WHITE PAPER

big.LITTLE Technology: The Future of Mobile

Making very high performance available in a mobile envelope without sacrificing energy efficiency

Introduction

With the evolution from the first mobile phones through smartphones to today's superphones and tablets, the demand for compute performance in mobile devices has grown at an incredible rate. Today's devices need to service smarter and more complex interactions, such as voice and gesture control, combined with seamless and reliable content delivery. Gaming and user interfaces have also grown in complexity, with mobile devices now increasingly being used as gaming platforms.

High performance requires fast CPUs which in turn can be difficult to fit in a mobile power or thermal budget. At the same time battery technology has not evolved at the same rate as CPU technology. Therefore today we are in a situation where smartphones require higher performance, but the same power consumption.

The development and design of next generation mobile processors is necessarily guided by the following factors:

1. At the high performance end: high compute capability but within the thermal bounds
2. At the low performance end: very low power consumption

ARM big.LITTLE™ technology has been designed to address these requirements. Big.LITTLE technology is a heterogeneous processing architecture which uses two types of processor. "LITTLE" processors are designed for maximum power efficiency while "big" processors are designed to provide maximum compute performance. Both types of processor are coherent and share the same instruction set architecture (ISA). Using big.LITTLE technology, each task can be dynamically allocated to a big or LITTLE core depending on the instantaneous performance requirement of that task. Through this combination, big.LITTLE technology provides a solution that is capable of delivering the high peak performance demanded by the the latest mobile devices, within the thermal bounds of the system, with maximum energy effiency.

This paper is an overview of the technical aspects of big.LITTLE technology including the hardware components required for a big.LITTLE system, and the software required to manage it.

Same architecture but different micro-architectures

The first big.LITTLE processing pair consists of the ARM Cortex®-A15 and Cortex-A7 processors. Since both processors support the same ARMv7-A ISA, the same instructions or program can be run in a consistent manner on both processors. Differences in the internal microarchitecture of the processors allow them to provide the different power and performance characteristics that are fundamental to the big.LITTLE processing concept. Future designs will also utilise the Cortex-A53 and Cortex-A57 processors in a big.LITTLE implementation.

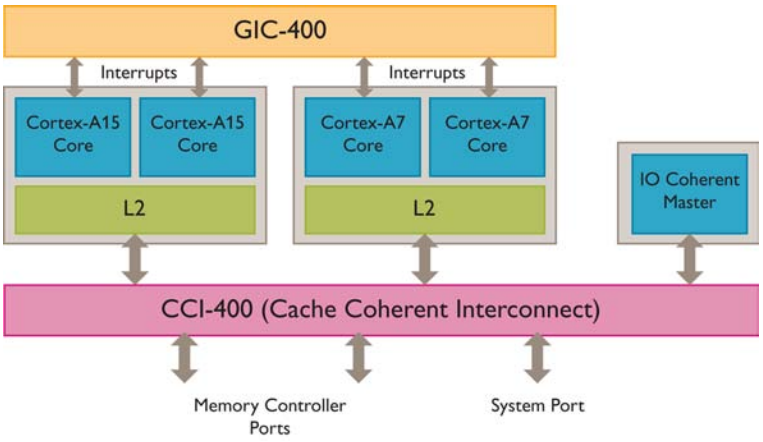

Fig 1: Typical big.LITTLE system

As an example,

Figure 2 describes the pipeline designs for the Cortex-A15 and Cortex-A7 cores. The Cortex-A15 core is designed to achieve high performance by running more instructions in parallel on a bigger and more complex pipeline. On the other hand, the Cortex-A7 core's pipeline is relatively simple and is designed to be extremely power efficient. The Cortex-A7 core's performance is lower than the Cortex-A15 core's but it is sufficient for most common usage scenarios executed by modern mobile devices. In fact, the Cortex-A7 core's performance is close to Cortex-A9 core, which powers most smartphones today.

