Abstract:
To make an informed choice among the available embedded microcontrollers for automotive applications, designers need benchmarks that are specific to these applications and provide a realistic representation of how the device will perform in the automotive environment. This class gives an overview of the challenges of developing such benchmarks, the hardware required for their implementation, and the process by which a new suite of benchmarks that addresses these concerns is being implemented by EEMBC. Key points include the creation of benchmarks that can be generated to match specific OEM specifications and which can evolve in tandem with automotive standards as soon as requirements are known, rather than requiring a long development cycle.

Real-Time Benchmarks for Automotive Applications

By Shay Gal-On and Markus Levy, EEMBC

Since the early 1980s, advanced microcontrollers (MCUs) have been adding digital intelligence and sophisticated control to an ever-increasing variety of applications in the automobile. Some MCUs optimize the engine performance, while others monitor and operate critical systems like airbags and antiskid braking. MCUs may control simple power windows and air conditioning or operate the audio, video, and navigation systems in the car. To select the best MCU for each of those applications, the designer must properly evaluate the MCU’s capability in the specific application.

The performance and capabilities of earlier 8-bit MCUs were relatively easy to assess, but evaluating modern processors with 32-bit capabilities, multilevel memory structures, and complex peripherals can be a challenge – and MCU selection must be done before any real work on the actual design begins. System partitioning and software development choices can impact application performance as much as MCU characteristics. Benchmarks are a popular method of evaluating a processor’s performance but can only predict true performance if they closely approximate the target application and its development environment.

A team of programmers and applications engineers within EEMBC is developing a benchmark suite that gives the automotive application designer an improved tool for evaluating processor choices for this unique environment. The goal of the project is to develop a benchmark suite capable of determining the right MCU for the automotive application and is also accepted as a standard measuring tool by automotive developers around the world. To do this, every factor influencing the processor’s performance must be considered, including MCU characteristics, the instruction set, buses, peripherals, memory, compilers, profilers, operating systems, software, drivers, and automatic code generators.

Introduction

Microcontrollers have come a long way since those early 8-bitters appeared under the hood. Microcontrollers have evolved to 16-bit and 32-bit instruction sets and data types, providing the
performance to keep pace with increasing demands in powertrain for pollution control, fuel economy, efficiency, and versatility. Transmission control and integration with the engine, and even vehicle handling, have added to the burden on the automotive MCU.

![Figure 1: Increasing performance demand](image)

MCUs have been able to match the auto industry’s demands by riding on Moore’s Law, allowing semiconductor vendors to double performance every 18 to 24 months without increasing costs. Meanwhile, the automotive industry has become one of the most stringent for improved quality and reliability in spite of the fact that the delicate electronics it uses must operate in some of the harshest environments.

For powertrain applications (which might manage the engine, transmission, and possibly differential or transaxle), tougher emission control regulations such as ULEV in the U.S. and Euro IV in Europe require sophisticated modeling to define engine behavior. Consumers join government agencies in keeping the pressure on for smaller, lighter motors generating more power with less fuel, and that becomes more urgent when oil supplies are less certain or more expensive.

Earlier engine control units (ECU) used relatively simple state machine and lookup tables to describe parameters for fuel delivery and spark generation. More recent methods utilize models that refine lookup tables by including many more parameters, such as wear-and-tear, for more sophisticated control algorithms. Over time, improved real-time analysis of the combustion process within a cylinder has moved traditional lookup tables to a dynamic fuel management system. Such changes drive demand for increasing processing capabilities for the engine. Add in tighter coordination between the engine and transmission or move to adaptive body control, and the system becomes much more complex.

Modern hybrid automobiles place even more demands on under-the-hood processors, coupling additional control strategy into the mix. Hybrid’s recent success also underlines that the consumer, as well as government incentives, will continue to push for better fuel economy, whatever the fuel may be.
Handling Growing System Complexity

Fortunately, MCUs and microprocessors (MPU) have been on a fast track, going multicore, increasing frequency, extending memory, and adding new peripherals and enhancing the older ones. Like most semiconductors, today’s MCU is hundreds of times more capable than its early predecessors. This capability shows itself in better processing throughput, bigger memory size, and higher transistor count. Processors now support greatly enhanced instruction sets. 32-bit ($2^{32}$) instructions offer far more possibilities than 8-bit ($2^8$) instruction words. Plus many MCUs integrate digital signal processor (DSP) functionality, floating-point operations, multiple processors, hardware accelerators, and customized peripherals. All of these enhancements provide greatly increased performance.

