Efficient C Code for ARM Devices

ARM Technology Conference 2010, Santa Clara CA, Session ATC-152

Chris Shore, ARM, September 2010

Introduction

Our industry moves incredibly quickly. The hot topic last year is very rarely so important this year – either we will have solved it or some other issue will have risen to even greater prominence. The issue of efficiency, however, has been a relatively constant theme through my time in the industry, now some 24 years but efficiency has had many faces over that time.

In the early days, I can recall developing local area networking software on small 8-bit microcontrollers where RAM was the scarce resource. We worked long and hard to come up with cleverer and cleverer ways of storing, compressing, reusing and encoding our necessary data structures to make them fit.

Later on, I can recall a controller for a touch screen radio operator station in which the scarce commodity was code space. Limited to 32k and writing in C, the first version or two didn't present any problems. But by the third and fourth versions, with the customer requesting added functionality all the time, we were banging hard on the ceiling of the ROM. Again, we put in the nights and weekends working out better ways of reusing common code sequences, re-coding some functions in assembler by hand to make them smaller, repartitioning the C library to remove even more unused sections.

Then there were the systems which needed to respond so quickly. There is nothing like the demand of hitting the frame rate of a time-division multiplexed cellular phone system to focus the mind on interrupt latency and processing deadlines.

These days, it seems that we very often have RAM and ROM coming out of our ears and we have processing power to burn. So what is the pressing issue of today? So often, it is power-efficiency or, perhaps more exactly, energy-efficiency. A huge proportion of today's electronics is portable and battery-powered. We have got used to our cell phones running for a week between charges and our expectations of what they will do on that limited amount of energy goes up year on year.

Hardware engineers have been in the business of saving energy for longer than we software engineers but, increasingly, it is our job to utilize hardware features to the maximum and then realize even greater savings by writing efficient software.

A double-sided coin

One of the fundamental changes in our field in recent years has been the increasing use of platform operating systems in the embedded world. The majority of ARM-based systems these days, large and surprisingly small, will use either one of the many proprietary real-time operating systems available or will use a home-brewed scheduler of some kind. Except at the small end, the people who develop the applications are generally not the same people as those who develop the operating systems under which they run. Both groups of engineers, though, need increasingly to focus on being energy-efficient.

There is some common ground between the two – indeed, much of what can be said about writing efficient applications applies equally to operating system development – but each contains specialist areas and focuses of its own.

The two sides of efficiency

Application	Operating System
Minimal instruction count	System power management
Minimal memory access	Subsystem power
Cache-friendly data accesses	DVFS etc
Efficient stack usage	Multicore power regimes
Efficient procedure calls	Power-friendly spinlocks
Task/thread partitioning	Intelligent task scheduling
SIMD and Vectorization	Cache configuration and management

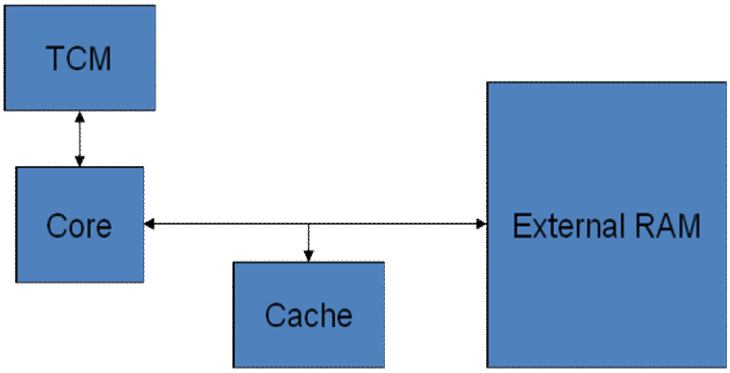
Some basic stuff

There are some things which are so basic I hope I don't have to say them. But, just in case, I will get them out of the way early on!

