Quantifying the Energy Cost of Data Movement in Scientific Applications

Gokcen Kestor*, Roberto Gioiosa*, Darren J. Kerbyson*, Adolfy Hoisie*
* Pacific Northwest National Laboratory
{gokcen.kestor, roberto.gioiosa, darren.kerbyson, adolfy.hoisie}@pnnl.gov

***Abstract*—In the exascale era, the energy cost of moving data across the memory hierarchy is expected to be two orders of magnitude higher than the cost of performing a double-precision floating point operation. Despite its importance, the energy cost of data movement in scientific applications has not be quantitatively evaluated even for current systems.**

In this work we present a methodology to estimate the energy cost of moving data across the memory hierarchy for scientific applications on state-of-the-art compute node systems. To this extent, we implement a set of highly-tuned micro-benchmarks that move data from a determined level of the memory hierarchy to the processor's registers. We then correlate the data movement latencies with external and internal power measurements to determine the energy cost of moving data.

We use this information to characterize current and future scientific applications from the DoE Exascale Co-Design Centers and Office of Science and estimate the percentage of application's energy that goes into data movement. Our results on current systems show that 28-40% of the total energy cost is spent in moving data, 19-36% of the energy is wasted in stalled cycles and that data prefetcher may waste energy by moving unnecessary data to the processor's caches.

I. Introduction

The energy cost of powering a supercomputer is rapidly approaching the machine's acquisition cost. This energy cost will keep increasing in the future, hence next generation exascale systems need to be considerably more power- and energy-efficient than current supercomputers to be of practical use. Unlike petascale systems, where the primary concern was performance, exascale systems need to climb the power and energy walls in order to deliver sustainable exaFLOPS performance. In other words, the scalability of exascale systems will be limited by their power and energy consumption.

Among the major energy limiting factors, several studies [11] and workshop reports [3] point out that the ratio between the energy cost of moving data from memory to registers and the energy cost of performing a register-to-register floating point operation is quickly increasing. This trend is bound to increase, as future systems will approach near-threshold-voltage (NTV) operation levels: the energy consumption of a double precision register-to-register floating point operation is expected to reduce by 10x by 2018 [3], [8]. The energy cost of moving data from memory to processor is not expected to reduce by as much, hence the relative energy cost of data movement with respect to performing a register-to-register operation will increase (*energy wall* [3]).

Despite its importance in current and future systems, the impact of data movement on the energy consumption of scientific applications has been only qualitatively evaluated [11]. The cost of data movement is often quoted as being the major component of the total energy consumption but very little has been done to quantify it on current systems. Previous studies attempted to characterize the power and energy profiles of HPC applications but they often rely on simulated environments [8] or specific hardware instrumentation that is not commonly available and/or too expensive to be applied to machines with a large number of nodes [11]. While these studies provide essential information, such as the energy cost of a specific operation, they do not answer important questions such as what is the amount of energy spent in data movement with respect to the total energy consumption or what is the dominant component of data movement energy for current and future parallel applications.

In this work we perform a quantitative analysis of the energy cost of data movement in scientific applications commonly executed on large cluster systems and representative of exascale workloads. To this extent, our evaluation methodology consists of using a set of micro-benchmarks that continuously access data stored in a given level of the memory hierarchy, e.g., the L2 cache. We then correlate the operations performed by the micro-benchmarks with external and internal power measurements obtained from a power meter connected to the compute node power supply and the processor's internal power sensor [7], [19]. From these samplings we compute the energy consumption of each micro-benchmark and then derive the energy cost of each load operation. Finally, we follow an incremental-step approach to estimate the energy cost of moving data between any two levels of the memory hierarchy.

Although measuring the energy consumption of a single operation by using micro-benchmarks may appear straightforward, an accurate estimation of the energy costs of data movement on a real system requires the use of a combination of measuring techniques. Both architectural features (such as memory prefetching and speculation) and compiler optimizations make it difficult to isolate the energy cost of a single load instruction that retrieves data from one of the cache levels or the main memory with a single experiment. For example, the energy consumption of the processor's fans, circuitry and memory should be properly evaluated and subtracted from the energy cost of each micro-benchmark. Similarly, the cost of speculation, resolving data dependencies and/or scheduling of out-of-order instructions should be isolated and removed from the energy cost of data movement. We evaluate these extra contributions to the total energy consumption with different versions of the same micro-benchmark: for example, by varying the memory access pattern stride we can enable or disable the memory prefetcher. By properly combining experiments and by running different versions of the same micro-benchmarks, we are able to estimate the energy cost of stalled cycles, memory prefetching, processor's fans, uncore components, etc. Once these extra components are known, we can isolate the energy cost of each single data movement instruction, memory to L3 cache, L3 to L2 cache, L2 to L1 cache, L1 cache to processor's registers. To the best of our knowledge, this is the first study that provides such level of detail on a real system without using additional expensive power instrumentation and/or relying on simulated environments.

In order to validate our methodology, we build micro-benchmarks that perform register-to-register operations (such as `add` instructions) and `nop` operations and evaluate the energy cost of each operation. We combine register-to-register and data movement instructions into test micro-benchmarks and compare the estimated energy cost of each test micro-benchmark to the energy cost derived from external measurements. Our results show very high accuracy (generally within 1-2%) with only one exception for which we underestimate the energy cost by 5.5%.

We use the estimated cost of moving data to compute the total energy cost of data movement of HPC benchmarks from the NAS Benchmark suite (NPB) [5] and scientific applications from the Exascale Co-Design Centers [1], *NEK-Bone* and *LULESH*, and the DoE Office of Science, *GTC*. Our results, performed on a AMD Opteron Interlagos, show that the energy cost of data movement impact differently on each application, ranging from 18% to 40%. This percentage might increase in the future, as the energy cost of performing computation decreases. To avoid such scenario, new technologies, such as Processing-In-Memory [6], [10], Non Volatile RAM (NVRAM) [12] and 3D stack memory [9], [16], becomes essential for the development of sustainable exascale computing. We also show that the energy spent in resolving data dependency, speculation and out-of-order scheduling of instructions (stalled cycles) accounts for a considerable part of the total dynamic energy, between 22% and 35%. This cost can be reduced with simpler processor core designs that are more energy efficient.

Finally we show how the energy of data movement is spent among the different levels of the memory hierarchy. Our experiments show that, for well-optimized and regular applications, the memory prefetcher is capable of proactively bringing data from memory to the processor's cache. While this cost still accounts to the cost of data movement (25% for optimized applications), performance of applications with high data locality and regular memory access patterns generally benefits from memory prefetching. For other applications, instead, the memory prefetcher may bring unnecessary data to the processor's cache: this data will reside in the processor's cache (eventually evicting useful cache lines) but will not be moved to registers. This waste of energy can be reduced with more effective memory prefetchers or by dynamically disabling memory prefetching if performance improvement is overly expensive in terms of energy.

