

Developing Power-Efficient Software Systems on ARM Platforms

By Chris Shore, ARM Training Manager

Given the emphasis on battery life for portable devices, it would seem strange that there are very few software engineers who actually have energy reduction in their daily project accountabilities. I suspect that those who do give the subject some thought are likely to do it on a “commendation vs. court martial” basis. We are entering a period when this will have to change. As battery life and performance requirements continue to fight with each other we, as software engineers, will be forced to spend a lot more time looking at how we can design and write our software in an energy-efficient way – at least until the tools catch up with us.

As engineers, we all love finding geeky solutions to the problems which we come across. It may come as a surprise to find that, in this particular area, there are none. Clever tricks may save some power, but the field is dominated by other, simpler considerations. There are several very large elephants in this room and we must be careful to hunt the elephants we can see, before spending significant effort chasing smaller mammals.

When looking at the power consumption of a system, it is important to be clear about what we are actually measuring. When we say “power-efficient” we can mean several things. Do we mean “power” or “energy.” In practice we need both. Most handheld portable devices have two distinct budgets here: a power budget which governs instantaneous power consumption and prevents over-heating or thermal stress, and an energy budget for total long-term energy use. Software is required to stay within the power budget over the short-term and within the energy budget over the longer term.

Clearly, we could reduce the power consumption of any device to near-zero simply by having it do nothing – or nothing meaningful at any rate! Unavoidably, carrying out useful functions requires energy use. So, this game is a continuous compromise between doing something meaningful and saving power. In order to carry out the functions we need, we must burn energy; consequently, we must ensure that those functions are carried out in as power-efficient a manner as possible.

The Energy-Delay Product

A better metric, commonly used in the academic material on this subject, is the “Energy-Delay Product.” Although it has neither standard units nor methodology, it combines energy consumption with a measure of performance. Increasing energy use or decreasing performance will increase EDP, so what we seek is the lowest acceptable value of EDP – in other words, the lowest energy use consistent with carrying out the required tasks within the time allowed.

Where does the energy go?

All computing machines carry out two essential functions. And they are both essential – without both no meaningful tasks can be accomplished.

Computation – or data manipulation – is naturally what we think of first. Typically, computation is carried out on values held in machine registers. In order to carry out computational tasks as efficiently as possible, we need to execute the smallest possible number of instructions in the shortest possible time. Most importantly, efficient computation allows one of two things: either we can finish earlier and go to sleep or we can turn down the clock speed and still complete within the allotted time.

What is often neglected is the aspect of communication - moving data around. In the majority of architectures (ARM, as a load-store architecture, is no exception) data movement is essential. You cannot process any information without moving it from one place to

another and often back again. Values in memory, for instance, need to be moved into registers for processing and then results written back.

But which of these consumes greater energy? Where is the largest payback?

Figure 1 is generally accepted truth – the memory accesses associated with a program are made up of approximately 60% instruction fetches and 40% data accesses.

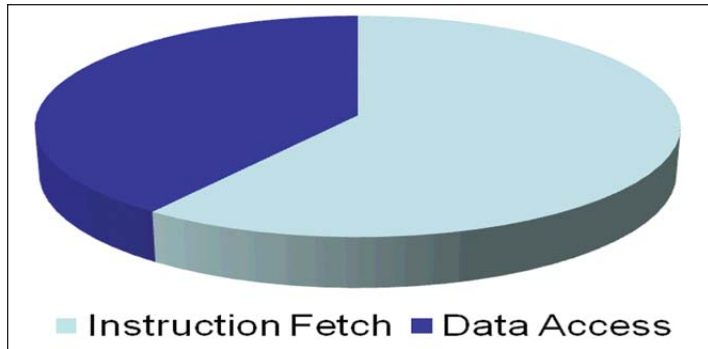


Figure 1: Memory access distribution

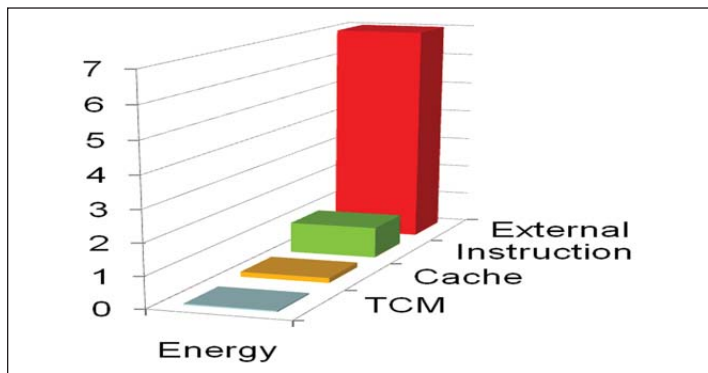


Figure 2: Energy cost of memory accesses

Figure 2 shows some research conducted at ARM. If the cost of using an instruction is 1, then the cost of a Tightly Coupled Memory (TCM) access is roughly 1/25, the cost of a cache access around 1/6. The cost of an external RAM access is 7 times the cost of an instruction execution.

