Multi-threading technology and the challenges of meeting performance and power consumption demands for mobile applications

September 2013

Navigating between ever-higher performance targets and strict limits on power consumption and heat dissipation is a major challenge faced by engineers today designing next-generation silicon platforms for mobile applications. One increasingly heralded technology is the implementation of multi-threading in multi-core processor systems. This article examines the pros, the cons and the opportunities for multi-threading.

Introduction

The increasing ubiquity of the smartphone along with a myriad of other smart mobile consumer devices is changing personal communication. But it is the smartphone that perhaps has become the leading wirelessly connected device used by people to browse the web, interact on social networks, play audio or video, or receive and send emails – in addition of course to the more traditional and prosaic activity of using it to speak to another human being. However, rather than the 'always connected' tasks of voice calls, text messages and emails, it is applications such as browsing, video streaming, gaming and navigation that are increasing the demand for higher performance silicon. What is more, this next-generation silicon has to enable increased battery life and also meet strict thermal requirements.

There are well-known question marks over the ability of single-core processors to meet these increasing demands as they start to reach performance limits. The latest microprocessors can now integrate billions of transistors on a single silicon die. Historically this increasing integration has been largely governed by Moore's Law, which concerns the doubling every two years of the number of transistors that can be implemented in a given area of silicon. However, it is proving increasingly difficult to utilize these additional transistors to accelerate the execution of a single sequential program.

The issue is that processors tend to become less efficient with increasing single-thread performance. The rationale being that to achieve higher single thread performance requires an increasing number of complex structures to exploit higher levels of Instruction Level Parallelism (ILP). However, these additional structures are not necessarily performing useful computational tasks, as they are more likely to be involved in scheduling and managing instructions in an order

that can get the most out of the hardware.

An examination of a multi-issue out-of-order processor core[1] will indicate that a substantial amount of the logic, and also the power, of the processor is dedicated to this scheduling and management functionality, and that the proportion of this functionality is much greater than in a more simple, single or dual issue core. This scheduling work is effectively overhead; it adds to the energy required to perform the computation tasks, such as data processing, so inherently undermining efficiency.

In addition, in order to provide ever more instruction level parallelism, more and more execution units are placed within a single processor core. This, combined with the scheduling hardware, offers the possibility of higher single thread performance, but the challenge of exposing sufficient instruction parallelism to keep the execution units busy grows rapidly. On the occasions that such parallelism cannot be found, or when bubbles in the pipeline exist as occurs on a branch mis-prediction, these execution units are under-utilized. Such under-utilization is clearly a source of inefficiency.

Single-thread multicore solutions

Over the past few years there has been a general move to using a higher number of individual processor cores in microprocessors to overcome these single-core performance limits. This model has seen wide acceptance across the industry with increasing proliferation of multicore solutions not only in mobile but also in a multitude of general computing and embedded application environments.

Each of these cores runs a separate thread of execution, and so by having multiple processor cores, the overall aggregate throughput from a given transistor budget can be increased, provided of course the computational problem can be expressed with thread-level parallelism.

Multi-threading

Multi-threading technology was first developed in the 1990s and famously introduced into mainstream PC processor applications in 2002. In essence multi-threading uses one physical processor core to run two threads of execution simultaneously, creating two ‘logical’ processors. This offers the promise of addressing the under-utilization problem of the complex processor by using a second thread as a source of independent instructions.

By addressing the under-utilization, it seems that the efficiency of the complex core can be improved; there are fewer occasions when the expensive multiple execution units are not being used, so the efficiency, expressed as computation performed per unit of energy consumed, is improved.

[1] *An out-of-order processor reschedules a sequence of instructions that were previously designed for execution in a specific order to maximize use of processor resources. The processor executes these rescheduled instructions and rearranges them in the original order to be written to memory.*

It is important to realize that when the physical processor core is running two threads of execution in this way, then the throughput of each thread is usually substantially reduced when compared with the throughput that one of those threads would have achieved had it been running on its own. However, by finding opportunities to use the under-utilized execution units, the prospect of multi-threading is that the aggregate throughput of the two threads running simultaneously is greater than would have been achieved had they run sequentially, so leading to higher performance, and lower energy consumption.