In summary, this paper makes the following contributions:

- We present a methodology to accurately estimate the energy cost of data movement on real systems.

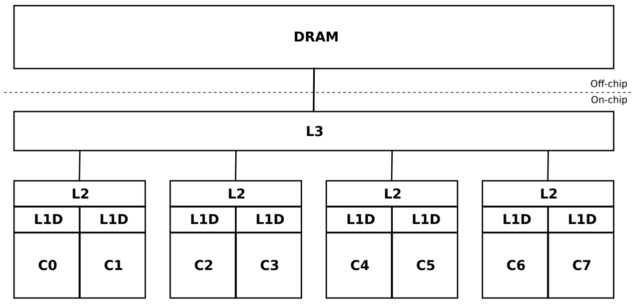


Fig. 1: Memory hierarchy for a generic multicore system.

- Using our methodology, we compute the energy cost of moving data across the memory hierarchy for a state-of-the-art compute node system.
- We analyze the energy cost of data movement and stalled cycles for scientific applications from NPB suite, DoE Exascale Co-Design Centers and DoE Office of Science.

The rest of this paper is organized as follows: Section II introduces background; Section III provides details about the design and implementation of our micro-benchmarks; Section IV describes our power and energy measurement techniques; Section V validates our methodology and analyzes the energy cost of data movement for scientific applications. Section VI lists related work and Section VII concludes this work.

II. Background

Measuring the energy cost of a specific instruction on a real system is a complicated task. Previous work focused on measuring power of sub-system components (CPU, memory, I/O, etc.) eventually using custom external instrumentation [11], [17]. The energy consumption of each sub-system can then be derived from the power profile of that sub-system and the elapsed time between two consecutive power samples. This approach provides only qualitative measurement of the power and energy cost of the memory sub-system and does not provide insights in the energy cost of moving data from one level of the memory hierarchy to the next one. In fact, modern out-of-order architectures overlap instructions to hide memory latency and data dependencies, which affects power and time measurements. Hardware vendors can perform fine-grained power and energy measurements, usually on prototype systems or simulation environment [8]. These measurements report the energy cost of each operation but do not specifically focus on a particular workload and are performed on systems that are ever so slightly different from the commercial versions of the same products. While the energy cost of an operation per se is an important information (usually made available by hardware vendors), in this work we are interested in the overall energy cost of data movement for exascale applications on real, commonly-available compute nodes.

Modern processor architectures feature a hierarchical memory structure, similar to the one depicted in Figure 1. This means that accessing data stored in memory implies moving data across all the cache hierarchy levels: from memory to the L3 cache, then to the L2 cache, the L1 cache and, finally, the processor register. In order to accurately measure the energy cost of data movement, we follow a incremental-step methodology in which we start from analyzing the cost of a register-to-register operation and move up in the hierarchy until we reach the DRAM memory. To isolate the cost of accessing data store in a specific level of the memory hierarchy, we design a set of highly-tuned micro-benchmarks that continuously access data in a specific level of the memory hierarchy. We then correlate our experiments with power readings obtained from an external power meter and internal processor power sensors and compute the energy cost of each data movement operation.

III. Micro-Benchmarks Workload

Isolating the cost of a specific data movement instruction with an elaborated benchmark is complicated on modern architectures. Out-of-order execution and speculation help hiding memory latency but also makes it difficult to measure the energy contribution of single instructions. To be able to perform accurate energy measurements, we design a new set of well-engineered micro-benchmarks that minimize the effects of speculation, out-of-order execution and other optimizations. All micro-benchmarks follow the same general structure:

```
MB_init();
for(i=0; i<N; i++){
    UNROLL_X{
        <body-loop>
    }
}
MB_finit();
```

The central part of each micro-benchmark is the `<body-loop>`, where the micro-benchmarks perform the same operation (e.g., a load instruction that retrieves data from the L2 cache) `N` times. `N` is chosen to be large enough to minimize the impact of initializing/finalizing the micro-benchmark. This value depends on the particular benchmark, as each operation in the body loop has a specific latency. The value of the loop unrolling `X` is large enough to minimize the impact of the control operation (`i<N`) and the loop counter increment (`i++`). We set `X=100`, which means that we tolerate an error $< 3\%$ (one control flow operation, one jump and one integer increment every 100 operations).

Although the micro-benchmarks follow the same general structure, each micro-benchmark differs from the others because of 1) the particular operation(s) executed in the `body-loop`, and 2) the input set used during the execution. For this study we designed the following micro-benchmarks:

- `MB_L1`: continuously accesses data stored in the L1 cache (`load`). We use pointer-chasing to obtain the address of the next memory location that should be accessed. We initialize and pre-load the array containing the addresses of the memory location to be accessed during the initialization phase (`MB_init()`).
- `MB_L1asm`: assembly implementation of `MB_L1`. It does not present any dependency among consecutive loads from the L1 cache (all data dependencies are preventively resolved when writing the benchmark).
- `MB_L2`: continuously accesses data stored in the L2 cache. The structure of the benchmark is similar to `MB_L1`, except that `MB_L2` always misses in the L1 cache when accessing the next memory location.
- `MB_L3`: behaves similarly to `MB_L1` and `MB_L2` but every access causes a miss in the L1 and L2 cache and a hit in the L3 cache.
- `MB_MEM`: continuously accesses memory locations that are not stored in any level of the cache hierarchy. The internal behavior is the same as the previous benchmarks but every memory access misses in all levels of the cache hierarchy, hence data is moved from memory to the processor's register.
- `MB_MEM64`: performs the same operations like `MB_MEM`, except that it uses 64 bytes as the memory stride size rather than 512 bytes.
- `MB_iADD`: performs register-to-register `add` operations. The input registers are pre-loaded during the initialization phase. We used this micro-benchmark primarily for validation.
- `MB_NOP`: performs `nop` operations. We used this micro-benchmark in Section V-A to validate our methodology.

TABLE I: Runtime characteristics of micro-benchmarks.

