Embedded Software Optimization and Power Aware Software Development

Steven P. Smith

Spring 2012

Agenda

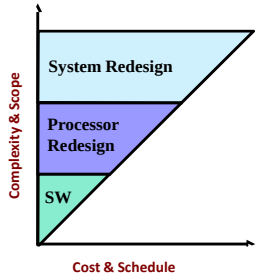
- Scope of our treatment of software optimization
- The software optimization process
- Some generally useful embedded software optimization tricks
- Power aware software development introduction
- Software controllable knobs for power management
- Calculating power consumption and battery lifetimes
- Conclusions

Scope of our Treatment of Optimization

- Optimization is a very broad term. For our purposes, assume:
  - An embedded system is at capacity in some sense.
  - Additional functionality must be added.
- Need typically arises near project completion during system integration and testing.
  - Schedule pressures are at their peak.

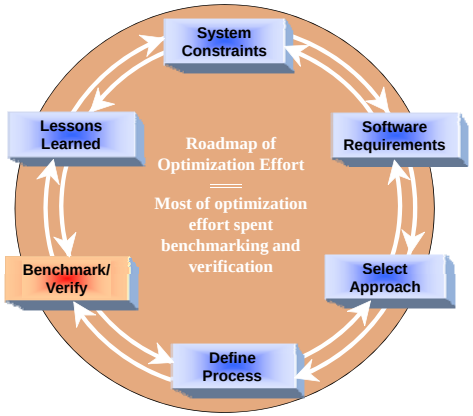
Optimization

- Allow addition of incremental functionality into system already at capacity.



- Subsystem modified, but external interfaces remain unchanged
- Processor centric upgrades both computer hardware and software
- Software centric using existing processor
  - Execute faster
  - Use less memory
  - Reduce input/output blockages
  - Allow integration and testing to continue

Software Centric Optimization



Step 1: Define the Specific Process to be Followed

- Determine what tools are available to assist in optimization.
  - Emulators
  - Instruction set simulators (ISS)
  - Compilers and assemblers
  - Linkers
  - Code profiler
  - Debugger
  - System and sub-system tests
- Identify the specific tools to be used.
- Define the objective in specific and measurable terms. E.g.,
  - Capture 1 KB of RAM for code execution.
  - Add a specific function or set of functions whose resource requirements are known.
  - Force all program constants to fit into 4 KB of non-volatile memory.

Step 2: Establish a Baseline

- Quantify current software in precise terms relevant to the optimization objective.
  - Code-only footprint size in bytes
  - RAM-based dynamic data storage requirements in bytes
  - Constant or non-volatile data storage requirements in bytes
  - Slack-time in the current real-time schedule
  - Processor utilization
- Use linker to discover code footprint and constant storage size.
- Instrument malloc() to track maximum dynamic memory use.
  - May be data-dependent, so a number of trials may be needed.
  - May be worthwhile to revisit the original decision to have dynamically
  - allocated storage.
- Identify and size sources of variability in baseline

Step 3: Modify Software

- Edit software source files to move closer to the optimization objectives.
- Ensure that all "knobs" in the tool chain are tuned properly.
- May also involve modifying the software operating environment
  - Change memory model.
    - Many 8-bit and 16-bit microprocessors support multiple memory models.
  - Reorganize code to maximize reference locality.
- Examine assembly code generated by compiler!
  - Compilers for embedded architectures vary widely in quality.
- Simple changes to HLL source code can have dramatic effects.
- Consider rewriting portions in assembly language.

Potentially Helpful Software Modifications

- More efficient conditionals
- Combine loops
- Unroll short loops
- Convert array indexing to pointer offset-based addressing
- In-line short functions
- Collapse functions called only once
- Reduce parameter passing

Step 4: Verify Functionality of Modified System

- Regression test suite
- Test bed
- Function hit counts
- Compare telemetry data sets

Step 5: Measure Enhancements

- Use the same methodology as employed for baseline.
- Assess whether objectives have been met.
- Review.
  - If objectives have not been met, assess options.
    - Additional software modifications
    - Try a "different set of eyes" on the code
    - "Plan B"

Software Optimization Examples

- Replace compound conditionals with arithmetic expressions, using bit position coding instead of incrementing integers.
  - OK:
```
for  (i = 0 ; i <=31 ; i++) {
        if ((i ==0) || (i==5) || (i==11) || (i==14) || (i==20))  { …
```
  - Better:
```
for (i = 1 ; i != 0 ; i = i << 1) {
        if ((i  & (1<<0|1<<5|1<<11|1<<14|1<<20)) != 0) { …
```
- Minimize variable sizes.
  - OK:
```
unsigned long i ;
for (i = 0 ; i < 100 ; i++) { …
```
  - Better:
```
unsigned char i ;
for (i = 0 ; i < 100 ; i++) { …
```

