Archives
Columns
Features
Print Archives 1994-1998

Special
BYTE Digest
Michael Abrash's *Graphics Programming Black Book*
101 Perl Articles

About Us
How to Access BYTE.com
Write to BYTE.com
Advertise with BYTE.com

BYTEmark

This is release 2 of BYTE Magazine's BYTEmark benchmark program (previously known as BYTE's Native Mode Benchmarks). This document covers the Native Mode (a.k.a. Algorithm Level) tests; benchmarks designed to expose the capabilities of a system's CPU, F PU, and memory system.

The Tests

The Native Mode portion of the BYTEmark consists of a number of well-known algorithms; some BYTE has used before in earlier versions of the benchmark, others are new. The complete suite consists of 10 tests:

Numeric sort - Sorts an array of 32-bit integers.

String sort - Sorts an array of strings of arbitrary length.

Bitfield - Executes a variety of bit manipulation functions.

Emulated floating-point - A small software floating-point package.

Fourier coefficients - A numerical analysis routine for calculating series approximations of waveforms.

Assignment algorithm - A well-known task allocation algorithm.

Huffman compression - A well-known text and graphics compression algorithm.

IDEA encryption - A relatively new block cipher algorithm.

Neural Net - A small but functional back-propagation network simulator.

LU Decomposition - A robust algorithm for solving linear equations.

A more complete description of each test can be found in later sections of this document.

BYTE built the BYTEmark with the multiplatform world foremost in mind. There were, of course, other considerations that we kept high on the list:

Real-world algorithms . The algorithms should actually do something. Previous benchmarks often moved gobs of bytes from one

point to another, added or subtracted piles and piles of numbers, or (in some cases) actually executed NOP instructions. We should not belittle those tests of yesterday, they had their place. However, we think it better that tests be based on activities that are more complex in nature.

Easy to port . All the benchmarks are written in "vanilla" ANSI C. This provides us with the best chance of moving them quickly and accurately to new processors and operating systems as they appear. It also simplifies mai ntenance.

This means that as new 64-bit (and, perhaps, 128-bit) processors appear, the benchmarks can test them as soon as a compiler is available.

Comprehensive . The algorithms were derived from a variety of sources. Some are routines that BYTE had been using for some time. Others are routines derived from well-known texts in the computer science world. Furthermore, the algorithms differ in structure. Some simply "walk" sequentially through one-dimensional arrays. Others build and manipulate two-dimensional arrays. Finally, some benchmarks are "integer" tests, while others exercise the floating-point coprocessor (if one is available).

Scalable . We wanted these benchmarks to be useful across as wide a variety of systems as possible. We also wanted to give them a lifetime beyond the next wave of new processors.

To that end, we incorporated "dynamic workload adjustment." A complete description of this appears in a later section. In a nutshell, this allows the tests to "expand or contract" depending on the capabilities of the system under test, all the while providing consistent results so that fair and accurate comparisons are possible.

Honesty In Advertising

We'd be lying if we said that the BYTEmark was all the benchmarking that anyone would ever need to run on a system. It would be equally inaccurate to suggest that the tests are completely free of inadequacies. There are many things the tests do not do, there are shortcomings, and there are problems.

BYTE will continue to improve the BYTEmark. The source code is freely available, and we encourage vendors and users to examine the routines and provide us with their feedback. In this way, we assure fairness, comprehensiveness, and accuracy.

Still, as we mentioned, there are some shortcomings. Here are those we consider the most significant. Keep them in mind as you examine the results of the be nchmarks now and in the future.

At the mercy of C compilers . Being written in ANSI C, the benchmark program is highly portable. This is a reflection of the "world we live in." If this were a one-processor world, we might stand a chance at hand-crafting a benchmark in assembly language. (At one time, that's exactly what BYTE did.) Not today, no way.

The upshot is that the benchmarks must be compiled. For broadest coverage, we selected ANSI C. And when they're compiled, the resulting executable's performance can be highly dependent on the capabilities of the C compiler. Today's benchmark results can be blown out of the water tomorrow if someone new enters the scene with an optimizing strategy that outperforms existing competition.

This concern is not easily waved off. It will require you to keep careful track of compiler version and optimization switches. As BYTE builds its database of benchmark results, version number and switch setting will bec ome an integral part of that data. This will be true for published

information as well, so that you can make comparisons fairly and accurately. BYTE will control the distribution of test results so that all relevant compiler information is attached to the data.

As a faint justification -- for those who think this situation results in "polluted" tests -- we should point out that we are in the same boat as all the other developers (at least, all those using C compilers -- and that's quite a sizeable group). If the only C compilers for a given system happen to be poor ones, everyone suffers. It's a fact that a given platform's ultimate potential depends as much on the development software available as on the technical achievements of the hardware design.

It's just CPU and FPU . It's very tempting to try to capture the performance of a machine in a single number. That has never been possible -- though it's been tried *a lot* -- and the gap between that i deal and reality will forever widen.

These benchmarks are meant to expose the theoretical upper limit of the CPU, FPU, and memory architecture of a system. They cannot measure video, disk, or network throughput (those are the domains of a different set of benchmarks). You should, therefore, use the results of these tests as part, not all, of any evaluation of a system.

Single threaded . Currently, each benchmark test uses only a single execution thread. It's unlikely that you'll find any modern operating system that does not have some multitasking component. How a system "scales" as more tasks are run simultaneously is an effect that the current benchmarks cannot explore.

BYTE is working on a future version of the tests that will solve this problem.

The tests are synthetic . This quite reasonable argument is based on the fact that people don't run benchmarks for a living, they run applications. Consequently, the only true measure of a system is how well it performs whatever applications you will be running. This, in fact, is the philosophy behind the BAPCo benchmarks.

This is not a point with which we would disagree. BYTE regularly makes use of a variety of application benchmarks. None of this suggests, however, that the BYTEmark benchmarks serve no purpose.

BYTEmark's results should be used as predictors. They can be moved to a new platform long before native applications will be ported. The BYTEmark benchmarks will therefore provide an early look at the potential of the machine. Additionally, the BYTEmark permits you to "home in" on an aspect of the overall architecture. How well does the system perform when executing floating-point computations? Does its memory architecture help or hinder the management of memory buffers that may fall on arbitrary address boundaries? How does the cache work with a program whose memory access favors moving randomly through memory as opposed to moving sequentially through memory?

The answers to these questions can give you a good idea of how well a system would support a particular class of applications. Only a synthetic benchmark can give the narrow view necessary to find the answers.

Dynamic Workloads

Our long history of benchmarking has taught us one thing above all others: Tomorrow's system will go faster than today's by an amount exceeding your wildest guess -- and then some. Dealing with this can become an unending race.

It goes like this: You design a benchmark algorithm, you specify its parameters (how big the array is, how many loops, etc.), you run it on

today's latest super-microcomputer, collect your data, and go home. A new machine arrives the next day, you run your benchmark, and discover that the test executes so quickly that the resolution of the clock routine you're using can't keep up with it (i.e., the test is over and done before the system clock even has a chance to tic k).

If you modify your routine, the figures you collected yesterday are no good. If you create a better clock routine by sneaking down into the system hardware, you can kiss portability goodbye.

The BYTEmark benchmarks solve this problem by a process we'll refer to as "dynamic workload adjustment." In principle, it simply means that if the test runs so fast that the system clock can't time it, the benchmark increases the test workload -- and keeps increasing it -- until enough time is consumed to gather reliable test results.

Here's an example.

The BYTEmark benchmarks perform timing using a "stopwatch" paradigm. The routine `StartStopwatch()` begins timing; `StopStopwatch()` ends timing and reports the elapsed time in clock ticks. Now, "clock ticks" is a value that varies from system to system. We'll presume that our test system provides 1000 clock ticks per second. (We'll also presume that the system actuall y updates its clock 1000 times per second. Surprisingly, some systems don't do that. One we know of will tell you that the clock provides 100 ticks per second, but updates the clock in 5- or 6-tick increments. The resolution is no better than somewhere around 1/18th of a second.) Here, when we say "system" we mean not only the computer system, but the environment provided by the C compiler. Interestingly, different C compilers for the same system will report different clock ticks per second.

Built into the benchmarks is a global variable called `GLOBALMINTICKS` . This variable is the minimum number of clock ticks that the benchmark will allow `StopStopwatch()` to report.

Suppose you run the Numeric Sort benchmark. The benchmark program will construct an array filled with random numbers, call `StartStopwatch()` , sort the array, and call `StopStopwatch()` . If the time reported in `StopStopwatch()` is less than `GLOBALMINTIC KS` , then the benchmark will build two arrays, and try again. If sorting two arrays took less time than `GLOBALMINTICKS` , the process repeats with more arrays.

This goes on until the benchmark makes enough work so that an interval between `StartStopwatch()` and `StopStopwatch()` exceeds `GLOBALMINTICKS` . Once that happens, the test is actually run, and scores are calculated.