Micro benchmark	Body-loop Instruction %	L1 Miss Rate	L2 Miss Rate	L3 Miss Rate	IPC
MB_NOP	99.98	-	-	-	3.86
MB_iADD	99.92	-	-	-	2.03
MB_L1	99.89	0.03	0.01	0.01	0.25
MB_L2	99.12	99.96	0.13	0.12	0.05
MB_L3	99.45	99.58	99.47	0.56	0.02
MB_MEM	99.87	99.48	99.48	99.18	0.01
MB_L1asm	99.32	0.01	0.01	0.01	2.03
MB_MEM64	99.69	99.33	99.94	2.15	0.03

The memory micro-benchmarks (`MB_L1`, `MB_L2`, `MB_L3`, and `MB_MEM`) perform the same operation (a `load` instruction) and use pointer chasing to obtain the address of the next memory location to load. However, each micro-benchmark accesses a data set that only fits in the corresponding cache level. For example, the data set of `MB_L2` completely fits in the L2 cache but is larger than the L1 cache. Compared to a stencil approach, pointer-chasing avoids the use of an extra addition operation to increment the array index.

Although designing micro-benchmarks may appear straightforward, their implementation may hide unexpected runtime behaviors. This is caused by architecture features, such as data prefetching, caches and speculation, as well as compiler optimizations. To avoid effects of data locality, two consecutive accesses are separated by a stride that is larger than the cache line (64 bytes in our case). Moreover, to avoid the effects of memory prefetcher, the stride must be large enough (512 bytes) so that the memory prefetcher does not recognize the stream of data and pre-load data from memory.

We use performance counters to validate the runtime behavior of our micro-benchmarks. Table I reports the dynamic characteristics of each micro-benchmark. As we can see from the Table, all micro-benchmarks execute around 95% of their instructions in their body-loop. The remaining instructions are branches, additional add operations and tests, which are

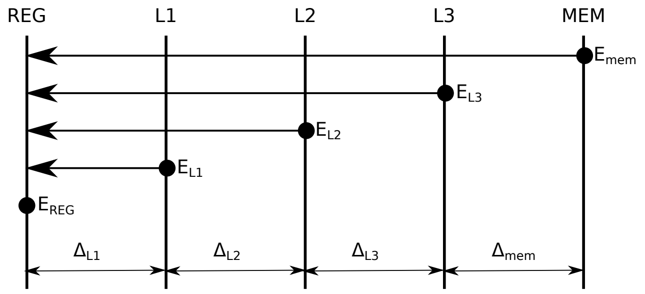


Fig. 2: Energy cost of moving data across the memory hierarchy to processor registers.

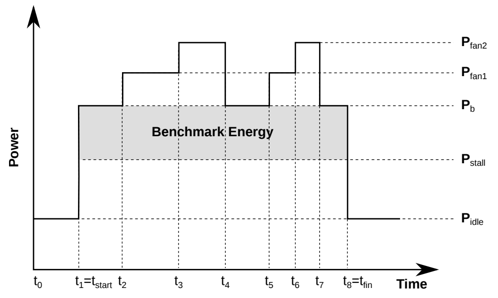


Fig. 3: Measuring the energy of a benchmark through an external power meter.

necessary for the body-loop execution. We also check whether each memory micro-benchmark retrieves data from the correct level of the memory hierarchy by computing the L1, L2, L3 miss rates. Table I reports that `MB_L1`, `MB_L2`, `MB_L3` fetch 99% of their data from L1, L2 and L3 respectively.

IV. Energy Measurement

Accurately measuring the energy cost of single operations in a generic compute node requires particular attention and meticulous planning. In this Section we illustrate the techniques we used to remove or isolate the components of the system that do not specifically relate to energy of data movement, such as idle energy, processor's fans, and stalled cycles. We then show how we derive the energy cost of data movement operations and data prefetcher.

Figure 2 depicts our approach: we first compute the energy cost of moving data from the L1 cache to the processor register E_{L1} and then we incrementally determine the other moving costs. We measure the total energy consumption E_b of each micro-benchmark and then derive the per-operation energy using the effective number of operations performed by the micro-benchmark obtained from performance counters. Since the micro-benchmarks perform order of billions instructions in the body loop, this ratio gives us a good approximation.

A. Measuring Micro-Benchmark Dynamic Energy

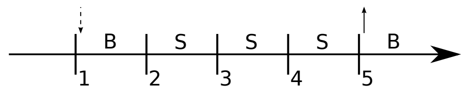
Our test system, based on two AMD Opteron 6272 Interlagos processors [7], is instrumented with an external power meter that measures the power consumption of the entire compute node P_{total}. Moreover, similarly to Intel SandyBridge [19], the AMD Opteron 6272 Interlagos features a power sensor that provides the power consumption of the processor chip P_{cpu}. This internal power sensor is more accurate than external power meters and can be queried at a higher frequency. On the other hand, socket-level sensors do not provide information about the power consumption of memory activity, which is essential to understand the total energy cost of data movement. In our experiments we combine the information provided by the internal processor power sensor and the external power meter to isolate the energy consumption of components that are external to the processor, such as DRAM, circuitry or fans.

Our external power meter only provides a power sample every two/three seconds. To overcome this limitation we run the benchmarks for about five minutes and collect performance counters and power readings. We also leave the system idle for about 90 seconds between the execution of two micro-benchmarks to cool down the system and to be able to identify, in the power profile, the beginning and end of each experiment.

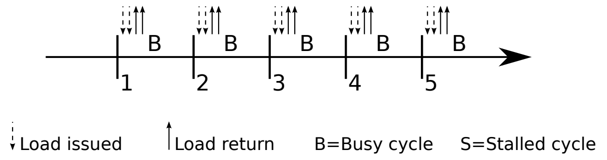
Figure 3 shows an example of a typical power profile for one of our micro-benchmarks obtained from the external power meter. The system is idle from time $t = t_0$ to $t = t_1 = t_{start}$, when the micro-benchmark starts its main loop, and for $t > t_8 = t_{fin}$, once the micro-benchmark has completed. During this idle time we measured an idle power $P_{idle} = 259W$, which consists of the processors' idle power (including cache and memory controller leakage), memory power, chipset power (circuitry, PCI buses, etc.) and device power (including processor's fans running at normal speed). We can isolate the idle power consumption of non-processor components (memory, chipset, etc.) as the difference between P_{idle} and the value read from the internal power sensor P_{cpu} when the system is idle. Given that our micro-benchmarks present a stable computation phase (once the initialization phase is terminated at time $t = t_{start}$), we can safely assume a constant power consumption during the execution of the body loop $[t_{start}; t_{fin}]$. We refer to this power consumption as P_b, hence $P(t) = P_b \ \forall t \in [t_{start}; t_{fin}]$. The total active energy E_b consumed by the micro-benchmark during the execution of the body loop $\Delta t = t_{fin} - t_{start}$, measured from the processor's time stamp counter, can then be computed as:

$$E = \int_{t_{start}}^{t_{fin}} P(t)\,\mathrm{d}t = P_b \int_{t_{start}}^{t_{fin}} \mathrm{d}t = P_b\,\Delta t \tag{1}$$