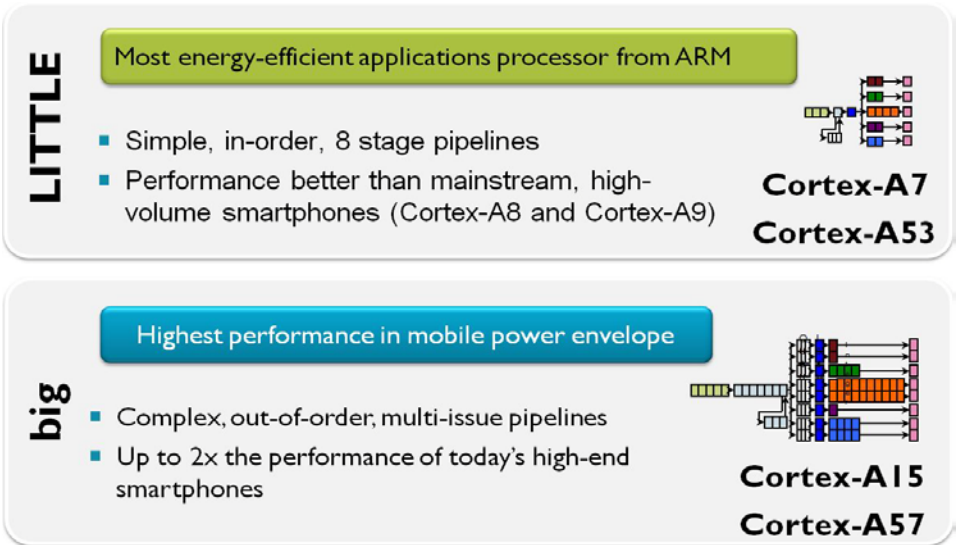

Figure 2: Cortex-A7 and Cortex-A15 pipeline

The basic idea of big.LITTLE technology is to dynamically allocate tasks to the right processor according to their instantaneous performance requirement. Different tasks have different and constantly changing performance and power requirements. In a typical system, most tasks can be carried out perfectly adequately by a Cortex-A7 core. However if the performance requirement goes above what can be delivered by Cortex-A7 cores alone, then one or more Cortex-A15 cores can be turned on. Performance hungry tasks can then be migrated to the Cortex-A15 core cluster. This provides great acceleration when it is needed. Thereafter, when the performance requirement reduces, tasks can be re-allocated to the Cortex-A7 cluster and one or more of the Cortex-A15 cores may then be turned off, quickly reducing power consumption.

Cache Coherency Interface and big.LITTLE Technology

The key ingredient that makes big.LITTLE technology possible is coherency. big.LITTLE software models require transparent and performant transfer of data between big and LITTLE processors. Hardware coherency enables this, transparently to the software. Without hardware coherency, the transfer of data between big and LITTLE cores would always occur through main memory - this would be slow and not power efficient. In addition, it would require complex cache management software, to enable data conherency between big and LITTLE processors

Figure 4 is an example of CPU subsystem consisting of a Cortex-A7 cluster, a Cortex-A15 cluster and a set of system fabric components which enable the seamless data transfer between clusters. This fabric is collectively referred to as a ”Cache Coherent Interconnect” – in this case the ARM CoreLink™ CCI-400 interconnect IP. The system is completed by the CoreLink GIC-400, which provides dynamically configurable interrupt distribution to all the cores.

