Energy Efficient Processors – ARM big.LITTLE Technology (January 2016)

Philipp Gsching, *Student (B.Sc.), Heidelberg University, Germany*

***Abstract*—Power consumption and battery life is one of the main limiting factors for the capabilities of mobile devices these days. It is crucial to build processors which consume as little power as possible. At the same time expectations and demand for performance are constantly increasing. It seems like those two requirements are hard to combine in one device and therefore compromises have to be made inevitably; compromises which end with reduced performance, higher power consumption or both. ARM Limited tried to tackle this problem heads on: being able to combine a low power with a high performance processor in one chip and being able to switch between them on-the-fly depending on the current requirement would reduce the need for compromise and allow for mobile devices with little average power consumption, while still meeting our performance requirements. In this paper I am going to take a closer look at ARMs big.LITTLE technology; assessing how it works as well as how it performs.**

***Index Terms*—Energy efficient processing, heterogeneous microarchitecture, ARM big.LITTLE**

I. Introduction

With mobile devices such as smartphones and tablet computers we can generally distinguish between two main – but very different – categories of usage: on one hand the devices idle, are in a low power mode (since they are usually never truly powered off – not even at night), or are used for low performance applications such as e-mail or text messaging for most of the day; on the other hand we demand high performance for a select few (but increasing) number of applications such as playing 3D games, watching HD videos etc. – for these tasks we expect performance similar to modern computers and laptops.

Both of these scenarios would call for a very different choice in processors when looked at independently. In the first case one would want to equip the device with a low power processor for maximum battery life. The latter case would require a high performance processor – which inevitably would mean high power consumption and low battery life.

This conflict of interest is especially drastic considering the fact that power consumption and battery life are among the main limiting factors for mobile devices these days.

Choosing one processor for both cases means a compromise when it comes to performance as well as power consumption. It would be ideal to have the right processor for each scenario within the same device or even die (heterogeneous processor) and to be able to switch back and forth between them depending on demand for maximum performance and maximum battery life.

This is what ARM tried to achieve with the development of their big.LITTLE technology.

In this paper I am going to take a closer look at energy efficient processing in general as well as ARMs big.LITTLE technology in particular. For this purpose the paper is divided into two main parts. In the first section I am going to discuss general computer architectural concepts for energy efficient processing with a brief introduction to the ARM micro-architecture. The second part will introduce and analyze big.LITTLE in more detail.

II. Energy Efficient Microarchitecture

A. Instruction Set Architecture

One of the most prominent and prevalent opinions about energy efficient processing is that the instruction set architecture (ISA) has a major impact on performance and power consumption of a processor. The two major ISAs to be distinguished are those labeled *Reduced Instruction Set Computing* (RISC) and *Complex Instruction Set Computing* (CISC). The general consensus (established in the 1980s) was that RISC architectures – which implement a smaller, simpler and fixed-size instruction set – are a better choice for low power applications but lack performance. CISC architectures on the other hand – implementing a large number of instructions, varying greatly in complexity and instruction length – are better suited for high performance systems where energy efficiency does not matter as much.

The fact that ARM solely produces RISC processors and at the same time controls the majority of the mobile market (for which energy efficiency is crucial) might seem like proof for this consensus.

However, as *Emily Blem et al.* [1] presented in their work "ISA Wars" published in 2015, there is no difference between RISC and CISC architectures anymore; ISA does not affect performance and power in modern processors. Instead, the authors keep emphasizing that microarchitecture and the decisions made regarding power/performance trade-offs are the major origin for the significant differences we see in processors today.

B. Microarchitectural Innovations

There are several principles, tools and technologies on the microarchitecture level that can be applied to a processor during its design and utilized to optimize it for either performance or power consumption. Since the subject of this paper is processors for mobile devices, I will present some of the most common of these techniques with an emphasis on energy efficiency.