In other words, for each external RAM access, we can execute 7 instructions, access cache 40 times or TCM around 170 times for the same energy cost.

Computation is cheap – Communication is expensive

So it appears that moving data around is much more expensive than processing it. The first elephant, therefore, is that of data efficiency.

We can propose two rules for managing the energy cost of memory accesses:

Keep it close - The relative cost, in energy terms, of accessing memory reduces the “closer” the memory is to the core.

Access it less - Reducing memory accesses is much more important than reducing instruction count.

Making good use of Scratchpad Memory

From our energy figures, it is clear that TCM is by far the most efficient type of memory we have in a system. Not all systems have what ARM calls TCM (connected to the core by a dedicated and optimized interface) but most do have at least some type of on-chip, fast, wide memory. For the purposes of this discussion we shall refer to the general case of Scratch Pad Memory (SPM).

Given that a single SPM access uses approximately 170th the energy of an external RAM access, making good use of memory of this type should be a priority.

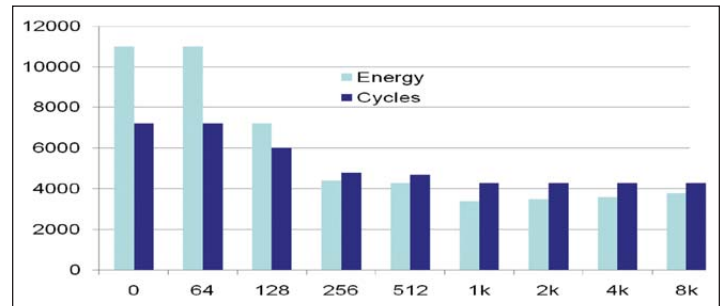


Figure 3: Energy benefit of SPM

The graph in Figure 3 shows that for the simple “multisort” benchmark, an SPM region of even 128 bytes can reduce power consumption by nearly 50%. The maximum reduction, with a memory of 1k bytes is 70%. The methodology used in this study (Marwedel, 2004) was to dynamically relocate code and data segments from external RAM to SPM dynamically. Even with the overhead of moving items around on demand, not only is the energy consumption reduced, but performance is also improved by some 60%.

It is clear that we are getting into diminishing returns at a certain point. In this case, while the performance improvement flattens out for SPM sizes greater than 1k, the overall system energy goes up slightly. Here we are effectively carrying the energy cost of SPM which this particular application cannot use – the application itself just isn’t big enough.

You can also note that this particular application, when coupled with the allocation algorithm used, cannot make use of an SPM region smaller than 64 bytes. None of the available segments are small enough to fit. A more sophisticated algorithm is shown in the study to increase the saving to over 80% in the best case.

Being cache-friendly

The analysis of the benefit of cache is somewhat more complicated than that for SPM. On the one hand, caches are essentially self-managing. On the other hand, they operate not on individual memory locations but on “lines” of a fixed size. So, an access to a single cacheable memory location may cause a burst of memory accesses to load an entire line. If this additional data is never accessed, the resulting energy expended is wasted.

Another downside is the cost (in terms of silicon area and power consumption) of the extra logic required for caches.

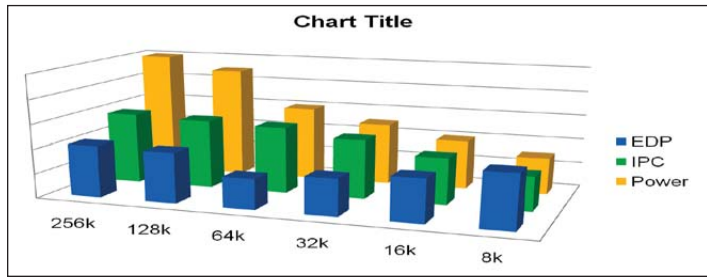


Figure 4: Energy benefits of cache

Figure 4, taken from a paper from Princeton (Brooks, 2000), shows three sets of figures for a simple application benchmark. For varying cache size, the bars show performance (Instructions Per Cycle), power consumption and EDP (Energy-Delay Product). In general, performance increases as cache size increases. However, the power consumption of the system also increases – due to the power consumption of larger cache elements. The Energy-Delay Product allows us to balance these two. In this case, there is a sweet spot at a cache size of 64k when the EDP hits a minimum.