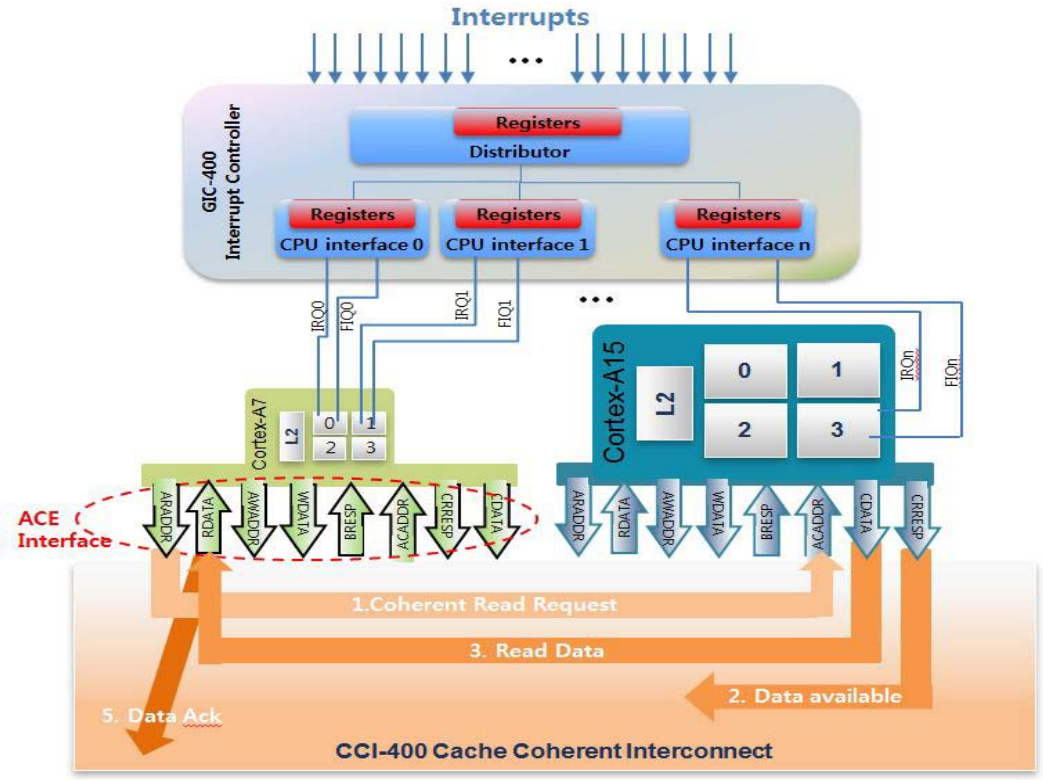

Figure 3: Cache coherency in a big.LITTLE system

As shown in Figure 3, the bus interfaces of Cortex-A15 and Cortex-A7 processors make use of the AMBA® AXI Coherency Extensions (ACE) to the widely-used AMBA AXI protocol. This protocol provides for coherent data transfer at the bus level. In the AMBA ACE protocol, three coherency channels are added in addition to the normal five channels of AMBA AXI. As an example, the lower part of Figure shows the steps in a coherent data read from the Cortex-A7 cluster to the Cortex-A15 cluster. This starts with the Cortex-A7 cluster issuing a Coherent Read Request through the RADDR channel. The CCI-400 hands over the request to the Cortex-A15 processor's ACADDR channel to snoop into Cortex-A15 processor's cache. On receiving the request from CCI-400, the Cortex-A15 processor checks the data availability and reports this information back through the CRRESP channel. If the requested data is in the cache, the Cortex-A15 processor places it on the CDATA channel. Then the CCI-400 moves the data from the Cortex-A15 processor's CDATA channel to the Cortex-A7 processor's RDATA channel, resulting in a cache linefill in the Cortex-A7 processor. The CCI-400 and the ACE protocol enable full coherency between the Cortex-A15 and Cortex-A7 clusters, allowing data sharing to take place without external memory transactions.

Software execution models for big.LITTLE

There are two major software models shown below in Figure 4.

