Lies, damned lies and benchmarks

INMOS Technical Note 27

INMOS Limited

72-TCH-027-00

Contents

1 Introduction

A benchmark is supposed to be a standard measure of performance that enables one computer to be compared with another. However, a car is a simpler machine than a computer, and yet no-one expects all the relevant features of a car to be contained in a single number. Even in the specialised world of motor-racing, knowing the b.h.p. or the top speed is not enough to predict which car will be fastest round the track, and computing equivalents such as 'MIPS' or 'MFlops' are similarly misleading.

For any application it is performance on that application which counts, and benchmarks are relevant only so far as they resemble it. For example, some microprocessors can match the speed of super-minicomputers on non-numerical benchmarks, although their floating-point performance and input-output capability can be substantially inferior. Also, microprocessor architectures tend to give atypically high performance on small programs, by making good use of small register sets, caches, on-chip memory etc., and nearly all benchmark programs are very small in order to be easily disseminated.

Ideally, computers should be compared by running the intended application on each of them, but usually this is impractical, and benchmarks are often used instead. Some benchmarks have been carefully constructed and, in context, they can be a good guide to processor performance, provided their limitations are clearly understood. The Whetstone benchmark is one such, and is widely used as an indicator of performance on numerical tasks, although it omits some aspects of such applications, which we consider separately. The Savage benchmark tests only a narrow aspect of performance, but is often included in sets of benchmarks, so we consider it briefly. On the other hand, there are benchmarks which are badly constructed and cannot be related to any real application. An example is the Dhrystone benchmark, which, regrettably, is also widely used as a vague measure of processor power.

It is important to realise that all of these benchmarks are intended as tests for single-processor machines. None of them are particularly suited to parallelism; but then none of them are real application programs! Real programs are generally used to process data of some kind, and very often different parts of the data can be dealt with independently, allowing for large performance gains when several processors are used. Applications designed with parallelism in mind can often also be split into parts which can perform successive operations on the same flow of data in parallel, using a pipeline or other structure, allowing still more processors to be used effectively.

It is likely that the wide variety of possible architectures for parallel machines will render benchmarking impractical. Until that time we must live with

benchmarks, so in this note we look at these three: the Whetstone, the Savage and the Dhrystone. We consider their merits and limitations, and provide performance figures and source listings.

2 The Whetstone benchmark

The Whetstone benchmark program [1] was constructed to compare processor power for scientific applications. Running the program is considered equivalent to executing (approximately) one million 'Whetstone' instructions. Performance, as measured by the benchmark, is quoted in 'Whetstones per second' and differs from any measure of pure floating-point performance given in 'flops'. In addition to floating-point operations, it includes integer arithmetic, array indexing, procedure calls, conditional jumps, and elementary function evaluations. These are mixed in proportions carefully chosen to simulate a 'typical' scientific application program of a decade ago.

2.1 Understanding the program

The virtue of the Whetstone benchmark is that it approaches real programs in complexity, whereas many other benchmarks only measure performance on simple loops. For example, a large part of the 'Linpack' benchmark effectively measures only the time to perform a loop of the form:

```
SEQ i = 0 FOR N
  a[i] := b[i]+(t*c[i])
```

However, this complexity means that in order to relate the resulting performance figures to a real application, it is necessary to consider the precise composition of the benchmark. The occam source of the Whetstone is given in section B.1. This is a straightforward translation of the ALGOL original, which consists of a series of modules designed to typify different aspects of a scientific computation. The core of each module is performed a certain number of times, determined by a 'best fit' to statistics of actual programs.

The time taken to execute a particular module may depend more on the speed of floating-point operations than on the specific task it represents. For example, module 2 is concerned with 'array accessing', but for each iteration of the loop there are 20 array accesses and 17 floating-point operations. On machines where the duration of a floating-point operation is much longer than the time taken to load or store a number, the floating-point operations will dominate the time to perform the module. This is also true of other modules. So the overall Whetstone performance will be largely determined

by the floating-point speed of such machines. It will also depend on the speed of evaluation of elementary functions, because of the large number of such evaluations in modules 7 and 11. This is an area where applications vary widely, and the Whetstone represents an average which may be very different from any particular application.

2.2 The effect of optimisations

Since the benchmark is written in a high-level language (originally ALGOL; commonly FORTRAN; and in this case occam) it must be compiled before it can be executed. This makes the interpretation of the results more difficult since they depend not only on the hardware but also on the software which is used. As compilers become more sophisticated there is a danger that the original purpose of the benchmark will be lost in all the optimisations that can be done. The purpose of the benchmark is to cause the execution of (typically) one million 'Whetstone' instructions, which represent low-level operations of an abstract machine, and not to get through a particular FORTRAN program as fast as possible. Thus 'global' or source-level optimisations (either automatic or by hand) invalidate the benchmark since they miss out some of the 'Whetstone' instructions. Indeed, since no-one is interested in the results of the computations they could be optimised out altogether! By contrast the choice of high-level language to express the benchmark is relatively insignificant, provided its semantics are not too different from those of FORTRAN or ALGOL.

