, we have to implement the TCM driver and API enabling that TCM can be accessed in user space. It took us about 2 months to build an available runtime environment. For SQLite, our optimization strategies are as follows.

Database buffer. We allocate 16KB DTCM for database buffer, which will be dynamically managed by SQLite.

Special variables. We use the Linux perf to profile the SQLite's runtime and find that about 70% L1D cache load operations are issued by the `sqlite3VdbeExec()` function to execute the query plan. This phenomenon in x86_64 architecture is similar to it in ARM architecture. We allocate 4KB DTCM and put some key structures of `sqlite3VdbeExec()` in it, such as query plan (`Vdbe`), meta data (`Vdbe->db`), cursor (`Vdbe->apCsr` and `Vdbe->apCsr->aOffset`), head address of heap space (`Vdbe->aOp` and `Vdbe->aMem`), etc.

B tree. Every table in SQLite is organized as a B tree. The primary key or row ID will be the key of B tree. So, the top layers of B tree will be frequently read. Based on this, we allocate 12KB DTCM to put the root and first few layers of B-tree of current tables into DTCM. We divide DTCM memory evenly according to the number of tables being queried. In this way, we can ensure that more B tree data of small tables are loaded into DTCM.

Noting that our strategies are for L1D cache load operation. The energy cost optimization of L1D cache store operation is more difficult. We do not discuss it in this paper.

4.3 Evaluation Results

ARM does not support RAPL, so that we use the external power meter to measure the energy cost. We first design a micro benchmark B_DTCM_array, which only loads data from DTCM. It is similar to Algorithm 1 but allocates memory from DTCM, instead of main memory. Compared with B_L1D_array, the energy cost of B_DTCM_array can reduce by 10% with no performance loss. Therefore, 10% will be as the peak energy saving of DTCM in our experimental environment.

In Figure 13, we show the energy saving and performance improvement of the optimized SQLite. Noting that we compare whether SQLite uses DTCM on ARM, not SQLite on Intel CPU or ARM CPU. We use 10MB TPCH data and the `small` setting to take our experiment. Our optimization makes SQLite save average 6% energy and improve average 1.5% performance. It means that our approach can achieve 60% of the peak energy saving. The dominant factor is that accessing the hot data in DTCM can bypass the L1D cache leading to L1D cache energy saving. In addition, the performance of 64% queries can be further improved due to the avoidance of hot data misses. For the non-optimized SQLite, 25% L1D miss rate in ARM1176JZF-S could cause the hot data to swap in and out the L1D cache from main memory. However, DTCM has the fixed physical address, so that the hot data in DTCM is not loaded from main memory.

Advantages. Limited to the hardware implementation, 6% energy saving in our experiments may seem to be less, but our approach has three advantages as follows. First, compared with the existing approaches, our approaches can save energy with no performance reduction. It is advantageous for energy-strict real-time tasks, such as UPS-powered data centers and databases in smart phones. Second, our approach is orthogonal to existing approaches. Our approach tends to save energy from the view of CPU architecture, so that it can work together with application-level energy optimization approaches, such as energy-oriented query optimization and DVFS-based approaches. Third, our approach is depend on the implementation of TCM. So, these results in our paper only suggest that our approach can achieve 60% of the peak energy saving, and do not mean the final energy-saving potential. The existing work shows that the optimized TCM has got 40% energy saving compared with L1D cache[9]. If integrating such an optimized TCM into ARM1176JZF-S, our approach should get a maximum 24% energy saving.

In summary, our optimized SQLite can achieve 60% of the peak energy saving. It can also achieve 1.5% performance improvement due to the avoidance of hot data misses.

Table 5: Energy cost bottleneck of B_mem at different CPU frequencies and voltages

P-state	36 (3.6GHz)	24 (2.4GHz)	12 (1.2GHz)
Micro-operations	Energy cost (J) / Percent%		
E_{mem}	295.4 (16.6%)	284.2 (29.8%)	284.6 (47.4%)
E_{stall}	**1416.1 (79.8%)**	**630.6 (66.2%)**	**310.1 (51.6%)**
E_{active}	1772.5 (100%)	952.9 (100%)	600.5 (100%)

5 POTENTIAL OPTIMIZATIONS

Based on the Busy-CPU energy breakdown results, we also find some additional interesting energy cost phenomena in Figure 6 and 7 that suggests other optimization approaches.

