Exploring CoreMark™ – A Benchmark Maximizing Simplicity and Efficacy

By Shay Gal-On and Markus Levy

There have been many attempts to provide a single number that can totally quantify the ability of a CPU. Be it MHz, MOPS, MFLOPS - all are simple to derive but misleading when looking at actual performance potential. Dhrystone was the first attempt to tie a performance indicator, namely DMIPS, to execution of real code - a good attempt, which has long served the industry, but is no longer meaningful. BogoMIPS attempts to measure how fast a CPU can “do nothing”, for what that is worth.

The need still exists for a simple and standardized benchmark that provides meaningful information about the CPU core - introducing CoreMark, available for free download from www.coremark.org. CoreMark ties a performance indicator to execution of simple code, but rather than being entirely arbitrary and synthetic, the code for the benchmark uses basic data structures and algorithms that are common in practically any application. Furthermore, EEMBC carefully chose the CoreMark implementation such that all computations are driven by run-time provided values to prevent code elimination during compile time optimization. CoreMark also sets specific rules about how to run the code and report results, thereby eliminating inconsistencies.

CoreMark Composition

To appreciate the value of CoreMark, it’s worthwhile to dissect its composition, which in general is comprised of lists, strings, and arrays (matrixes to be exact). Lists commonly exercise pointers and are also characterized by non-serial memory access patterns. In terms of testing the core of a CPU, list processing predominantly tests how fast data can be used to scan through the list. For lists larger then the CPU’s available cache, list processing can also test the efficiency of cache and memory hierarchy.

List processing consists of reversing, searching or sorting the list according to different parameters, based on the contents of the list data items. In particular, each list item can either contain a pre-computed value or a directive to invoke a specific algorithm with specific data to provide a value during sorting. To verify correct operation, CoreMark performs a 16b cyclic redundancy check (CRC) based on the data contained in elements of the list. Since CRC is also a commonly used function in embedded applications, this calculation is included in the timed portion of the CoreMark.

In many simple list implementations, programs allocate list items as needed with a call to malloc. However, on embedded systems with constrained memory, lists are commonly constrained to specific programmer-managed memory blocks. CoreMark uses the latter approach to avoid calls to library code (malloc/free).

CoreMark partitions the available data space into 2 blocks, one containing the list itself, and the other containing the data items. This also applies to embedded designs where data can accumulate in a buffer (items) and pointers to the data are kept in lists (or sometimes ring buffers). The *data16* items are initialized based on data that is not available at compile time.

```
typedef struct list_data_s {
        ee_s16 data16;
        ee_s16 idx;
} list_data;
```

Each *data16* item really consists of two 8-bit parts, with the upper 8b containing the original value for the lower 8b. The data contained in the lower 8b is as follows:

0..2: Type of function to perform to calculate a value.
3..6: Type of data for the operation.
7 : Indicator for pre-computed or cached value.

The benchmark code modifies the *data16* item during each iteration of the benchmark. The *idx* item maintains the original order of the list items, so that CoreMark can recreate the original list without reinitializing the list (a requirement for systems with low memory capacity).

```
typedef struct list_head_s {
        struct list_head_s *next;
        struct list_data_s *info;
} list_head;
```

The list head is modified during each iteration of the benchmark and the *next* pointers are modified when the list is sorted or reversed. At each consecutive iteration of the benchmark, the algorithm sorts the list according to the information in the *data16* member, performs the test, and then recreates the original list by sorting back to the original order and rewriting the list data.

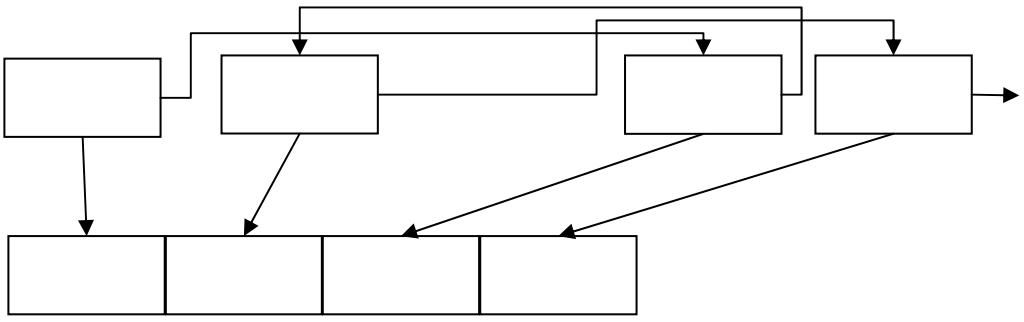

Figure 1. Basic structure of linked-list access mechanism. Each list item has a pointer to the data and a pointer to the next item in the list.

