OABench™1.1

software benchmark data book

www.eembc.org

An Industry-Standard Benchmark Consortium

Table of Contents

OABench™ Version 1.1 Benchmark Name: Dithering

Highlights

- **Benchmarks Potential Performance of a Printer Application**
- **Uses Floyd-Steinberg Error Diffusion Dithering Algorithm (1975)**
- **Converts 8bpp Grayscale Image to Binary**
- **Largely Integer Math with Shifts and Logical Compares**

Application

The Dithering Benchmark is representative of color and monochrome printer applications. The algorithm converts a grayscale image into a form ready for printing using the Floyd-Steinberg Error Diffusion dithering algorithm. This algorithm propagates an error quantity from image row to image row, effectively diffusing errors from the rendering calculations and preventing unwanted printing artifacts, such as banding.

References: Robert Ulichney (1987); Digital Halftoning, The MIT Press, Cambridge, Massachusetts; pp. 239-242

Benchmark Description

The benchmark changes a 64K byte grayscale 8bpp image to a 8K binary image, using a Floyd-Steinberg Error Diffusion dithering algorithm. It uses two image buffers (one for the source image and a second for the generated output), and two line buffers to hold error data.

Two “error” arrays are used - one for saving the errors from the current row (used to dither the next row) and one from the previous row, used to diffuse the errors from that row to the current pixel. This array must be zeroed out first thing before the first row, to ensure that no spurious data is left there.

The error array is created such that there is one extra int at either end. This eliminates special processing at the start and end of each row (but requires zeroing the additional columns).

Each pixel of the input image file is processed as follows:

1. Calculate an “error” value using the history buffer (weighted values of surrounding pixels).
2. Calculate a binary output pixel value and store.
3. Store "error" value to next line history buffer.

Analysis of Computing Resources

The benchmark effectively stresses four areas of the target CPU:

- Its indirect references used for managing internal buffers.
- Its manipulation of large data sets, since large images will stress the cache.
- Its ability to manipulate packed-byte quantities, which are used to hold grayscale pixel information.
- Its ability to perform four byte-wide multiply-accumulate operations

per pixel.

This benchmark uses an instruction mix of integer Add/Subtract instructions (35%), Compare/Branch instructions (25%), Loads/Stores (20%), Shift/Rotate instructions (10%) and integer Multiply instructions (5%). The percentages are approximate and may vary across architectures. The C library function memset() is called twice per iteration, once for 8196 bytes and again for 2064 bytes. No floating-point calculations are used. The code size is small and the data size is large.

Special Notes: The Dithering Benchmark is part of the EEMBC OA™ score.

OABench™ Version 1.1

Benchmark Name: Image Rotation

Highlights

- **Benchmarks Potential Performance of a Printer Application.**
- **Uses a Bitmap Rotation Algorithm to Perform a Clockwise, 90° Rotation on a Binary Image.**
- **Tests Bit Manipulation, Comparison and Indirect Reference Capabilities.**
- **Largely Logical Compares/Branches and Integer Addition/Subtraction**

Application

The Image Rotation Benchmark is representative of color and monochrome printer applications that must rotate an arbitrary binary image 90 degrees, for example, to switch between portrait and landscape modes. This benchmark uses a bitmap rotation algorithm to perform a clockwise, 90-degree rotation on a binary image. Rotated images are assumed to be a complete image (i.e. not rotating a bitmap within a larger image), with rows padded out to byte boundaries.

Benchmark Description

The bitmap rotation algorithm is primarily aimed at testing the bit manipulation, comparison and indirect reference capabilities of the microprocessor. The algorithm uses a series of indirect references and bit masks to check and set individual bits in a data buffer representing a binary image. The implementation supports 8-, 16- and 32-bit data as well as little and big Endian memory architectures. Two buffers are used, one for input and one for output, rather than trying to rotate the image in place.