1) Technology-node and feature size

Reducing the feature size of processors does not only allow for more and more complex structures on each die, it also reduces the capacitance of these structures and therefore plays a major role in reducing energy consumption.

2) Dynamic voltage and frequency scaling

Dynamically adjusting supply voltage as well as clock speed of a processor allows matching the performance with the current task and avoids wasting energy during idling and low performance applications.

3) Clock-gating

Through supplying different sections of a processor with individual independent clocks, these individual components can be stopped independently if not needed. They will retain their current state, but use significantly less power than before.

4) Power domains

By dividing the processor up into different (logically coherent) domains, unused parts of the processor can be turned off entirely to avoid any idle power consumption.

5) Pipelining

An execution pipeline tries to minimize idle time for different execution stages of a processor which otherwise would have to wait on previous stages to complete their current task before continuing execution. This increases performance and efficiency.

6) Caches

Accesses of the processor to main memory are not only time but also power intensive. On-chip caches storing the current working set data drastically reduce the frequency of main memory accesses.

7) SoC-Design

In mobile devices *Systems-On-A-Chip* (SoC) – which combine most of the key components of a system, such as CPU, GPU, periphery controllers etc. on one die – are commonly chosen over discrete chips for each of these components – the usual approach for laptop and desktop computers. All these components put restrictions on and need to be adjusted to one another. This can mean limitations as well as opportunities for designers regarding performance and power consumption.

Most of these techniques contain some sort of power/performance trade-off which can and must be used to optimize a chip for a given task.

TABLE I
ARM PROCESSORS

	Cortex A53 (LITTLE)	Cortex A57 (big)
64 Bit	Yes	Yes
Cores	1 - 4	1 - 4
Frequency[a]	1.3 GHz	1.9 GHz
L1-Cache	8 – 64 kB	48 / 32 kB[b]
L2-Cache	128 – 248 kB	512 – 2048 kB
Pipeline stages[c]	8	15
Out-of-Order	No	Yes
Performance[d]	2.3 DMIPS / MHz	4.1 DMIPS / MHz
Process[a]	20 nm	20 nm
Core size[a]	0.70 mm²	2.05 mm²
Cluster size[a]	4.58 mm²	15.10 mm²

Comparison of two processors used in an ARM big.LITTLE System
[a]Values vary depending on implementation. Values presented are taken from Samsung SoC Exynos 5433 used in Samsung Galaxy Note 4. [13]
[b]First value is size of instruction cache, second value is size of data cache[11]
[c]Integer depth [10][11]
[d]Unit DMIPS / MHz stands for Dhrystone MIPS per Megahertz; see Appendix for additional information

C. ARM Processors

In this section I want to introduce and take a closer look at actual real-life implementations of the ARM architecture. The processors presented will be the components of the big.LITTLE system analyzed in the second part of this paper.

The processors most commonly found in the current (second) generation of big.LITTLE systems are the ARM Cortex A53 (A53) and the ARM Cortex A57 (A57). Both are 64-bit processors that share the same microarchitecture. However, they differ greatly in complexity. As shown in Table I the A57 outperforms the A53 by almost a factor of 2 when running at the same clock speed. On the other hand the Cortex A53 consumes significantly less power, as is demonstrated in Fig. 1. It is also noteworthy that the A53 running all four cores still consumes less or equal power than only one running core of the A57 at any clock-rate.

