

Using Virtual System Prototyping Technology to Optimize Real-time Systems for Power

Introduction

The designers of most electronic systems are concerned with minimizing the amount of power that the system consumes. This is particularly true in the case of battery-powered portable (often wireless) consumer electronics systems such as personal digital assistants (PDAs) and cell phones.

These portable products are becoming physically smaller with each new generation, yet consumers have grown to expect more and better functionality (which requires increased processing capability and performance) and to demand longer battery life. In addition to actually making telephone calls, for example, a modern cell phone may include features such as the ability to act as a personal organizer; play games; take, transmit, and receive still pictures and/or short videos; browse the internet; and so forth.

Modern consumers are becoming increasingly aware of product aspects such as battery life, and such features strongly effect their purchasing decisions. Despite its increased functionality, for example, users expect a contemporary cell phone's battery to last several hours when in continuous use and five or more days while in standby mode. Products that consume less power have a very significant advantage in today's extremely competitive markets. Each generation of product planning must satisfy substantial increases in functionality and performance plus substantial reductions in power consumption.

In the past, the focus of next-generation product planning has been concentrated largely on the micro-architecture of the underlying microprocessing units. However, the improvement of the processor micro-architecture typically yields only second- or third-order effects with regard to improving performance. By comparison, the overall hardware (platform) architecture and the architecture and algorithmic content of the software that runs on it both have first-order effects at the system level.

Creating optimal low-power designs requires making sophisticated tradeoffs in the hardware architecture, the software architecture, and the underlying software algorithms. The creation of successful power-sensitive designs requires system architects and engineers (both hardware and software) to have the ability to accurately and efficiently perform and quantify such tradeoffs. In order to achieve this, the architects and engineers require the ability to access and analyze power data early in the design process.

This paper first introduces the concept of virtual prototypes (VPs) and virtual system prototypes (VSPs). Next, the paper discusses how VSPs can be used to model, analyze, and optimize real-time systems in the context of power. Finally, the paper presents results from some real-world experiments showing how small changes in the hardware architecture and software algorithms can significantly affect the power consumption of a system.

The VP and VSP Concepts

Characteristics such as performance and power for a complex system such as a cell phone – including its software – cannot be represented and computed as a formal mathematical problem. The only realistic solution for determining such characteristics is some form of simulation.

One option for this simulation is hardware acceleration and/or emulation. Unfortunately, in addition to providing only limited visibility into the inner working of the system, the highest level of abstraction supported by these solutions are register transfer level (RTL) representations. As a result, development and evaluation cannot commence until a long way into the design cycle when the hardware portion of the design is largely completed. In turn, this limits the design team's ability with regard to exploring, evaluating and optimizing the hardware architecture. In addition, FPGA implementations of processors typically are slow,

executing software at around 1 MIP – about 50 times slower than a virtual processor model of the same processor.

The related concepts of virtual prototypes (VPs) and virtual system prototypes (VSPs) provide a solution. A VP is a functionality-accurate and timing-accurate software model of the hardware portions of an electronic system. Such a model will typically include processor cores, memory subsystems, peripherals, buses, bridges, mechanical and RF devices, and so on. By comparison, a VSP is a model of the entire system: that is, the combination of the VP and the software that will run on it.

Fully evaluating the characteristics of a complex system may require performing many hundreds of experiments on various system configurations. Furthermore, it is not unusual for a single simulation to require that 100 billion instructions be run to reproduce a problem or to compute a representative result. This represents less than one hour of simulation time using a high-performance, timing-accurate VSP. By comparison, the same software simulation would take between 100 to 500 hours or more using a typical timing-accurate structural instruction set simulator (ISS) model and 100,000 hours or more using an RTL model.

A key advantage of using a VSP is that the hardware and software portions of the system can be developed and evaluated concurrently. A VSP allows different hardware architectures to be quickly and easily tested and analyzed under real software workloads. While hard real-time software code is being developed, its execution yields trace data (from probes inserted into the models) from which performance (timing, reaction times, latency times, etc.) and power data alongside normal debug data.