Minimize data memory access It is a feature of the ARM architecture that constants are problematic - in particular, it is not possible to place an arbitrary 32-bit constant into a register in a single instruction. Coupled with the fact that all memory accesses must be made to addresses held in a register, this means that programs frequently need to place addresses and other constants in registers and that this is difficult. The standard method for doing this is to embed constants as literal data in the code segment and to load them using PC-relative loads at run time.

It is useful, therefore, to minimize the impact of this. Ensure that constants are known at compile-time and, where possible, can be encoded in single ARM instructions. Minimize the need to load base pointers in order to access global variables. This essentially involves ensuring that global variables are co-located in memory at run-time so that single pointers can be used to access multiple variables. The easiest way to achieve this is to place global variables in a structure.

Although stack accesses on ARM are relatively efficient (they make good use of load and store multiple instructions), there are many things the programmer can do to reduce stack access: minimize live variables; avoid taking the address of local variables; make use of tail-call optimizations where possible; reduce the number of parameters passed to functions to fewer than four; allow the compiler to inline functions aggressively.

The case for and against recursion is a little more complex. Often the compiler can tail-optimize a recursive function very nicely. The fact that all the data is stored on the stack can make for better locality than the alternatives. Perhaps the advice can be best phrased as “don’t use recursion unless either the alternatives are worse for data locality, or you are confident that the recursive call can be tail-optimized by the compiler.”

Exception handlers should be written to maximize the opportunities for tail-chaining in order to avoid unnecessary saving and restoring of context on the stack.

Now to turn our attention to the second elephant in the jungle – instruction execution.

Minimizing Instruction Count On the face of it, minimizing instruction execution is essentially the same as optimizing for performance - executing fewer instructions inevitably leads to lower energy consumption. Again, some obvious pointers.

Firstly, configure the tools correctly. The compiler and linker are unable to carry out even some basic optimizations unless they are fully aware of the target platform.

Write code sensibly to avoid unnecessary operations. On the ARM architecture 32-bit data types are efficient: in general 8-bit and 16-bit types, while they occupy less storage, are less efficient to process. The packing and unpacking instructions and the SIMD operations in v6 and v7 of the architecture go some way to helping with this but be aware that, in the main, these instructions are not accessible from C.

Write loops carefully Loops should be written following some simple rules: use unsigned integer counters, count down and test for equality with zero as a termination condition. This will produce shorter and faster loops which use fewer registers. Loops should also be written with vectorization in mind. Some simple rules about control structures and data declarations can make the job of the compiler much easier when trying to unroll and vectorise even the simplest of loops.

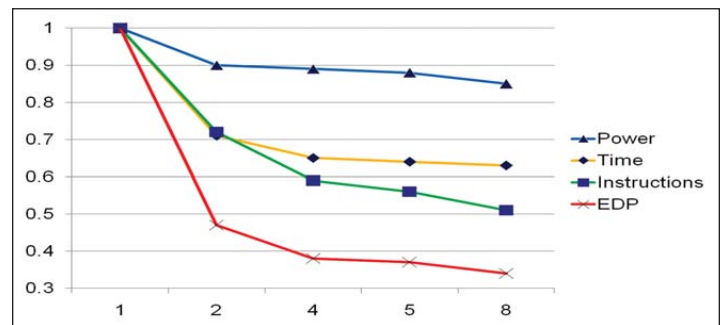


Figure 5: Loop unrolling

Figure 5 shows some figures relating to one specific loop optimization, loop unrolling (Brooks, 2000). As expected, execution time and instruction count decrease as the unroll factor increases. We are seeing the effect of reduced loop overhead and a reduction in address calculations. The power results are more interesting and not as obvious. Since there are fewer branches on which the predictor can train its behavior and the final mis-predict on the fall-through at the bottom of the loop now forms a greater proportion of the total, branch predictor accuracy decreases as the loop is unrolled further. However, the fetch stages are able to be more efficient as the continuous stream of sequential fetches is less frequently disrupted.

The combined effect is a net reduction in energy consumption *per instruction*.

So, although the execution time essentially levels off beyond an unroll factor of 4, since the power consumption continues to decrease, so does the all-important energy-delay product. An energy-aware compiler or developer might tend therefore to unroll somewhat more aggressively than one which is simply concerned with execution time.