There are multiple input data buffers available to debug the benchmark, but the "Medium" image must be used in the certified benchmark. This image is 295 wide and 345 bits high, or about 12K. The input buffer is included in the benchmark as statically initialized data and the output buffer is created by calling the test harness memory allocation routine, th_malloc(). After the timed iterations have been completed, the test is run one additional time so that the results can be checked by calculating a CRC check of the output buffer.

The C library routine memset() is called at the beginning of each iteration to set the output buffer to zeroes.

Analysis of Computing Resources

The benchmark effectively stresses the bit manipulation capabilities of the target CPU.

This benchmark uses an instruction mix of Compare/Branch instructions (45%), integer Add/Subtract instructions (25%) and Loads/Stores (12%). The percentages are approximate and may vary across architectures. The C library function memset() is called once per iteration to initialize the 12K output buffer to zeroes. No floating-point calculations are used. The code size is small and the data size is moderate.

Special Notes:

The Image Rotation Benchmark is part of the EEMBC OA_{mark}™ score.

OABench™ Version 1.1

Benchmark Name: Text Processing

Highlights

- **Benchmarks Potential Performance of a Printer Interpretive Control Language.**
- **Parses Boolean Expressions Made Up Of Text Strings.**
- **Tests bit manipulation, comparison and indirect reference capabilities.**
- **Largely Shift/Rotates with Integer Math and Logical Compares/Branches**

Application

The Text Processing Benchmark is representative of a printer application where an interpretive control language is parsed. The algorithm parses boolean expressions represented as text lines made up of variables, constants and operators. The variables are space separated words, from 1 to 64 characters long, the constants are single character "T" or "F" and the operators may either be single character symbols (& | !) or their phonetic equivalents (and, or, not). Standard precedence rules for expression parsing apply.

Benchmark Description

Input to the benchmark consists of a statically declared array of variable length strings. The strings consist of variables, constants and operators separated by spaces. For example:

```
"sss and fred implies ( red & blue ) or fred"
```

The expression is broken down into a binary tree structure, with each branch on the tree being an operand (a single variable, or a constant, or a reference to yet another tree node representing another expression). Unary operators are stored as modifiers to each of the branches. The resulting structure is then traversed to evaluate the value of the expression.

The benchmark avoids calling a memory allocation routine by statically declaring and managing a 1000 node buffer.

After the timed iterations have been completed, the test is run one additional time and a CRC is calculated for the binary tree to be used for checking for correct operation.

Analysis of Computing Resources

This benchmark exercises the byte manipulation, pointer comparison, indirect reference handling and stack manipulation capabilities of a processor.

This benchmark uses an instruction mix of Compare/Branch instructions (35%), Load/Store instructions (30%), Add/Subtract instructions (20%) and Logical/Shift instructions (8%). About 30% of the memory accesses are

for characters or strings. The percentages are approximate and may vary across architectures. The C library functions `strcmp()` and `strncpy()` are used extensively by this benchmark. No floating-point calculations are used. The code size and the data size are moderate.

Special Notes:

The Text Processing Benchmark is part of the EEMBC OA™ score.

Optimizations Allowed

Out of the Box/Standard C

- The C code must not be changed unless it must be modified to get it to compile. All changes must be documented and must not have a performance impact.
- Additional hardware can be used if it does not require code changes.
- All optimized libraries must be part of the standard compiler package, and/or available to all customers.
- The EEMBC Test Harness Regular or Test Harness Lite may be used. Test harness changes may be made for portability reasons if they do not impact performance.

Full Fury / Optimized

ASM	*Opt. Libs*	*Inlining*	*Re-write Alg.*	*Hard-ware*
Yes	**Yes**	**Yes**	**No**	**Yes**

- The basic algorithm may not be rewritten.
- The code may be rewritten in assembler, as long as it doesn't change the algorithm.
- Optimized libraries can be used if they are publicly available.
- Hardware assist can be used if it is on the same processor as that being benchmarked.
- In-lining is allowed.
- Additional data files may be used by ECL during certification to ensure the correctness of the optimized benchmark.