Notice that the benchmark didn't make bigger arrays, it made more arrays. That's because the time taken by the sort test does not increase linearly as the array grows, it increases by a factor of N*log(N) (where N is the size of the array).

This principle is applied to all the benchmark tests. A machine with a less accurate clock may be forced to sort more arrays at a time, but the results are given in arrays per second. In this way fast machines, slow machines, machines with accurate clocks, machines with less accurate clocks, can all be tested with the same code.

Confidence Intervals

Another built-in feature of the BYTEmark is a set of statistical-analysis routines. Running benchmarks is one thing; the question arises as to how many times should a test be run until you know you have a good sampling. Also, can you determine whether the test is stable (i.e., do results vary widely from one execution of the benchmark to the next)?

The BYTEmark keeps score as follows: Each test (a test being a numeric sort, a string sort, etc.) is run five times. These five scores are averaged, the standard deviation is determined, and a 95% *confidence half-interval* for the mean is calculated. This tells us that the true average lies -- with a 95% probability -- within plus or minus the confidence half-interval of the calculated average. If this half-interval is within 5% of the calculated average, the benchmarking stops. Otherwise, a new test is run and the calculations are repeated.

The upshot is that, for each benchmark test, the true average is -- with a 95% level of confidence -- within 5% of the average reported. Here, the "true average" is the average we would get were we able to run the tests over and over again an infinite number of times.

This specification ensures that the calculation of results is controlled; that someone running the tests in California will use the same technique for determining benchmark results as someone running the tests in New York.

Interpreting Results

Of course, running the benchmarks can present you with a boatload of data. It can get mystifying, and some of the more esoteric statistical information is valuable only to a limited audience. The big question is: What does it all mean?

First, we should point out that the BYTEmark reports both "raw" and indexed scores for each test. The raw score for a particular test amounts to the "iterations per second" of that test. For example, the numeric sort test reports as its raw score the number of arrays it was able to sort per second.

The indexed score is the raw score of the system under test divided by the raw score obtained on the baseline machine. As of this release, the baseline machine is a DELL 90 Mhz Pentium XPS/90 with 16 MB of RAM and 256K of external processor cache. (The compiler used was the Watcom C/C++ 10.0 compiler; optimizations set to "fastest possible code", 4-byte structure alignment, Pentium code generation with Pentium register-based calling.) The indexed score serves to "normalize" the raw scores, reducing their dynamic range and making them easier to grasp. Simply put, if your machine has an index score of 2.0 on the numeric sort test, it performed that test twice as fast as a 90 Mhz Pentium.

If you run all the tests (as you'll see, it is possible to perform "custom runs", which execute only a subset of the tests) the BYTEmark will also produce two overall index figures: Integer index and Floating-point index. The Integer index is the geometric mean of those tests that involve only integer processing -- numeric sort, string sort, bitfield, emulated floating-point, assignment, Huffman, and IDEA -- while the Floating-point index is the geometric mean of those tests that require the floating-point comprocessor -- Fourier, neural net, and LU decomposition. You can use these scores to get a general feel for the performance of the machine under test as compared to the baseline 90 Mhz Pentium.

What follows is a list of the benchmarks and associated brief remarks that describe what the tests do: What they exercise; what a "good" result or a "bad" result means. Keep in mind that, in this expanding universe of faster processors, bigger caches, more elaborate memory architectures, "good" and "bad" are indeed relative terms. A good score on today's hot new processor will be a bad score on tomorrow's hot new processor.

These remark s are based on empirical data and profiling that we have done to date. (NOTE: The profiling is limited to Intel and Motorola 68K on this release. As more data is gathered, we will be refining this section. 3/14/95--RG)

Benchmark	Description
Numeric sort	Generic integer performance. Should exercise non-sequential performance of cache (or memory if cache is less than 8K). Moves 32-bit longs at a time, so 16-bit processors will be at a disadvantage.
String sort	Tests memory-move performance. Should exercise non-sequential performance of cache, with added burden that moves are byte-wide and can occur on odd address boundaries. May tax the performance of cell-based processors that must perform additional shift operations to deal with bytes.
Bitfield	Exercises "bit twiddling" performance. Travels through memory in a somewhat sequential fashion; different from sorts in that data is merely altered in place. If properly compiled, takes into account 64-bit processors, which should see a boost.
Emulated F.P.	Past experience has shown this test to be a good measurement of overall performance.
Fourier	Good measure of transcendental and trigonometric performance of FPU. Little array activity, so this test should not be dependent of cache or memory architecture.
Assignment	The test moves through large integer arrays in both row-wise and column-wise fashion. Cache/memory with good sequential performance should see a boost (memory is altered in place -- no moving as in a sort operation). Processing is done in 32-bit chunks -- no advantage given to 64-bit processors.
Huffman	A combination of byte operations, bit twiddling, and overall integer manipulation. Should be a good general measurement.
IDEA	Moves through data sequentally in 16-bit chunks. Should provide a good indication of raw speed.
Neural Net	Small-array floating-point test heavily dependent on the exponential function; less dependent on overall FPU performance. Small arrays, so cache/memory architecture should not come into play.
LU decomp.	A floating-point test that moves through arrays in both row-wise and column-wise fashion. Exercises only fundamental math operations (+, -, *, /).

The Command File

Purpose

The BYTEmark program allows you to override many of its default parameters using a command file. The command file also lets you request statistical information, as well as specify an output file to hold the test results for later use.

You identify the command file using a command-line argument. E.G.,

C: NBENCH -cCOMFILE.DAT

tells the benchmark program to read from COMFILE.DAT in the current directory.

The content of the command file is simply a series of parameter names and values, each on a single line. The parameters control internal variables that are either global in nature (i.e., they effect all tests in the program) or are specific to a given benchmark test.

The parameters are listed in a reference guide that follows, arranged in the following groups:

Global Parameters

Numeric Sort

String Sort

Bitfield

Emulated floating-point

Fourier coefficients

Assignment algorithm

IDEA encryption

Huffman compression

Neural net

LU decomposition

As mentioned above, those items listed under "Global Parameters" affect all tests; the rest deal with specific benchmarks. There is no required ordering to parameters as they appear in the command file. You can specify them in any sequence you wish.

You should be judicio us in your use of a command file. Some parameters will override the "dynamic workload" adjustment that each test performs. Doing this completely bypasses the benchmark code that is designed to produce an accurate reading from your system clock. Other parameters will alter default settings, yielding test results that cannot be compared with published benchmark results.

A Sample Command File

Suppose you built a command file that contained the following:

```
ALLSTATS=T

CUSTOMRUN=T
```

```
OUTFILE=D:\DATA.DAT

DONUMSORT=T

DOLU=T
```

Here's what this file tells the benchmark program:

ALLSTATS=T means that you've requested a "dump" of all the statistics the test gathers. This includes not only the standard deviations of tests run, it also produces test-specific information such as the number of arrays built, the array size, etc.

CUSTOMRUN=T tells the system that this is a cus tom run. Only tests explicitly specified will be executed.

OUTFILE=D:\DATA.DAT will write the output of the benchmark to the file DATA.DAT on the root of the D: drive. (If DATA.DAT already exists, output will be appended to the file.)

DONUMSORT=T tells the system to run the numeric sort benchmark. (This was necessary on account of the CUSTOMRUN=T line, above.)

DOLU=T tells the system to run the LU decomposition benchmark.

Command File Parameters Reference

(NOTE: Altering some global parameters can invalidate results for comparison purposes. Those parameters are indicated in the following section by a bold asterisk (*****). If you alter any parameters so indicated, you may NOT publish the resulting data as BYTEmark scores.)

Global Parameters

`GLOBALMINTICKS=<n>`

This overrides the default `global_min_ticks` value (defined in NBENCH1.H). The `global_min_ticks` value is defined as the minimum number of c lock ticks per iteration of a particular benchmark. For example, if `global_min_ticks` is set to 100 and the numeric sort benchmark is run; each iteration MUST take at least 100 ticks, or the system will expand the work-per-iteration.

`MINSECONDS=<n>`

Sets the minimum number of seconds any particular test will run. This has the effect of controlling the number of repetitions done. Default: 5.

`ALLSTATS=<T|F>`

Set this flag to T for a "dump" of all statistics. The information displayed varies from test to test. Default: F.

`OUTFILE=<path>`

Specifies that output should go to the specified output file. Any test results and statistical data displayed onscreen will also be written to the file. If the file does not exist, it will be created; otherwise, new output will be appended to an existing file. This allows you to "capture" several runs into a single file for later review.

Note: the pat h should not appear in quotes. For example, something like the following would work: `OUTFILE=C:\BENCH\DUMP.DAT`

`CUSTOMRUN=<T|F>`

Set this flag to T for a custom run. A "custom run" means that the

program will run only the benchmark tests that you explicitly specify. So, use this flag to run a subset of the tests. Default: F.