Software Optimization Examples

- Make frequently used literal constants into global constants
  - OK:
```
#define ERROR_STRING "Input error – try again"
...
display_message(ERROR_STRING) ;
```
  - Better:
```
const char * Error_string = "Input error – try again" ;
...
display_message(Error_string ) ;
```
- Replace divides with multiplies.
  - OK: `if ((f / g) < h) { …`
  - Better: `if (f < (h * g)) { …`

Software Optimization Examples

- Use reciprocals to replace divides.
  - OK: `if ((1/y) < 0.5) { …`
  - Better: `if (y < 2.0) { …`
- Use multiplication instead of square root calculations.
  - OK: `if (y < sqrt(z)) { …`
  - Better: `if (y * y) < z) { …`
- For better *typical* execution times, sort compound conditionals in order.
  - For Boolean AND conditionals, place terms with the highest probability of evaluating to a logic 0 first.
  - For Boolean OR conditionals, place terms with the highest probability of evaluating to a logic 1 first.
  - For terms that have roughly equal probabilities, put simpler terms first.

Software Optimization Examples

- For array accesses (especially multi-dimensional arrays), use pointer expressions instead of array reference syntax.
  - OK:
```
for (j = 0 ; j < ASIZE ; j++) {
        for (k = 0 ; k < BSIZE ; k++) {
                a[j][k] = 0 ;          …
```
  - Better:
```
p = &a[0][0] ;
for (j = 0 ; j < ASIZE * BSIZE ; j++) {
        *p++ = 0.0 ;          …
```
  - For some compilers, the difference in the size and execution time of the code generated can be dramatic.

Software Optimization Examples

- For initialization of structures, declare a constant instance and use memcpy() to initialize entries instead of using individual assignments. Use memset() to initialize entire struct to 0.
  - OK:
```
if ((p = (struct mystruct *) malloc(sizeof(struct mystruct)) !=
                        NULL) {
        p->field1 = 1 ;
        p->field2 = 2 ;      …
```
  - Better:
```
if ((p = (struct mystruct *) malloc(sizeof(struct mystruct)) !=
                        NULL) {
        memcpy(p, &const_mystruct, sizeof(struct mystruct)) ;
```

Software Optimization Summary

- Software optimization is the lowest cost and fastest means of reducing resource requirements and execution times to make room for additional functionality.
- Begin with a precisely quantified baseline.
- Set specific target objectives.
- Use the compiler's assembly language output option to examine code generated (e.g., gcc –S option).
- Get to know your compiler and all its knobs.
- Consider using assembly code where significant gains appear to be achievable.

Power Aware Software Development

Steven P. Smith

Spring 2010

Opportunities to Reduce Power Consumption

Algorithms	Minimize Operating Time
HLL Source Code	Optimized Code
Compiler	Energy Miser
Operating System	Scheduling
Instruct Set Architecture	Energy Exposed
Microarchitecture	Clocked Gating
Circuit Design	Low Voltage Swings
Manufacturing	Low-k dielectric

Power Aware Software

- Software structured to minimize system power consumption
- Most often associated with battery-powered devices 
- Becoming increasingly important in all classes of systems, including line-powered 
- As with hardware, power is becoming another key optimization criteria.
- In many embedded systems, power is the most critical factor.
- There is no Moore's Law for batteries.
  - Progress in battery efficiency has been comparatively slow.

Hardware Techniques for Reducing Energy Consumption

- Modern embedded systems make a variety of hardware-based energy consumption reduction controls accessible to software.
  - Voltage scaling
  - Frequency scaling
  - Power domains and sleep modes
  - Clock domains and gated clocking
- A great many other techniques exist, of course, but are beyond the scope of this discussion.

Other Important Aspects of Power Aware Systems

- Beyond the topic of local power aware software, there exist other issues that also play a key role in certain classes of embedded systems.
  - For wireless devices such as sensor networks, the routing algorithm used by the MANet is a key factor in overall energy consumption.
    - Shortest path routing may lead to over-stressed nodes.
    - It is wise to ensure that battery-powered operating times be generally equivalent throughout the network.
    - Synchronizing ON intervals to facilitate routing is a challenge.
  - For distributed embedded systems, load balancing in general can play a significant role in overall energy consumption.

How can Software Reduce Power Consumption?

- Minimize execution time through selection of a fast algorithm.
- Minimize bit widths for all data.
- Minimize off-chip accesses.
- Optimize cache usage through data and code structuring.
- For devices with current-driven switching (TTL), encode data to maximize the occurrence of lower-power logic values (logic 1).
- Take advantage of hardware power management modes accessible to software.
- Put idle hardware elements in sleep mode where possible.
- Use low-power instructions where possible.
- Power aware task scheduling

Power Consumption and Software

- Power consumed by a microprocessor
  $P = V_{dd} * I$
- Energy required for program execution
  $E = P * T_p$ T_p = Program execution time
- Program execution time
  $T_p = N * T_{clk}$ N = Cycles required for program execution

$$E = V_{dd} * I * N * T_{clk}$$

- If V_{dd} and T_{clk} are fixed, N is the only variable under software control.