Since pointers on CPUs can range from 8 bits to 64 bits, the number of items initialized for the list is calculated such that the list will contain the same number of items regardless of pointer size. In other words, a CPU with 8-bit pointers will use ¼ of the memory that a 32-bit CPU uses to hold the list headers).

Matrix Processing

Many algorithms use matrixes and arrays, warranting significant research on optimizing this type of processing. These algorithms test the efficiency of tight loop operations as well as the ability of the CPU and associated toolchain to use ISA accelerators such as MAC units and SIMD instructions. These algorithms are composed of tight loops that iterate over the whole matrix. CoreMark performs simple operations on the input matrixes, including multiplication with a constant, a vector, or another matrix. CoreMark also tests operating on part of the data in the matrix in the form of extracting bits from each matrix item for operations. To validate that all operations have been performed, CoreMark again computes a CRC on the results from the matrix test.

Within the matrix algorithm for CoreMark, the available data space is split into three portions: an output matrix (with a 32b value in each cell) and two input matrixes (with 16b values in each cell). The input matrixes are initialized based on input values that are not available at compile time. During each iteration of the benchmark, the input matrixes are changed based on input values that cannot be computed at compile time. The input matrixes are recreated with the last operation and the same function can be invoked to repeat exactly the same processing.

State machine processing

An important function of a CPU core is the ability to handle control statements other than loops. A state machine based on switch or ‘if’ statements is an ideal candidate for testing that capability. There are 2 common methods for state machines – using switch statements or using a state transition table. Because CoreMark already utilizes the latter method in the list processing algorithm to test load/store behavior, CoreMark uses the former method, switch and ‘if’ statements, to exercise the CPU control structure.
The state machine tests an input string to detect if the input is a number, if it is not a number it will reach the “invalid” state. This is a simple state machine with 9 states. The input is a stream of bytes, initialized to ensure we pass all available states, based on an input that is not available at compile time. The entire input buffer is scanned with this state machine.