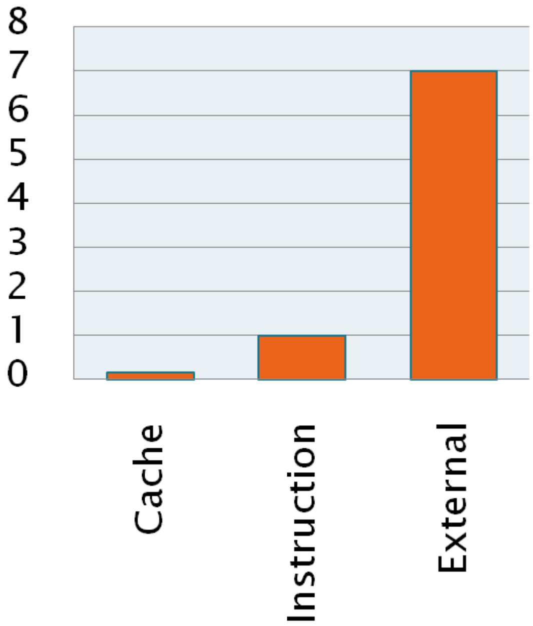
We have a very nice training course on optimizing embedded software on ARM platforms. In a series of optimization steps over 3 days, we take a standard piece of video codec software and progressively apply varying optimizations to it. Some are more effective than others but the cumulative effect is to improve performance by between 200% and 250%. We then note that all of this would be completely wasted work without correctly enabling and configuring the data cache as that simple omission reduces performance by approximately 5000%. Modern ARM cores are specifically designed and optimized to run from caches as much as possible. Neglecting this simple fact is catastrophic. We shall have more to say about cache-friendly programming later on but, at this point, it is simply sufficient to say "Turn it on!"

Memory use is key

The fact that cache is so important is a direct result of the fact that memory access in general is costly – both in terms of time and energy. Almost all systems have some kind of memory hierarchy. It might be as simple as some on-chip scratchpad RAM and some external off-chip RAM; it might be as complex as two or three levels of cache between the core and the external memory system.



A rough rule of thumb might be that if an L1 cache access takes one time period, an L2 cache access might take around 10 and an external memory access of the order of 100. There is an increasing energy cost with this as well. The increasing expense of memory accesses the further they are removed from the core is something which you need to bear in mind at all times when looking at writing efficient code. The graphic [1] shows the measured energy cost of memory accesses, benchmarked against the cost of executing an instruction. An external memory access is typically going to take 100 times longer than accessing cache and cost 50-60 times the energy.



So, message number two, following on from simply turning on the cache, is to make sure that you use it as much as possible.

Instructions count too

In terms of energy and time cost, instruction execution come after memory access. Put simply, the fewer instructions you execute, the less energy you consume and the shorter time you take to do it. We are assuming here that instructions are, in the main, executed from cache. For most applications, with sensible cache configuration and reasonable cache size, this will generally be the case.

So, message number three is “optimize for speed”. The efficiency comes in two ways: firstly, the fewer instructions you execute, the less energy that takes; secondly, the faster you accomplish your task, the sooner you can sleep the processor and save power. We will have more to say about dynamic and static power management later, when we get to operating systems but, for now it suffices to say that fast code is, in general, power efficient code.

Further, notice that executing instructions is cheap compared to accessing memory. This leads us to conclude that algorithms which favor computation (data processing) over communication (data movement) will tend to be more power-efficient. Another way of looking at this is to say the programs which are CPU-bound tend to be more efficient than those which are memory-bound.

There is one complication which we would do well to mention at this point. Code which is compiled for size (i.e. smallest code size) may, in some circumstances be more efficient than code which is compiled for speed. This will occur if, by virtue of being smaller, it uses the cache better. I think this may be regarded as an exception to the general rule discussed above but it may apply in some circumstances. If it does, due to the greatly increased efficiency of cache accesses, its effect could be huge.

Good coding practice

A good rule of thumb here is “Make things Match.” So, match data types to the underlying architecture, code to the available instruction set, memory use to the configuration of the platform, coding conventions to the available tools.