![Size of Flash in KByte](image_url)

Figure 2: Memory growth in engine management

For an automotive application, some of this performance goes toward increasing functionality, robustness, or self-test. An even larger part goes to managing complexity and lowering system costs through use of software platforms and reusable software modules across multiple applications. Key to this effort is development of hardware-independent software. To lower costs, functions like knock signal conditioning for engine management, which were previously implemented in hardware, can now be performed in software on higher performance MCUs.

Moving to hardware-independent software makes software development more critical. Fortunately, automatic code generation has the potential to speed up code development and algorithm quality verification. It will also help programmers more readily extract legacy code. This performance increase opens new possibilities across applications and the industry for standardization of application interfaces, development of encapsulated transparent software, and software self-test.

Evaluating the Performance-Related Features of a Microcontroller

A microcontroller consists of a processor, memory, and peripherals, all on a single piece of silicon. Advances in technology for each of the three elements have led to performance increases for the
MCU. Instructions in the processor core have been made more powerful and efficient, allowing more work to be done with each clock cycle. MCUs are clocked at much higher speeds than was possible 20 years ago so that more work can be performed faster as well. Development tools have evolved that let the MCUs be programmed using automatic coding tools.

Memory has also contributed to increased performance. More memory is needed with each passing generation for additional application features. The memory cell has gotten smaller, but the memory access speed has not proportionally increased. For memory speeds to keep up with the increases of instruction speeds, exotic mechanisms such as burst access, double data rate, and caches have been developed.

The peripherals off-load work from the processor and decrease power consumption. More sophisticated peripherals have been developed to minimize attention from the processor while refining the service they perform. Meanwhile, even more peripherals have been added from network interfaces to high-resolution analog-to-digital converters (ADC).

Choosing the right MCU for an application is a complicated task, with the processor core, the on-chip memory, and the array of peripherals all affecting system performance, both on their own as well as working together. Does the processor have adequate performance to perform the necessary tasks, including coordination of the chip itself? What size memory will be needed to hold the programs and the working data? What is the right mix of peripherals? How much programming is needed to drive one manufacturer’s CAN module compared to that from another manufacturer? Can the job still be done using the less expensive part with one less timer? Obviously, the right way to answer these questions is to build the system and test it, but the time and resources to do that for each MCU is impractical, especially if a different system must be built for each MCU being compared. The industry must rely on a benchmarking process that reaches across a wide spectrum of platforms for comparison. The complexity of evaluating whether an MCU has the bandwidth to handle the job is a reflection of the system itself. A person can imagine that selecting the right MCU is no small feat.
Evaluating MCUs with Standard Benchmarks

A benchmark specific to powertrain applications should demonstrate how that device would perform in that environment. Today’s EEMBC Automotive/Industrial benchmarks, which have been available for eight years, consist of a variety of algorithms frequently used in automotive and industrial control applications (see examples in the LIST.ALG).

LIST ALG

EEMBC Automotive Benchmark Algorithms

- Angle to Time Conversion
- Tooth-to-Spark
- Road Speed Calculation
- Floating Point
- Bit Manipulation
- “Cache Buster”
- Pointer Chasing
- CAN Remote Data Request
- Pulse Width Modulation (PWM)
- Table Lookup and Interpolation
- Fast Fourier Transform (FFT)
- Inverse Fast Fourier Transform (IFFT)
LIST ALG (continued)

Finite Impulse Response (FIR) Filter
Infinite Impulse Response (IIR) Filter
Inverse Discrete Cosine Transform (iDCT)
Matrix Arithmetic

The algorithms listed above test the instruction set of a processor. However, they do not consider use of the common peripherals. The next step in EEMBC’s evolution is to test these peripherals and to include more real-time tests to determine how a processor behaves inside the target application. The new benchmarks will utilize a special hardware exerciser that provides a stimulus to the I/O and interrupt pins of the MCU and captures and verifies the output. The purpose is to gain insight into the MCU’s ability to acquire data, process it, and then generate an output as a result.

This new benchmark suite will perform physical hardware tests for CAN, PWMs, ADC, serial port interfaces (SPI), UARTs, and capture/compare timers. The processor’s DMA controller, interrupt control, and context switch capabilities will be observable indirectly through a combination of different tasks. An exerciser board will generate the appropriate signals to emulate each of these peripheral functions and thus enable the test.