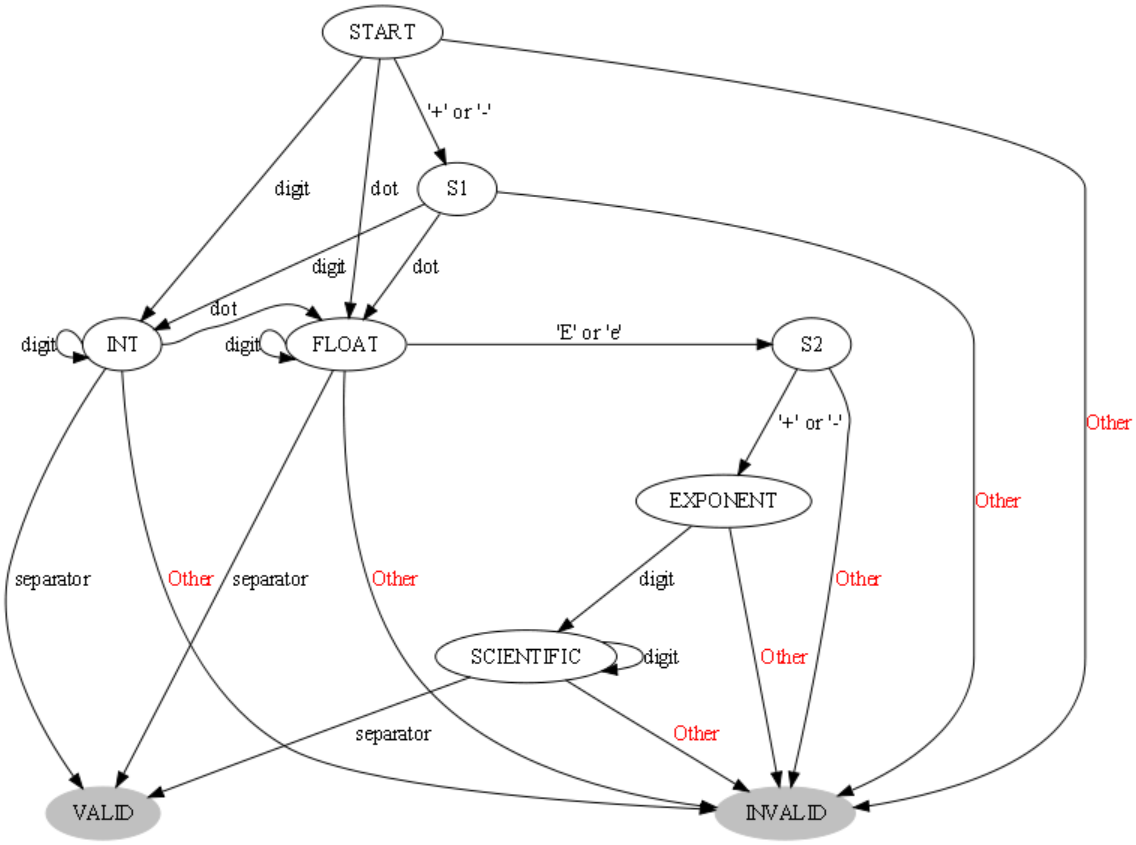

Figure 2. Overall functionality of CoreMark's state machine processing.

To validate operation, CoreMark keeps count of how many times each state was visited. During each iteration of CoreMark, some of the data is corrupted based on input that is not available at compile time. At the end of processing, the data is restored based on inputs not available at compile time.

CoreMark Profiling

Since CoreMark contains multiple algorithms, it is interesting to demonstrate how the behavior changes over time. For example, looking at the percentage of control code executed (samples taken at each 1000 cycles) and branch mis-predictions in Figure 3, it is obvious where the matrix algorithm is being called. This is portrayed by the low mis-prediction rate and high % of control operations, indicative of tight loops) (for example, between points 330-390).

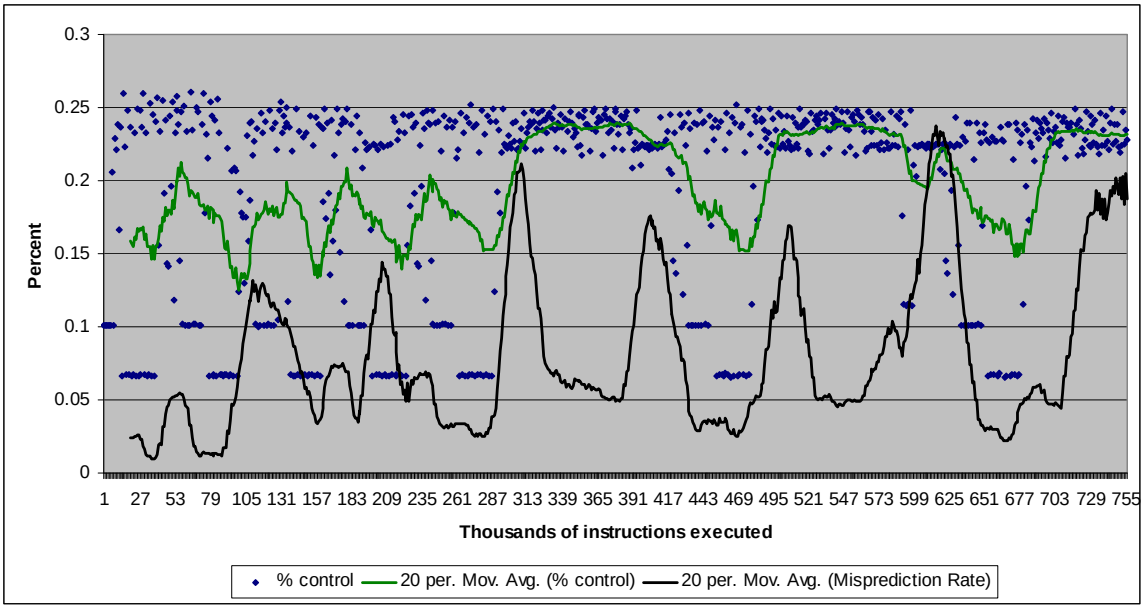

Figure 3. Distribution of control instructions and mispredictions over CoreMark execution. Note that the graph contains both the exact data points and the moving average for control operation. This is to emphasize the extremes.