Similarly “Write sensible code” and make sure you know at least something about how the tools work. If you know even a little about how procedure calls and global data are implemented on the ARM architecture, there is a lot you can do to make your code more efficient.

To some extent, be driven by the way the tools work, rather than trying to drive the tools to work the way you do. Very often, the tools work they way they do for very valid architectural reasons!

Data type and size

ARM cores are 32-bit machines. That is to say that they have 32-bit registers, a 32-bit ALU, and 32-bit internal data paths. Unsurprisingly, they are good at doing things with 32-bit quantities. Some cores do implement wider buses than this between, say, the core and the cache but this does not change the fact that the core is fundamentally a 32-bit machine.

For instance, simple ALU operations on 16-bit data (signed or unsigned short) will involve extra instructions to either truncate or sign-extend the result in the 32-bit register.

```
unsigned int a, b;

a = a + b;

ADD r0, r0, r1
```

```
signed short a, b;

a = a + b;

ADD r2, r0, r1
LSL r2, r2, #16
ASR r0, r2, #16
```

The core can, sometimes, hide these extra operations. For instance, truncating a 32-bit value to 16-bits can be accomplished at the same time as storing it to memory if a halfword store instruction is used. This does indeed accomplish two operations for the price of one but does depend on storing the item being what you actually want to do with it.

Later versions of the architecture provide sub-word SIMD instructions which can get around some of these inefficiencies but in most cases these instructions are hard, if not impossible, to use from C.

Remember, too, that local variables, regardless of size, always take up an entire 32-bit register when held in the register bank and an entire 32-bit word in memory when spilled on to the stack.

There is very little advantage in using sub-word types on an ARM core (as compared with many other architectures in which it can save both time and storage). That’s not to say that it doesn’t work or isn’t sometimes the right thing to do – just that there is little or no efficiency advantage in doing so.

Data alignment

In keeping with this, the ARM core has historically imposed very strict alignment requirements on both code and data placement. It is still true today that ARM cores cannot execute unaligned instructions. However, more recent cores, from architecture ARMv6 onwards, have relaxed somewhat the requirements for data accesses. These cores support access to unaligned words and halfwords. It is tempting, therefore, to abandon all consideration of alignment in your code, enable unaligned support in the hardware and declare everything as unaligned. This would be a mistake! For instance, loading an unaligned word may indeed take one instruction where, on earlier cores, it would have taken three or four but there is still a performance penalty due to the need for the hardware to carry out multiple bus transactions. Those transactions are hidden from the program and from the programmer but, at hardware level, they are still there and they still take time.

Structures and arrays

If storage is not at too much of a premium, there are good reasons for choosing carefully the size of array elements. Firstly, making the length of each element a power of 2 simplifies the offset calculation when accessing an individual element. This is a result of the fact that the ARM instruction set allows shift operations to be combined with ALU operations and to be built into addressing modes.

For an array of elements of size 12, based at address r3, accessing the element at index r1 would take a sequence like this:

```
ADD r1, r1, r1, LSL #1         ; r1 = 3 * r1
LDR r0, [r3, r1, LSL #2]       ; r0 = *(r1 + 4 * r1)
```

By comparison, if the element size is 16, the address calculation is much simpler:

```
LDR r0, [r3, r1, LSL #4]       ; r0 = *(r3 + 16 * r1)
```

Although we will look at cache-friendly data access in more detail later, it is worth noting at this point that elements which fit neatly into cache lines make more efficient use of the automatic prefetching behavior of cache linefills.

Efficient parameter passing

Each subword-sized parameter to a function will be passed using either a single 32-bit register or a 32-bit word on the stack. This information is buried inside a specification called the Procedure Call Standard for the ARM Architecture (AAPCS). This, in turn, is part of a larger document called the ARM Application Binary Interface (ARM ABI). The latest version of both can be found on ARM's website and, while the ABI document is aimed largely at tools developers, the AAPCS information is directly relevant to applications programmers.

