EMBEDDED SYSTEMS AND MULTI-CORE - AN INTRODUCTION

Pete Wilson • Kiva Design • pete@kivadesigngroupe.com.

Introduction

Embedded systems have always spanned a very broad spectrum in many dimensions - in cost, size, power, what they did, how they did it and what technologies were used. And so any trumpeting of major changes in how embedded systems will be made has to be looked at with the right degree of scepticism - far too often one will find someone trying to make a buck by selling some new panacea in architecture, tools, methodology, silicon or IP.

But there really is emerging a new dynamic - many more embedded systems than before are going to be based on multiple programmable engines than previously, and in many cases, the multiple engines will be living on the same chip. Such systems are increasingly referred to as 'multicore systems', and making good use of such platforms will require changes in the way most folk have traditionally approached system design. While the major impact is likely to be on software architecture (given that we've already accepted the hardware architec-true change as given), this will, over time, have knock-on effects on tools and languages.

This is the first in a series of essays discussing what this will all mean for embedded systems developers and the community that supports them. This first essay will touch on some major issues associated with "going multi-core"; later essays will dig a little deeper into the areas identified, bring up new areas, and respond to feedback.

Why Multicore?

It's worth revisiting the question of just why multicore has become a hot subject recently. Firstly, it's important to realise that physics hasn't changed - multicore has *always* been a Good Idea for many application domains. But our industry is driven much more by perception than physics, and so it has chosen, for several decades, to obtain higher performance by building significantly more expensive processors and support infrastructure so that uniprocessor software could be re-used with little or no redesign. And why was the software designed for uniprocessors? Because everybody knows that multiprocessor, multiprocess, multithread software is hard to design, write, get correct and deploy. And how do we know this? Because no-one does it; so there's no knowledge or support for it; so it's hard.

This is, of course, a little unfair. But the industry perceptions are driven by the dynamics of the larger mainstream computing industry; for many real-world reasons, embedded re-uses the investments and approaches deployed in that industry, where they have long embraced a computational model which says that computers run programs; that programs don't talk to each other very much; and that programs are sequential programs - just one thread of action[1]. And as long as the processors that were provided provided increased performance

[1] Although there has of late been an increase in the use of multithreading in the writing of 'ordinary' programs, it's still looked upon as arcane.

through increased performance on sequential programs, this was a convenient universe for software (although it added rather high, unpleasant costs to the semiconductor manufacturers, who took on the burden of not only developing the next generation of silicon, but also finding ways to get significantly higher performance than their prior generation of processor).

Now it turns out that customers of semiconductor companies are lazy. They know that they want higher performance; and they know that performance is not a single number, but depends on what the application does and what's important to the application. But there's a disappointingly large percentage of them who don't know what their software does, nor how it does it, let alone what bits need speeding up and what the opportunities and constraints are on such speed-ups. So they are easily seduced by simplistic stories from their vendors. Not long ago it was the Dhrystone performance of a processor which was the official descriptor of performance; Intel simplified even this, and made simple clock frequency a first-class discriminator of performance.

Until one day Intel discovered that driving clock rate higher was not only as painful as it ever had been, but that it simply didn't work well enough. And, worse, that the amount of power needed to get acceptable performance from a very high clock-rate processor was beyond unacceptable.

Since then, they have changed course somewhat, and are now pushing processor solutions which deploy simpler, more-efficient cores in configurations which provide several cores; initially, these are two-core chips, but over time - as the silicon geometries shrink - the count will likely increase to 4, 8, 16 or more.

For computers to show performance improvements with these platforms, the software is going to have to mutate into a form which can leverage multiple cores - it is going to have to change from being basically sequential to become basically concurrent.

For some computer applications, this is not new news. Any server application is already highly concurrent - the requests from the clients can generally be handled entirely independently, and servers have long been multiprocessor-based. The situation is a little different for workstations, PCs and laptops - in general, there's just one user at any one time, and so each program that is performance sensitive needs restructuring to be able to leverage the multiple cores.[2]

Again, some performance-critical PC programs are already architected to make use of multiple cores - the image-processing 'filters' in Adobe Photoshop are a prime example. These are efficiently parallelised, since the problem they're dealing with is itself gratuitously parallel.