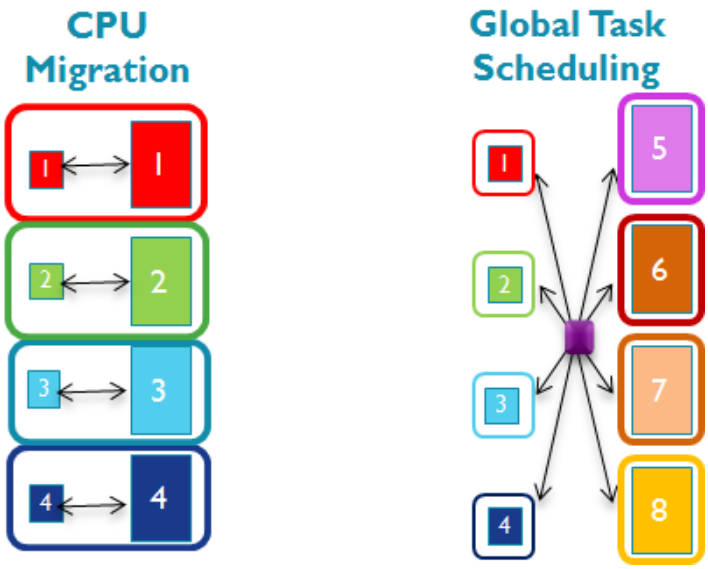

Figure 4 big.LITTLE Software Models

CPU Migration: In this model, each big core is paired with a LITTLE core. Only one core in each pair is active at any one time, with the inactive core being powered down. The active core in the pair is chosen according to current load conditions. Using the example in the figure above, the operating system sees 4 logical processors. Each logical processor can physically be a big or LITTLE processor and this choice is driven by dynamic voltage and frequency scaling (DVFS). This model requires the same number of processors in both the clusters.

The 'In Kernel Switcher' (IKS) solution from Linaro is an example of this model. This is available today from Linaro (http://www.linaro.org/linaro-blog/2013/05/02/the-linaro-iks-code-now-publicly-available).

Global Task Scheduling: In this model the scheduler is aware of the differences in compute capability capacity between big and LITTLE cores. Using stastistical data and other heuristics, the scheduler tracks the performance requirement for each individual thread, and uses that information to decide which type of processor to use for each thread. Unused processors can be powered off. If all processors in a cluster are off, the cluster itself can be powered off. This model can work on a big.LITTLE system with any number of processors in any cluster. Also, as we shall discuss in detail in later sections, the reaction time of this model to load variations of individual tasks can be much faster than that of the CPU migration model which is a significant advantage.

Through the development of big.LITTLE technology, ARM has evolved the software models starting with various migration models through to Global Task Scheduling (GTS). GTS is a sophisticated, flexible and popular model which shall be the focal point of all future development. This paper focuses exclusively on GTS and ARM's implementation of GTS, known as big.LITTLE MP.

Global Task Scheduling

In Global Task Scheduling, the OS task scheduler understands the differences in compute capacity between the big and LITTLE processors in the system. The scheduler tracks the compute requirements of each individual thread and the current load state of each processor, and uses this information to determine the optimal balance of threads between big and LITTLE processors. This approach has a number of advantages over CPU Migration:

1. The system can have different numbers of big and LITTLE cores.
2. Any number of cores may be active at any one time. When peak performance is required the system can deploy all cores. With CPU Migration only half of the cores may be active at any one time.
3. It is possible to isolate the big cluster for the exclusive use of intensive threads, whilst light threads run on the LITTLE cluster . With CPU Migration, all the threads in a processor transfer together. This allows heavy compute tasks to complete faster, as there are no additional background threads.
4. It is possible to target interrupts individually to big or LITTLE cores. The CPU Migration model assumes all context, including interrupt targetting, migrates between big and LITTLE processors.

The crux of this solution is being able to determine which tasks are intensive and which are light and to track this in real-time. The scheduler does this by tracking the load average of each thread across its running time. The basic idea behinds ARM big.LITTLE MP solution is depicted in Figure 6 below. The scheduler tracks the load of each thread, as a historical weighted average ofacross the thread's running time. The calculation is weighted so that recent task activity contributes more strongly than older past activity.