According to the AAPCS, we have four registers available for passing parameters. So, up to four parameters can be passed very efficiently simply by loading them into registers prior to the function call. Similarly, a single word-sized (or smaller) return value can be passed back in a register and a doubleword value in two registers.

It is plain to see that trying to pass more than four parameters involves placing the remainder on the stack with the attendant cost in memory accesses, extra instructions and, therefore, energy and time. The simple rule "keep parameters to four or fewer" is well worth keeping in mind when coding procedures.

Further, there are "alignment" restrictions on the use of registers when passing doublewords as parameters. In essence, a doubleword is passed in two register and those registers must be an even-odd pair i.e. r0 and r1, or r2 and r3. The following function call will pass 'a' in r0, 'b' in r2:r3 and 'c' on the stack. R1 is unable to be used due to the alignment restrictions which apply to passing the doubleword.

```
fx(int a, double b, int c)
```

Re-declaring the function as shown below is functionally identical but passes all the parameters in registers.

```
fx(int a, int c, double b)
```

The C compiler is not omniscient

Simply put, the C compiler cannot read your mind! Much as you try, it cannot divine your attentions simply by looking at the code you write. Further, it is restricted to examining the single compilation unit with which it is presented. In order to guarantee correct program execution, it is programmed for maximum paranoia in all circumstances. So it must make “worst-case” assumptions about everything. The most obvious and best known example of this is “pointer aliasing.” This means that the compiler must make the assumption that any write through any pointer can change the value of any item in memory whose address is known to the program. This frequently has severe repercussions for compiler optimization.

Other examples would be that the compiler must assume that all global data is volatile across external function boundaries, loop counters may take any value, loop tests may fail first time round. There are many other examples.

The good news is that it is, in most cases, very easy for the programmer to provide extra information which helps the compiler out. In other cases, you can rewrite your code better to express your intentions and better to convey the particular conditions which prevail. For instance, if you know that a particular loop will always execute at least once, a `do-while` loop is a much better choice than a `for(;;)` loop. The `for` loop in C always applies the termination test before the first iteration of the loop. The compiler is therefore forced to either place duplicate tests at top and bottom, place a branch at the top to a single test at the bottom or a branch at the bottom to a single test at the top which is executed first time round. All three approaches involve either extra code, extra branches or both. Yes, you can argue that modern sophisticated branch prediction hardware reduces the penalty of this but it is, in my view, still lazy code and you should be doing better!

The ARM compiler also provide several keywords which are available for giving the compiler “meta-information” about your code at various points.

`__pure` Indicates that a function has no side-effects and accesses no global data. In other words, its result depends only on its parameters and calling it twice with the same parameters will always return the same result. This makes the function a candidate for common-subexpression elimination.

`__restrict` When applied to a pointer declaration, indicates that no write through this pointer changes the value of an item referenced by any other pointer. This is of particular benefit in loop optimization where it increases the freedom of the compiler to apply transformations such as rotation, unrolling and inversion etc.

`__promise` Indicates that, at a particular point in a program, a particular expression holds true.

Consider the following example.

```
void f(int *x, int n)
{
      int i;
      __promise((n > 0) && ((n&7)==0));
      for (i = 0; i < n; i++)
      {
            x[i]++;
      }
}
```

The `__promise` intrinsic here informs the compiler that the loop counter, at this particular point, will be greater than zero and divisible by eight. This allows the compiler to treat the `for` loop as a `do-while` loop, omitting the test at the top, and also to unroll the loop by any factor up to eight without having to worry about boundary conditions.

This kind of optimization is particularly important when combined with the vectorization capability of the NEON engine.

The C compiler is not omnipotent

As well as not knowing everything, the compiler cannot do everything either!

There are many instructions, particularly in more recent versions of the architecture, which cannot be generated by the C compiler at all. This is usually because they have no equivalent in the semantics of the C language. Using SIMD instructions to operate on complex numbers stored as packed signed halfwords is a good example. The C compiler has no complex type, certainly not one which fits this storage method and so it cannot use these very efficient instructions for carrying out very straightforward operations.