Using the premises above, a common conclusion is that a processor designed for multi-threading can bring performance and efficiency advantages: specifically, additional throughput performance against approximately half the additional power consumption – although it does comes with a slight increase in silicon die area because of the requirement for additional hardware execution resources. The addition of multi-threading to an out-of-order core delivers a small improvement in throughput per Watt, and also in terms of throughput per mm^2 compared with that achieved by the equivalent out-of-order core without multi-threading. So on initial observation multi-threading begins to sound like a compelling proposition, but looking deeper into the subject unveils a number of uncertainties.

Two logical processors

One way to look at multi-threading is that it is a method of making a large processor core look like two small processor cores, each one of which offers relatively low single-thread performance but with the native efficiency of the large core. As observed earlier when the large processor core is using its multi-threading capabilities simultaneously, the throughput of each of the threads is substantially less than it would have been had the thread had sole use of the large processor. However, this processor core still uses all of the complex scheduling and management structures that are inherent in that large core, and that are contributing so much to decreasing efficiency of the large processor.

Given that the management and scheduling structures of the large processor contribute so much to the energy consumption of a complex, out-of-order processor, it is inefficient to have to use such logic for each thread while achieving, for that thread, a throughput that could be achieved by simply using a much smaller, simpler and more efficient processor core.

A more-efficient alternative can be to implement two processors or cores with low single-thread performance. When the single-thread performance of a large processor is not required, it can be argued that it is preferable to put it on two small processors rather than using a large processor disguised as two small processors.

Large core performance

However, as discussed above, for larger cores and typically those embedding out-of-order execution, the efficiency of multi-threading becomes significantly less with the inherent inefficiencies from out-of-order mechanisms. As previously mentioned, increasing single-thread performance involves increasing the number of execution units and the exposure of increasing levels of ILP. This typically involves tracking increasing numbers of instructions, on-the-fly,

which requires larger rename pools and reservation stations, as well as more hazard checking. Practical experience of large processors shows that this additional logic tends to suffer from diminishing returns where the additional energy overhead of these resources increases faster than average performance.