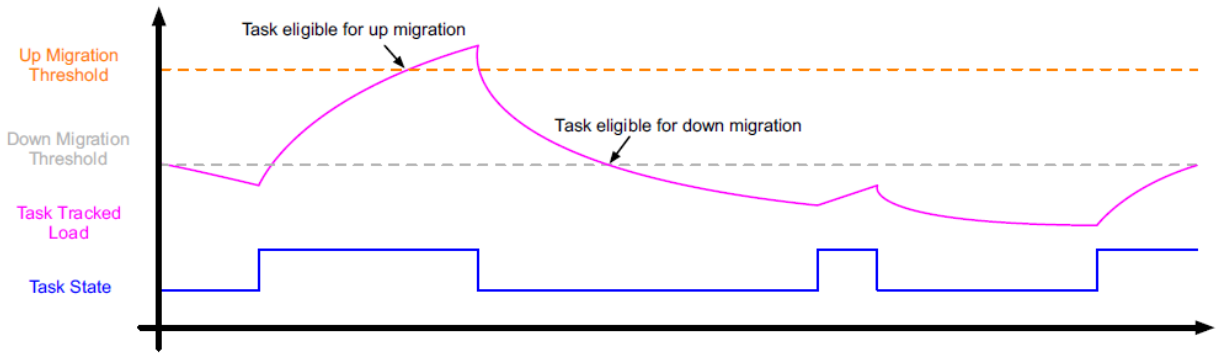

Figure 5: Tracking the load of a task

The ARM big.LITTLE MP solution uses the tracked load metric to decide whether and when to allocate a thread to a big or LITTLE core. This is done using two configurable thresholds: the "up migration threshold" and the "down migration threshold". When the tracked load average of thread, which is currently allocated to a LITTLE core, exceeds the up migration threshold, the thread is considered eligible for migration to a big core. Conversely, when the load average of a thread which is currently allocated to a big core drops below the down migration threshold, it is considered eligible for migration to a LITTLE core. In ARM's big.LITTLE MP solution these basic rules govern task migration between big and LITTLE

cores. Within the clusters, standard Linux scheduler load balancing applies. This tries to keep the load balanced across all the cores in one cluster.

We refine the model by adjusting the tracked load metric based on the current frequency of a processor. A task that is running when the processor is running at half speed, will accrue tracked load at half the rate that it would if the processor was running at full speed. This allows big.LITTLE MP and DVFS management to work together in harmony.

The ARM big.LITTLE MP solution uses a number of software thread affinity management techniques to determine when to migrate a task between big and LITTLE processors: fork migration, wake migration, forced migration, idle-pull migration and offload migration.

Fork Migration

Fork migration operates when the fork system call is used to ceate a new software thread. At this point, clearly no historical load information is available – the thread is new. The system defaults to a big core for new threads on the assumption that a ”light” thread will quickly migrate down to a LITTLE core as a result of wake migration (see below).

Fork migration benefits demanding tasks without being expensive. Threads that are low intensity and persistent, like Android system services, will only get migarted to the big processors once at creation time, quickly moving to the more suitable LITTLE processors therafter. Threads that are clearly demanding throughout, won’t get penalised by being made to launch on the LITTLE core first. Threads that are episodic but tend to require performance on the whole will benefit from being launched on the big core and will continue to run there as needed.

Wake Migration

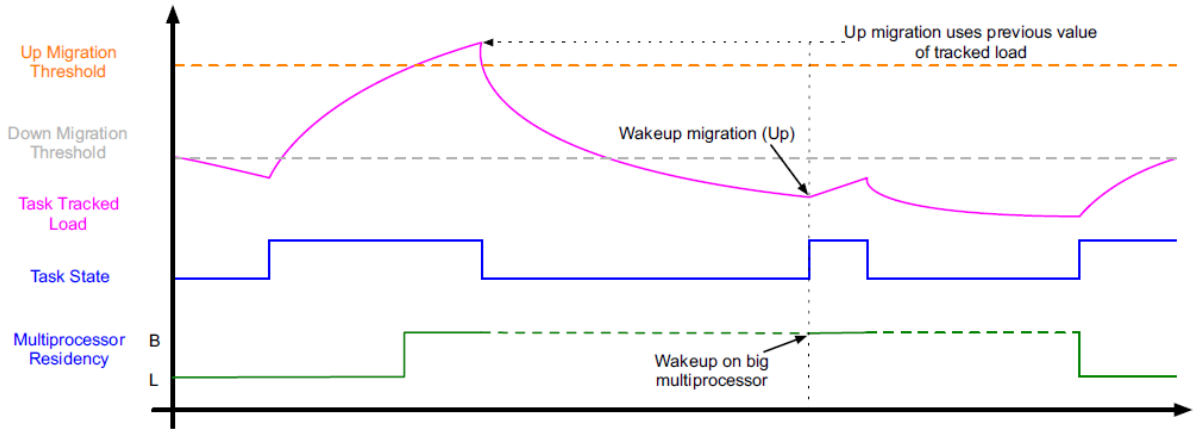

Figure 6: Wake Migration

When a task that was previously idle becomes ready to run, the scheduler needs to decide which processor will execute the task. To choose between big and LITTLE cores, the ARM MP solution uses the tracked load history of a task. Generally, the assumption is that the task will resume on the same cluster

as before. Critically, the load metric does not actually get updated for a task that is sleeping. Therefore, when scheduler checks the load metric of a task at wake up, before choosing a cluster to execute it on, the metric will have the value it had when the task last ran. This depicted above in Figure 6. This property means that tasks that a periodically busy will always tend to wake up on a big core. A task has to actually modify its behaviour, to change cluster.