The proficient programmer could write hand-coded assembly functions to implement these operations. However, it is often much easier to use the rich set of intrinsic functions provided by the ARM C compiler.

The following example shows a short function implementing a complex multiplication using the `SMUSD` and `SMUADX` instructions provided in architecture ARMv6.

```
// complex variables a and b stored as packed
// halfwords [Re]:[Im]
//
unsigned int cmultiply(unsigned int a,unsigned int b)
{
      return((__smusd(a, b) << 16)+ __smuadx(a, b));
}
```

This produces the following output.

```
SMUSD       r2, r0, r1
SMUADX      r0, r0, r1
ADD         r0, r0, r2, LSL #16
BX          lr
```

If the compiler is able to inline the function, then there is no function call overhead either. The obvious advantages of this approach over coding in assembly are increased portability and readability.

NEON support in the compiler

The C compiler can be also used to access the functionality of the NEON Media Processing Engine via a comprehensive set of intrinsic functions and built-in data types.

Here is a straightforward implementation of an array multiplication function. The C code is shown on the left and the resulting assembly code on the right. Only the body of the loop is shown and this executes a total of eight times.

```
for(i = 0; i < 8; i++)
{
   dst[i] = src[i] * dst[i];
}
```

```
MOV      r2,#0
LDR      r3,[r1,r2,LSL #2]
LDR      r4,[r0,r2,LSL #2]
MUL      r3,r3,r4
STR      r3,[r0,r2,LSL #2]
ADD      r2,r2,#1
CMP      r2,#8
BLT      {pc}-0x18 ; 0x38
```

The next pair of sequences show the same loop implemented using NEON instrinsics. The key points to note are that the loop has been unrolled by a factor of four to reflect the fact that the NEON load, store and multiplication instructions are dealing with four 32-bit words each time. This greatly lowers the instruction count. Due to the reduced number of iterations, the loop overhead is also reduced.

```
for(i = 0; i < 8; i+=4)
{
   n_dst = vld1q_s32(dst+i);

   n_src = vld1q_s32(src+i);

   n_dst = vmulq_s32(n_dst,n_src);

   vst1q_s32(dst+i,n_dst);
}
```

```
MOV      r2,#0
ADD      r4,r1,r2,LSL #2
ADD      r3,r0,r2,LSL #2
ADD      r2,r2,#4
VLD1.32  {d0,d1},[r4]
CMP      r2,#8
VLD1.32  {d2,d3},[r3]
VMUL.I32 q0,q1,q0
VST1.32  {d0,d1},[r3]
BLT      {pc}-0x20 ; 0x8
```

If you look a little more closely, you can see that the compiler output does not exactly correspond to the C source code. The order of the instructions has been changed. The compiler has done this to minimize interlocks and thus maximize throughput. This is another advantage of using intrinsics over hand coding in assembly - the compiler is able to optimize the machine instructions in the context of the surrounding code and in response to the target architecture.

Intelligent data cache use

Most application programmers, often quite rightly, view the caches as something which is the province of the operating system. Certainly, configuration and management of the cache is something which the operating system takes care of and applications are not generally permitted to meddle with them.

That is not to say, however, that the application programmer should completely ignore the fact that there is a cache in the system and that it is capable of providing huge performance benefits. Those benefits can be improved if the application code is written with the cache operation and architecture in mind.

Aligning data structures to cache line boundaries in memory can make better use of the pre-loading effect of cache line fills, making it more likely that wanted data is loaded as part of the line and unwanted data is not. The compiler provides a data attribute for doing this:

```
int myarray[16] __attribute__((aligned(64)));
```

Some very common-place algorithms can also benefit from being written in a cache-friendly way. It is well known that caches perform much better when data is accessed sequentially – the data automatically loaded as part of a line fill is automatically reused. Operations like matrix multiplication pose an interesting problem in this regard. The following example is a simplistic implementation of a matrix multiplication function.