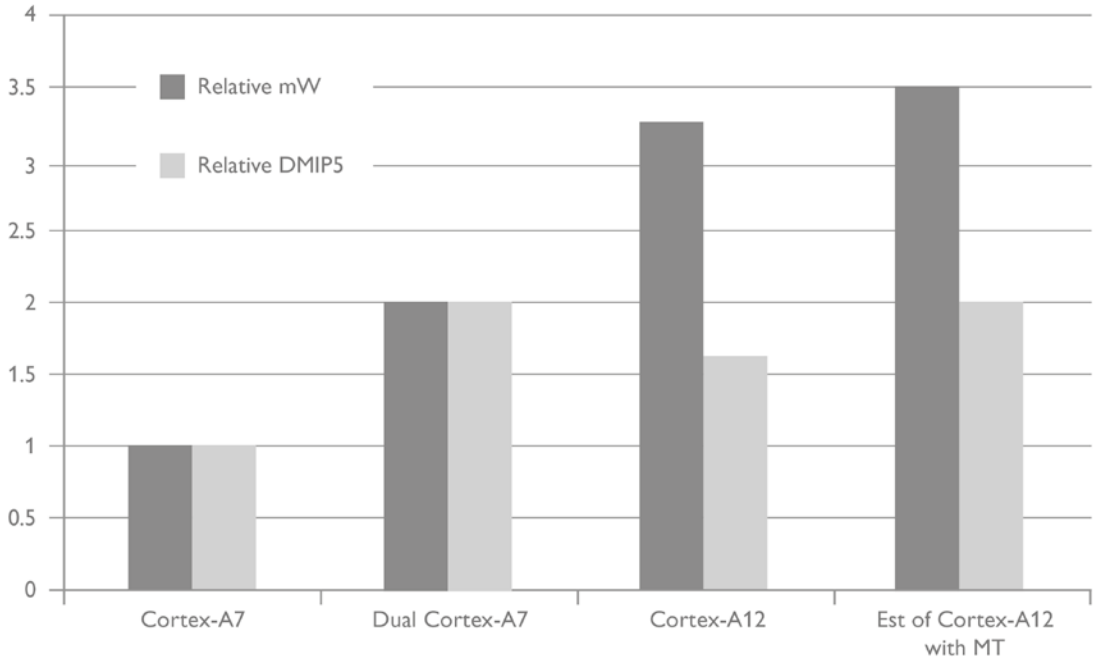


Fig. 1 Power vs performance - single thread Cortex-A7 vs multi thread Cortex-A12

When compared with an in-order processor such as the Cortex-A7 processor, an out-of-order high performance processor such as the Cortex-A12 processor adds significant additional hardware resources, including register rename engines and execution dependency tracking. This additional logic enables a higher level of single-thread performance but it also means higher energy consumption per instruction. Evaluation tests by ARM have shown that performance scales such that for highly complex processors, increasing performance by 50% will cost more than a 50% increase in terms of power. The figure above shows an example of that scaling comparing a Cortex-A7 processor with a Cortex-A12 processor and an ARM estimated multi-threaded version of the Cortex-A12 processor.

Looking at the typically reported costs in die area and power consumption versus the performance-throughput benefits of adding multi-threading capability to a typical out-of-order processor core, the incremental throughput benefits are approximately twice that of the incremental costs in die area and power consumption.

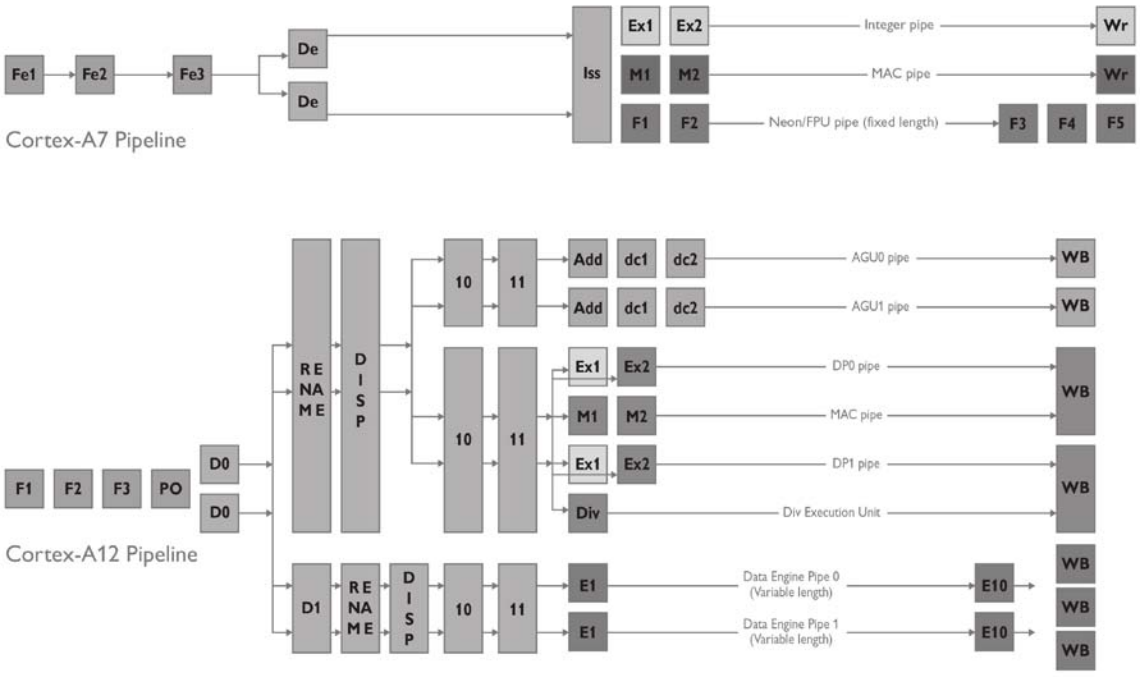


Fig.2 Relative pipeline complexity, Cortex A7 vs Cortex-A12

Based on this premise, it has been estimated that adding multi-threading to a high-performance Cortex core, the Cortex-A12 core for example, would result in a processor that offers approximately the same throughput as two Cortex-A7 cores. However, this hypothetical multi-threaded high-performance Cortex core, while delivering an equivalent throughput would use twice the aggregate power of the two Cortex-A7 cores and would have a die area approximately 10 percent larger than the combined Cortex-A7 cores. This assumes that both systems have a level 2 cache of the same size, which would be consistent with delivering an equivalent throughput.