But performance-sensitive problems which are currently implemented as sequential programs will need restructuring to make use of multiple cores; this means redesigning, re-architecting and rewriting fairly large amounts of software over the coming years.

---

[2] This isn't entirely true - modern operating systems are generally running multiple tasks concurrently; some of the tasks are separate user programs, but most of the tasks are stuff to keep the machinery running right, watching the network, and so forth. These OS's can go further in exposing concurrency - they can decouple the computation needed by the program from that needed for the pretty user interface; and the application from the file system; and so forth (and run the extra processes on multiple cores); but this OS-provided partitioning doesn't seem likely to be able to fill up more than some small number of processors.

This change in the mainstream world will produce expectations, tools and knowledge which will be available to the embedded space for use in our own systems.

And in the embedded space, it's not only higher maximum performance which is a driver for multicore - it's also lower cost and lower power and software re-usability.

Multicore systems can in principle be lower cost because a collection of small cores can use less silicon area than a single core with the same overall capability. The simpler processors have a significantly higher percentage of their gates (or transistors) dedicated to actual program execution - instruction decoding, registers, function units and so forth than the higher performance cores, which dedicate a rather large amount of silicon to making it possible for the 'useful' gates to do their work. In very large, complex designs, there can be an order of magnitude more silicon (and more) provided for this sort of 'support'; unless the result is a machine which executes programs an order of magnitude faster, a collection of small processors will provide more useful MIPS per square centimetre than the large, complex core. This also results in lower power for the replicated simple processor approach, along with better power scaling (if the machinery doesn't have much work to do, it's simpler to turn the clock rate down on one processor and turn the others off entirely than to work out which bits of a complex processor can usefully be powered up and down in cunning manners[3]).

And multicore software holds the promise of a greater *dynamic range* than uniprocessor, sequential code - at least for applications in which there is discoverable concurrency and where that concurrency can be efficiently expressed. The greater dynamic range arises from the possibility of vastly increasing the high-performance bar for such software through the use of many processors; appropriate software can speed up almost linearly over quite a range, allowing performance an order of magnitude or more higher than with a single processor. This flexibility makes the software more scalable, and simplifies re-use.

But the potential benefits of multicore don't come for free, and the benefits are not equally applicable to all applications.

The Basics

For software to run efficiently or usefully on a multicore system it must be organised in such a way that there is useful work for most of the cores to perform most of the time. So a first requirement for multicore-capable software is that it already be organised into concurrent chunks, which somehow work together to get the system's needs accomplished.

The next major point is that if the software is a collection of tasks running concurrently, there are opportunities for errors completely unavailable to an entirely sequential program. The key opportunity is the need to properly housekeep access to any resource used by more than one concurrent task; getting this wrong can lead to either deadlock (the system locks up, with several tasks each waiting for another to finish what it's doing) or data corruption (when concurrent tasks read and write shared data without taking appropriate precautions).

---

[3] Of course, the standard tricks of voltage and frequency scaling used in the complex machines can also be employed in the simple machines.

Further nightmares include finding that the unimportant things get done but the important things don't (because the system happened not to schedule the tasks in a way conducive to getting the desired results). Or over-provisioning the system (with, for example, way too much processor capability) in an attempt to avoid the scheduling problems but with the result that the thing now costs too much. Or uses too much power.

Probably, most embedded systems of any complexity are already constructed as a collection of tasks and shared resources managed by an RTOS (or a 'real' OS). One might be forgiven for thinking that this means that all such systems are all ready for multicore prime time. But this would be a mistake.

Current systems divide the work up into as few tasks as practical, both for reasons of design complexity and for reasons of execution efficiency.

As noted earlier, most folk don't naturally think of concurrent algorithms, or concurrency-based problem solutions - they think stuff out in a sequential manner, a style which fits well with available mainstream programming languages. So the natural style and the programming tools drive current systems to be as sequential as possible, with concurrency being rather coarse-grained. Taking a coarse-grained solution of this nature and dropping it on a platform with (say) eight times as many cores will probably result in a system in which most of the cores are quiescent most of the time - there just isn't enough concurrency in the software as written.