Numeric Sort

`DONUMSORT=<T|F>`

Indicates whether to do the numeric sort. Default is T, unless this is a custom run (`CUSTOMRUN=T`), in which case default is F.

`NUMNUMARRAYS=<n>`

Indicates the number of numeric arrays the system will build. Setting this value will override the program's "dynamic workload" adjustment for this test. *****

`NUMARRAYSIZE=<n>`

Indicates the number of elements in each numeric array. Default is 8001 entries. (NOTE: Altering this value will invalidate the test for comparison purposes. The perform ance of the numeric sort test is not related to the array size as a linear function; i.e., an array twice as big will not take twice as long. The relationship involves a logarithmic function.) *****

`NUMMINSECONDS=<n>`

Overrides `MINSECONDS` for the numeric sort test.

String Sort

`DOSTRINGSORT=<T|F>`

Indicates whether to do the string sort. Default is T, unless this is a custom run (`CUSTOMRUN=T`), in which case the default is F.

`STRARRAYSIZE=<n>`

Sets the size of the string array. Default is 8111. (NOTE: Altering this value will invalidate the test for comparison purposes. The performance of the string sort test is not related to the array size as a linear function; i.e., an array twice as big will not take twice as long. The relationship involves a logarithmic function.) *****

`NUMSTRARRAYS=<n>`

Sets the number of string arrays that will be created to run the t est. Setting this value will override the program's "dynamic workload" adjustment for this test. *****

`STRMINSECONDS=<n>`

Overrides `MINSECONDS` for the string sort test.

Bitfield

`DOBITFIELD=<T|F>`

Indicates whether to do the bitfield test. Default is T, unless this is a custom run (`CUSTOMRUN=T`), in which case the default is F.

`NUMBITOPS=<n>`

Sets the number of bitfield operations that will be performed. Setting this value will override the program's "dynamic workload" adjustment for this test. *****

`BITFIELDSIZE=<n>`

Sets the number of 32-bit elements in the bitfield arrays. The default value is dependent on the size of a `long` as defined by the current compiler. For a typical compiler that defines a `long` to be 32 bits, the default is 32768. (NOTE: Altering this parameter will invalidate test results for comparison purposes .) *****

`BITMINSECONDS=<n>`

Overrides `MINSECONDS` for the bitfield test.

Emulated floating-point

`DOEMF=<T|F>`

Indicates whether to do the emulated floating-point test. Default is T, unless this is a custom run (`CUSTOMRUN=T`), in which case the default is F.

`EMFARRAYSIZE=<n>`

Sets the size (number of elements) of the emulated floating-point benchmark. Default is 3000. The test builds three arrays, each of equal size. This parameter sets the number of elements for EACH array. (NOTE: Altering this parameter will invalidate test results for comparison purposes.) *****

`EMFLOOPS=<n>`

Sets the number of loops per iteration of the floating-point test. Setting this value will override the program's "dynamic workload" adjustment for this test. *****

`EMFMINSECONDS=<n>`

Overrides `MINSECONDS` for the emulated floating-point test .

Fourier coefficients

`DOFOUR=<T|F>`

Indicates whether to do the Fourier test. Default is T, unless this is a custom run (`CUSTOMRUN=T`), in which case the default is F.

`FOURASIZE=<n>`

Sets the size of the array for the Fourier test. This sets the number of coefficients the test will derive. NOTE: Specifying this value will override the system's "dynamic workload" adjustment for this test, and may make the results invalid for comparison purposes. *****

`FOURMINSECONDS=<n>`

Overrides `MINSECONDS` for the Fourier test.

Assignment Algorithm

`DOASSIGN=<T|F>`

Indicates whether to do the assignment algorithm test. Default is T, unless this is a custom run (`CUSTOMRUN=T`), in which case the default is F.

`ASSIGNARRAYS=<n>`

Indicates the number of arrays that will be built for the test. Specifying this value will overrid e the system's "dynamic workload" adjustment for this test. (NOTE: The size of the arrays in the assignment algorithm is fixed at 101 x 101. Altering the array size requires adjusting global constants and recompiling; to do so, however, would invalidate test results.) *****

ASSIGNMINSECONDS=<n>

Overrides MINSECONDS for the assignment algorithm test.

IDEA encryption

DOIDEA=<T|F>

Indicates whether to do the IDEA encryption test. Default is T, unless this is a custom run (CUSTOMRUN=T), in which case the default is F.

IDEAARRAYSIZE=<n>

Sets the size of the plaintext character array that will be encrypted by the test. Default is 4000. The benchmark actually builds 3 arrays: 1st plaintext, encrypted version, and 2nd plaintext. The 2nd plaintext array is the destination for the decryption process [part of the test]. All arrays are set to the same size. (NOTE: Specifying this value will invalidate test results for comparison purposes.) *****

IDEALOOPS=<n>

Indicates the number of loops in the IDEA test. Specifying this value will override the system's "dynamic workload" adjustment for this test. *****

IDEAMINSECONDS=<n>

Overrides MINSECONDS for the IDEA test.

Huffman compression

DOHUFF=<T|F>

Indicates whether to do the Huffman test. Default is T, unless this is a custom run (CUSTOMRUN=T), in which case the default is F.

HUFFARRAYSIZE=<n>

Sets the size of the string buffer that will be compressed using the Huffman test. The default is 5000. (NOTE: Altering this value will invalidate test results for comparison purposes.) *****

HUFFLOOPS=<n>

Sets the number of loops in the Huffman test. Specifying this value will override the system's "dynamic workload" adjustment for this test. *****

HUFFMINSECONDS=<n>

Overrides MINSECONDS for the Huffman test.

Neural net

DONNET=<T|F>

Indicates whether to do the Neural Net test. Default is T, unless this is a custom run (CUSTOMRUN=T), in which case the default is F.

NNETLOOPS=<n>

Sets the number of loops in the Neural Net test. NOTE: Altering this value overrides the benchmark's "dynamic workload" adjustment algorithm, and may invalidate the results for comparison purposes. *****

NNETMINSECONDS=<n>

Overrides MINSECONDS for the Neural Net test.

LU decomposition

`DOLU=<T|F>`

Indicates whether to do the LU decomposition test. Default is T, unless this is a custom run (`CUSTOMRUN=T)` , in which case the default is F.

`LUNUMARRAYS=<n>`

Sets the number of arrays in each iteration of the LU decomposition test. Specifyi ng this value will override the system's "dynamic workload" adjustment for this test. *****

`LUMINSECONDS=<n>`

Overrides `MINSECONDS` for the LU decomposition test.

Numeric Sort

Description

This benchmark is designed to explore how well the system sorts a numeric array. In this case, a numeric array is a one-dimensional collection of signed, 32-bit integers. The actual sorting is performed by a heapsort algorithm (see the text box following for a description of the heapsort algorithm).

It's probably unnecessary to point out (but we'll do it anyway) that sorting is a fundamental operation in computer application software. You'll likely find sorting routines nestled deep inside a variety of applications; everything from database systems to operating-systems kernels.

The numeric sort benchmark reports the number of arrays it was able to sort per second. The array size is set by a global constant (it can be ov erridden by the command file -- see below).

Analysis

Optimized 486 code : Profiling of the numeric sort benchmark using Watcom's profiler (Watcom C/C++ 10.0) indicates that the algorithm spends most of its time in the `numsift()` function (specifically, about 90% of the benchmark's time takes place in `numsift()`). Within numsift(), two if statements dominate time spent:

`if(array[k]<array[k+1L])` and `if(array[i]<array[k])`

Both statements involve indexes into arrays, so it's likely the processor is spending a lot of time resolving the array references. (Though both statements involve "less-than" comparisons, we doubt that much time is consumed in performing the signed compare operation.) Though the first statement involves array elements that are adjacent to one another, the second does not. In fact, the second statement will probably involve elements that are far apart from one another during early passes t hrough the sifting process. We expect that systems whose caching system pre-fetches contiguous elements (often in "burst" line fills) will not have any great advantage of systems without pre-fetch mechanisms.

Similar results were found when we profiled the numeric sort algorithm under the Borland C/C++ compiler.

680x0 Code (Macintosh CodeWarrior): CodeWarrior's profiler is function based; consequently, it does not allow for line-by-line analysis as does the Watcom compiler's profiler.

However, the CodeWarrior profiler does give us enough information to note that `NumSift()` only accounts for about 28% of the time consumed

by the benchmark. The outer routine, `NumHeapSort()` accounts for around 71% of the time taken. It will require additional analysis to determine why the two compilers -- Watcom and CodeWarrior divide the workload so differently. (It may have something to do with compiler architecture, or the act of profiling the code may pr oduce results that are significantly different than how the program runs under normal conditions, though that would lead one to wonder what use profilers would be.)

Porting Considerations

The numeric sort routine should represent a trivial porting exercise. It is not an overly large benchmark in terms of source code. Additionally, the only external routines it calls on are for allocating and releasing memory, and managing the stopwatch.