You can see that the ‘a’ array is accessed sequentially, because the rightmost index varies most quickly – this is cache-friendly. In contrast, the ‘c’ array is accessed by column – the leftmost index varies most quickly. This is decidedly not cache-friendly. Each access will result in the loading of a complete cache line of data, only one element of which is likely to be used before being evicted. Since cache accesses often happen in the background, there may not be a speed penalty associated with this but there will definitely be an energy cost due to unnecessary external memory accesses.

```
for (i = 0; i < SIZE; i++)
{
  for (j = 0; j < SIZE; j++)
  {
    for (k = 0; k < SIZE; k++)
    {
      a[i][j] += b[i][k] * c[k][j];
    }
  }
}
```

Various techniques are well known for avoiding this behavior – strip mining and tiling being the most common.

Certain access patterns are also pathologically bad in terms of cache reuse and are best avoided completely. For instance, in the case of a write-allocate cache, writing a large amount of data which is not to be immediately re-used will result in large areas of the cache being populated with useless data. Some recent ARM cache controllers take note of this and, in the case of the Cortex-A9, will temporarily disable write-allocation if they detect certain unfavorable access patterns. This behavior is automatic and transparent to the programmer but awareness of the cache allocation strategy can be a useful guide when writing code.

Global data access

It is a feature of the ARM architecture that you cannot specify a full 32-bit address in a memory access instruction. Accessing a variable held in memory requires its address to be placed in a register. Or, at least, a base address from which it can be accessed via a simple offset. This gives rise to problems with accessing external global variables in that the compiler must store at compile time and load at run time a base pointer for each and every such variable. If a function accesses a number of external globals, the compiler must assume that they are in separate compilation units and therefore not guaranteed to be co-located at run-time. A separate base pointer is required for each one.

If you are able to allow the compiler to infer that a group of globals are co-located in the memory address map then they can be accessed as different offsets from a single base pointer. The easiest way of achieving this is simply to declare globals in the module in which they are used. However, applications which do not need items with global scope are few and far between so a more pragmatic solution is required.

The most common solution is to place globals, or more usually groups of related globals, into structures. These structures are then guaranteed to be placed as a single unit at link-time and can be accessed using a single base pointer.

Tools features like multi-file compilation can go some way to getting round this problem but they lack flexibility in other ways.

System power management

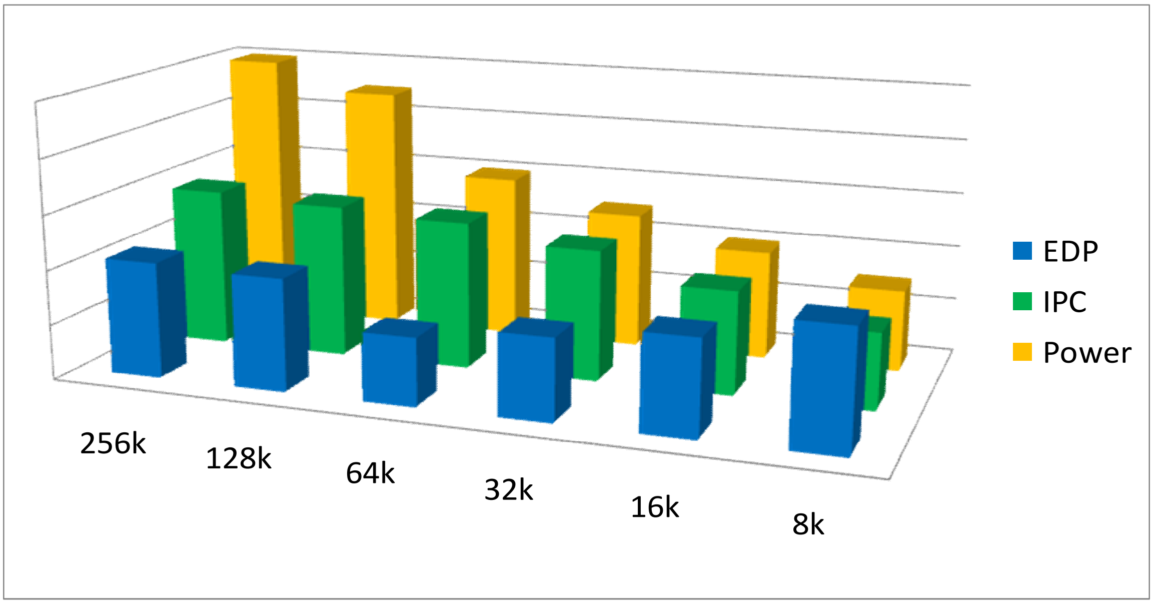
We turn now to wider system issues which are generally regarded as the domain of the operating system. In most systems, it is the operating system which has control of things like clock frequency, operating voltage, power state of individual cores etc. The application programmer is not generally aware of these.