In index-intensive scenario, especially for index scan of PostgreSQL and MySQL, the percent of both E_{mem} and E_{pf} becomes prominent, also causing a high E_{stall}. Similar phenomenon can also be seen in PostgreSQL's basic query operations compared with them of two other database systems. In addition, as the data size increases, the main memory access is also more frequent, especially for MySQL. They imply a memory-bound tendency. In this section, we will focus on the energy cost optimization of memory-bound workloads. Actually, we think that improving main memory performance or radically lowering CPU frequency and voltage are efficient ways to save energy.

To explain our idea, we first profile the energy cost of a typical memory-bound workload under different CPU frequencies and voltages. The micro-benchmark B_mem is a typical memory-bound workload and we break down its energy cost shown in Table 5. Interestingly, such a slight change to ΔE_{mem} does not imply that the low P-state will cause the energy cost to increase for memory-bound workloads, although the elapsed time may be increased. Actually, B_mem's performance bottleneck is main memory, the energy cost bottleneck is the CPU (E_{stall}, not E_{mem}). This result suggests that the energy cost bottleneck is in the CPU, even if for non-CPU bound workloads. So, the ultra-linear decrease in E_{stall} causes a reduction in E_{active} with slight performance loss. For example, B_mem only trades 7% performance loss for 46% E_{active} saving when lowering P-state from 36 to 24. The energy-efficiency ($\frac{Perf}{Energy}$)[14] is improved by 70%. In addition, EIST cannot work well on memory-bound workloads. When turning on the EIST knob, the percent of P-state 36 is 98.6% due to the high CPU load (99.8% CPU usage), implying failure of dynamic energy saving.

Actually, for memory-bound query scenarios, an energy-saving chance is to reduce E_{stall}. Improving main memory performance tends to reduce N_{stall} or radically lowering CPU frequency and voltage tends to reduce ΔE_{stall}. We take a preliminary experiment on PostgreSQL's index scan to confirm the second approach. When lowering P-state from 36 to 24, PostgreSQL's index scan only trades 20% performance loss for 27% E_{active} saving, showing that the energy-efficiency is improved by 10%. However, our strategy is not trivial. PostgreSQL's table scan, a CPU-bound workload, has to trade 30% performance loss for 28% E_{active} saving, i.e., the energy-efficiency is reduced by 3%. So, a customized DVFS approach is expected for memory-bound query scenarios. It should analyze the query plan, such as index-intensive or not, and monitor the main memory access to employ a more radical DVFS strategy.

The percent of MySQL's E_{other} is higher than that of two other databases, especially for basic query operations, so that energy-efficient calculation components or instruction-related components, e.g., instruction TCM (ITCM), should be considered.

6 RELATED WORK

Energy characterization and optimization in database systems is the basic work to design an energy-efficient database systems. There are extensive researches on this topic from different aspects, including (1) macro energy cost breakdown, (2) trade-off based energy optimization and (3) employing TCM. However, we take a micro analysis of the Busy-CPU energy cost of database systems and then provide a customized CPU architecture to enable the energy-efficient database systems.

Macro energy cost breakdown. The energy-efficient database design is systematically reported in [14]. From then on, the researchers had made much effort to make sense of its energy bottleneck. Research work focuses on the breakdown of the energy cost of the major resources, i.e., the CPU, main memory and disk, on various of system architectures. For the classic x86_64 architecture with local disk, the CPU is identified as the energy cost for disk-based database systems[19, 25] and in-memory database systems[13]. For the architecture of ARM+RDRAM (Rambus DRAM), the main memory is identified as the bottleneck[23]. For the system with the remote disk array, the disk is the energy bottleneck[24]. These above conclusions are difficult to explain whether the power is consumed by the database system or by the hardware itself because the measurement of the total energy cost contains the Background energy cost and Idle-CPU energy. Our work divides the Busy-CPU energy cost into the Active energy cost and the Background energy cost, and only profile the Active energy cost which is consumed by database systems. In addition, the existing work focuses on the macro energy breakdown of major resources. However, the main drawback is that they cannot give the clear optimization suggestions on hardware architecture due to the lack of the fine-grained information. We study the micro energy analysis inside CPU and have the ability to make sense of microscopic energy cost distribution. It is helpful to design the CPU architecture for energy-efficient database systems.