Calculating Energy and Power in CMOS circuits

Recall the basic current equation for a capacitor:

$I = C * dv/dt$

$P = Vdd * I$

$\therefore P = C * dv/dt * V$

Also recall that

$dt = 1/frequency$ and generally $dv = V$

For example if Freq=10MHz, Capacitance = 5pf, Vdd = 1.0Volts

$I = 5e\text{-}12f * 10e+6 = 5e\text{-}6$ amps (50 µA)

and

$P = 50\ \mu A * 1.0V\ 50e\text{-}6$ watts (50µW)

and

$E = 5\ \mu W/MHz$

Therefore every clock cycle 50 µA of current is used

Calculating Energy and Power in a Battery

Battery capacity is measured in mA-Hour. Typically it is the amount of current (milli-Amp) that can be sustained for 20 Hours.

Example: an AA battery (1.5V) with 2890 mA-Hour capacity can provide 144.5 mA for 20 Hours (2890 ma-Hour ÷ 20 Hours)

Energy in a battery is measured in Joules:

3600 Joule = 1 Watt-Hour (W-H)

An AA Battery can theoretically provide approximately 15600 Joules of energy, e.g.,

2890 mA-Hour * 1.5 V = 4.333 Watt-Hour * 3600 Joules/W-H

= 15606 Joules

Example

- Assume a system with two basic power states: fully operational and asleep.
- Assume that the power consumed per unit time in both states is:
  - 40mW when fully operational
  - 200µW when in sleep mode
- Assume that 20% of the time the system is active and 80% of the time it is in sleep mode. How long would a AA battery last?
  - We know that Vdd = 1.5V (how?)

$\therefore$ 40mW ÷ 1.5 = 26.66 mA of operational current * 20% = 5.33mA

200µW ÷ 1.5 = 0.133 mA of sleep current * 80% = .1 mA

Total = 5.43mA

Total time = 144.5mA ÷ 5.43mA * 20hours = 532 hours (22.1 days)

Question: What is the optimal current draw from a battery such that it provides the stated mA-Hour capacity

Models for Power Aware Software

- Instruction-level power modeling.
  - Information needed:
    - Power (or current) for each specific instruction type
    - The number of cycles required for each specific instruction
    - Overhead of executing the next instruction (stall probability)
  - Power consumed by instruction type is not widely available from vendors, but can be measured empirically.
    - Block of identical instructions configured to execute from cache.
    - Other inputs such as data values should be randomized across a number of trials.
  - An instruction set simulator (ISS) can be instrumented to predict power.
  - Some, but not all, modeling tools account for potentially important but complex effects, such as cache behavior, pipeline stalls, data values, branch prediction accuracy, etc. (e.g., POET from Politecnico di Milan).

Low-level Code Optimization for Reducing Power

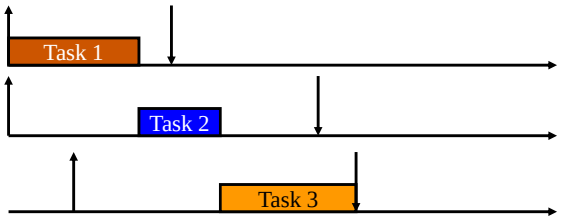
- If power consumption per instruction information is available, then prefer lower-power instructions where possible.
- Unroll small loops to reduce overhead.
- Use the smallest possible data types.
- Pack data.
- Organize data and instructions to optimize cache performance.
  - Avoid going to external memory wherever possible.
  - It may be practical to tune certain embedded applications to achieve near ideal cache performance, and the effort may be justified for high volume gadgets.
    - I.e., approaching Belady's optimal cache algorithm, where cache replacements are selected based on perfect knowledge of which entry will be unneeded for the longest period in the future.

System-level Power Aware Programming Techniques

- Leverage all available controls that influence power consumption while still meeting system performance criteria.
- Reduce voltage or clock frequencies whenever workload is light.
- Put power domains that are not currently in use in sleep mode.
  - Many devices used in embedded systems contain a number of separately controllable power domains.
    - Cypress PSoC separate analog power domains from digital.
    - Atmel ATMega processors have separately controllable on-chip peripherals.
- Put entire device in sleep mode when idle.
  - But how will you awaken?
    - Countdown timer
    - External event triggers an interrupt

Real-Time Scheduling in Power Aware Programming

- Goal is to modulate voltage, clock frequency, and/or sleep periods to minimize system power consumption while still meeting real-time performance criteria.



What is the optimal supply voltage, clock frequency, etc., for each task in order to obtain minimum energy consumption?

Power Aware Programming Summary

- Power aware programming is a key issue in most embedded systems development.
- Low-level power aware coding techniques are helpful, but are typically heavily outweighed by the influence of system-level software controllable power management.
  - Cache optimization may offer the best return per unit engineering time among the low-level techniques.
- Exploiting system-level software-based power management techniques in real-time systems presents some considerable challenges.
- Similarly, power aware programming for networked devices requires careful synchronization and consideration of routing.