Be only as accurate as you need to be The required accuracy of output should also be considered. Fixed point implementations are usually computationally much more efficient than floating point ones, even when floating point hardware is available. If you are rendering an image for onscreen viewing, full compliance with the standard may not be required – go only as far as you need to render an acceptable image.

One study (Shin, 2002), involving progressive optimization of a standard MPEG-4 decode function, has shown that switching from soft FP to fixed point binary can reduce energy consumption by 72%. The loss of accuracy means that the result is no longer compliant with the standard but it is still adequate for rendering purposes on the system under study.

What about Thumb? While the thumb instruction set is designed explicitly to improve code density it can also improve performance for narrow memory systems. However, while code density certainly improves, the instruction count increases at the same time. This is due to the reduced functionality of individual Thumb instructions as compared with ARM instructions. So, it seems reasonable that recompiling in Thumb will result in an increase in energy consumption and this is indeed what we see.

The study cited above shows that while code size decreases by 4%, the instruction execution count increases by 38% and the energy requirement by 28%.¹

To find our third elephant we need to move outside the realm of the processor and its memories and into the wider system. The systems with which we work these days have been cleverly put together by our hardware designing colleagues to offer a multitude of power saving options.

Power-saving in the wider system Obviously, components which are not in use should be put into as low power a state as possible. This should be an integral part of all sensibly designed systems and should include memory and cache systems and even the processor itself. In a multi-core system we should consider the possibility of powering down one or more cores when processing requirements are low.

Firstly, a small point but one well worth making: when dealing with peripherals, always try to use an interrupt mechanism rather than polling. Polling loops simply burn energy for no purpose. Almost all architectures include some kind of Wait-For-Interrupt instruction

which can be used to put the system into a standby state in this situation. On ARM systems, the core will typically be clock-gated, leaving only static power leakage.

Unnecessary sleep-wake cycles can often be avoided by designing the interrupt architecture to maximize tail-chaining. The ARM Cortex-M3 architecture does this automatically.

It is fairly easy to come up with a selection of shutdown schemes for individual computation units. Those which are required on a predictable basis can simply be shut down by the application or OS when not required. Those which are required unpredictably can be powered up on demand and powered down again automatically when idle for longer than a certain period. The timescale over which you power down a subsystem will be derived from two things: its power consumption when powered but idle and the cost of the sleep-wake cycle. The decision is fundamentally application-dependent. A simple cycle count of the power cycle code, however, would be the most obvious place to start.

Measurements indicate that the Neon engine adds about 10% to the running power of a core like the Cortex-A9. However, it provides 40% - 150% improvement in performance for typical signal processing algorithms. The benefit of enabling Neon for the duration of the task and then shutting it down when not required is clear. Very often, it is not just the Neon engine which can be shut down on task completion but the entire processor system giving increased savings.

One, often difficult, choice is that between enabling computation components to complete earlier (and therefore shut down for longer) and slowing the processor to reduce power consumption while still completing just-in-time. Figure 6 shows energy consumption figures *per iteration* for a simple benchmark (Domeika, 2009). The benchmark is run four times at each of two clock speeds with various combinations of instruction cache and floating point coprocessor.

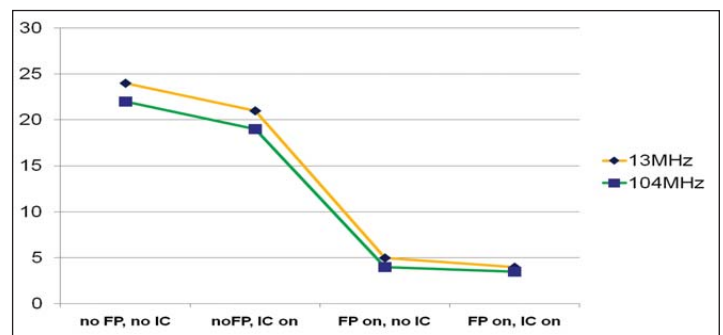


Figure 6: Power usage of system components

There are two clear points of interest. First, although both the instruction cache and floating point unit reduce the energy consumption, the floating point unit makes a much greater difference than the instruction cache.

¹ It is worth noting that Thumb-2, implemented in ARM's Cortex processors, does not suffer from this problem.

Second, energy consumption per iteration is essentially the same, in all configurations, *regardless of the clock speed*. Given this, running at full speed with all features enabled in order to complete the task more quickly will be significantly more efficient than turning the clock speed down.