This advanced benchmark suite will also be able to show the extent to which different peripherals actually succeed in unburdening the processor. Measuring processor core performance as well as the microcontroller’s peripherals will give automotive powertrain designers a much better feeling for each MCU’s capabilities.

Advantages of Advanced Benchmarking

In some ways, evaluating a ROM-less and peripheral-free microprocessor is relatively easy. Granted, there are branch predictions, hit-rates of cache configurations, and instruction overlap to make it more interesting than just comparing clock rates. But the limited memory space and plethora of peripherals on microcontrollers make them far more interesting, and difficult, to assess.

But comparing one version of a peripheral to another, from different chips and even different vendors, can be difficult. If a simpler version of a timer is chosen, the processor must perform more of the higher-level timing operation, consuming power and code and data memory space. The time it takes may delay another function from being serviced as quickly.

Benchmarks that can comprehend peripherals as well as the processor allow a much more accurate comparison of MCUs. The work that peripherals perform is targeted, but significant, and frees the processor core to be more of a system manager than a “bit banger.” The timer on one MCU may be simple and require the processor to perform higher level timing operations, consuming power, time, and code and data memory space. Another MCU’s more-sophisticated timer may cost more but require little overhead from the processor. Even the overhead of changing state when servicing the peripheral can differ greatly between MCUs.

Because peripherals are so critical to MCU-based systems, it is vital to have benchmarks that can effectively measure the work they perform. The ability to apply random inputs and read the outputs of the peripherals is required to replicate the chaotic nature of the system in actual use and verify
correct operation while measuring performance.

My Code – The Next Step

Benchmarks for microprocessors, microcontrollers, and DSPs can help designers determine the processor architecture and configuration that will best perform in the application. The more benchmarks succeed in breaking down an end-system into its many component algorithms and tasks, the more effective they will be in predicting performance. For example, EEMBC’s digital entertainment suite involves 15 discrete benchmarks and as many as 69 executions with different datasets.

Standardized benchmarks aim to fairly represent the operations typically performed within a given application. But every implementation is different. One might offer more features, another might be compiled for speed, and still another might have been written directly in assembly rather than bulkier compiled code.

Designers are often frustrated by the task of determining how well their own algorithms and system management software will run on different processors. To be absolutely thorough, a different system and different code would need to be written for each test case. Such a rigorous approach eliminates the need for guesswork, but it can also significantly delay time to market.

Rigor and speed can be combined, however, if the designer has access to a characterization of the target application workload and a workload characterization of the benchmarks. With this information at hand, the designer’s own code can be tested with relative ease. The next section discusses how such workload and benchmark characterizations can be defined and delivered.

Standard Benchmark Characterization

With reasonable effort put into selecting the right benchmarks, an excellent proxy for an application being designed can be found, leading to the selection of the right processor for the job. It is not likely as simple as dividing a chip's SPECint numbers by its price, or even looking at the MCU with the most aggressive Automotive Benchmark total. A much better method of finding the right processor matches key characteristics of the benchmark with the workload of the target application.

The first step is to characterize the behavior of numerous standard benchmarks according to a number of carefully selected parameters. The difficult part is determining the most significant characteristics and finding a method for evaluating them.

LIST INSTR

Instruction Types within Program

Load
Store
Move
Flow Change
By analyzing the type of instructions the benchmark program executes, we can learn about the instruction mix and how it affects performance. How many loads and stores of data take place? How frequently are conditions tested? How many branches taken? How many simple arithmetic or logical operations are run? How many complex instructions are executed? And finally, what kind of parallelism is possible running the code and what is the optimum number of instructions per cycle that this program can take advantage of? Each of these can be expressed as a benchmark that can be situated within a graphic like that in FIGURE 4.

Additionally, LIST CACHE shows some relevant characteristics of a program that will highlight the physical environment in which the program will run best, such as the impact of cache organization, memory footprint, and branch prediction accuracy. Optimum instructions per cycle is derived from analysis of the instruction mix and usage.