Trade-off based energy optimization. For the x86_64 architecture, the CPU has been identified as the energy bottleneck for database systems. Because the energy cost distribution inside the CPU is unknown, the existing work sees the CPU as an inseparable whole to design the energy-oriented query optimizer or tune the external DVFS knobs. Inspired by the performance-oriented query optimizer, PET [28, 29] as an energy-aware query optimization constructs a cost model to choose the low-energy query plans under a DBA-specified energy/performance trade-off level. QED [21] uses query aggregation to leverage common components of queries to reduce energy cost. These query optimization techniques are used to gracefully trade response time for energy. DVFS knobs can be configured by users to trade the voltage and CPU frequency for energy. It provides the chance to optimize the energy. PVC[21] and sweet spots[13] attempt all of combinations between the voltage and CPU frequency for a specific workload to choose one combination which can minimize the energy cost. Their drawback is hard to apply to all queries. Other approaches leverage feedback-control loops to dynamically set DVFS knobs using the load profile and can obey a query

latency limit as a soft constraint. Based on this idea, lots of work has achieved the adaptive energy control for various of databases, such as the disk-based transaction-oriented DBMS [20, 26, 30] and data-oriented in-memory DBMS [19]. In order to get a good energy-saving effect, these techniques expect a relaxed run-time constraint, so that they in essence trade the performance for the energy. Through our micro analysis of the Busy-CPU energy cost, we have found the micro-operation level energy bottleneck. With the help of the L1D cache energy-efficient CPU architecture, we can cut down the energy cost of database systems with slight performance improvement.

Employing TCM. TCM as on-chip memory is usually used for performance improvement by combining the micro performance feature of applications, such as digital signal processing[12], MapReduce framework[16] and embedded multi-media[15]. However, we attempt to use TCM to reduce the energy cost, combining the micro energy feature of database systems. The main idea of our optimization is to put the hot data into TCM to save the energy. The similar idea for TCM has appeared. They at compile-time analyze a given piece of program, identify the hot data and put them into TCM at runtime to reduce the energy cost of data movement, such as a SPM management framework for nest-loop[17], a data program relationship graph for global and stack variables[27] and heap data[11]. Facing the complex database systems, these program-level optimization methods under specific premises may not work. Actually, our optimization is system-level. We profile the runtime behavior of database system, review its source code and elaborately identify the hot data. Although the optimized SQLite is only a proof-of-concept system, it confirms that our method is feasible. Providing the customized CPU architecture is a possible way to enable energy-efficient database systems.

7 SUMMARY & FUTURE WORK

In this paper, we propose a novel idea to reduce the energy cost based on profiling the energy cost of CPU micro-operations for databases systems. Our approach can break down the majority of Busy-CPU energy cost and isolate the accurate energy cost of data movement. The CPU energy breakdown method exposes that L1D cache load/store is the energy bottleneck of database systems. The finding supposes that we may achieve energy efficiency by adopting a customized CPU architecture with lower L1D energy cost. TCM can meet this requirement well and an optimized system-level co-design solution for SQLite is implemented to evaluate the proposed idea. The experimental result of the proof-of-concept system shows that our method can achieve 60% of the peak energy saving with further performance improvement.

In future, we will try to profile the energy cost of other typical database systems, such as NoSQL systems to identify their energy distribution feature on CPU and check if our method can be employed into more type of database systems.

8 ACKNOWLEDGEMENT

This research was partially supported by the grants from the Natural Science Foundation of China (No. 61532016, 91646203, 91846204, 61532010, 61762082).