Determining P_b, however, is not trivial using external coarse-grained power meters. Many factors may influence the power profile and, thus, lead to erroneous conclusions. For example, at time $t = t_2$ in Figure 3 the temperature of the processor increases as a result of the micro-benchmark activity performed on that processor. Consequently the processor fan increases its speed, which increases the system power consumption up to P_{fan1}. If the temperature of the processor does not reduce below the safe threshold, the processor fan may further increase its speed, hence its power consumption increases up to P_{fan2}, as shown in the example in Figure 3 for time $t \in [t_3; t_4]$. When the temperature of the processor reduces below the safe threshold, the fan speed is reduced to the default speed used when the system is idle or to the intermediate speed observed between $[t_2; t_3]$. This behavior results in an irregular power profile, such as the one depicted in Figure 3. To perform an accurate measurement we need to determine P_b before the



(a) The processor core stalls while waiting for a data dependency to be resolved.



(b) If there are no data dependencies to be resolved, the processor core is able to issue instructions at every cycle.

Fig. 4: Pipelined execution of *MB_L1* and *MB_L1asm*.

processor fans increase their speeds or, equivalently, remove the power consumption of the fans from the micro-benchmark power profiles. We estimate the power consumption of the fans (our system features two processors) by subtracting the value read from the internal processor sensor, which does not change when the fan speed increases, from the external power meter. Our results show that the power consumption of each fan is about 7W when running at the highest speed.

B. Stalled Cycles

Once we determined the value of P_b for each micro-benchmark, the second problem is to determine how much energy consumption derives from the micro-benchmark activity and how much derives from powering up the system or processor overhead. Clearly, the idle power P_{idle} does not depend on the micro-benchmark and can be removed from P_b. Moreover, the processor's core performs activities that are not directly related to the instruction itself or consume power while waiting for a data dependency to be resolved. Consider the example in Figure 4a: the picture shows the execution of a `load` instruction that retrieves data from the L1 cache (`MB_L1`) when the pipeline is full. The `load` instruction is issued at cycle 1 and the latency of accessing the L1 cache is 4 cycles, hence the data accessed is available in the register at cycle 5. Only one instruction is completed in 4 cycles (IPC = 0.25, as reported in Table I). During cycles 2, 3, and 4 the processor core is stalled (back-end stall cycles) waiting for the data to come back but still consuming energy. This energy translates into the power consumption P_{stall} shown in Figure 3 and should not be included in the cost of moving data from the L1 to the processor's register. Note that this stall energy is not included in P_{idle} either, as the operating system performs relax instructions when the a core is idle (a discussion about the difference between idle power and baseline power is also available in previous work [18]). In fact, the active energy cost of a benchmark is only the shadow area in Figure 3.

In order to evaluate the energy consumption of each stalled cycle, we use the `MB_L1asm` micro-benchmark. Since `MB_L1asm` does not present any dependency among consecutive loads, the processor core is able to issue two instructions per cycle at every cycle, as depicted in Figure 4b. Comparing Figure 4a and Figure 4b we note how, in the latter, the core's utilization increases considerably and all cycles are busy. The resulting IPC for `MB_L1asm` is 2.03. We can then evaluate the energy cost of a stalled cycle as follows: First, we compute the energy cost of a L1 instruction E_{L1} using `MB_L1asm`, for which we know there are no stalled cycles, hence $P_{stall} = 0$. Then we subtract the total energy cost of L1 instructions ($E_{L1} * N_{L1}$) from the total energy cost of `MB_L1` E_{MB_L1}, which also includes the energy cost of stalled cycles. Finally, we note that, during 4 cycles, `MB_L1asm` performs 8 `load` instructions while `MB_L1` performs only 1 instruction. This means that even during the busy cycles, the processor's core has hardware resources that are not fully utilized when running `MB_L1`. Indeed, each core is capable of issuing two loads per cycle but, because of the pointer chasing data dependency, `MB_L1` can issue only one instruction. Another way to look at this computation is to note that in the time `MB_L1` performs one load operation (4 cycles), `MB_L1asm` can perform 7 more load instructions, i.e., 1.75 more load instructions per cycle. We conclude that the energy cost of a stalled cycle is

$$E_{stall} = \frac{E_{MB\_L1} - 1.75 * E_{L1} * N_{L1}}{N_{stall}}$$

This energy cost includes the cost of performing several activities that consume considerable energy, such as determining access strides to eventually activate the data prefetcher or speculate on the next instruction to execute. All these activities are considered "overhead" and may represent a considerable cost in modern out-of-order processor, as reported in [8].

C. Long Latency Memory Operations

For micro-benchmarks that perform long latency memory operations, such as `MB_L2`, `MB_L3`, and `MB_MEM`, we could not implement the same assembly version of the respective original benchmark as we did for `MB_L1`. The latency of those operations, in fact, is quite large compared to the processor clock rate (20 cycles for accessing data in the L2, 60 cycles for the L3, and 150 cycles for memory) and the benchmarks completely fills the load/store queue, stalling the processor core. In other words, even with assembly implementations that present no data dependency between two consecutive load instructions, we would still need to remove the energy cost of stall cycles E_{stall} from the total energy cost of performing a load instruction. We, thus, use the original `MB_L2`, `MB_L3`, and `MB_MEM` but subtract E_{stall} from the energy cost measured for the micro-benchmark. For example, for `MB_L2`, we compute the energy cost of performing a load instruction that retrieves data from the L2 cache as:

$$E_{L2} = \frac{E_{MB\_L2} - E_{stall} * N_{stall}}{N_{L2}}$$

where E_{MB_L2} is the total energy cost of executing `MB_L2` computed as in (1), N_{stall} and N_{L2} are the stalled cycles and the number of accesses to the L2 cache obtained from performance counters, respectively.

`MB_L1`, `MB_L2`, and `MB_L3` do not perform any access to memory, as their data sets fit into their respective cache level. For these benchmarks we can directly compare the power

TABLE II: Energy cost of moving data across the memory hierarchy in a AMD Interlagos 6227 system.

Operation	Energy Cost (nJ)	Δ Energy (nJ)	Eq. Ops
`NOP`	0.48	-	-
`ADD`	0.64	-	-
`L1->REG`	1.11	1.11	1.8 ADD
`L2->L1`	2.21	1.10	3.5 ADD
`L3->L2`	9.80	7.59	15.4 ADD
`MEM->L3`	63.64	53.84	99.7 ADD
`stall`	1.43	-	-
`prefetching`	65.08	-	-

information obtained from the power meter with the one obtained from the processor internal power sensor to validate our external measurements. `MB_MEM`, instead, performs memory activity, thus the total power consumption of the benchmark $P_b = P_{cpu} + P_{dram}$, where P_{cpu} can be measured from the processor internal power sensor and P_b from the external power meter. We can thus derive the power consumption related to the memory activity P_{dram}: our results show that this power consumption is about 0.82W per thread.