Using VSPs to Model, Analyze, and Optimize Real-Time Systems for Power

Consider the underlying hardware architecture associated with a realistic cell phone system, as illustrated in Figure 1. This system includes two ARM926E processors, a StarCore SC1200 processor, hierarchical bus and memory subsystems, and a variety of peripherals.

There are many techniques for constructing objective functions for system attributes such as power. The classical technique is to track event frequencies and/or latencies and to construct the power function based on events that contribute significantly to the computation of power. A VSP enables power analysis to be made in the context of alternative hardware and software architectures running real software workloads.

The first step with the VSP is to assign “weights” for each class of function that contributes to the system’s overall power consumption. As a simple example, we could start by assigning a default weight of 1.0 to a generic register file access, and to then base other weights as multiples of this default weight. Consider some of the weights that might be associated with the ARM926E CPU and its cache and memory accesses as shown in Table 1.

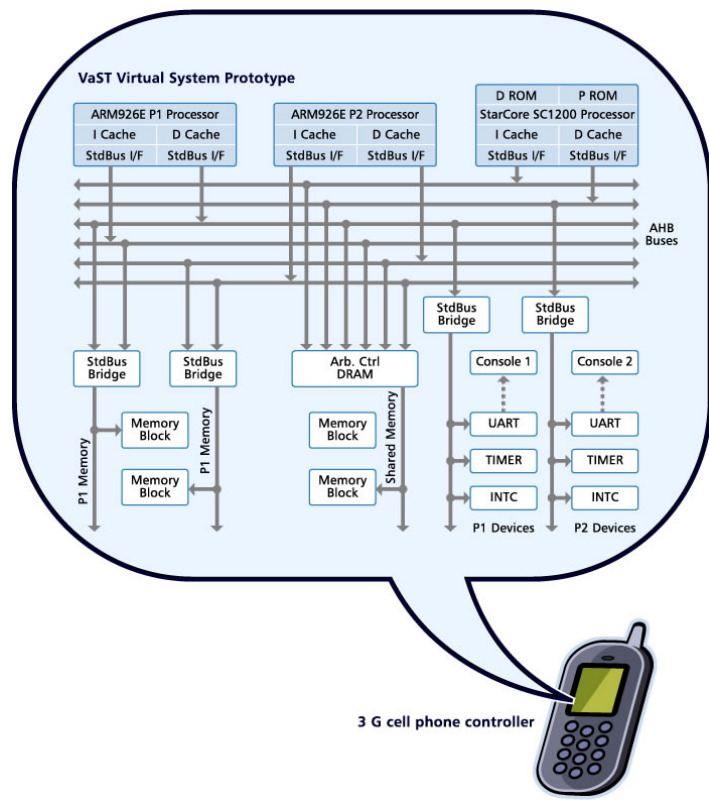


Figure 1. Cell phone architecture

Function Type	Events	Weight Functions
Generic Register Access	RegAcc	1.0
Pipeline	Pipe	6.0
Instruction Types		
Jump	iJmp	2.0
Arithmetic	iArith	1.0
Co-processor	iCopro	12.0
etc.	:	:
Cache		
Instruction hit	iCache_hit	5.0
Instruction miss	iCache_miss	5.0
Data hit	dCache_hit	5.0
Data miss	dCache_miss	5.0
etc.	:	:
Memory (inc. bus transactios)	Membus	50.0
etc.	:	:

Table 1. ARM926E function types, events, and weighting functions for power

As seen here, the weights (W) for each of these function classes have been set to constant multiples of the generic register access function. However, these weights may be represented by more complex functions; for example, the cache hit/miss weights could each be a function of the cache structure (size, wayness, policies, etc.).

The next step is to build an interpretation table that defines the component bindings, as shown in Table 2. Although these tables are large, the event bindings themselves are simple to implement, since each is typically a pointer to a function and a history buffer of events.