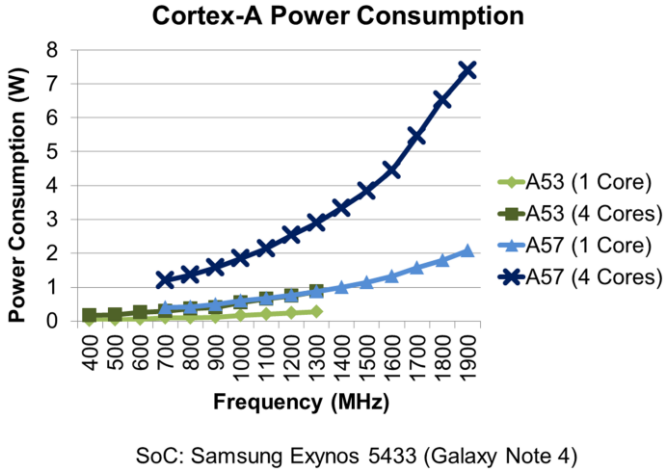

Fig. 1. Cortex-A power consumption in relation to clock frequency.
The Exynos 5433 SoC contains 4 Cortex A53 and 4 Cortex A57 processors. For the analysis of one cluster, the other was completely powered off. When measurements for only one core were conducted, the other three cores of the cluster were powered off entirely as well. [13]

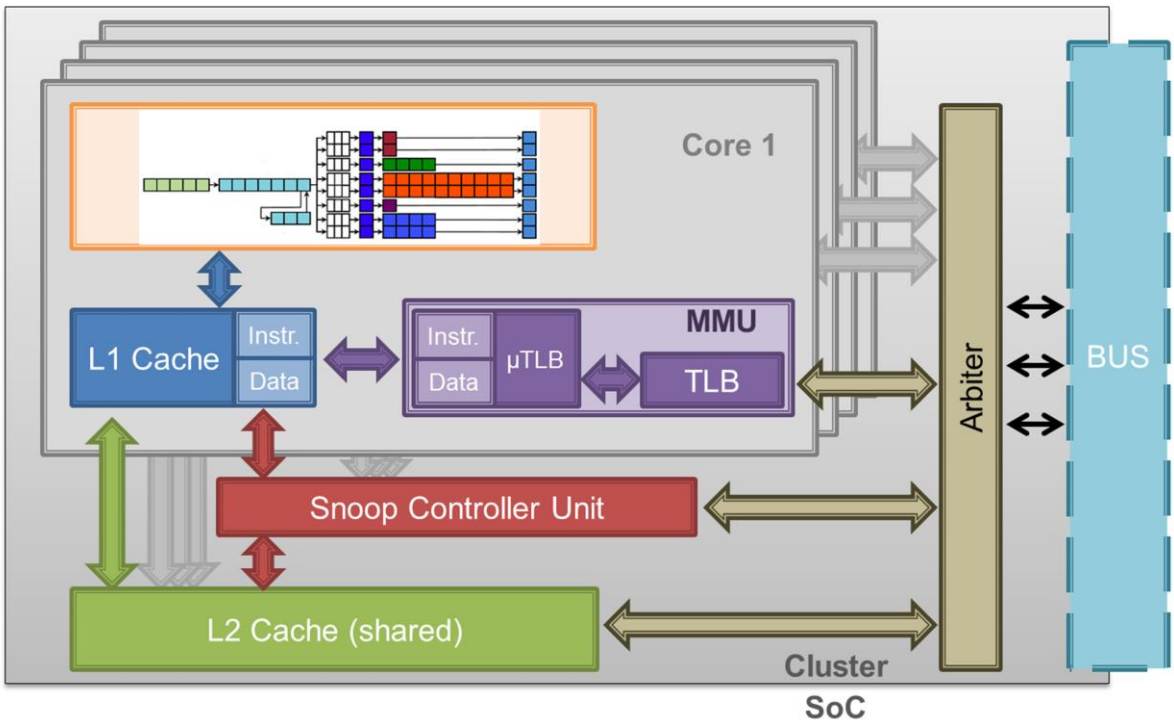

Fig. 2. Major components of a Cortex-A cluster and core [9][10][12]

Fig. 2 illustrates the basic architecture of Cortex-A processors. Next to the execution pipeline (orange) each core contains a *level-1* (L1) cache (blue) for instructions as well as data. The same is true for the *memory management unit* (MMU) (purple). All cores in one cluster share a large *level-2* (L2) cache (green). In addition there is a so called *snoop controller unit* (SCU) (red) in the cluster, located close to the L2 cache. The SCU keeps track of the contents and status of all L1 caches; however, it does not store the actual cache data itself. The SCU serves an important purpose in the big.LITTLE system as will be discussed in section III. The arbiter administers access of the individual cores to the shared resources and the bus.