D. Memory Prefetcher

Modern architectures proactively prefetch data from memory to the processor caches to hide memory latency and improve overall performance. This data movement is not directly initiated by the programmer and it is not reflected in the number of load operations or the cache misses. Prefetchers typically recognize strided accesses but non-strided prefetchers are also common in recent systems [4]. AMD 6272 processors feature two strided prefetchers: the L1 prefetcher is activated by L1 cache misses and brings data from memory to the L1 core cache. Similarly, the L2 prefetcher reacts to L2 cache misses and move data from memory to the L2 cache. The two prefetchers work in a coordinated way: the L1 memory prefetcher also drives the L2 memory prefetcher, hence the same data is not prefetched twice. Additionally, the processor features a non-strided prefetcher.

In the previous experiments we deliberately calibrated our micro-benchmark with a stride access that is not recognized by the prefetchers in our compute node. However, scientific applications heavily use memory prefetchers, hence we need to evaluate the energy cost of prefetching data from memory as it accounts for the total energy cost of data movement. To estimate the energy cost of data prefetching, we use the `MB_MEM64` micro-benchmark that presents 64 bytes stride size between consecutive accesses rather than 512. This stride is recognized by the memory prefetcher that, indeed, succeeds to proactively move data from memory to the processor's cache *before* the data is actually requested. The result is that the number of L3 cache misses considerably reduces, as shown in Table I. We measure the power consumption of this benchmark P_b from the power meter and subtract the power consumption of the processor measured by the internal processor power sensor, the idle power and the fan power. The net difference is the power consumed by the prefetcher when moving data from memory to the processor's cache P_{dram}. We add this value to the processor power P_{cpu} to obtain P_b, as we did for the original `MB_MEM`, and the energy is computed as in (1). The energy cost of data prefetching E_{pf} is, not surprisingly, very close to the energy cost of retrieving data from memory E_{MEM}. Indeed, although prefetchers may directly move data to a lower cache level, such as the L1 or the L2 cache, the largest component of the energy cost is spent moving data from memory to the processor.

Summary: Table II reports our energy cost estimates: the first column in the table shows the cost of moving data from a given level in the memory hierarchy to the processor's register. For example, moving data from the memory to the processor's register consumes 63.64 nJ. The second column reports the energy cost of moving data from one level in the memory hierarchy to the next one, e.g., from the L2 to the L1 cache. The table also reports the energy cost of performing operations such as `nop` and `add` and the energy cost of a stall cycle E_{stall} and prefetching data from memory E_{pf}. We also report the ratio between the energy cost of moving data from one level of the memory hierarchy to registers and the cost of performing a register-to-register add operation. As we can see, moving data from memory to processor's register (E_{MEM}) is equivalent to 99 `add` instructions. Note that this ratio is lower than the ones reported in the previous works because it does not include the energy cost of stall cycles.

V. Experimental Results

In this Section we validate our measurements and analyze the energy cost of data movement for scientific applications. We perform our experiments on a dual-socket AMD Opteron 6272 (Interlagos) [7] compute node equipped with 64GB of DRAM memory: each socket is a Clustered Multi-Threaded (CMT) chip divided into two Multi-Chip Modules (MCMs), and each MCM consists of four Clustered Integrated Cores (CIC), a shared 8MB L3 cache, and a memory controller. A Clustered Integrated Core couples two "conventional" x86 out-of-order Integer Cores together; the two cores share fetch and decode units, the 64KB L1I and the 2MB L2 cache (the 64KB L1D is private to a single Integer Core), and the floating point unit. Like many modern processor architectures in the market, our AMD 6272 system features mostly inclusive cache structure.[1] This compute node system is part of the PNNL Institutional Computing (PIC) cluster that includes 141 compute nodes equipped with two AMD 6272 Interlagos processors, for a total of 4512 cores. From this study's point of view, we are particularly interested in the memory hierarchy of the system: Figure 1 shows the topology of a MCM and how the different levels of the cache hierarchy are connected and shared by the processor's cores.

All applications are compiled with `pathscale 4.0.10` for 64-bit architectures with optimization `-O3` and linked to `openmpi 1.5.4`. NPB applications [5] use the class C input set, while the *GTC*, *NEK-Bone*, and *LULESH* use their respective reference input sets. Unless otherwise specified, all applications are executed with 16 MPI tasks[2] and each task is bound to a specific core through the `--bind-to-core`

[1]In fact the cache hierarchy is not fully inclusive, as the memory prefetcher copies data directly to one level of the cache hierarchy. However, in the validation experiments, we use access strides not recognized bye the prefetchers, hence the system behaves as a fully inclusive cache system.

[2]We do not use the second Integer Core on each CIC because this generally creates contention in the floating point unit that reduces performance.

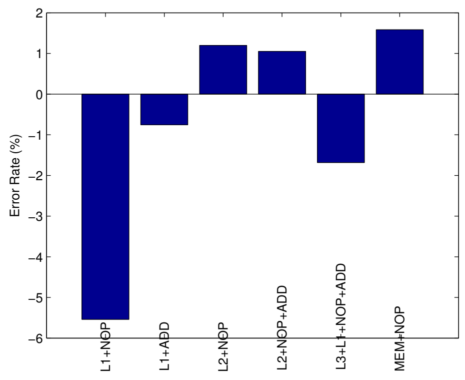


Fig. 5: Validation error rate.

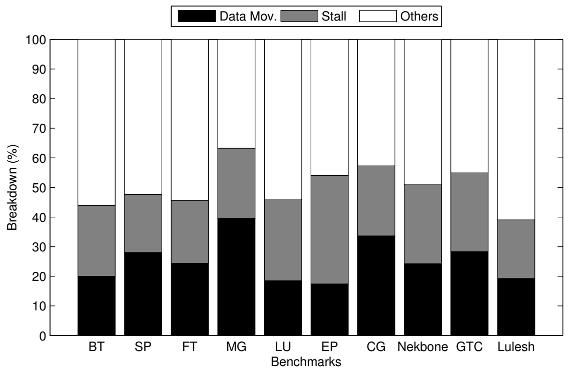


Fig. 6: Energy Breakdown.

option of `mpiexec`. *LULESH* requires a cube number of MPI processes, thus we run 27 MPI tasks.