The numeric sort benchmark depends on the following global definitions (note that these may be overridden by the command file):

NUMNUMARRAYS -- Sets the upper limit on the number of arrays that the benchmark will attempt to build. The numeric sort benchmark creates work for itself by requiring the system to sort more and more arrays...not bigger and bigger arrays. (The latter case would skew results, because the sorting time for heapsort is N log2 N - e.g., doubling the array size does not double the sort time.) This constant se ts the upper limit to the number of arrays the system will build before it signals an error. The default value is 100, and may be changed if your system exceeds this limit.

NUMARRAYSIZE - Determines the size of each array built. It has been set to 8111L and should not be tampered with. The command file entry NUMARRAYSIZE=<n> can be used to change this value, but results produced by doing this will make your results incompatible with other runs of the benchmark (since results will be skewed -- see preceding paragraph).

To test for a correct execution of the numeric sort benchmark, #define the DEBUG symbol. This will enable code that verifies that arrays are properly sorted. You should run the benchmark program using a command file that has only the numeric sort test enabled. If there is an error, the program will display "SORT ERROR." (If this happens, it's possible that tons of "SORT ERROR" messages will be emitted, so it's best not to redirect output to a file.)

References

Gonnet, G.H. 1984, Handbook of Algorithms and Data Structures (Reading, MA: Addison-Wesley).

Knuth, Donald E. 1968, Fundamental Algorithms, vol 1 of The Art of Computer Programming (Reading, MA: Addison-Wesley).

Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T. 1989, Numerical Recipes in Pascal (Cambridge: Cambridge University Press).

Heapsort

The heapsort algorithm is well-covered in a number of the popular computer-science textbooks. In fact, it gets a pat on the back in *Numerical Recipes* (Press et. al.), where the authors write:

Heapsort is our favorite sorting routine. It can be recommended wholeheartedly for a variety of sorting applications. It is a true "in-place" sort, requiring no auxiliary storage.

Heapsort works by building the array into a kind of a queue called a

heap. You can imagine this heap as being a form of in-memory binary t ree. The topmost (root) element of the tree is the element that -- were the array sorted -- would be the largest element in the array. Sorting takes place by first constructing the heap, then pulling the root off the tree, promoting the next largest element to the root, pulling it off, and so on. (The promotion process is known as "sifting up.")

Heapsort executes in N log2 N time even in its worst case. Unlike some other sorting algorithms, it does not benefit from a partially sorted array (though Gonnet does refer to a variation of heapsort, called "smoothsort," which does -- see references).

String Sort

Description

This benchmark is designed to gauge how well the system moves bytes around. By that we mean, how well the system can copy a string of bytes from one location to another; source and destination being aligned to arbitrary addresses. (This is unlike the numeric sort array, which moves bytes longword-at-a-time.) The strings themselves are built so as to be of random length, ranging from no fewer than 4 bytes and no greater than 80 bytes. The mixture of random lengths means that processors will be forced to deal with strings that begin and end on arbitrary address boundaries.

The string sort benchmark uses the heapsort algorithm; this is the same algorithm as is used in the numeric sort benchmark (see the sidebar on the heapsort for a detailed description of the algorithm).

Manipulation of the strings is actually handled by two arrays. One array holds the strings themselves; the other is a pointers array. Each member of the pointers array carries an offset that points into the string array, so that the *i* th pointer carries the offset to the *i* th string. This allows the benchmark to rapidly locate the position of the *i* th string. (The sorting algorithm requires exchanges of items that might be "distant" from one another in the array. It's critical that the routine be abl e to rapidly find a string based on its indexed position in the array.)

The string sort benchmark reports the number of string arrays it was able to sort per second. The size of the array is set by a global constant.

Analysis

Optimized 486 code (Watcom C/C++ 10.0) : Profiling of the string sort benchmark indicates that it spends most of its time in the C library routine `memmove().` Within that routine, most of the execution is consumed by a pair of instructions: `rep movsw` and `rep movsd.` These are *repeated string move -- word width* and *repeated string move -- doubleword width* , respectively.

This is precisely where we want to see the time spent. It's interesting to note that the `memmove()` of the particular compiler/profiler tested (Watcom C/C++ 10.0) was "smart" enough to do most of the moving on word or doubleword boundaries. The string sort benchmark specifically sets arbitrary boundaries, so we'd expect to see lots of byte-wide moves. The "smart" `memmove()` is able to move bytes only when it has to, and does the remainder of the work via words and doublewords (which can move more bits at a time).

680x0 Code (Macintosh CodeWarrior): Because CodeWarrior's profiler is function based, it is impossible to get an idea of how much time the test spends in library routines such as memmove(). Fortunately, as an artifact of the early version of the benchmark, the string sort algorithm makes use of the `MoveMemory()` routine in the `sysspec.c` file (system specific routines). This call, on anything other than

a 16-bit DOS system, calls `memmove()` directly. Hence, we can get a good approximation of how much time is spent moving bytes.

The answer is that nearly 78% of the benchmark's time is consumed by `MoveMemory()`, the rest being taken up by the other routines (the `str_is_less()` routine, which performs string c omparisons, takes about 7% of the time). As above, we can guess that most of the benchmark's time is dependent on the performance of the library's `memmove()` routine.

Porting Considerations

As with the numeric sort routine, the string sort benchmark should be simple to port. Simpler, in fact. The string sort benchmark routine is not dependent on any `typedef` that may change from machine to machine (unless a `char` type is not 8 bits).

The string sort benchmark depends on the following global definitions:

NUMSTRARRAYS - Sets the upper limit on the number of arrays that the benchmark will attempt to build. The string sort benchmark creates work for itself by requiring the system to sort more and more arrays, not bigger and bigger arrays. (See section on Numeric Sort for an explanation.) This constant sets the upper limit to the number of arrays the system will build before it signals an error. The default value is 100, and may be changed if your system exceeds this limit.

STRARRAYSIZE - Sets the default size of the string arrays built. We say "arrays" because, as with the numeric sort benchmark, the system adds work not by expanding the size of the array, but by adding more arrays. This value is set to 8111, and should not be modified, since results would not be comparable with other runs of the same benchmark on other machines.

To test for a correct execution of the string sort benchmark , #define the DEBUG symbol. This will enable code that verifies the arrays are properly sorted. Set up a command file that runs only the string sort, and execute the benchmark program. If the routine is operating properly, the benchmark will complete with no error messages. Otherwise, the program will display "Sort Error" for each pair of strings it finds out of order.

References

See the references for the Numeric Sort benchmark.

Bitfield Operations

Description

The purpose of this benchmark is to explore how efficiently the system executes operations that deal with "twiddling bits." The test is set up to simulate a "bit map"; a data structure used to keep track of storage usage. (Don't confuse this meaning of "bitmap" with its use in describing a graphics data structure.)

Systems often use bit maps to keep an inventory of memory blocks or (more frequently) disk blocks. In the case of a bit map that manages disk usage, an operating system will set aside a buffer in memory so that each bit in that buffer corresponds to a block on the disk drive. A 0 bit means that the corresponding block is free; a 1 bit means the block is in use. Whenever a file requests a new block of disk storage, the operating system searches the bit map for the first 0 bit, sets the bit (to indicate that the block is now spoken for), and returns the number of the corresponding disk block to the requesting file.

These types o f operations are precisely what this test simulates. A block of memory is set allocated for the bit map. Another block of

memory is allocated, and set up to hold a series of "bit map commands". Each bitmap command tells the simulation to do 1 of 3 things:

1) Clear a series of consecutive bits,

2) Set a series of consecutive bits, or

3) Complement (1->0 and 0->1) a series of consecutive bits.

The bit map command block is loaded with a set of random bit map commands (each command covers an random number of bits), and simulation routine steps sequentially through the command block, grabbing a command and executing it.

The bitfield benchmark reports the number of bits it was able to operate on per second. The size of the bit map is constant; the bitfield operations array is adjusted based on the capabilities of the processor. (See the section describing the auto-adjust feature of the benchmarks.)

Analysis

Optimized 486 code : Using the Watcom C/C++ 10.0 profiler, the Bitfield benchmark appears to spend all of its time in two routines: `ToggleBitRun()` (74% of the time) and `DoBitFieldIteration()` (24% of the time). We say "appears" because this is misleading, as we will explain.

First, it is important to recall that the test performs one of three operations for each run of bits (see above). The routine `ToggleBitRun()` handles two of those three operations: setting a run of bits and clearing a run of bits. An `if()` statement inside `ToggleBitRun()` decides which of the two operations is performed. (Speed freaks will quite rightly point out that this slows the entire algorithm. `ToggleBitRun()` is called by a `switch()` statement which has already decided whether bits should be set or cleared; it's a waste of time to have `ToggleBitRun()` have to make that decision yet again.)