Component Types Binding	Component Instance Binding	Component Event Binding
f_{CPU}	$f_{ARM926E}$	$f_{ARM926E.RegAcc}$ $f_{ARM926E.Pipe}$:
	f_{SC1200}	$f_{SC1200.TCMwrite}$ $f_{ESC1200.Branch(Taken)}$:
:	:	:

Table 2. Component event binding table

In the case of power calculations, the basic function to be computed is that of instant power, which calculates the total energy consumed over some period of time or some number of events (such as clock cycles). Based on this instant power, the two main derived functions that are of interest for optimization purposes are:

The maximum power consumed over a particular period of time (this will be the maximum of the instant powers).

- The average power consumed over the course of an entire experiment.

If we assume that only one of the ARM926E processors is enabled (to match the experiments presented in the next section), then a simplified accumulating function used to compute the instant power per k -cycles is as follows:

$$f_{\text{Power}} = (W_{\text{Inst}} \times f_{\text{Inst}}) + (W_{\text{Pipe}} \times f_{\text{Pipe}}) + (W_{\text{Cache}} \times f_{\text{Cache}}) + \dots$$

Where:

$$f_{\text{Inst}} = (W_{\text{iJump}} \times N_{\text{iJump}}) + (W_{\text{iArith}} \times N_{\text{iArith}}) + (W_{\text{iCoprocc}} \times N_{\text{iCoprocc}}) + \dots$$

Where N_x means “the sum of the instructions of type ‘x’ over the course of ‘k’ cycles and W_{iJump} , W_{iArith} , W_{iCoprocc} , etc. are the weights associated with the function types and events from Table 1. Thus:

$$f_{\text{Inst}} = (2.0 \times N_{\text{iJump}}) + (1.0 \times N_{\text{iArith}}) + (12.0 \times N_{\text{iCoprocc}}) + \dots$$

Similar sub-functions occur for f_{Pipe} , f_{Cache} , and so forth. Note that the weights for each of the classes of functions contributing to the main accumulating function f_{Power} (that is, W_{Inst} , W_{Pipe} , W_{Cache} , etc.) may be assumed to be a constant value of 1.0 for the purposes of this paper. In more sophisticated studies, however, these weights might be replaced with more complex functions relevant to computing power in ways not considered for the purposes of the simple examples presented in this paper. For example, such functions might include history-dependent and implementation-dependent attributes.

Experimental Results

For the purposes of these simple example experiments, we created a VSP corresponding to Figure 1, but with only one of the ARM processors enabled and its instruction and data busses bridged to a shared memory. (We put the second ARM processor and the StarCore SC1200 processor in reset mode so that they consumed no cycles and no power).

We then constructed four suites of experiments (58 experiments in all) to investigate the effects of various arrangements of cache, busses, memory hierarchy, and algorithms with regard to power consumption and performance (speed).

With regard to the software workloads used to exercise our experiments, we employed Viterbi and Sieve programs from the Embedded Microprocessor Benchmark Consortium (EEMBC) test suite (www.eembc.org), a prime number program downloaded from the web, and a Linux boot of MontaVista Linux v2.6.

Viterbi: The results from seven Viterbi-based experiments used for calibration were expected; an uncached implementation was poor in regard to speed and power. However, when the cache was enabled, even a minimal cache of 1,024 bytes proved sufficient for this algorithm. Furthermore, due to the fact that there was a better than 99.5% hit rate on the data and instruction caches, the cache line size was shown to be immaterial, as was the bus width and memory type (either DDR or SDR).

Linux Boot: The results from nine structural variants of the experimental VSP were computed while booting Linux. These variants were generated as a subset of all possible variants based on different mixtures of cache size (1K, 8K, 32K), cache line size (16B, 32B), memory configured as DDR (first word delayed five cycles, subsequent words available per half cycle) and SDR (first word delayed five cycles and subsequent words available per cycle), and a bus data width of four bytes.