A. Validation

In this subsection we validate our methodology and measurements with a set of test micro-benchmarks. To the contrary of the micro-benchmarks described in Section III, which perform only one type of operation in the body loop, these test micro-benchmarks combine different operations within the same body loop. Since we know the energy cost of each of those operations, including the energy cost of stalled cycles, we can estimate the total energy cost of each test micro-benchmark by summing the energy consumption of each group of operations. For example, the estimated energy cost of *L1+NOP* benchmark is:

$$E_{L1+NOP} = E_{NOP} * N_{NOP} + E_{L1} * N_{L1} + E_{stall} * N_{stall}$$

where N_{NOP} and N_{L1} are the number of `nop` instructions and accesses to the L1 cache, respectively, and N_{stall} is the number of stall cycles, obtained from performance counters.

Figure 5 shows the accuracy of our methodology for the test micro-benchmarks. The graph reports the error between the estimated energy and the energy obtained from external measurements of each test micro-benchmark. The graph shows that our approach is able to accurately estimate the energy consumption of the test benchmarks, generally within 1-2% error rate. Only for one case (*L1+NOP*) we underestimate the cost of energy by 5.5%.

B. Analysis of Scientific Applications

We use our methodology and measurements to analyze the cost of data movement in scientific applications. The total dynamic energy consumption (including the dynamic energy consumption of fans and uncore power but not considering the idle energy portion) is derived from power measurements obtained from the external power meter. We collect N sample P_i and compute the total energy consumption of each application E using the trapezoidal rule in the time interval $[t_{start};t_{fin}]$:

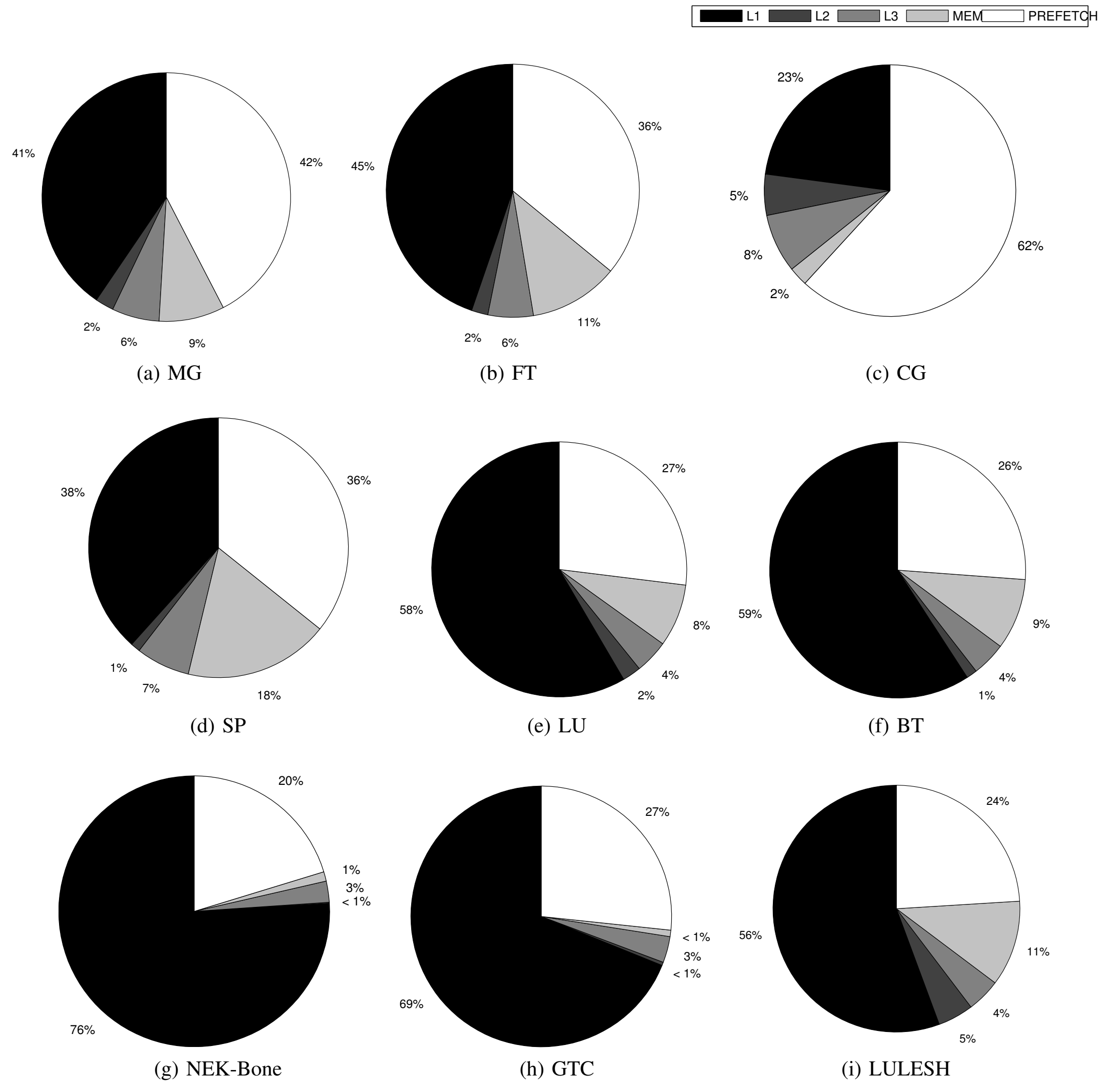
$$E = \int_{t_{start}}^{t_{fin}} P(t)\,\mathrm{d}t = \sum_{i=0}^{k} \overline{P}_i t_i$$

Independently from the power measurements obtained from the external power meter, we collect performance counters to estimate the cost of data movement. In particular, we collect the number of accesses to L1 (`L1->REG`), L1 misses (`L2->L1`), L2 misses (`L3->L2`), L3 misses (`MEM->L3`), and memory prefetching requests. For completeness, we also collect back-end stalled cycles and estimate their impact on the energy profile of each application.

We show the impact of energy cost of data movement and stalled cycles on scientific applications in Figure 6. The energy cost of data movement for the tested applications on our system ranges between 18% (*EP*) and 40% (*MG*), with an average percentage of about 25%. This means that, on average, a considerable portion of the total dynamic energy consumption, 1/4 of the total energy, is spent in moving data across the memory hierarchy. This percentage is bound to increase in future exascale system, as we expect the energy cost of performing a double-precision floating point operation to reduce by 10x while the cost of moving data will not likely reduce by the same factor.