III. ARM BIG.LITTLE – CONCEPT

After having introduced the two processors that will be used in the big.LITTLE system and some of their underlying microarchitectural features the following section is dedicated to explain the fundamental concept of big.LITTLE as well as its implementation. In the end I am going to take a closer look at performance and efficiency of such a system.

The Idea of big.LITTLE is to establish an environment in which the operating system (OS) does not have to distinguish between the two clusters anymore; instead of seeing two different quadcore clusters it appears as if there was only one octacore processor on the die. Such a system is no different from existing multicore processors and would allow handling multiprocessing in the traditional way. This "illusion" of one coherent cluster of cores is created by ensuring *binary compatibility*, *cache coherency* and *memory coherency*.

A. Binary compatibility

The first of the three might seem obvious but it is still noteworthy. Binary compatibility means that any program compiled for one of the two processors (A53 or A57) will also be executable on a core of the other cluster without any alterations to the code or having to recompile. [14]

B. Cache coherency

Once this prerequisite is met, the next task at hand is to physically connect the two processors and establish cache coherency of all eight cores.

1) Interface and Interconnect

This is achieved by extending the existing ***A**dvanced **E**xtensible **I**nterface* (AXI) – the common bus interface in ARM devices – with three more channels (address, data and response) called the *AXI Coherency Extension* (ACE). [8][12] Through this extended (yet backwards-compatible) interface the "key component" of the system can be connected to the two processors. This key component is the so called *Cache Coherency Interconnect* (CCI). It is a separate co-processor on the die which administers all the communication as well as establishes and maintains cache coherency between the processors. [6] Fig. 3 illustrates how the two clusters are connected through the CCI. To aide visualization and to put it its location and relationship to other components into perspective, the big.LITTLE system is shown in a simplified sample layout of a SoC design.

2) Coherency protocol

Cache coherency is a known concept already utilized in all conventional multicore processors. All of these implement some form of categorization for cache lines to trace their status within the system. All of these implementations share an underlying concept similar to the MESI (***M**odified **E**xclusive **S**hared **I**nvalid*) or MOESI (same as MESI with addition of ***O**wned*) protocol. [9]

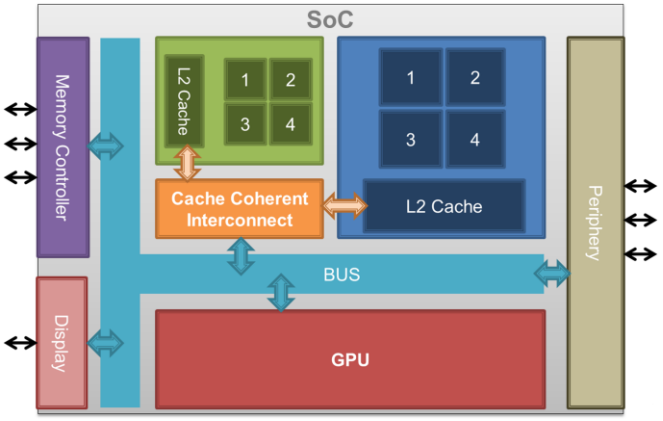

Fig. 3. Connecting big and LITTLE via the Cache Coherent Interconnect in a simplified sample layout of an SoC

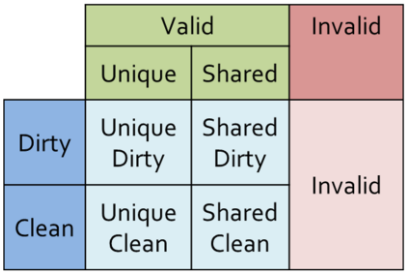

Fig. 4. ARM cache coherency states [9]