As is expected in an environment where the working set size of the target code greatly exceeds the cache size, the impact of the memory hierarchy on power and speed is considerable. With regard to booting Linux in our VSP, setting the ARM296E cache size to 32 Kbytes, the cache line size to 32 bytes, and using DDR memory yielded the minimum power consumption and maximum performance. However,

reducing the cache size to 16 Kbytes adversely impacted both power and performance by only around 1%, whereas this reduction would proportionally reduce the silicon cost by around 30%. Similarly, further reducing the cache size to 8 Kbytes negatively impacted power and speed by 5-10% while yielding a further 25% reduction in silicon cost.

Alternate Memory Hierarchies: This portion of our experiments was used to investigate the best tradeoff between power consumption, performance, and silicon cost for a controller executing a limited amount of code: a prime number generator using the *Sieve of Eratosthenes* algorithm. One of the things we were very interested in with regards to this series of experiments was determining the near-minimum cache size that would still yield within 5% of optimum performance and power.

In these experiments, we considered instruction and data cache characteristics of size (0B, 64B, 128B, 256B, 1 KB, 4 KB, 8 KB), cache line size (16B, 32B), wayness (1, 2, 4), cache power rating (3, 4, 5 – we varied the relative power consumption depending on size), and memory type (DDR, SRD). The results were as expected except that the transition between a cache size of 64B and 128B was very sharp, and at 128B we essentially achieved full speed. By comparison, in the case of power, the uncached power consumption was 20-35% less than the power consumed by 64B caches, and 200% higher than that consumed by 128B caches.

From this we determined that the effect of installing a small cache in a processor to achieve a four-fold increase in performance has a detrimental effect on power consumption due to the infrastructure required to support the cache. The cost of the cache (and its infrastructure) is also high in terms of silicon real estate. These considerations led to an investigation of alternative memory hierarchies that might achieve a better tradeoff between speed, power, and cost for a controller running a limited amount of code in an embedded application.

Thus, we decided to mimic the relative power consumed by a dedicated external 128B buffer (essentially a small, physically-addressed, direct-mapped, on-chip cache external to the processor). The results were that we could achieve a further 40% saving in power while maintaining optimum speed. This is significantly better than the earlier results because it minimizes power consumption and cost while maximizing performance.

Algorithmic Optimization: Last but certainly not least, our final ten experiments considered the effect of using different versions of an algorithm to optimize the combination of hardware and software for a particular (embedded) application. Since we already had good empirical data for our initial Sieve-based prime number generation algorithm, we acquired another version known as Kazmierczak's algorithm (www.mkaz.com/matg/primes.html).

The Kazmierczak algorithm required a small external 512B buffer to achieve its maximum speed, which was 40% faster than the Sieve-based algorithm while consuming only 15% more power. This provides an alternate solution that is optimized for speed where the tolerance to power and cost are more elastic.

Summary

Optimizing the hardware and software components of systems with complex objective functions is non-intuitive. Sophisticated tradeoffs between the hardware architecture and the software and algorithmic loads that are run on the hardware cannot be achieved by intuition or by formal mathematical analysis alone.

In the case of the Sieve and Kazmierczak algorithmic experiments discussed above, for example, it is inconceivable that the optimal hardware-software architectures determined by these experiments could have been determined simply by looking at – or mathematically analyzing – these two algorithms.

The solution is to use a high-performance, functionality-accurate and timing-accurate VSP that allows objective functions such as power consumption to be determined in the context of alternative hardware and software architectures running real software workloads.

Graham Hellestrand is founder and chief technology and strategy officer for VaST Systems Technology. He is also emeritus professor of computer science at the University of New South Wales, Australia. Hellestrand holds B.Sc. and Ph.D. degrees in computer science and engineering, and an MBA. He is a Fellow of IEEE and Institution of Engineers (Australia). You can reach him at g.hellestrand@vastsystems.com.