Another interesting observation is that 19-36% of the total dynamic energy cost is spent into processor stalled cycles, i.e., in resolving data dependencies, speculation actions, out-of-order execution of instructions, etc. This high energy cost, referred as "overhead" in [8], suggests that future exascale processor architecture can increase their energy efficiency by simplifying core's architectural design, i.e., small, in-order cores with smaller cache-coherent domains. Finally, the energy cost of computation and operating the processor fans ("Others" in Figure 6) varies between 36% and 60%, with average of 50%. This cost is still dominant on current systems, though we expect it to reduce by the exascale time-frame.

We also analyzed the energy cost of data movement breakdown for the studied applications. Figure 7 shows the relative percentage of energy spent into moving data between any two levels of the memory hierarchy (energy costs of computation and stalled cycles are not reported in these graphs). The tested applications are considered regular applications, hence we expect a high data locality (i.e., data will resides in the lower levels of the cache hierarchy before it is requested) and the memory prefetcher to be able to proactively move data to the processor's caches. As we can see from the graphs, each application presents a different data movement energy



(a) MG (b) FT (c) CG

(d) SP (e) LU (f) BT

(g) NEK-Bone (h) GTC (i) LULESH

Fig. 7: Relative percentage of energy spent into moving data from all levels of the cache hierarchy for scientific applications.

breakdown: several applications (*LU*, *BT*, *GTC*, and *LULESH*) show that the memory prefetcher consumes about 1/4 of the energy cost of data movement. This cost increases for *CG* (62%), *MG* (42%) *SP* and *FT* (36% for both of them) while it is slightly lower for *NEK-Bone* (20%). In general, however, the memory prefetcher is effective in proactively moving data from memory to the processor's cache: the graphs in Figure 7 show that only a small percentage of energy (less than 10%) is spent in moving streams of data not captured by the prefetcher from memory to the processor's cache. The only exceptions are *LULESH* (11%), *FT* (11%) and *SP* (18%). The regularity of the applications can be also observed by the low percentage of data moved from the L2 (generally below 2%) and L3 cache (below 9%). This is the result of combining good data locality (a cache line in the L1 cache is accessed several times) and the work of the memory prefetcher that, in our architecture, brings data directly to the L1 cache.

The energy cost of moving data from the L1 cache to the processor's registers is the dominant factor in most of the cases, with only *FT* (45%), *CG* (23%), *MG* (41%) and *SP* (38%) consuming less than half of their data movement energy cost in moving data from the L1 to the processor's registers. Obviously, this is also due to the cache design of the AMD 6272 processor: all data must be moved from the L1 cache to the processor's register, no matter whether that data originally came from the L2 cache or the memory prefetcher. For *CG* the memory prefetcher energy is the dominant factor in the data movement energy cost (62%). However, Figure 7c shows that the largest part of data brought to the L1 by the prefetcher is not actually used, thus this energy is wasted. In fact, all data that reside in the processor's registers must come from the L1 cache: the fact that only 23% of the data movement energy is spent moving data from the L1 cache suggests that a considerable portion of energy is spent moving data from memory to the L1 but that data is not moved to the processor's register, hence it is not used.

LU and *BT* (Figure 7e and 7f) show typical cases of optimized scientific applications: the memory prefetcher is generally able to move data from memory to the L1 cache before this data is actually required. The applications spend the largest part of data movement energy in moving data from the L1 cache to the processor's registers (58% and 59%) and less than 10% of the relative energy is spent moving extra data (i.e., streams not captured by the prefetcher) from memory.

Figure 7a, 7b, and 7d show applications (*MG*, *FT* and *SP*) that present some form of irregularity. The data prefetcher still succeeds, to some extent, to bring useful data to the L1 cache but a considerable amount of extra data must still be moved from memory to the L3 cache (9%, 11% and 18% of the data movement energy, respectively). As for *CG*, the relatively low energy spent in moving data from the L1 cache to the processor's registers (41%, 45% and 38%, respectively) suggests that a large part of the data prefetched is not used.

EP (not shown in Figure 7 because of space constraints) is an embarrassingly parallel application, hence there is no communication among the MPI tasks except for a final global barrier at the end of the application. When running 16 MPI tasks and the class C input set, each task's working set fits in the L1 cache, hence 99% of the data movement energy cost is spent in moving data from the L1 to the processor's register, while data movement form either level of the cache hierarchy or memory is only performed at the beginning of the application. Note that, since the memory prefetcher is driven by the L1 cache misses, there is no energy spent in prefetching data from memory.

The last row in Figure 7 shows the two applications from the Exascale Co-Design Centers [1] (*NEK-Bone* and *LULESH*) and *GTC* from the DoE Office of Science. *NEK-Bone* was developed by the CESAR Co-Design Center to mimic the behavior of *NEK5000*, a computational fluid dynamics solver based on the spectral element method. *GTC* (3D Gyrokinetic Toroidal Code) solves the gyro-averaged Vlasov equation in real space using global gyrokinetic techniques and an electrostatic approximation. This code is used to study microturbulence in magnetically confined toroidal fusion plasmas via the Particle-In-Cell (PIC) method. Both these applications have been optimized for our compute node: Figures 7g and 7h show that, except from the prefetching energy and the energy spent moving data to the processor's registers from the L1 cache, there is practically not other data movement activity. The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) presented in Figure 7i has not been optimized for our cluster compute node, as the NPB benchmarks. The data movement energy breakdown is similar to *BT* and *LU* with the memory prefetcher able to proactively move most of the data from memory to the processor's cache but not all. As a result, the application still spends 11% of the data movement energy in moving extra data from memory to the L3 cache and then down to the other cache levels and the processor's registers.

Summary: The experiments performed in this Section confirm that data movement represents a considerable part of the total energy cost, up to 40% of the total dynamic energy consumption. Additionally, and for the first time, we show the impact of data movement on the total energy consumption of scientific applications, including three DoE applications, two of which (*NEK-Bone* and *GTC*) have been optimized for our compute node. Our results show that 1) the energy cost of data movement for scientific applications is, on average, 25% of the total dynamic energy consumption; 2) the energy cost of the "overhead" (i.e., stalled cycles) is also considerable in current systems, up to 36%; 3) the prefetcher is generally able to proactively move data to the the processor's cache but in some cases (e.g., *CG*) part of this energy is wasted; 4) for well-optimized applications (*NEK-Bone* and *GTC*) only 25% of energy is spent moving data from memory or across the different levels of the cache hierarchy while 56-76% of the energy is spent moving data from the L1 cache to the processor's registers.

These results highlight clear directions for the design of future exascale systems, such as simpler processor core that minimizes the wasted energy of stalled cycles, precise memory prefetchers capable of properly recognizing memory access patterns or increasing the number of available register to minimize the cost of moving data from the L1 to the processor's registers. Moreover, new memory technologies, such as PIM, NVRAM and 3D stack are essential to reduce the the increasing ratio between the energy cost of data movement and the energy cost of performing computation.