By default, CoreMark only requires the allocation of 2 Kbytes to accommodate all data. This minimal memory size is necessary to support operation on the smallest microcontrollers, so that it can truly be a standard performance metric for any CPU core. Figure 4 examines the memory access pattern during the benchmark's execution. The information is represented as a percentage of memory operations that access memory within a certain distance from the previous access. It is easy to deduce that the distance peaks are caused by switching between the different algorithms (since each algorithm operates on a slice of 1/3 of the total available data space).

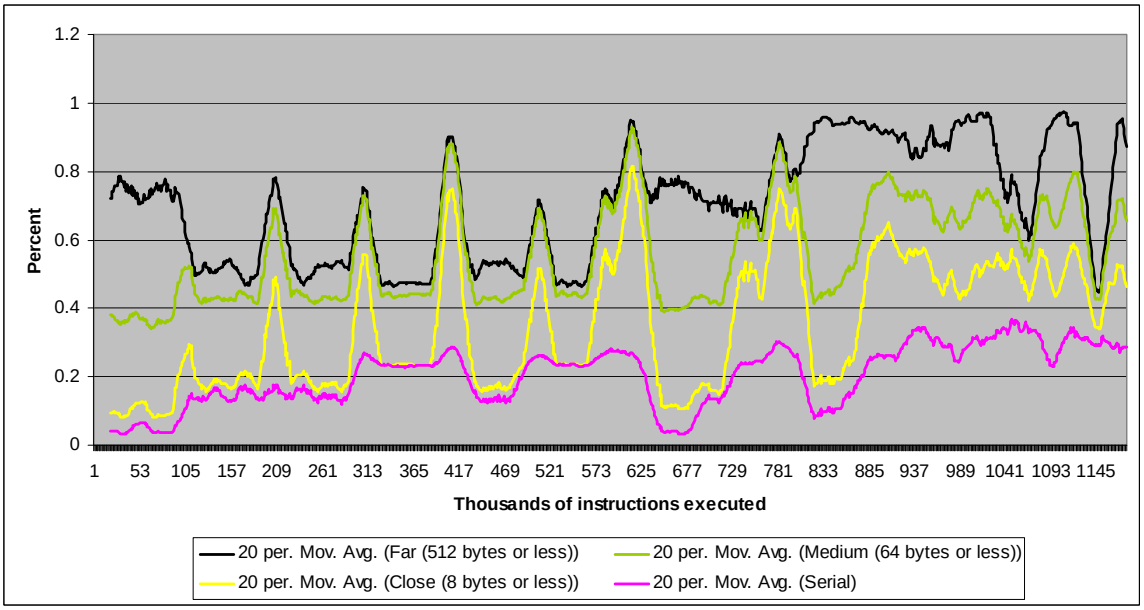

Figure 4. Distribution of memory access distance over time during CoreMark execution. Globally, about 20% of the time, memory access is serial, with peaks of serial access likely due to state machine

operation. This access pattern will test the cache mechanism (if any) and memory access efficiency on systems without cache.

Results

There are more than 120 CoreMark results available online at www.CoreMark.org, but Figure 5 shows a few results that display some interesting patterns. All results that depend on the compiler version and flags make it clear that these details must be included, otherwise it is impossible to make a useful comparison. The run and reporting rules for CoreMark require that exact tool versions be reported along with any performance results.