If a task modifies its behaviour, and the load metric has crossed either of the up or down migration thresholds, the task may be allocated to a different cluster. Rules are defined which ensure that big cores generally only run a single intensive thread and run it to completion, so upward migration only occurs to big cores which are idle. When migrating downwards, this rule does not apply and multiple software threads may be allocated to a little core.

Forced Migration

Forced migration deals with the problem of long running software threads which do not sleep, or do not sleep very often. Periodically the scheduler checks the current thread running on each LITTLE core. If it's tracked load exceeds the upmigration threshold the task is transfered to a big core. This is depicted in Figure 7 below.

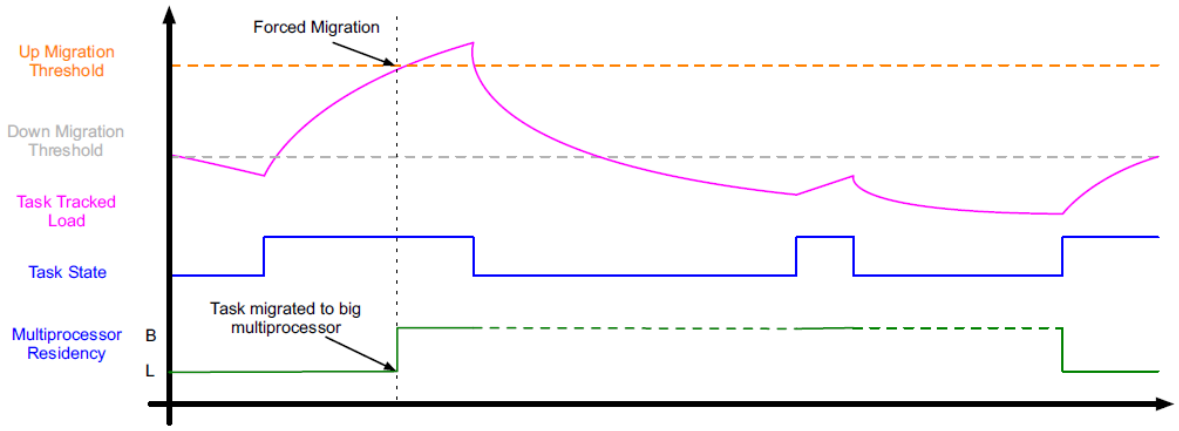

Figure 7: Forced Migration

Idle Pull Migration

Idle Pull Migration is designed to make best use of active big cores. When a big core has no task to run, a check is made on all LITTLE cores to see if a currently running task on a LITTLE core has a higher load metric that the up migration threshold. Such a task can then be immediately migrated to the idle big core. If no suitable task is found, then the big core can be powered down.

This technique ensures that big cores, when the are running, always take the most intensive tasks in a system and run them to completion. Idle-pull migration is very beneficial for performance benchmarks.

Offload Migration

The big.LITTLE MP solution requires that normal scheduler load balancing be disabled. The downside of this is that long-running threads can concentrate on the big cores, leaving the LITTLE cores idle and under-utilized. Overal system performance, in this situation, can clearly be improved by utilizing all the cores.

Offload migration works to peridically migrate threads downwards to LITTLE cores to make use of unused compute capacity. Threads which are migrated downwards in this way remain candidates for up migration if they exceed the threshold at the next scheduling opportunity.

Similar to idle-pull migration, offload migration is very beneficial for performance benchmarks.

Results

Figure 8 shows CPU and SoC level power savings for a variety of representative mobile use-cases. When compared to a system composed only of big Cortex-A15 processors, a big.LITTLE system running ARM big.LITTLE MP implementation shows substantial power savings.