There is an ongoing debate over the most fundamental issue: is it better to complete the necessary computing work at maximum speed and then sleep for the longest possible time, or is it better to wind down voltage and/or clock frequency to reduce power consumption while completing the computation just in time.

The major factor these days in this debate is the growing contribution of static power consumption in a system. Historically, static power consumption (primarily leakage) has been much smaller than dynamic power consumption. As chip geometries become ever smaller, it is a fact that leakage increases, this making static consumption more of a contributor to overall energy consumption. These days, this leads to the conclusion that it may be better to "race to completion" and then power down completely (to eliminate leakage), rather than to keep executing for longer, albeit at a lower frequency.

A suitable metric

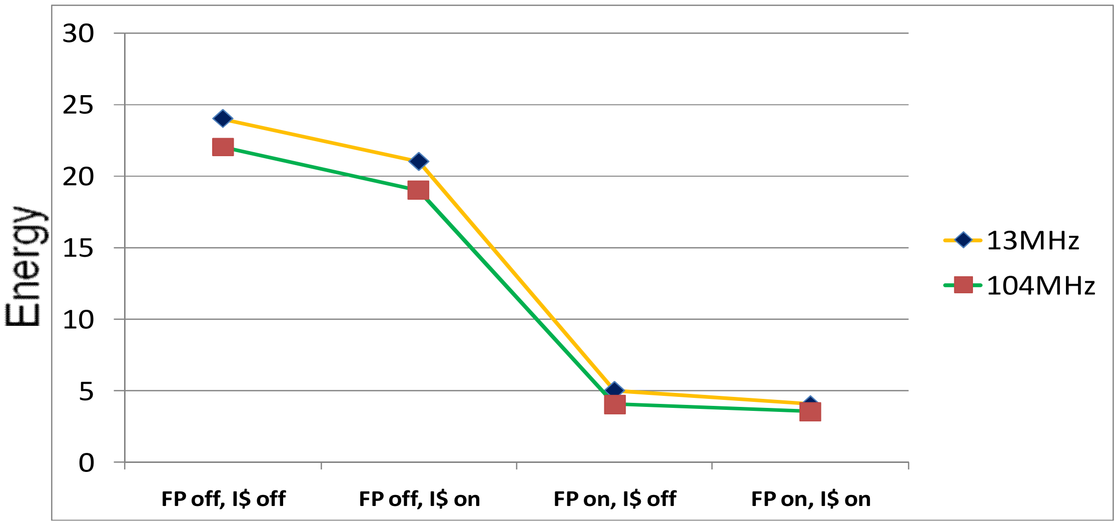
What is needed is a metric which somehow combines energy consumption with an idea of how long a particular computation takes. Such a metric is often referred to as "Energy Delay Product", or EDP. Although such metrics have been widely used in circuit design for many years, there is as yet no generally accepted methodology for deriving or using such a metric in the field of software development. The example above [2]



shows one instance where the EDP metric assists in making a decision as to the optimum size of cache for a specific processor undertaking a specific task. It is clear that a larger cache increases performance but that this is at the expense of higher power consumption. However, the EDP metric shows that there is a "sweet spot" with a cache size of 64k at which the energy saving due to higher performance is maximized. At greater cache sizes, performance still increases but energy usage per unit of computation also increases, negating the benefit.