	Processor	Compiler	CoreMark/MHz
1	Analog Devices BF536 0.3 393 MHz	gcc4.1.2	1.01
2	Analog Devices BF536 0.3 393 MHz	gcc4.3.3	1.12
3	Microchip PIC24FJ64GA004 32MHz	gcc4.0.3 (dsPIC30, Microchip v3_20)	0.93
4	Microchip PIC24FJ64GA004 32MHz	gcc4.0.3 (dsPIC30, Microchip v3_20)	0.75
5	Microchip PIC24HJ128GP202 40MHz	gcc4.0.3 (dsPIC30, Microchip v3.12)	1.86
6	Microchip PIC24HJ128GP202 40MHz	gcc4.0.3 (dsPIC30, Microchip v3.12)	1.29
7	Microchip PIC32MX360F512L (MIPS32 M4K) 72MHz	gcc3.4.4 MPLAB C32 v1.00-20071024	1.71
8	Microchip PIC32MX360F512L (MIPS32 M4K) 72MHz	gcc3.4.4 MPLAB C32 v1.00-20071024	1.90
9	Microchip PIC32MX360F512L (MIPS32 M4K) 80MHz	gcc4.3.2 (Sourcery G++ Lite 4.3-81)	2.30
10	NXP LPC1114 48MHz	Keil ARMcc v4.0.0.524	1.06
11	NXP LPC1114 48MHz	gcc 4.3.3 (Code Red)	0.98
12	NXP LPC1768 100MHz	armcc 4.0	1.75
13	NXP LPC1768 72MHz	Keil ARMCC V4.0.0.524	1.76
14	Texas Instruments OMAP3530 500MHz	gcc4.3.3	2.42
15	Texas Instruments OMAP3530 600MHz	gcc4.3.3	2.19
16	TI Stellaris LM3S9B96 Cortex M3 50MHz	Keil ARMCC V4.0.0.524	1.92
17	TI Stellaris LM3S9B96 Cortex M3 80MHz	Keil ARMCC V4.0.0.524	1.60
18	Xilinx MicroBlaze v7.20.d in Spartan XC3S700A FPGA, 3-s	gcc4.1.1 (Xilinx MicroBlaze)	1.48
19	Xilinx MicroBlaze v7.20.d in Spartan XC3S700A FPGA, 5-s	gcc4.1.1 (Xilinx MicroBlaze)	1.66

Figure 5. CoreMark/MHz performance data, and complete benchmark environment details are available at www.coremark.org.

A. Blackfin results (1,2) show a 10% increase in performance when moving from GCC 4.1.2 to GCC 4.3.3, a reasonable expectation for a newer compiler version.
B. Results (8,9) show an even more pronounced difference of 18% for a mature compiler, while other results (10,11) (12,13) show minor effects only, as all of those compilers are based on the GCC4 series.
C. The compiler can also balance code size vs. performance as we can see in results (3,4). The compiler and platform are the same, but when directed to build a smaller executable (-Os -mpa), performance drops 19% vs. optimizing the code with –O3. The distinction is even sharper with results (5,6) at 30% performance difference (using –mpa switch). Note: compiler switch information is available from CoreMark website reports.
D. Other compiler options affect how much the compiler tries to optimize the code. Results (7,8) show a typical effect from safest (-O2) vs. normal (-O3) optimizations of about 10%.
E. When operating frequency is scaled up, the system memory and/or on-chip flash cannot always maintain a 1:1 ratio. It is common to see extra wait states on the flash when using higher processor frequencies. When the code resides in flash, the efficiency (as expressed in CoreMark/MHz) is impacted; (14,15) shows the

efficiency dropping almost 10%. For (16,17) the wait-state effect is even more pronounced as the CPU:memory ratio can only be maintained 1:1 up to 50MHz; when operating at the devices' highest frequency (80MHz), the ratio drops to 1:2 resulting in an efficiency drop of 15%. However, running at 80MHz still yields an absolute performance improvement of 25% vs. running at 50MHz.

F. The results (18,19) explore the situation where the cache is too small to contain all of the data. In 18, the cache is exactly 2K, which will fit all the data just barely, but have no room left for function arguments that must be passed on the stack. This causes a small amount of bus traffic external to the cache, but when the cache is enlarged (result 19) performance improves 10% even though the pipeline has been changed to a less efficient 5 stage pipeline vs. a 3 stage pipeline for the first implementation.

Overall CoreMark is well suited to comparing embedded processors. It is small, highly portable, well understood, and highly controlled. CoreMark verifies that all computations were completed correctly during execution, which helps debug any issues that may come up. The run rules are clearly defined and reporting rules are enforced on the CoreMark web site. In addition, EEMBC offers certification for CoreMark scores and even has a standardized method to measure energy consumption while running the benchmark.