`DoBitFieldIteration()` is the "outer" routine that calls `ToggleBitRun()` . `DoBitFieldIteration()` also calls `FlipBitRun()` . This latter routine is the one that performs the third bitfield operation: complementing a run of bits. `FlipBitRun()` gets no "air time" at all (while `DoBitFieldIteration()` gets 24 % of the time) simply because the compiler's optimizer recognizes that `FlipBitRun()` is only called by `DoBitFieldIteration()` , and is called only once. Consequently, the optimizer moves `FlipBitRun()` "inline", i.e., into `DoBitFieldIteration()` . This removes an unnecessary call/return cycle (and is probably part of the reason why the `FlipBitRun()` code gets 24% of the algorithm's time, instead of something closer to 30% of its time.)

Within the routines, those lines of code that actually do the shifting, the and operations, and the or operations, consume time evenly. This should make for a good test of a processor's " bit twiddling" capabilities.

680x0 Code (Macintosh CodeWarrior): The CodeWarrior profiler is function based. Consequently, it is impossible to produce a profile of machine instruction execution time. We can, however, get a good picture of how the algorithm divides its time among the various functions.

Unlike the 486 compiler, the CodeWarrior compiler did not appear to collapse the `FlipBitRun()` routine into the outer `DoBitFieldIteration()` routine. (We don't know this for certain, of course. It's possible that the compiler *would* have done this had we not been profiling.)

In any case, the time spent in the two "core" routines of the bitfield test are shown below:

`FlipBitRun()` - 18031.2 microsecs (called 509 times)

ToggleBitRun() - 50770.6 microsecs (called 1031 times)

In terms of total time, FlipBitRun() takes about 35% of the time (it gets about 33% of the calls). Remember, Tog gleBitRun() is a single routine that is called both to set and clear bits. Hence, ToggleBitRun() is called twice as often as FlipBitRun().

We can conclude that time spent setting bits to 1, seting bits to 0, and changing the state of bits, is about equal; the load is balanced close to what we'd expect it to be, based on the structure of the algorithm.

Porting Considerations

The bitfield operations benchmark is dependent on the size of the long datatype. On most systems, this is 32 bits. However, on some of the newer RISC chips, a long can be 64 bits long. If your system does use 64-bit longs, you'll need to #define the symbol LONG64 .

If you are unsure of the size of a long in your system (some C compiler manuals make it difficult to discover), simply place an ALLSTATS=T line in the command file and run the benchmarks. This will cause the benchmark program to display (among other things) the size of the data types int , short , and long in bytes.

BITFARRAYSIZE - Sets the number of longs in the bit map array. This number is fixed, and should not be altered. The bitfield test adjusts itself by adding more bitfield commands (see above), not by creating a larger bit map.

Currently, there is no code added to test for correct execution. If you are concerned that your port was incorrect, you'll need to step through your favorite debugger and verify execution against the original source code.

References

None.

Emulated Floating-point

Description

The emulated floating-point benchmark includes routines that are similar to those that would be executed whenever a system performs floating-point operations in the absence of a coprocessor. In general, this amounts to a mixture of integer instructions, including shift operations, integer addition and subtraction, and bit testing (among others).

The benchmark itself is remarkably simple. The test builds three 1-dimensional arrays and loads the first two up with random floating-point numbers. The arrays are then partitioned into 4 equal-sized groups, and the test proceeds by performing addition, subtraction, multiplication, and division -- one operation on each group. (For example, for the addition group, an element from the first array is added to the second array and the result is placed in the third array.)

Of course, most of the work takes place inside the routines that perform the addition, subtraction, multiplication, and division. These routines operate on a special data type (referred to as an InternalFPF number) that -- though not strictly IEEE compliant -- carries all the necessary data fields to support an IEEE-compatible floating-point system. Specifically, an InternalFPF number is built up of the following fields:

Type (indicates a NORMAL, SUBNORMAL, etc.)

Mantissa sign

Unbiased, signed 16-bit exponent

4-word (16 bits) mantissa.

The emulated floating-point test reports its results in number of loops per second (where a "loop" is one pass through the arrays as described above).

Finally, we are aware that this test could be on its way to becoming an anachronism. A growing number of systems are appearing that have coprocessors built into the main CPU. It's possible that floating-point emulation will one day be a thing of the past.

Analysis

Optimized 486 code (Watcom C/C++ 10.0): The algorithm's time is distributed across a number of routines. The distribution is:

`ShiftMantLeft1()` - 60% of the time

`ShiftMantRight1()` - 17% of the time

`DivideInternalFPF()` - 14% of the time

`MultiplyInternalFPF()` - 5% of the time.

The first two routines are similar to one another; both shift bits about in a floating-point number's mantissa. It's reasonable that ShiftMantLeft1 () should take a larger share o f the system's time; it is called as part of the normalization process that concludes every emulated addition, subtraction, mutiplication, and division.

680x0 Code (Macintosh CodeWarrior): CodeWarrior's profiler is function-based; consequently, it isn't possible to get timing at the machine instruction level. However, the output to CodeWarrior's profiler has provided insight into the breakdown of time spent in various functions that forces us to rethink our 486 code analysis.

Analyzing what goes on inside the emulated floatingpoint tests is a tough one to call because some of the routines that are part of the test are called by the function that builds the arrays. Consequently, a quick look at the profiler's output can be misleading; it's not obvious how much time a particular routine is spending in the test and how much time that same routine is spending setting up the test (an operation that does not get timed).

Specifically, the routine that loads up the arrays w ith test data calls `LongToInternalFPF()` and `DivideInternalFPF(). LongToInternalFPF()` makes one call to `normalize()` if the number is not a true zero. In turn, `normalize()` makes an indeterminate number of calls to `ShiftMantLeft1(),` depending on the structure of the mantissa being normalized.

What's worse, `DivideInternalFPF()` makes all sorts of calls to all kinds of important low-level routines such as `Sub16Bits()` and `ShiftMantLeft1().` Untangling the wiring of which routine is being called as part of the test, and which is being called as part of the setup could probably be done with the computer equivalent of detective work and spelunking, but in the interest of time we'll opt for approximation.

Here's a breakdown of some of the important routines and their times:

`AddSubInternalFPF()` - 1003.9 microsecs (called 9024 times)

`MultiplyInternalFPF()` - 20143 microsecs (called 56 10 times)

`DivideInternalFPF()` - 18820.9 microsecs (called 3366 times).

The 3366 calls to `DivideInternalFPF()` are timed calls, not setup calls -- the profiler at least gives outputs of separate calls made to the same routine, so we can determine which call is being made by the benchmark, and which is being made by the setup routine. It turns out that the setup routine calls `DivideInternalFPF()` 30,000 times.

Notice that though addition/subtraction are called most often, multiplication next, then finally division; the time spent in each is the reverse. Division takes the most time, then multiplication, finally addition/subtraction. (There's probably some universal truth lurking here somewhere, but we haven't found it yet.)

Other routines, and their breakdown:

`Add16Bits()` - 115.3 microsecs

`ShiftMantRight1()` - 574.2 microsecs

`Sub16Bits()` - 1762 microsecs

`StickySiftRightMant` - 40.4 micr osecs

`ShiftMantLeft1()` - 17486.1 microsecs

The times for the last three routines are suspect, since they are called by `DivideInternalFPF()` , and a large portion of their time could be part of the setup process. This is what leads us to question the results obtained in the 486 analysis, since it, too, is unable to determine precisely who is calling whom.

Porting Considerations

Earlier versions of this benchmark were extremely sensitive to porting; particularly to the "endianism" of the target system. We have tried to eliminate many of these problems. The test is nonetheless more "sensitive" to porting than most others.

Pay close attention to the following defines and typedefs. They can be found in the files EMFLOAT.H, NMGLOBAL.H, and NBENCH1.H:

`u8` - Stands for unsigned, 8-bit. Usually defined to be `unsigned char` .

`u16` - Stands for unsigned, 16-bit. Usually defined to be `uns igned short` .

`u32` - Stands for unsigned, 32-bit. Usually defined to be `unsigned long`.

`INTERNAL_FPF_PRECISION` - Indicates the number of elements in the mantissa of an InternalFPF number. Should be set to 4.

The exponent field of an InternalFPF number is of type `short` . It should be set to whatever minimal data type can hold a signed, 16-bit number.

Other global definitions you will want to be aware of:

CPUEMFLOATLOOPMAX - Sets the maximum number of loops the benchmark will attempt before flagging an error. Each execution of a loop in the emulated floating-point test is "non-destructive," since the test takes factors from two arrays, operates on the factors, and places the result in a third array. Consequently, the test makes more work for itself by increasing the number of times it passes through the arrays (# of loops). If the system exceeds the limit set by CPUEMFLOATLOOPMAX, it will signal an error.

Th is value may be altered to suit your system; it will not effect the benchmark results (unless you reduce it so much the system can never generate enough loops to produce a good test run).

EMFARRAYSIZE - Sets the size of the arrays to be used in the test. This value is the number of entries (InternalFPF numbers) per array. Currently, the number is fixed at 3000, and should not be altered.