Predictability and additional hardware cost

Multi-threading has other considerations that further undermine its value. For one thing, it is much less predictable than those of multi-core solutions. Multi-threading involves sharing pipeline resources between two independent threads; and the characteristics of each thread determine its use of pipeline resources. This leads to an unenviable choice for the processor designer: either the processor will be optimized for single-thread use, at which point it risks starving one of the threads when used in multi-threading; or it will be designed for multi-thread use, at which point it is carrying more resources than is efficient for single-thread performance.

In addition, sharing a level 1 cache leads to cache interference effects such as cache thrashing and destructive cache interference, which have been reasonably well documented in research into multi-threading. However, carefully written code can actually deliver ‘constructive cache interference’, where one thread conveniently brings in a piece of data that a second thread is just about to use, thereby enabling a prefetching effect. But this is only really applicable in well-controlled closed environments. For the independent workloads that are commonplace in open

systems it is in a minority compared with the destructive effect and trying to use it is extremely difficult, especially under a single operating system such as Linux or Android. Of course, a bigger cache can be used, but with a resulting increase in silicon area, thereby undermining any saving in area that comes from implementing multi-threading.

Power saving and design complexity

Another issue is that of power-down modes, which are important for energy saving and extending battery life. In multi-threading, one core is executing two threads which means significantly more sharing of resources between the two different threads; control logic for the second thread is heavily integrated with the control logic for the first thread on a single core. Therefore, turning off one of the two 'logical' cores because of low workload results in a significantly more complex task of turning off the myriad of logic circuits spread throughout the core. Compare that with the relatively simple task of turning off a discrete and separate core.

In addition, as two completely separate threads are being executed on the same physical piece of silicon hardware, it becomes much harder to guarantee perfect isolation than if they were being executed on separate cores. Potentially, this can lead to significantly longer processor validation for silicon vendors.

The Case for Small Core Multi-threading

The argument above is all about using multi-threading on large, complex cores with substantial hardware and energy overhead for scheduling and control. But multi-threading can also be used in smaller processors, where the argument becomes fascinatingly different.

Some tasks are inherently ill-suited to using large complex processors due to a fundamental lack of predictability in the memory accesses. Such *memory-bound* tasks tend to stall processor cores, while they wait for data from slow memory to enable the processing to proceed. If this happens there is very little benefit in using a large complex processing core capable of exploiting high instruction level parallelism, as such a processing core remains stalled. A simple analogy would be taking a high performance sports car through the centre of a city with many red lights. Running such tasks on simple processors cores, that don't have the hardware and energy overhead for scheduling and control, is a very efficient solution.

In such workloads, which are common in situations such as networking data plane applications, it can be beneficial to being able to run a second thread, or more, in the times when other threads are stalled. The act of stalling is wasteful in energy, as parts of the system will inevitably consume power during these stalls, either through leakage or from dynamic power of clocks that cannot be fully stopped.

Such a system is different from the large core multi-threading as each thread is not going through a processor core that has been designed for much higher throughput, and the vast majority of the energy spent in the processor core is being spent doing the essential tasks of computation, rather than management overhead. Correspondingly, there are efficiencies to be