ARMs version of such a protocol is presented in Fig. 4. A protocol like this assigns each cache line one of five statuses. First of all, it can be *valid* or *invalid*. An invalid cache line either has not been filled with any data so far (e.g. after powering on) or it previously contained data which has recently been evicted to make room for data with higher priority. Valid cache lines are categorized even further. They can be *dirty* or *clean* depending on whether the data they contain has been altered from what is still stored in main memory (and hence main memory needs updated sooner or later) or not. Lastly they can also be *unique*, which means the cache line is only held in one cache and no others, or *shared*, indicating that several cores hold the line in their caches; these lines need to be kept coherent to avoid operations on invalid data. This protocol is then applied to the entire big.LITTLE system to establish cache coherency across clusters.

Since both processors – big and LITTLE – are multicore processors, cache coherence *within* a cluster is already assured by default. Therefore, the task at hand is to make those two previously independent cache systems coherent with one another as well. This is achieved by the CCI as follows:

3) Load commands

If the core wants to read data from main memory, the CCI receives that request and then automatically issues a snoop request to all the other clusters through the ACE interface. The snoop controller unit (SCU) receives that snoop request. It will – again automatically – snoop the L2 cache of its cluster as well as all the L1 caches of each core within the cluster. Keep in mind that meta-data for all L1 cache entries is stored within the SCU. Therefore the sole act of snooping the L1 caches does not actually require accessing them. If the requested data can be found in one of the caches, the SCU will return the data to the CCI through its ACE interface. The CCI eventually returns that data to the initiating core through the regular AXI interface. The data in both processors will be marked as "shared", indicating that the data is now present in more than one cache. In case the snoop request is unsuccessful the CCI will simply fetch the data from main memory, mark it as "unique", and also return it through the AXI interface. This means, that the initiating core cannot tell the difference between a successful (or unsuccessful) snoop of the CCI and a regular main memory read without interference of a CCI. The SCU on the receiving side also handles snoop requests independently. Therefore, the receiving cluster is also unaware of the snoop request taking place. Both processors are entirely unaffected by the operation of the CCI in the background. [7][9][10]

4) Store commands

For store/write commands also two cases can be distinguished. If the cache line has been marked as "unique" it can simply be stored in a conventional way. If, however, it is present in more than one cache and therefore marked "shared", cache coherency needs to be maintained.

The CCI will receive the request and broadcast a signal to all other clusters demanding the cache line to be made unique. This means, the receiving clusters now know that they have an out-of-date cache line and will evict it from their caches. After eviction each cluster responds to the CCI with an acknowledge-signal. The CCI will collect these acknowledgements until the cache line is unique again. Then it will allow the initiating core to perform the store command. Should other cores require that data again, they need to fetch it again as explained in the previous paragraph. [7][9][10]

C. Memory Coherency

In a multicore system threads running on several different cores might share the same virtual memory (e.g. if they belong to the same process). This virtual memory needs to stay coherent as well. To achieve this in a big.LITTLE system, where threads sharing the same virtual memory can be spread not only across cores but also across clusters (called *Distributed Virtual Memory* (DVM)), ARM uses a concept similar to the cache coherency approach.

Each core of a cluster contains a *Memory Management Unit* (MMU), which mainly consist of a *translation lookaside buffer* (TLB) (compare Fig. 2). TLBs are similar to caches. They save the results of the last few memory address translations. Addresses accessed several times in a given time-frame (which is very common in computer programs) do not have to be translated. The translation result can simply be looked up in the TLB.

However, if several cores share the same virtual memory,

any changes to these addresses need to be synchronized among all the cores involved.