Currently, there is no means of testing correct execution of the benchmark other than via debugger. There are routines available to decode the internal floating point format and print out the numbers, but no formal correctness test has been constructed. (*This should be available soon. -- 3/14/95 RG*)

References

Microprocessor Programming for Computer Hobbyists, Neill Graham, Tab Books, Blue Ridge Summit, PA, 1977.

Apple Numerica Manual, Second edition, Apple Computer, Addison-Wesley Publishing Co., Reading, MA, 1988.

Fourier Series

Description

This is a floating-point benchmark designed primarily to exercise the trigonometric and transcendental functions of the system. It calculates the first n Fourier coefficients of the function (x+1)x on the interval 0,2. In this case, the function (x+1)x is being treated as a cyclic waveform with a period of 2.

The Fourier coefficents, when applied as factors to a properly constructed series of sine and cosine functions, allow you to approximate the original waveform. (In fact, if you can calculate all the Fourier coefficients -- there'll be an infinite number -- you can reconstruct the waveform exactly). You have to calculate the coefficients via intergration, and the algorithm does this using a simple trapezoidal rule for its numeric integration function.

The upshot of all this is that it provides an exercise for the floating-point routines that calculate sine, cosine, and raising a number to a power. There are also some floating-point multiplications, divisions, additi ons, and subtractions mixed in.

The benchmark reports its results as the number of coefficients calculated per second.

As an additional note, we should point out that the performance of this benchmark is heavily dependent on how well-built the compiler's math library is. We have seen at least two cases where recompilation with new (and improved!) math libraries have resulted in two-fold and five-fold performance improvements. (Apparently, when a compiler gets moved to a new platform, the trigonometric and transcendental functions in the math libraries are among the last routines to be "hand optimized" for the new platform.) About all we can say about this is that whenever you run this test, verify that you have the latest and greatest math libraries.

Analysis

Optimized 486 code : The benchmark partitions its time almost evenly among the modules `pow387` , `exp386` , and `trig387` ; giving between 25% and 28% of its time to ea ch. This is based on profiling with the Watcom compiler running under Windows NT. These modules hold the routines that handle raising a number to a power and performing trigonometric (sine and cosine) calculations. For example, within `trig387` , time was nearly equally divided between the routine that calculates sine and the routine that calculates cosine.

The remaining time (between 17% and 18%) was spent in the balance

of the test. We noticed that most of that time occurred in the routine `thefunction()` . This is at the heart of the numerical integration routine the benchmark uses.

Consequently, this benchmark should be a good test of the exponential and trigonometric capabilities of a processor. (Note that we recognize that the performance also depends on how well the compiler's math library is built.)

680x0 Code (Macintosh CodeWarrior): The CodeWarrior profiler is function based, therefore it is impossible to get performance results for individua l machine instructions. The CodeWarrior compiler is also unable to tell us how much time is spent within a given library routine; we can't see how much time gets spent executing the `sin()` , `cos()` , or `pow()` functions (which, unfortunately, was the whole idea behind the benchmark).

About all we can glean from the results is that `thefunction()` takes about 74% of the time in the test (this is where the heavy math calculations take place) while `trapezoidintegrate()` accounts for about 26% of the time on its own.

Porting Considerations

Necessarily, this benchmark is at the mercy of the efficiency of the floating-point support provided by whatever compiler you are using. It is recommended that, if you are doing the port yourself, you contact the designers of the compiler, and discuss with them what optimization switches should be set to produce the fastest code. (This sounds simple; usually it's not. Some systems let you decide be tween speed and true IEEE compliance.)

As far as global definitions go, this benchmark is happily free of them. All the math is done using `double` data types. We have noticed that, on some Unix systems, you must be careful to include the correct math libraries. Typically, you'll discover this at link time.

To test for correct execution of the benchmark : It's unlikely you'll need to do this, since the algorithm is so cut-and-dried. Furthermore, there are no explicit provisions made to verify the correctness. You can, however, either dip into your favorite debugger, or alter the code to print out the contents of the abase (which holds the A[i] terms) and bbase (which holds the B[i] terms) arrays as they are being filled (see routine `DoFPUTransIteration`). Run the benchmark with a command file set to execute only the Fourier test, and examine the contents of the arrays. The first 4 elements of each array should be:

A[i] B[i]

2.8377707 56 n/a

1.045784473 -1.879103261

.2741002242 -1.158835123

.0824148217 -.8057591902

Note that there is no B[0] coefficient. If the above numbers are in the arrays shown, you can feel pretty confident that the benchmark it working properly.

References

Engineering and Scientific Computations in Pascal, Lawrence P. Huelsman, Harper & Row, New York, 1986.

Assignment Algorithm

Description

This test is built on an algorithm with direct application to the business world. The assignment algorithm solves the following problem: Say you have X machines and Y jobs. Any of the machines can do any of the jobs; however, the machines are sufficiently different that the cost of doing a particular job can vary depending what machine does it. Furthermore, the jobs are sufficiently different that the cost varies depending on which job a given machine does. You therefore construct a matrix; machines are the rows, jobs are the col umns, and the [i,j] element of the array is the cost of doing the jth job on the ith machine. How can you assign the jobs so that the cost of completing them all is minimal? (This also assumes that one machine does one job.)

Did you get that?

The assignment algorithm benchmark is largely a test of how well the processor handles problems built around array manipulation. It is not a floating-point test; the "cost matrix" built by the algorithm is simply a 2D array of long integers. This benchmark considers an iteration to be a run of the assignment algorithm on a 101 x 101 - element matrix. It reports its results in iterations per second.

Analysis

Optimized 486 code (Watcom C/C++ 10.0) : There are numerous loops within the assignment algorithm. The development system we were using (Watcom C/C++ 10.0) appears to have a fine time unrolling many of them. Consequently, it is difficult to pin down the execution impact of single lines (as in, for example, the numeric sort benchmark).

On the level of functions, the benchmark spends around 70% of its time in the routine `first_assignments()` . This is where a) lone zeros in rows and columns are found and selected, and b) a choice is made between duplicate zeros. Around 23% of the time is spent in the `second_assignments()` routine where (if `first_assignments()` fails) the matrix is partitioned into smaller submatrices.

Overall, we did a tally of instruction mix execution. The approximate breakdowns are:

move - 38%

conditional jump - 12%

unconditional jump - 11%

comparison - 14%

math/logical/shift - 24%

Many of the move instructions that appeared to consume the most amounts of time were referencing items on the local stack frame. This required an indirect reference through EBP, plus a constant offset to resolve the address.

This should be a good exercise of a cache, since operations in the `first_as signments()` routine require both row-wise and column-wise movement through the array. Note that the routine could be made more "severe" by chancing the `assignedtableau[][]` array to an array of `unsigned char` -- forcing fetches on byte boundaries.

680x0 Code (CodeWarrior): The CodeWarrior profiler is function-based. Consequently, it's not possible to determine what's going on at the machine instruction level. We can, however, get a good idea of how much time the algorithm spends in each routine. The important routines are broken down as follows:

`calc_minimum_costs()` - approximately 0.3% of the time

(250 microsecs)

`first_assignments()` - approximately 79% of the time

(96284.6 microsecs)

`second_assignments()` - approximately 19% of the time

(22758 microsecs)

These times are approximate; some time is spent in the `Assignment()` routine itself.

These figures are reas onably close to those of the 486, at least in terms of the mixture of time spent in a particular routine. Hence, this should still be a good test of system cache (as described in the preceding section), given the behavior of the `first_assignments()` routine.

Porting Considerations

The assignment algorithm test is purely an integer benchmark, and requires no special data types that might be affected by ports to different architectures. There are only two global constants that affect the algorithm:

ASSIGNROWS and ASSIGNCOLS - These set the size of the assignment array. Both are defined to be 101 (so, the array that is benchmarked is a 101 x 101 -element array of longs). These values should not be altered.

To test for correct execution of the benchmark : #define the symbol DEBUG, recompile, set up a command file that executes only the assignment algorithm, and run the benchmark. (You may want to pipe the output through a paging filter, like the `more` program.) The act of defining DEBUG will enable a section of code that displays the assigned columns on a per-row basis. If the benchmark is working properly, the first 25 numbers to be displayed should be:

37 58 95 99 100 66 9 52 4 65 43 23 16 19 62 13 77 10 11 95 4 64 2 76 78

These are the column choices for each row made by the algorithm. (For example, row 0 selects column 37, row 1 selects column 58, etc.) Odds are extremely good that, if you see these numbers displayed, the algorithm is working correctly.

References

Quantitative Decision Making for Business, Gordon, Pressman, and Cohn, Prentice-Hall, Englewood Cliffs, NJ, 1990.

Quantitative Decision Making, Guiseppi A. Forgionne, Wadsworth Publishing Co., California, 1986.