Multi-processing It is well known that higher performance can be achieved at greater energy efficiency by employing multiple cores rather than cranking up the power of a single core. When using a multi-core system, we should consider the option of shutting down one or more cores when not required. Work at ARM has shown that the cost of cycling a core in an SMP Linux system would be of the order of 50000 cycles (most of this is consumed by cleaning the L1 caches). This implies that this operation would be undertaken on a timescale of hundreds of milliseconds rather than anything shorter, otherwise the cost clearly outweighs the benefit.

ARM is actively researching architectures containing two cores, one high-performance core for full operation and a companion core, much smaller, for low-power operation within a lower-performance envelope. When high processing power is required, the system runs on the larger core. As tasks are complete, the system can transfer all context to the smaller core and shut the larger one down. Switching back to the larger core involves reversing the context move. If the cores are connected as part of a coherent system, then the cost of making the switch can be minimized.

What about Operating Systems? When running on an operating system, unfortunately much of this flexibility is denied to the application programmer. Cache configuration, use of SPM, power cycling of components, etc. is very much at the discretion of the architect of the operating system architect and the device drivers. However, there is still much the application programmer should consider.

Studies have shown that poorly designed inter-process communication (IPC) can increase the energy cost of a system considerably. One simple technique involves “vectorizing” IPC - batching it up and sending a number of small messages as a single larger one, thus incurring the overhead of a context switch less often. Alongside this, reducing the number of processes clearly reduces the need for communication between them. Processes which need to communicate frequently could be merged into a single process.

A recent study (Tan, 2003), running under embedded Linux, shows a potential improvement in energy consumption of up to 60% resulting from analysing and rationalising IPC.²

Conclusions Although I have highlighted many areas in which there is ongoing academic work, there is much we can do now. The conclusions are relatively simple: minimize external memory access, minimize instruction execution and turn things off when you’re not using them.

By way of conclusion, I recall a conversation I had with a customer group on a training course in mid-2009. They were concerned with implementation of signal-processing algorithms on a Cortex-A8 platform incorporating Neon and wanted to know the exact energy consumption of individual instructions. I explained the fact that much of this information is simply not known and is, in any case, extremely hard to derive using current tools. In retrospect, we should have recognized that they were at the very insignificant end of a very long chain of elephants. The elephant they were hunting was, in reality, extremely small compared to others in the room. Better advice would have been to estimate, either from analysis or from program trace data, the number and type of data accesses involved in each implementation. From this, together with an instruction count, much more informed choices could have been made. The power consumption of individual instructions pales into insignificance when placed alongside the cost of badly placed memory accesses.

It falls to us, software developers, to keep up the pressure on the academics and the tools vendors to build these capabilities into the next generation of tools. This won’t be easy but it will come.

Finally, I must caution that it all depends. On your system, your platform, your application, your OS, your batteries and your users. To coin a phrase “Your Mileage May Vary.”




END

Works Cited

- Brooks. (2000). Wattch: A framework for Architectural-Level Power Analysis and Optimizations. ISCA. Vancouver: ACM.
- Domeika, M. (2009). Evaluating the performance of multi-core processors. Embedded.com .
- Loghi, M. (2004). Cycle-Accurate Power Analysis for Multiprocessor Systems-on-a-Chip. GLSVLSI. Boston: ACM.
- Marwedel. (2004). Fast, predictable and low energy memory references through architecture-aware compilation. Asia and South Pacific Design Automation Conference. IEEE.
- Mattei. (n.d.). Reducing Energy Consumption by Balancing Caches and Scratchpads on ARM Cores. University of Siena .
- Shin, D. (2002). Energy-Monitoring Tool for Low-Power Embedded Programs. IEEE .
- Tan, T. (2003). Software Architectural Transformations: A New Approach to Low Energy Embedded Software. DATE. IEEE.

Many of these subjects are covered in detail on ARM’s software development training courses. For more details of courses offered by ARM’s training team, please go to <http://www.arm.com/support/training.html>.

² In the example system, the number of processes was reduced from 4 to 2, the number of IPC routes from 5 to 1.

**Only Embedded Developer
lets you compare more
than  and  You
can compare  s
And devices and tools.
Then you can buy them.
(Wow).**

**www.embeddeddeveloper.com
One Stop. Shop.**

**And now, try the EmbeddedDeveloper.cn, Chinese site
and EmbeddedDeveloper.de, the new German site.**

EMBEDDED DEVELOPER 
FIND. COMPARE. BUY.