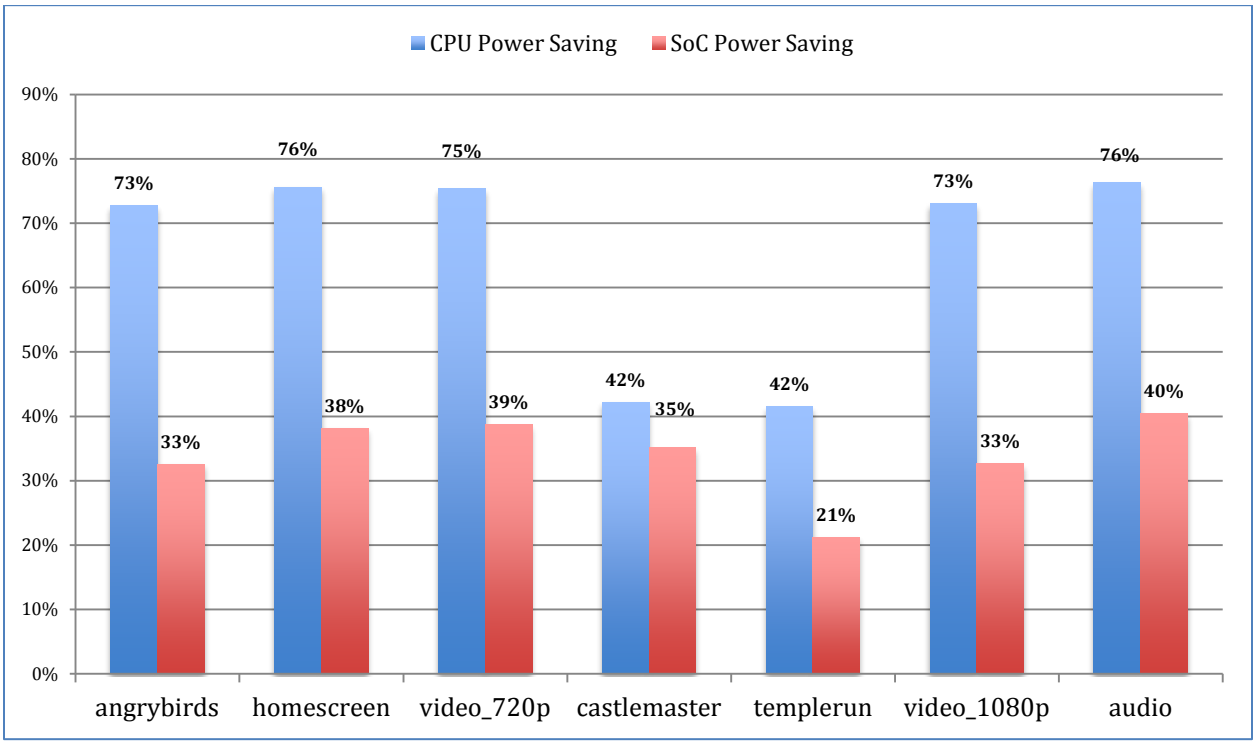

Figure 8 big.LITTLE MP Power Savings compared to a Cortex-A15 processor-only based system