In such a case an invalidation message is sent to the CCI, which in return will send a broadcast-signal to all other clusters and cores. The affected TLB entry will be invalidated. This is a one-way communication because TLBs are read-only. TLB entries can only be invalidated; there will be no response from the cores back to the CCI. Altered addresses have to be translated again by each core individually and stored in the respective TLB. [7][9]

After ensuring binary compatibility, cache and memory coherency there is no limitation anymore to scheduling threads to any of the available cores, no matter what cluster (big or LITTLE) they belong to.

IV. ARM big.LITTLE – Performance

A. ACE and CCI performance

The ACE interface can be clocked at integer fractions of the CPU clock, including 1:1, which allows operation at CPU clock speed. [8]

One cluster can issue and queue a total of 16 write commands. Each core can issue and queue 8 simultaneous read commands (a total of 32 per cluster). Each request is assigned an ID and the CCI will automatically keep track of the status of every request. [7] [10]

B. Snoop performance

The SCU is located within a cluster and therefore clocked at CPU clock speed. [10]

The SCU can handle 8 simultaneous snoops per cluster.

A snoop response will be ready after 13 cycles, if there has been a L2 cache hit. If the cache line was found in one of the L1 caches, the response will be ready after 16 cycles, which is also the worst case scenario. Note that the L1 and L2 caches are not inclusive.

If there is a cache miss and the requested cache line cannot be found in any of the caches, the negative response will be ready in 6 cycles. [10]

C. Performance example

Combining the results of *A* and *B* one can see, that a given core can issue 8 read requests and a receiving core can handle 8 snoop requests. Therefore, a maximum of 8 transfer transaction can be active between two cores at a time.

Since cache lines have a width of 64 byte and the ACE data channel is 128 bit wide, transferring one cache line will take 4 cycles (on top of the initial 13 - 16 cycle wait for the snoop response).

This means, that after a one-time wait of 16 cycles (worst-case scenario) 128-bits (16 Bytes) of data can be transmitted every cycle.

Transferring all 32kB of data contained in a L1 cache (compare Table I) will take

$$16 + \frac{32\ kB}{16\ B} = 2{,}013 \text{ cycles.}$$

Transferring the entire content of a 2MB L2 cache will take

$$13 + \frac{2\ MB}{16\ B} = 131{,}088 \text{ cycles.}$$

Assuming a A53 clocked at 1.3 GHz the L1 transfer would take about 1.5 µs, whereas the L2 transfer would take approximately 100 µs. As a comparison, an Intel E5520 desktop multicore processor clocked at 2.26 GHz will take about 6.5 µs for a context switch with a working set size of 32kB and 30µs for one with 2MB. [15]

A regular context switch on the big.LITTLE system will usually take about 20 – 30 µs, and is therefore absolutely comparable to any current processor and can even be considered fast if one keeps in mind that this context switch is not happening from core to core within a given processor but from one CPU *cluster* to another. [16]

D. Summary

Combining the measurements presented in Fig. 1 with the Dhrystone performance of each processor and assuming linear scalability (each core runs a separate Dhrystone thread) we can put performance and power consumption in a simple relation. Fig. 5 shows the result of this in comparison to having just a A53 or A57 cluster. Despite the linear and simple approach, this result is consistent with the findings of *S. Yoo et al.* [4].

The chart illustrates the two main advantages of using a big.LITTLE system: because LITTLE cores can be utilized over big cores whenever possible, the big.LITTLE system always consumes less power than a processor just consisting of a Cortex A57 given a specific performance level. This power advantage can be quite substantial with up to 70%.

There is also a smaller performance advantage of up to 25% due to the simple fact that there are eight cores in the system which all can be utilized at once. At this point thermal limitations need to be considered though. Mobile devices usually do not have any active cooling and having 8 cores run at full speed might easily increase the device temperature above a save threshold of 40-50°C.

The most important advantage of a big.LITTLE system in comparison to a processor just containing a A57, however, might be the availability of the LITTLE cores when idling or executing very low performance tasks. Here a big.LITTLE system can operate at power consumptions which are simply inaccessibly to the big-cluster-only processor.