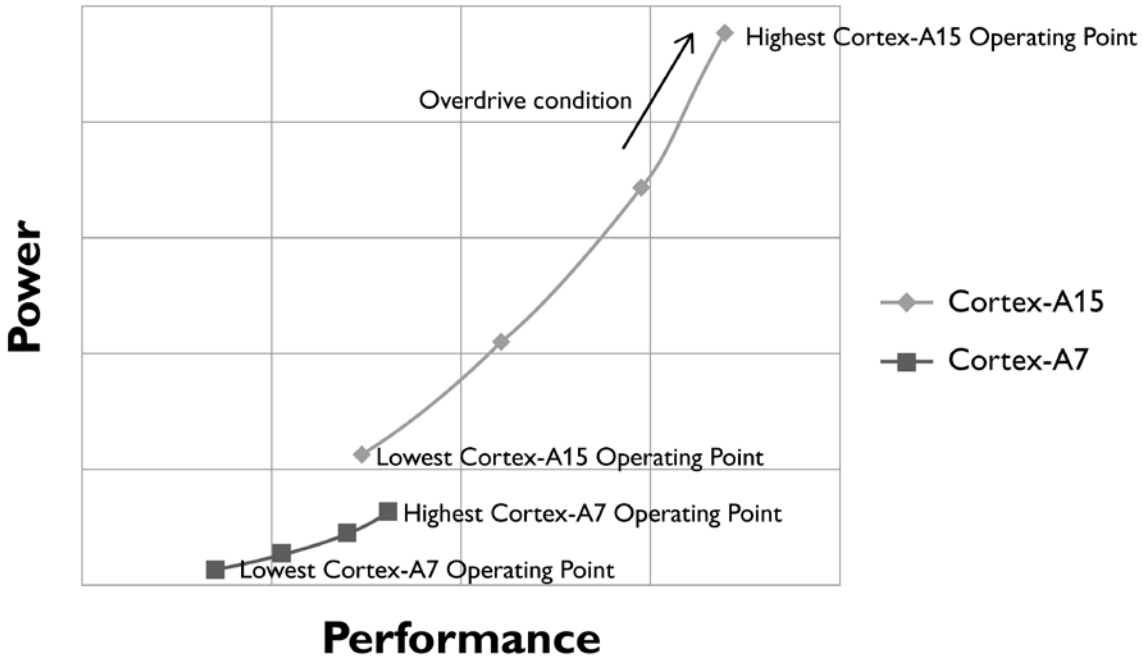
gained by finding alternative threads to run while the first task is stalled, so constructively using the energy that would be otherwise wasted during the stalls.

Core Combinations

As previously mentioned, the dominant solution used today in the majority of mobile phone platforms is the single-threaded multi-processor solution. And increasingly, this solution is evolving to be a combination of multiple processors that offer significantly different levels of performance. A key industry example is the ARM big.LITTLE processing approach, which combines high-performance cores with highly efficient cores in a multi-core processor system. A central tenet of the platform is running small workloads on small and efficient processors, whereas multi-threading is about running small workloads on larger, more complex and therefore less-efficient processors.

In big.LITTLE processing, workloads are dynamically and instantly transitioned to the appropriate core based on the performance demand, Low-intensity tasks such as scrolling, messaging, text entry or simple applications will run on the ‘LITTLE’ core leading to better efficiency. High-intensity tasks such as application loading and initializing, video conversion and stream and complex games will run in the ‘big’ processor to deliver the high single thread throughput needed to make the system responsive to the user. There is a trade-off with die area, but in most systems the more important factors are power consumption and efficiency, which helps provide longer battery life.

In the first big.LITTLE system, the ‘big’ core is a Cortex-A15 processor, paired with the ‘LITTLE’ Cortex-A7 processor. By coherently connecting the Cortex-A15 and Cortex-A7 processors, the system is flexible enough to support various use models, which can be tailored to the processing requirements of the tasks.



With big.LITTLE processing, if the application to be run requires a level of performance that can be met by running on the Cortex-A7 core, then the operating system will schedule that task to run on the Cortex-A7 core. This exploits the inherent efficiency of the Cortex-A7 core, while meeting the needs of the application. However, if the task requires the performance of the Cortex-A15 core, then the task will be scheduled by the operating system to the Cortex-A15 core.

While this appears to create complexity for the operating system, the complexity is in principle little different from the complexity of an operating system that schedules effectively for multi-threading. In a multi-threading system, the operating system must decide whether the application to be run can have its performance requirements met by running as a second thread on a multi-threaded core, or has a higher performance requirement meaning it must be run as the sole occupant of a core. Indeed this scheduling problem for multi-threading is made worse by the fact that the decision to schedule a second thread on a multi-threaded core affects the performance of both threads on that core, so the scheduler must also consider the needs of the current occupant of that core.

Further information on big.LITTLE is available at thinkbiglittle.com.

Conclusions

Multi-threading is a significant processor technology and it is highly likely that it will see growing implementation in a few specific applications. Particularly of interest is the technology's possibilities in networking applications and more specifically 'small-core' multi-threading. This is undoubtedly a compelling use that should easily find a place in network data plane applications.

However, in mobile applications, where the performance/power efficiency ratio is crucial, and particularly where applications require larger superscalar and out-of-order processor based multicore designs, then single-threaded multicore implementations such as big-LITTLE are the most efficient solution, certainly today and probably for the next few generations.

ENDS