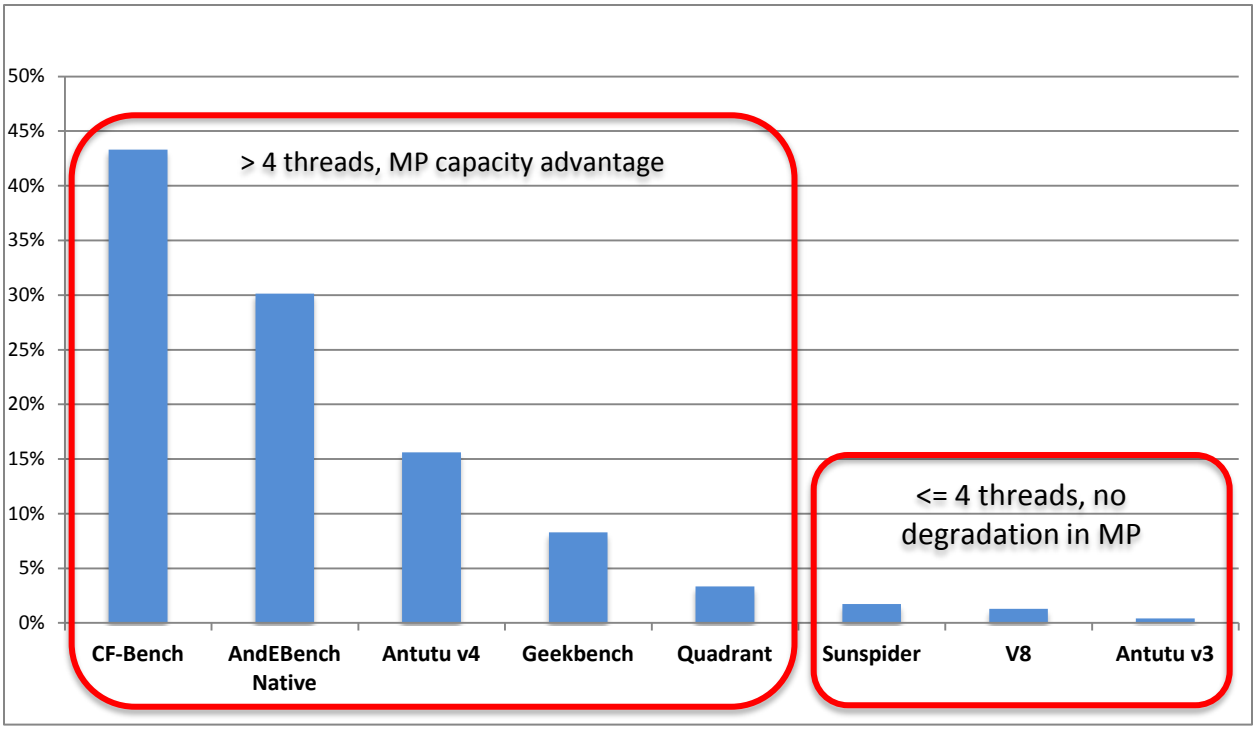

Figure 9: big.LITTLE MP Benchmark Improvements

Figure 9 shows how the big.LITTLE MP model benefits benchmarks. The comparison is between a big.LITTLE system composed of four LITTLE processors and four big processors and a system composed of only four big processors.

The software thread affinity management techniques discussed earlier result in substantial performance gains for threaded benchmarks where the number of threads is greater than four. In this situation on the system under test, big.LITTLE MP enables the use of more processors to aid the benchmark. Offload migration helps with spreading the number of compute intensive benchmark threads to the LITTLE processors when the big processors are busy and overloaded. Idle-pull migration results in the best utilisation of the big processors which effectively work as accelerators.

For those benchmarks with fewer threads, using big.LITTLE MP either provides no degradation or a marginal but noticeable improvement. Compared to the test system with only four big processors, the dynamic software thread affinity management will promote better utilisation of the big processors which will not be encumbered with low intensity and frequent running threads (such as system services) or interrrupts.

Conclusion

The ARM big.LITTLE MP technology has been well qualified with Android on multiple silicon implementations. The code is self contained and freely available as a drop-in into the vendor stack. It is interesting to note that the code doesn't require any significant modification or tuning. The only requirement is that the platform board-support package be well tuned in terms of DVFS and idle power management, allowing the scheduler extensions to focus on getting the job done.

The big.LITTLE MP scheduler extensions are available in two forms:

1. As a part of monthly Linaro Stable Kernel releases for the ARM TC2 platform. These releases, also known as LSK releases, contain a complete Android software stack for TC2 based on a very recent linux-stable kernel. The stack is available in source form and also as a pre-built binary set complete with boot firmware, boot loaders, ramdisk images and an Android root filesystem image. See https://releases.linaro.org/13.09/android/vexpress-lsk for details on the LSK.

2. As an isolated patch set against the LSK's kernel. See <https://wiki.linaro.org/ARM/VersatileExpress?action=AttachFile&do=get&target=big-LITTLE-MP-scheduler-patchset-13.08-lsk.tar.bz2>.