Embedded Software Optimization Techniques

Hardware and Software Design for Cryptographic Applications

February 12, 2013

Software Performance

Let $T = \{t_1, t_2, ..., t_n\}$ be the set of "tasks" in a program.

$$\text{Total time} = \sum_{t_i \in T} time(t_i)$$

$$time(t_i) = \frac{\text{work of } t_i}{\text{rate of work of } t_i}$$

Obvious Questions

- How do we measure the amount and rate of work?
- How can we change our software to decrease the amount of work or increase the rate of work?
  - Either change will yield a lower execution time

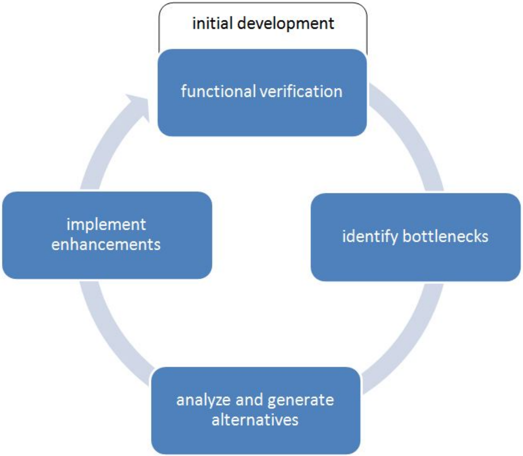
Levels of Optimization

- Code architecture and design
  - Application and algorithm design
- High-level source code changes
- Compiler settings
- Assembly tweaks

Iterative Process

1. Measure performance
   - Dynamic program analysis using a software profiler
2. Identify hotspots
   - Portions of the code that consume the most CPU cycles and/or computation time
3. Identify cause of hotspots
   - I/O overhead, inefficient algorithm, poor design?
4. Change the program
   - Source code tweaks or design changes?

Optimization Cycle



“We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil."

- Donald Knuth

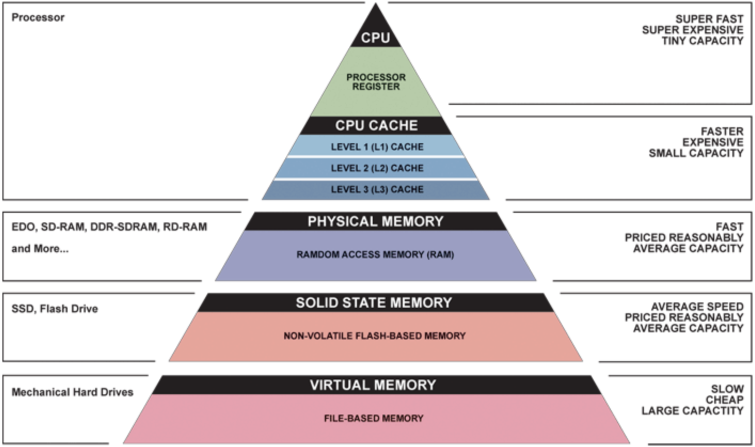
Code Architecture

- Design changes tend to have the biggest impact on code performance
- Analysis of the code architecture is the best starting point
  - Mathematical (asymptotic) analysis
  - Technological constraints
  - Investigate candidates for parallelism
  - Change the scope of analysis (e.g. module or global scope)

Technological Considerations

- Data access
  - Keep data in physical locations that can be accessed faster or with higher throughput
- Arithmetic operation
  - Know the performance of mathematical operations and functions
  - Think at the bit-level (e.g. shifting versus division, masking for modular arithmetic)
- Control flow
  - Software control flow structures (e.g. indirect function calls, switch statements, branches) perform differently
  - Be conscious of processor pipeline predictions
- Memory usage
  - Strive for data reuse (especially with embedded devices)
- External hardware peripherals
  - Consider the communication interface

Memory Hierarchy



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

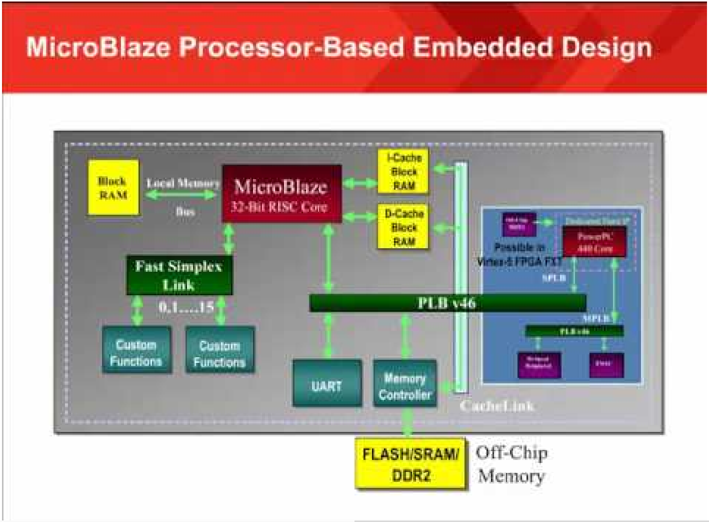
MicroBlaze Specification

- Soft-core processor designed specifically for Xilinx FPGAs
- Implemented using general-purpose memory and logic fabric of the FPGA
- Versatile interconnect system to support communication between components connected to the PLB, its primary I/O bus
- User-configured memory aspects (e.g. cache size, pipeline depth, embedded peripherals, MMU, etc)
- Supports operating systems that require hardware support (e.g. page tables and address space protection in Linux)

Processor Local Bus

- 128-bit, 64-bit, and 32-bit data transfer support for masters and slaves
  - We are working with the 32-bit wide version
- Selectable shared bus or point-to-point interconnect topology
- Fully synchronous to one clock
- A PLB-to-PLB bridge is required when two PLB segments are connected
  - Different bus speeds and widths

MicroBlaze


MicroBlaze Processor-Based Embedded Design
Block RAM
Local Memory Bus
MicroBlaze 32-Bit RISC Core
I-Cache Block RAM
D-Cache Block RAM
Fast Simplex Link
0,1....15
Custom Functions
Custom Functions
PLB v46
UART
Memory Controller
CacheLink
Possible in Virtex-5 FPGA FXT
PowerPC 440 Core
SPLB
MPLB
FLASH/SRAM/ DDR2
Off-Chip Memory

Parallelism

- Is it an option on the target platform?
- Can the work of an algorithm be broken down into a set of independent tasks or operations?
  - SIMD instructions
  - Hardware acceleration
- Can other hardware components perform computations in parallel with the processor?

Analysis Scope and Dimensions

- Look at the software from both a source code and design perspective
- Analyze the flow of data in your algorithm
- High-level API usage
  - What kind of performance hits are caused by not subverting these API calls?
  - `xil_printf()` vs `printf()`
- Code size

Optimization Misconceptions

- Improved hardware makes software optimization unnecessary
- Using (look-up) tables always beats recalculating
- Using modern C compilers makes it impossible to manually optimize code for performance
- Globals can be accessed faster than local variables
- Using smaller data types is more efficient than larger ones

Flavors of Optimization

Different flavors of optimization (in traditional low-level languages):

- Computation-oriented
- Memory-oriented
- I/O-oriented

Computation-Oriented

- Pick a better algorithm for the average case
  - Mergesort versus Bubblesort
- Loop manipulation
  - Jamming, unrolling, and inversion
- Code for the common case
- Table look-ups
- Function calls
- Stack usage
- Get a better compiler!

Memory-Oriented

- Be conscious of the locality of reference
- Don't copy large blocks of data often (use pointers!)
- Seek divide and conquer approach to large data structures (if possible)
- Manage memory leaks