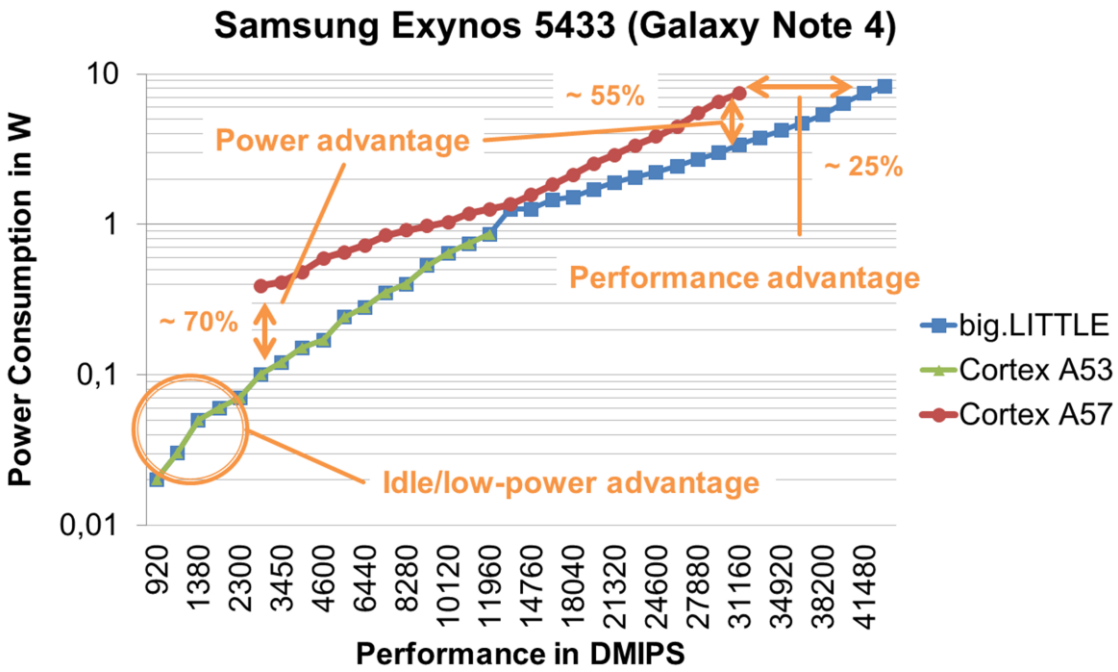

Fig. 5. Power / Performance comparison of a big.LITTLE system in comparison to the big and LITTLE processors individually.

V. Conclusion

To conclude this paper I want to stress again the three major advantages of ARMs big.LITTLE technology:

First, there is a performance advantage of up to 25% because of the availability of extra cores (8 versus 4).

Second, there are substantial power advantages of up to 70%; power is saved at all times and performance levels; low performance tasks benefit more.

Third, the availability of LITTLE cores allow extra low power consumption during idling or when in low-power modes (which is a very common scenario for mobile devices).

It is also noteworthy that the big.LITTLE system proved that a heterogeneous CPU is not only possible at all, but it is also feasible and performs well (enough). This new modularity opens up a lot of possibilities in the future to custom-make CPUs with exactly the properties required for a given task and reducing the need for compromise.

Appendix

A. *Dhrystone*

Dhrystone is a benchmark program for evaluating and comparing computer performance. Developed in 1984, the program only contains integer operations.

The output of the program is the number of Dhrystones per second. That is the number of iterations of the program per second, which is executed continuously in a loop.

The kind and amount of operations in the program are designed to mimic the typical utilization of a processor (with the exception of floating-point operations).

The benchmark result is said to be more appropriate for measuring processor performance than basic MIPS (million instructions per second). The result for MIPS can be very different depending on the (instruction set) architecture (e.g. RISC and CISC): processors with similar performance might need a very different amount of (simple or complex) instructions to achieve that performance. Therefore it might be more practical to measure how many times per second a given program can be executed (regardless of how many instructions are needed for executing the program once).