Huffman Compression

Description

This is a compression algorithm that -- while helpful for some time as a text compression technique -- has since fallen out of fashion on account of the supe rior performance by algorithms such as LZW compression. It is, however, still used in some graphics file formats in one form or another.

The benchmark consists of three parts:

Building a "Huffman Tree" (explained below),

Compression, and

Decompression.

A "Huffman Tree" is a special data structure that guides the compression and decompression processes. If you were to diagram one, it would look like a large binary tree (i.e., two branches per each node). Describing its function in detail is beyond the scope of this paper (see the references for more information). We should, however, point out that the tree is built from the "bottom up"; and the procedure for constructing it requires that the algorithm scan the uncompressed buffer, building a frequency table for all the characters appearing in the buffer. (This version of the Huffman algorithm compresses byte-at-a-time, though there's no reason why the same principle could not be applied to tokens larger than one byte.)

Once the tree is built, text compression is relatively straightforward. The algorithm fetches a character from the uncompressed buffer, navigates the tree based on the character's value, and produces a bit stream that is concatenated to the compressed buffer. Decompression is the reverse of that process. (We recognize that we are simplifying the algorithm. Again, we recommend you check the references.)

The Huffman Compression benchmark considers an iteration to be the three operations described above, performed on an uncompressed text buffer of 5000 bytes. It reports its results in iterations per second.

Analysis

Optimized 486 code (Watcom C/C++ 10.0) : The Huffman compression algorithm -- tree building, compression, and decompression -- is written as a single, large routine: `DoHuffIteration()` . All the benchmark's time is spent within that routine.

Components of `DoHuffIteration()` that consume the most time are those that perform the compression and decompression .

The code for performing the compression spends most of its time (accounting for about 13%) constructing the bit string for a character that is being compressed. It does this by seeking up the tree from a leaf, emitting 1's and 0's in the process, until it reaches the root. The stream of 1's and 0's are loaded into a character array; the algorithm then walks "backward" through the array, setting (or clearing) bits in the compression buffer as it goes.

Similarly, the decompression portion takes about 12% of the time as the algorithm pulls bits out of the compressed buffer -- using them to navigate the Huffman tree -- and reconstructs the original text.

680x0 Code (Macintosh CodeWarrior): CodeWarrior's profiler is function based. Consequently, it's impossible to get performance scores for individual machine instructions. Furthermore, as mentioned above, the Huffman compre ssion algorithm is written as a monolithic routine. This makes the results from the CodeWarrior profiler all the more sparse.

We can at least point out that the lowmost routines (`GetCompBit()` and `SetCompBit()`) that read and write individual bits, though called nearly 13 million times each, account for only 0.7% and 0.3% of the total time, respectively.

Porting Considerations

The Huffman algorithm relies on no special data types. It should port readily. Global constants of interest include:

EXCLUDED - This is a large, positive value. Currently it is set to 32000, and should be left alone. Basically, this is a token that the system uses to indicate an excluded character (one that does not appear in the plaintext). It is set to a ridiculously high value that will never appear in the pointers of the tree during normal construction.

MAXHUFFLOOPS - This is another one of those "governor" constants. The Huffman benchmark creates m ore work for itself by doing multiple compression/decompression loops. This constant sets the maximum number of loops it will attempt per iteration before it gives up. Currently, it is set to 50000. Though it is unlikely you'll ever need to modify this value, you can increase it if your machine is too fast for the adjustment algorithm. Do not reduce the number.

HUFFARRAYSIZE - This value sets the size of the plaintext array to be compressed. You can override this value with the command file to see how well your machine performs for larger or smaller arrays. The subsequent results, however, are invalid for comparison with other systems.

To test for correct execution of the benchmark : #define the symbol DEBUG, recompile, build a command file that executes only the Huffman compression algorithm, and run the benchmark. Defining DEBUG will enable a section of code that verifies the decompression as it takes place (i.e., the routine compares -- character at a time -- the un compressed data with the original plaintext). If there's an error, the program will repeatedly display: "Error at textoffset xxx".

References

Data Compression: Methods and Theory, James A. Storer, Computer Science Press, Rockville, MD, 1988.

An Introduction to Text Processing, Peter D. Smith, MIT Press, Cambridge, MA, 1990.

IDEA Encryption

Description

This is another benchmark based on a "higher-level" algorithm; "higher -level" in the sense that it is more complex than a sort or a search operation.

Security -- and, therefore, cryptography -- are becoming increasingly important issues in the computer realm. It's likely that more and more machines will be running routines like the IDEA encryption algorithm. (IDEA is an acronym for the International Data Encryption Algorithm.)

A good description of the algorithm (and, in fact, the reference we used to create the source code for the test) can be fou nd in Bruce Schneier's exhaustive exploration of encryption, "Applied Cryptography" (see references). To quote Mr. Schneier: "In my opinion, it [IDEA] is the best and most secure block algorithm available to the public at this time."

IDEA is a symmetrical, block cypher algorithm. Symmetrical means that the same routine used to encrypt the data also decrypts the data. A block cipher works on the plaintext (the message to be encrypted) in fixed, discrete chunks. In the case of IDEA, the algorithm encrypts and decrypts 64 bits at a time.

As pointed out in Schneier's book, there are three operations that the IDEA uses to do its work:

XOR (exclusive-or)

Addition modulo 216 (ignoring overflow)

Multiplication modulo 216+1 (ignoring overflow).

IDEA requires a key of 128 bits. However, keys and blocks are further subdivided into 16-bit chunks, so that any given operation within the IDEA encryption is performed on 16-bit quantities. (T his is one of the many advantages of the algorithm, it is efficient even on 16-bit processors.)

The IDEA benchmark considers an "iteration" to be an encryption *and* decryption of a buffer of 4000 bytes. The test actually builds 3 buffers: The first to hold the original plaintext, the second to hold the encrypted text, and the third to hold the decrypted text (the contents of which should match that of the first buffer). It reports its results in iterations per second.

Analysis

Optimized 486 code: The algorithm actually spends most of its time (nearly 75%) within the `mul()` routine, which performs the multiplication modulo 216+1. This is a super-simple routine, consisting primarily of `if` statements, shifts, and additions.

The remaining time (around 24%) is spent in the balance of the `cipher_idea()` routine. (Note that `cipher_idea()` calls the `mul()` routine frequently; so, the 24% is comprised of t he other lines of `cipher_idea()`). `cipher_idea()` is littered with simple pointer-fetch-and-increment operations, some addition, and some exclusive-or operations.

Note that IDEA's exercise of system capabilities probably doesn't extend beyond testing simple integer math operations. Since the buffer size is set to 4000 bytes, the test will run entirely in processor cache on most systems. Even the cache won't get a heavy "internal" workout, since the algorithm proceeds sequentially through each buffer from lower to higher addresses.

680x0 code (Macintosh CodeWarrior): CodeWarrior's profiler is function based; consequently, it is impossible to determine execution profiles for individual machine instructions. We can, however, get an idea of how much time is spent in each routine.

As with Huffman compression, the IDEA algorithm is written monolithically -- a single, large routine does most of the work. However, a special multiplication routine , `mul()` , is frequently called within each encryption/decription iteration (see above).

In this instance, the results for the 68K system diverges widely from those of the 486 system. The CodeWarrior profiler shows the `mul()` routine as taking only 4% of the total time in the benchmark, even though it is called over 20 million times. The outer routine is called 600,000 times, and accounts for about 96% of the whole program's entire time.

Porting Considerations

Since IDEA does its work in 16-bit units, it is particularly important that `u16` be defined to whatever datatype provides an unsigned 16-bit integer on the test platform. Usually, `unsigned short` works for this. (You can verify the size of a short by running the benchmarks with a command file that includes ALLSTATS=T as one of the commands. This will cause the benchmark program to display a message that tells the size of the int, short, and long datatypes in bytes.)

Also, the `mul()` routine in IDEA requires the `u32` datatype to define an

unsigned 32-bit integer. In most cases, `unsigned long` works.

To test for correct execution of the benchmark : #define the symbol DEBUG, recompile, build a command file that executes only the IDEA algorithm, and run the benchmark. Defining DEBUG will enable a section of code that compares the original plaintext with the output of the test. (Remember, the benchmark performs both encryption and decryption.) If the algorithm has failed, the output will not match the input, and you'll see "IDEA Error" messages all over your display.

References

Applied Cryptography: Protocols, Algorithms, and Source Code in C, Bruce Schneier, John Wiley & Sons, Inc., New York, 1994.

Neural Net

Description

The Neural Net simulation benchmark is based on a simple back-propagation neural network presented by Maureen Caudill as part of a BYTE article that appeared in the October, 1991 issue (see "Expert Networks" in that issue). The network involved is a simple 3-layer (input neurodes, middle-layer neurodes, and output neurodes) network that accepts a number of 5 x 7 input patterns and produce a single 8-bit output pattern.

The test involves sending the network an input pattern that is the 5 x 7 "image" of a character (1's and 0's -- 1's representing lit pixels, 0's representing unlit pixels), and teaching it the 8-bit ASCII code for the character.