Should we turn it on?

This example [3] shows the impact on energy consumption of Instruction Cache and a Floating Point Unit. The energy consumption is shown for all four possible configurations and for two different clock speeds. You can see that both units decrease overall energy consumption when carrying out the same amount of computation. However, the benefit is essentially independent of clock speed. In this particular system, therefore, it is clearly better to run to completion at maximum clock speed and then power down as early as possible.



Re-visiting the cost of memory access

We have already established that external memory access is one of the most energy-costly operations carried out by any system. The following is a list of techniques which can be used to minimize the cost of this.

- Use registers as much as possible
  Minimize live variables, limit parameter count, don't take the address of automatic variables, inline where possible to reduce stack usage.

- Make best use of cache
  Write-Allocate/Write-Back is generally the best configuration for stack, heap and global data; Write-Back/Read-Allocate is often better for buffers (as this kind of data is often written by one agent and then read by another). Multi-level cache systems are best configured with L1 as Write-Through/Read - Allocate and L2 as Write-Back/Write-Allocate. Beware of "false sharing" in systems with cache coherency management hardware.

Managing sub-systems

In a single-core system, we must ensure that additional computation engines (such as NEON) and external peripherals (serial ports and the like) are only powered when required. This is a system decision for the operating system developer and individual chips will provide features for managing this. The operating system will almost always need customizing to target specific devices. For instance, Freescale's i.MX51 range of chips incorporate a NEON hardware monitor which can be programmed automatically to turn off the NEON engine if it is unused for a configurable period of time. Power it on again is handled in the Undefined Instruction exception which results in trying to use it when it is not powered.

In multi-core systems, we have the option of turning individual cores off and on in order to match the system capability with current workload. The cost of power-cycling an individual core is often significant, so the decision as to what circumstances trigger this will always be system-dependent. Current implementations of ARM SMP Linux support:

- CPU hotplug
- Load-balancing and dynamic re-prioritization
- Intelligent and cache-friendly scheduling
- Dynamic Voltage and Frequency Scaling (DVFS) per CPU
- Individual CPU power state management

The kernel incorporates an interface to a generic external power management controller and this interface will need targeting for individual platforms to make use of available features.

By way of example, TI's OMAP4 platforms incorporate a range of voltage and frequency scaling options. They define a number of "Operating Performance Points" from which the operating system can select the most appropriate option at any particular time. The power consumption of the device can be varied from 600µW to 600mW depending on load.

The programmer's job

In multi-core systems it is easy to focus on the features provided by the hardware and think that this is the end of the story – that would be delinquent on the part of any serious programmer! It is still vital to bear in mind the following properties when writing software and configuring operating systems.

- System efficiency
  Intelligent and dynamic task prioritization; load-balancing; power-conscious spinlocks.
- Computation efficiency
  Parallelism at data, task and function level; minimized synchronization overhead.
- Data efficiency
  Efficient use of memory system features; careful use of coherent caches to avoid thrashing and false sharing.

Conclusions

The elephants in the room here are:

- Configure tools and platform properly
- Minimize external memory access by writing code carefully and configuring caches correctly
- Minimize instruction count by optimizing for speed and making effective use of computation engines

- Manage subsystems effectively

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.

Works Cited

[1] Reducing Energy Consumption by Balancing Caches and Scratchpads on ARM Cores, Mario Mattei, University of Siena

[2] Wattch: A framework for Architectural-Level Power Analysis and Optimizations, Brooks et al, Princeton University

[3] Evaluating the performance of multi-core processors – Part 1, Max Domeika, Embedded.com