VI. Related Work

Accurate power and energy profiling of scientific applications has become very important to be able to effectively design and deploy large-scale computer systems. Many approaches have been proposed in this area. Ge et al. [11] introduce a framework, called PowerPack, which enables system component-level power profiling correlated to application functions. This framework requires special hardwired connections to pins of hardware components, high-performance data acquisition equipments, and detailed event and log information for power measurement or estimation. By using this framework, the authors profiled applications from the NAS benchmark suites [5]. Although this approach can obtain detailed information about the energy and power consumption of the various system's components, it cannot be used on a common, un-instrumented compute node. Manousakis et al. [17] introduce a power instrumentation framework to measure power and energy consumption of processors and components of the memory hierarchy. The authors also explore the impact of off-chip memory accesses on energy efficiency and power consumption with a set of engineered micro-benchmarks. Alonso et al. [2] propose a framework of easy-to-use and scalable tools to analyze the power dissipation and the energy consumption of parallel MPI and/or multi-threaded scientific applications. These tools enable system administrators, library developers and scientists to identify hot spots in the hardware and power sinks in the system software, library code and applications. Unlike our work, the authors used only synthetic benchmarks to illustrate the use and benefits of their tools and they did not characterize the impact of data movement on the total energy consumption.

Several works [14], [15], [20], [21] propose the use of performance counters to estimate the power consumption for a specific load operation. Other works focus on the performance of data movement from different memory levels [13]. However, none of these works focus on the energy consumption of moving data between memory levels. Similar to our work, Molka et al. [18] characterizes the energy efficiency of arithmetic and memory operations across different cache levels. The authors develop a set of multi-threaded workloads that specifically stress certain CPU components and also enable to control the location of all data. By using these micro-benchmarks, they build a basic model to approximate the energy consumption of data transfers from different cache levels and main memory. In contrast with this work, in this paper we build a more elaborated energy model that considers the energy cost of stalled cycles, the impact of prefetching on data movement. Moreover, we profile real applications from the Exascale Co-Design Centers and DoE Office of Science while providing the energy cost data movement over total energy consumption.

VII. Conclusions

The energy cost of data movement has been identified as one of the major limiting factors for the development of efficient and sustainable exascale systems. Projections show that the cost of moving data from memory is two orders of magnitudes higher than the cost of computing a double-precision register-to-register floating point operation. These studies are usually performed on simulation environments or with custom power instrumentation and, although important, only provide qualitative information for scientific applications.

If this work, we perform a quantitative study of the energy cost of moving data across the memory hierarchy and analyze the energy impact of data movement on scientific applications. We developed a set of well-engineered micro-benchmarks and combine external and internal power measurements to isolate the cost of each data movement instructions. We then use an incremental-step approach to estimate the cost of moving data from one level to the next of the memory hierarchy.

Our results show that, on current systems, scientific applications spend 18-40% (25% on average) of their total dynamic energy in moving data and 19-36% in stalled cycles. We further divide the energy cost of data movement among the different levels of the memory hierarchy: we reported that, for well-optimized applications, the memory prefetcher is generally capable of proactively moving data from memory to the processor's cache but also that, in some case, this data is not actually useful, which results in energy waste.

We believe that this study provides a quantitative understanding of the impact of data movement on the energy consumption of scientific applications and important contributions for the development of future exascale systems.

Acknowledgments

This work was supported by the DOE Office of Science, Advanced Scientific Computing Research, under award number 62855 “Beyond the Standard Model – Towards an Integrated Modeling Methodology for Performance and Power”; Program Manager Sonia Sachs.

References

[1] Advanced Scientific Computing Research (ASCR). Scientific discovery through advanced computing (SciDAC) Co-Design. http://science.energy.gov/ascr/research/scidac/co-design/.

[2] P. Alonso, R. Badia, J. Labarta, M. Barreda, M. Dolz, R. Mayo, E. Quintana-Orti, and R. Reyes. Tools for power-energy modelling and analysis of parallel scientific applications. In *Proc. of the International Conference on Parallel Processing*, pages 420–429, 2012.

[3] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, P. Balaji, P. C. Diniz, A. Koniges, M. Snir, , and S. R. Sachs. Report of the Workshop on Exascale Programming Challenges. Technical report, US Department of Energy, 2011.

[4] BIOS and Kernel Developers Guide (BKDG) for AMD Family 15h Processor. http://support.amd.com/us/Processor.

[5] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical report, NASA Advanced Supercomputing (NAS) Division, March 1994.

[6] T. Barrett, M. Sumit, K. Taek-Jun, S. Ravinder, C. Sachit, J. Sondeen, and J. Draper. A double-data rate (ddr) processing-in-memory (pim) device with wideword floating-point capability. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2006.

[7] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro*, 31(2):6–15, Mar. 2011.

[8] B. Dally. Power, programmability, and granularity: The challenges of exascale computing. In *Proc. of International Parallel and Distributed Processing Symp.*, pages 878–878, 2011.

[9] H. Dong, H. Nak, D. Lewis, and H.-H. Lee. An optimized 3d-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[10] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th international conference on Supercomputing*, pages 14–25, 2002.

[11] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. W. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Trans. Parallel Distrib. Syst.*, 21(5):658–671, May 2010.

[12] Y. H., G. Huang, and P. L. Nonvolatile memristor memory: Device characteristics and design implications. In *IEEE/ACM International Conference on Computer-Aided Design.*, pages 485–490, 2009.

[13] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–422, 2009.

[14] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 135–140, 2001.

[15] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie. Enabling accurate power profiling of hpc applications on exascale systems. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, pages 4:1–4:8, 2013.

[16] G. Loh. 3D-Stacked memory architectures for multi-core processors. In *Proceedings of the International Symp. on Computer Architecture*, pages 453–464, 2008.

[17] I. Manousakis and D. Nikolopoulos. Btl: A framework for measuring and modeling energy in memory hierarchies. In *Proceedings of the IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, pages 139–146, 2012.

[18] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Characterizing the energy consumption of data transfers and arithmetic operations on x86-64 processors. In *Proceedings of the International Conference on Green Computing*, pages 123–133, 2010.

[19] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, 2012.

[20] K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, 37(2):46–55, 2009.

[21] W. Wu, L. Jin, J. Yang, P. Liu, and S. X.-D. Tan. A systematic method for functional unit power estimation in microprocessors. In *Proceedings of the 43rd annual Design Automation Conference*, pages 554–557, 2006.