At the time Dhrystone was created a popular benchmark program for floating-point operations was “Whetstone”. Therefore the name was derived as a play on words *whet* / *wet* and *dhry* / *dry* respectively.

B. *DMIPS*

DMIPS are derived from the basic Dhrystone count when dividing the result by 1757. This was the number of Dhrystone iterations per second achieved by the Vax-11/780 processor, nominally an even 1MIPS machine. It is therefore a way to normalize the Dhrystone result and make it comparable to other MIPS evaluations. The Vax instruction set architecture is a CISC architecture.

C. *DMIPS/MHz*

DMIPS per Megahertz can be derived by dividing the DMIPS result by the clock frequency in MHz (megahertz) of the processor. This allows performance comparisons between processors with different clock speeds.

REFERENCES

[1] E. Blem, J. Menon, T. Vijayaraghavan, K. Sankaralingam. (2015, March). ISA Wars: Understanding the Relevance of ISA being RISC or CISC to Performance, Power, and Energy on Modern Architectures. *ACM Transactions on Computer Systems.* [Type of medium]. *Vol. 33, No.1, Article 3*. Available: http://tocs.acm.org/

[2] T. Mitra. (2014). Energy-Efficient Computing with Heterogeneous Multi-Cores, Presented at International Symposium on Integrated Circuits (ISIC).

[3] V. Villebonnet, G. Da Costa, L. Lefevre, J.-M. Pierson, P. Stolf. (2014). Towards Generalizing "Big.Little" for Energy Proportional HPC and Cloud Infrastructures. Presented at IEEE Fourth International Conference on Big Data and Cloud Computing.

[4] S. Yoo, Y. Shim, S. Lee, S.-A. Lee, J. Kim. (2015, October). A case for bad big.LITTLE switching: How to scale power-performance in SI-HMP. Presented at Hotpower'15, Monterey, CA, USA.

[5] *ARMv7 Architecture Reference Manual*, ARM Ltd., Cherry Hinton, Cambridge, 2014.

[6] *ARMv8 Architecture Reference Manual*, ARM Ltd., Cherry Hinton, Cambridge, 2015.

[7] *CoreLink CCI-400 Cache Coherent Interconnect Technical Reference Manual*, ARM Ltd., Cherry Hinton, Cambridge, 2012.

[8] *AMBA AXI and ACE Protocol Specification*, ARM Ltd., Cherry Hinton, Cambridge, 2013.

[9] *Introduction to AMBA 4 ACE and big.LITTLE Processing Technology*, ARM Ltd., Cherry Hinton, Cambridge, 2013.

[10] *ARM Cortex-A53 MPCore Processor Technical Reference Manual*, ARM Ltd., Cherry Hinton, Cambridge, 2014.

[11] *ARM Cortex-A57 MPCore Processor Technical Reference Manual*, ARM Ltd., Cherry Hinton, Cambridge, 2014.

[12] *big.LITTLE Technology: The Future of Mobile*, ARM Ltd., Cherry Hinton, Cambridge, 2013.

[13] A. Frumusanu, R. Smith. (2015, February). *ARM A53/A57/T760 investigated - Samsung Galaxy Note 4 Exynos Review*. Available: http://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review/

[14] H.-D. Cho, K. Chung, T. Kim. (2012, February). *Benefits of the big.LITTLE Architecture*. Samsung Electronics, Seoul.

[15] B. Sigoure, (2010, November) *How long does it take to make a context switch?* Available: http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

[16] K. Yu, (2012) *big.LITTLE Switchers – Evaluation on Exynos.bl Processor*. Presented at 2012 Korea Linux Forum. Available: http://events.linuxfoundation.org/images/stories/pdf/klf2012_yu.pdf