REFERENCES

[1] 2019. ARM11. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf.
[2] 2019. CPU2006. http://www.spec.org/cpu2006/.
[3] 2019. Intel prefetcher. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.
[4] 2019. L2 prefetcher behavior. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/703019.
[5] 2019. Linux Perf. http://www.brendangregg.com/perf.html.
[6] 2019. ocperf. https://github.com/andikleen/pmu-tools.
[7] 2019. P-state and C-state. https://software.intel.com/en-us/blogs/2008/03/12/c-states-and-p-states-are-very-different/.
[8] 2019. TPCH. http://www.tpc.org/tpch/.
[9] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 2002 International Symposium on Hardware/Software Codesign*. 73–78.
[10] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design*. 189–194.
[11] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. 2005. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing* 1, 4 (2005), 521–540.
[12] Syed Z Gilani, Nam Sung Kim, and Michael Schulte. 2011. Scratchpad memory optimizations for digital signal processing applications. In *Proceedings of 2011 Design, Automation & Test in Europe*. 1–6.
[13] Sebastian Götz, Thomas Ilsche, Jorge Cardoso, Josef Spillner, Thomas Kissinger, Uwe Aßmann, Wolfgang Lehner, Wolfgang E Nagel, and Alexander Schill. 2014. Energy-efficient databases using sweet spot frequencies. In *Proceedings of the 2014 IEEE/ACM International Conference on Utility and Cloud Computing*. 871–876.
[14] Stavros Harizopoulos, Mehul Shah, Justin Meza, and Parthasarathy Ranganathan. 2009. Energy efficiency: The new holy grail of data management systems research. In *Proceedings of the 2009 biennial Conference on innovative data systems research*. 81–90.
[15] Wen Shu Hong, Hui Juan Cui, and Kun Tang. 2005. Scratchpad Memory Assignment in Embedded Multimedia Application. *Acta Electronica Sinica* 33, 11 (2005), 1937–1940.
[16] Christoforos Kachris, Georgios Ch. Sirakoulis, and Dimitrios Soudris. 2015. A MapReduce scratchpad memory for multi-core cloud computing applications. *Microprocessors & Microsystems* 39, 8 (2015), 599–608.
[17] Mahmut Kandemir, Jagannathan Ramanujam, Mary Jane Irwin, Narayanan Vijaykrishnan, Ismail Kadayif, and Amisha Parikh. 2001. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Design Automation Conference*. 690–695.
[18] Gokcen Kestor, Roberto Gioiosa, Darren J Kerbyson, and Adolfy Hoisie. 2013. Quantifying the energy cost of data movement in scientific applications. In *2013 IEEE international symposium on workload characterization*. 56–65.
[19] Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. 2018. Adaptive energy-control for in-memory database systems. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of data*. 351–364.
[20] Mustafa Korkmaz, Alexey Karyakin, Martin Karsten, and Kenneth Salem. 2015. Towards Dynamic Green-Sizing for Database Servers.. In *Proceedings of the 2015 International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*. 25–36.
[21] Willis Lang and Jignesh M Patel. 2009. Towards Eco-friendly Database Management Systems. *Proceedings of the 2009 biennial Conference on innovative data systems research* (2009).
[22] Dhinakaran Pandiyan and Carole Jean Wu. 2014. Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms. In *IEEE International Symposium on Workload Characterization*. 171–180.
[23] Jayaprakash Pisharath, Alok Choudhary, and Mahmut Kandemir. 2004. Reducing energy consumption of queries in memory-resident database systems. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. 35–45.
[24] Meikel Poess and Raghunath Othayoth Nambiar. 2008. Energy cost, the key challenge of today's data centers: a power consumption analysis of TPC-C results. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1229–1240.
[25] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A Shah. 2010. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 231–242.
[26] Yi-Cheng Tu, Xiaorui Wang, Bo Zeng, and Zichen Xu. 2014. A system for energy-efficient data management. *ACM SIGMOD Record* 43, 1 (2014), 21–26.
[27] Sumesh Udayakumaran and Rajeev Barua. 2003. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. 276–286.
[28] Zichen Xu, Yi-Cheng Tu, and Xiaorui Wang. 2010. Exploring power-performance tradeoffs in database systems. In *Proceedings of the 2010 IEEE International Conference on Data Engineering*. 485–496.
[29] Zichen Xu, Yi-Cheng Tu, and Xiaorui Wang. 2012. PET: reducing database energy cost via query optimization. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1954–1957.
[30] Zichen Xu, Xiaorui Wang, and Yi-cheng Tu. 2013. Power-aware throughput control for database management systems. In *Proceedings of the 2013 International Conference on Autonomic Computing*. 315–324.