A thorough description of how the back propagation algorithm works is beyond the scope of this paper. We recommend you search through the references given at the end of this paper, particularly Ms. Caudill's article, for detailed discussion. In brief, the benchmark is primarily an exercise in floating-point operations, with some frequent use of the `exp()` function. It also performs a great deal of array references, though the arrays in use are well und er 300 elements each (and less than 100 in most cases).

The Neural Net benchmark considers an iteration to be a single learning cycle. (A "learning cycle" is defined as the time it takes the network to be able to associate all input patterns to the correct output patterns within a specified tolerance.) It reports its results in iterations per second.

Analysis

Optimized 486 code : The forward pass of the network (i.e., calculating outputs from inputs) utilize a sigmoid function. This function has, at its heart, a call to the `exp()` library routine. A small but non-negligible amount of time is spent in that function (a little over 5% for the 486 system we tested).

The learning portion of the network benchmark depends on the derivative of the sigmoid function, which turns out to require only multiplications and subtractions. Consequently, each learning pass exercises only simple floating-point operations.

If we divide the time spen t in the test into two parts -- forward pass and backward pass (the latter being the learning pass) -- then the test appears to spend the greatest part of its time in the learning phase. In fact, most time is spent in the `adjust_mid_wts()` routine. This is the part of the routine that alters the weights on the middle layer neurodes. (It accounts for over 40% of the benchmark's time.)

680x0 Code (Macintosh CodeWarrior): Though CodeWarrior's profiler is function based, the neural net benchmark is highly modular. We can therefore get a good breakdown of routine usage:

`worst_pass_error()` - 304 microsecs (called 4680 times)

`adjust_mid_wts()` - 83277 microsecs (called 46800 times)

`adjust_out_wts()` - 17394 microsecs (called 46800 times)

`do_mid_error()` - 11512 microsecs (called 46800 times)

`do_out_error()` - 3002 microsecs (called 46800 times)

`do_mid_forward()` - 49559 microsecs (called 46800 times)

`do_out_forward()` - 20634 microsecs (called 46800 times)

Again, most time was spent in `adjust_mid_wts()` (as on the 486), accounting for almost twice as much time as `do_mid_forward()`.

Porting Consideration

The Neural Net benchmark is not dependent on any special data types. There are a number of global variables and arrays that should not be altered in any way. Most importantly, the #defines found in NBENCH1.H under the Neural Net section should not be changed. These control not only the number of neurodes in each layer; they also include constants that govern the learning processes.

Other globals to be aware of:

MAXNNETLOOPS - This constant simply sets the upper limit on the number of training loops the test will permit per iteration. The Neural Net benchmark adjusts its workload by re-teaching itself over and over (each time it begins a new training session, the network is "cleared" -- loaded with random va lues). It is unlikely you will ever need to modify this constant.

`inpath` - This string pointer is set to the path from which the neural net's input data is read. It is currently hardwired to "NNET.DAT". You shouldn't have to change this name, unless your filesystem requires directory information as part of the path.

Note that the Neural Net benchmark is the only test that requires an external data file. The contents of the file are listed in an attachment to this paper. You should use the attachment to reconstruct the file should it become lost or corrupted. Any changes to the file will invalidate the test results.

To test for correct execution of the benchmark : #define the symbol DEBUG, recompile, build a command file that executes only the Neural Net test, and run the benchmark. Defining DEBUG will enable a section of code that displays how many passes through the learning process were required for the net to learn. It should learn in 780 passes.

References

"Expert Networks," Maureen Caudill, BYTE Magazine, October, 1991.

Simulating Neural Networks, Norbert Hoffmann, Verlag Vieweg, Wiesbaden, 1994.

Signal and Image Processing with Neural Networks, Timothy Masters, John Wiley and Sons, New York, 1994.

Introduction to Neural Networks, Jeannette Stanley, California Scientific Software, CA, 1989.

LU Decomposition

Description

LU Decomposition is an algorithm that can be used as the heart of a program for solving linear equations. Suppose you have a matrix **A** . LU Decomposition determines the matrices **L** and **U** such that

L . U = A

where **L** is a lower triangular matrix and **U** is an upper triangular matrix. (A lower triangular matrix has nonzero elements only on the main diagonal and below. An upper triangular matrix has nonzero elements only on the main diagonal and above.)

Without going into the mathe matical details too deeply, having the **L** and **U** matrices makes the solution of linear equations (i.e., equations of the form **A** . **x** = **b**) quite easy. It turns out that you can also use LU decomposition to determine matrix inverses and determinants.

The algorithm used in the benchmarks was derived from *Numerical Recipes in Pascal* (there is a C version of the book, which we did not have on hand), a book we heartily recommend to anyone serious about mathematical and scientific computing. The authors are approving of LU decomposition as a means of solving linear equations, pointing out that their version (which makes use of what we would have to call "Crout's method with partial implicit pivoting") is a factor of 3 better than one of their Gauss-Jordan routines, a factor of 1.5 better than another. They go on to demonstrate the use of LU decomposition for iterative improvement of linear equation solutions.

The benchmark begins by crea ting a "solvable" linear system. This is easily done by loading up the column vector **b** with random integers, then initializing **A** with an identity matrix. The equations are then "scrambled" by either multiplying a row by a constant, or adding one row to another. The scrambled matrices are handed to the LU algorithm.

The LU Decomposition benchmark considers a single iteration to be the solution of one set of equations (the size of **A** is fixed at 101 x 101 elements). It reports its results in iterations per second.

Analysis

Optimized 486 code (Watcom C/C++ 10.0) : The entire algorithm consists of two parts: the LU decomposition itself, and the back substitution algorithm that builds the solution vector. The majority of the algorithm's time takes place within the former; the algorithm that builds the **L** and **U** matrices (this takes place in routine `ludcmp()`).

Within `ludcmp()` , there are tw o extremely tight for loops forming the heart of Crout's algorithm that consume the majority of the time. The loops are "tight" in that they each consist of only one line of code; in both cases, the line of code is a "multiply and accumulate" operation (actually, it's sort of a multiply and de-accumulate, since the result of the multiplication is subtracted, not added).

In both cases, the items multiplied are elements from the A array; and one factor's row index is varying more rapidly, while another factor's column index is varying more rapidly.

Note that this is a good overall test of floating-point operations within matrices. Most of the math is floating-point; primarily additions, subtractions, and multiplications (only a few divisions).

680x0 Code (Macintosh CodeWarrior): CodeWarrior's profiler is function based. It is therefore impossible to determine execution profiles at the machine-code level. The profiler does, however, allow us to det ermine how much time the benchmark spends in each routine. This breakdown is as follows:

`lusolve()` - 3.4 microsecs (about 0% of the time)

`lubksb()` 1198 microsec (about 2% of the time)

`ludcmp()` - 63171 microsec (about 91% of the time)

The above percentages are for the whole program. Consequently, as a portion of actual benchmark time, the amount attributed to each will be slightly larger (though the proportions will remain the same).

Since `ludcmp()` performs the actual LU decomposition, this is exactly where we'd want the benchmark to spend its time. The `lubksb()` routine calls `ludcmp()` , using the resulting matrix to "back-solve" the linear equation.

Porting Considerations

The LU Decomposition routine requires no special data types, and is immune to byte ordering. It does make use of a typedef (`LUdblptr`) that includes an embedded union; this allows the benchmark to "co erce" a pointer to double into a pointer to a 2D array of double. This arrangement has not caused problems with the compilers we have tested to date.

Other constants and globals to be aware of:

LUARRAYROWS and LUARRAYCOLS - These constants set the size of the coefficient matrix, A. They cannot be altered by command file. In fact, you shouldn't alter them at all, or your results will be invalid. Currently, they are both set to 101.

MAXLUARRAYS - This is another "governor" constant. The algorithm performs dynamic workload adjustment by building more and more arrays to solve per timing round. This sets the maximum upper limit of arrays that it will build. Currently, it is set to 1000, which should be more than enough for the reasonable future (1000 arrays of 101 x 101 floating-point doubles would require somewhere around 80 megabytes of RAM -- and that's not counting the column vectors).

To test for correct execution of the benchmark : Currently, there is no simple technique for doing this. You can, however, either use your favorite debugger (or embed a printf() statement) at the conclusion of the `lubksb()` routine. When this routine concludes, the array `b` will hold the solution vector. These items will be stored as floating-point doubles, and the first 14 are (with rounding):

46 20 23 22 85 86 97 95 8 89 75 67 6 86

If you find these numbers as the first 14 in the array `b[]` , then you're virtually guaranteed that the algorithm is working correctly.

References

Numerical Recipes in Pascal: The Art of Scientific Computing, Press, Flannery, Teukolsky, Vetterling, Cambridge University Press, New York, 1989.

Architect, SD Expo, SD Magazine, Sys Admin, The Perl Journal, UnixReview.com, Windows Developer Network