LIST CACHE

Architectural Characterization of Program Execution

- Instruction Type (Load, Store, Branch, Multicycle...)
- Scalar Parallelism (Instructions Per Clock)
- Branch Prediction Accuracy
- Cache Size, Organization (Associativity, Line, Refill...)
- Each Cache
- Memory Speeds, Bus Delays

Characterizing benchmarks in this way provides insights into why some processors perform better
than others on the designer’s code. What are the ideal cache sizes and association for the benchmark? How frequently are loops and branches executed? Will variability/randomness of the conditional data thwart branch prediction mechanisms? Are instructions and data diverse enough to allow maximum parallelism in the processor pipeline? Will data be found in the cache or is it all over the memory map?

While these questions call for descriptive answers, engineers like data, charts, and graphs. Properly measuring the program characteristics can yield data and effective means of quantitative comparison. One handy way to view the profile of a program is shown in FIGURE 5. The size and the shape of this Kiviat graph represent a benchmark for the mentioned parameters. Differing shapes of the image convey information about peculiarities of the program that it represents.

![Figure 5: Kiviat Graph Illustrates Program Characteristics](image)

For years microprocessor designers have studied typical program code in these ways to make decisions to optimize each processor they built. Ultimately, choices are made that define each processor: data and separate instruction cache, each 32K bytes, 4-way set associative, 32-byte cache line; multiply hardware; DSP instructions; superscalar, seven-stage pipe, and so forth. In the end, the processor fits nicely across a wide variety of applications, but may not do so well with any one application in particular.

**Workload Characterization**

The solution to this problem is to tie the characterization directly to a given application. Software for the target application can also be characterized by the factors of LIST CACHE and LIST INST. With the right instrument, characterization is made of an OEM's key application software, looking at the same parameters. This will determine the application's workload requirements and outline the behavior and architecture that is ideal for running that application.

This is a new and uniquely valuable step. The workload characterization creates a profile of the OEM's program, which is likely to be very proprietary and which the OEM is unlikely to share with
outsiders, benchmarkers, or chip vendors for direct analysis. The measured characteristics reveal no actual code—though they might indicate extremes such as straight-line code or code heavily laden with memory writes—yet they provide enough information to allow valid comparisons.

Next, the workload characterization of the target application is matched to the reference benchmark characterizations. The normal grouping of benchmarks into application areas may have to be ignored for best results. For example, JPEG decoding is not a normal component of an automotive benchmark suite, but such a benchmark might closely approximate various characteristics of a critical piece of an engine control application. Proper characterization of both the benchmark and the end-equipment software would match those two together.

The collection of benchmarks that is found to have a good fit of characterizations may need some weighting to assure proper balance for the target. The result is a model that is an excellent proxy of the target application, specifically determined by routines selected by the application's designers. It is a model that could never be arrived at by theoretical means, nor is it likely to bear much resemblance to a competitor’s application.

With this model in hand, the designer is ready to short-list the best processors for the application. Each processor architecture and configuration will perform differently on the model set of benchmarks, as will each compiler and option, singly and in combination. Studying the model and the processor scores of the underlying benchmarks will reveal the best processors and compilers for that application. For processors and compilers that have been scored on the standardized benchmarks, it should be easy to confidently size up their compatibility for and performance in the OEM's own application code.

In the end, the standard benchmarks are intermediaries that allow the system designer to get a realistic picture of the capabilities and performance of a spectrum of processors and configurations. The benchmarks are more than manufacturer’s claims, but significant, relevant routines run in a controlled manner on numerous platforms providing validated results. The workload characterization of the end-equipment programs show the designer which benchmarks are really relevant to the style and content of the design's specific software, not some generic version of it. There are many options to choose from, but with this process, the best technical solution should become readily apparent and the system can be designed with a processor that has all of the performance the system is going to need.

**Conclusion**

Electronics engineering in the automotive domain covers a very broad spectrum of applications, each with different needs. The performance requirements of microcontrollers are increasing, and designers are seeking innovative solutions to increased complexity and greater feature offerings. More than ever, it is difficult to specify, compare, and measure real performance. New benchmarks are mandatory to communicate equipment expectations and system validation among OEMs, Tier-ones, silicon vendors, and third-party suppliers. In order to manage the variety and complexity of such benchmarks, it is important to develop tools that characterize the work load, generate the benchmark program code, initiate the I/O stimulus, verify execution, and quantify system performance in a report.
The combination of benchmark characterization with workload characterization promises to let programmers essentially judge exactly how different processors will perform when executing their specific programs – beyond generalized standard benchmarks. This will neither require writing nor compiling to the architecture, nor exposing their proprietary code. These powerful advances in benchmarking are on the horizon and will soon be a part of the EEMBC automotive suites.