The concurrency in such existing systems is often driven by the major characteristics of the problem - an automotive engine controller could be constructed as a collection of tasks in which one strategy task looked at the machinery every (say) ten milliseconds, and decided on the general goals of the engine, and another n tasks ran one per engine cylinder, mapping the goals into specific actions - inject the right amount of fuel at this time, fire the spark plug at that time, and so forth. And the processor would be chosen so that it could do all the computation involved in these $n + 1$ tasks with time to spare. However, dropping the collection of tasks onto a 4-processor platform instead might well result in discovering that although the per-cylinder tasks distribute nicely, the slower cores in the 4-core platform are unable to execute the global strategy task in time. And it's one sequential process, and so there's no automatic way of getting it turned into a collection of concurrent tasks.

So a first problem which will be encountered in moving to multicore will be the need to rewrite existing sequential chunks of code so that they can be distributed over multiple cores in such a way that overall performance is improved compared with executing that code on a single processor.

A next problem is that the unicore code almost certainly was written believing that there was just one chunk of memory, or, more exactly, that every piece of code could see all of the memory. In a multicore system, this may not be true. For it to be true, then the cores all have to share memory. If they have caches, this means that the system must implement an appropriate cache coherence policy. Providing cache coherence does indeed make the old way of programming a bit simpler on multicore systems, but it brings its own costs - greater design and implementation complexity and difficulty (and therefore harder to verify at the level of silicon design); providing a failure mechanism which more or less guarantees that if a core or a piece of software go rogue, the whole system will become tainted and untrustworthy (and likely inoperable); and a marked lack of scalability (cache-coherent SMPs simply don't scale - to more processors - very well, unless they're being used for gratuitously-concurrent problems). The obvious alternative to a shared-memory architecture is a message-passing one.

So a second problem likely to be encountered is that not only does one have the pleasure of re-architecting one's software so it will run on multiple cores, but the further pleasure of throwing away some fundamental assumptions on what each chunk of the software can actually see - changing from shared-data to message-passing can change the shape of software quite a lot[4].

A third problem is that whatever RTOS one was using will need rejigging, and (quite likely) the libraries will need rework.

A fourth is that it's quite possible that your optimal platform will be a heterogeneous platform - multicore, yes, but not all the cores are the same. they may differ in performance, and/or they may differ in capabilities (a DSP, a communications-oriented core, a general purpose core..). And this will lead to further fun and games trying to compile, build and debug, especially if software functionality is moved around between the cores in an attempt to get a good balance[5].

And this leads to a further dimension of change - while it's always been important to *size* one's system (to understand the computational and I/O burden imposed, and to understand what platform resources will be needed to meet the burden) it's much easier to do this for a single processor than for a collection of them; and the collection sizing problem is simpler if the cores are homogeneous. So it's also likely that the coming multicore era will encourage rather better design analyses, probably involving somewhat more simulation, than in the past; similarly, there will likely be more.

Other Areas

Other areas are affected by the move to multicore, in ways which may not be overly apparent immediately. What sort of protection scheme should be used? Systems with caches can already display undesirable non-determinism - what will the effect of going multicore be? It's fine to claim that in general, for appropriate applications, multicore is "better"; but in what ways, for what applications, and how can one tell?

Looking Ahead

The next few essays will look at the topics mentioned above - simulation (what it is; how simulators work; simulating software running on multicore platforms); how RTOS's work (and how they should work), and the changes needed for multicore; shared memory and message-passing approaches; the effects of processor architecture on how well everything works; non-determinism and what to do about it; protection schemes; power and more.

---

[4] Of course, there's no absolute requirement to perform this transformation at any particular moment - the effects of the shared memory/cache coherence issues vary with application. But over time, it's increasingly likely that it will be necessary to do it

[5] Of course, there are heterogeneous multicore chips available today; and as a result, tools vendors are stepping up to provide appropriate tools. But it's most unusual to think about writing the code as a collection of concurrent tasks, and then seeing which cores should run which code. Rather, the partitioning is done upfront - functionality is rarely moved between the classes of cores.