The occam compilers used to benchmark transputers aim to produce efficient code, but do not perform global or source-level optimisations. Consequently all the 'Whetstone' instructions implicit in the original program are performed.

2.3 Limitations of the Whetstone

It is important to realise that significant aspects of many contemporary scientific calculations are absent from the Whetstone, whilst others are over-emphasised:

1. No consideration is given to the quality of floating-point calculations, and their speed is measured only indirectly.

2. There are no multi-dimensional arrays, which are common in numerical programs, and the arrays which are present are very small.

3. The number of elementary function evaluations is probably atypical of

modern programs, and despite this heavy usage no account is taken of their accuracy.

We examine these points in the following sections.

Floating-point operations on the IMS T414

The floating-point operations provided for the IMS T414 are both fast and of high quality. Although the IMS T414 was designed to provide fast arithmetic operations on 32-bit integer values, it was appreciated that for many applications it would be necessary to perform floating-point arithmetic and so there are special instructions in the IMS T414 to support the implementation of floating-point operations in software.

The use of formal program proving methods has ensured that the quality of the software implementation is very high [2]. The software packages correctly implement IEEE-standard floating-point arithmetic, including the handling of denormalised numbers.

Although implemented in software, floating-point operations on the IMS T414 are very fast, comparable with those performed by special floating-point co-processor chips. For example, the assignment in the occam fragment below:

```
REAL32 a, b, c :
SEQ
  ...
  a := b * c
  ...
```

will execute in about 11 μS, provided all the code and variables are in internal RAM. By comparison, the, same assignment on an 8 Mhz Intel 80286/80287 combination would take about 31 μS (using the fastest possible memory). Even on 64-bit floating-point numbers, where it might be expected that software would lose out against hardware, the IMS T414 would take about 38 μS whilst the Intel combination would take about 44 μS.

Floating-point operations on the IMS T800

To achieve even higher performance than the IMS T414, the IMS T800 has a 64-bit floating-point unit on-chip. Its microcode was derived from the formally-proven occam implementation, so that the results of floating-point calculations by the two processors are identical (and correct) - only the speed differs. On an IMS T800 the assignment above would take only 29 cycles (1.45 μS for a 20MHz version, 0.97 μS for a 30MHz version), again assuming internal RAM is used.

The table below gives the typical and worst case operation times for floating point arithmetic on an IMS 414, (50 nS cycle time) and on an IMS T800 (50 nS and 33 nS cycle times). For the IMS T414 this assumes the, code of the floating-point package is in the internal RAM.

	IMS T414-20		**IMS T800-20**		**IMS T800-30**	
	Typical	Worst	Typical	Worst	Typical	Worst
REAL32						
$+, -$	11.5 μS	15.0 μS	350 nS	450 nS	230 nS	300 nS
$*$	10.0 μS	12.0 μS	550 nS	900 nS	370 nS	600 nS
/	11.3 μS	14.0 μS	800 nS	1400 nS	530 nS	930 nS
REAL64						
$+, -$	28.2 μS	35.0 μS	350 nS	450 nS	230 nS	300 nS
$*$	38.0 μS	47.0 μS	1000 nS	1350 nS	670 nS	900 nS
/	55.8 μS	71.0 μS	1550 nS	2150 nS	1030 nS	1430 nS

Table 1: Floating-point operation times

Multi-dimensional arrays

Although not represented in the Whetstone benchmark, multi-dimensional arrays are common in many numerical applications. The IMS T414 and IMS T800 have a fast multiplication instruction ('product') which is used for the multiplication implicit in multidimensional array access. For example, in the following fragment of occam:

```
[20][20]REAL32 A :
SEQ
  ...
  B := A[I][J]
  ...
```

performing the assignment involves calculating the offset of element A[I][J] from the base of the array A.

The transputer compiler would generate the following code for this computation:

```
load local       I
load constant   20
product
load local       J
add
```

Since the product instruction executes in a time dependent on the highest bit set in its second operand, and the highest bit set in the constant 20 is

bit 5, in this case the 'product' instruction will execute in only 8 cycles. In general, the multiplication in an address calculation is performed in a time approximately proportional to the logarithm of the array dimension. When combined with the concurrent operation of the CPU and FPU on the IMS T800 this enables address calculations to be entirely overlapped with floating-point calculations in most cases.

Elementary functions on the IMS T414 and IMS T800

The implementation of elementary functions involves a trade-off between speed, accuracy, and code-size. Whilst total accuracy is mathematically impossible, errors must be kept within reasonable bounds or else the functions are useless. The need to constrain code-size precludes the use of certain very fast algorithms which make use of very large look-up tables and linear interpolation.

The elementary function libraries used on the INMOS transputers are written in occam. They use rational approximations (quotients of polynomials), rather than table look-up or 'CORDIC' methods, as this gives the fastest execution whilst remaining accurate and code-compact. The single-length functions typically require a few hundred bytes of code (approximately 400 on the IMS T414 and 300 on the IMS T800), and have average errors of less than half a unit in the last bit. The functions handle all IEEE-standard values, including denormalised numbers, Not-a-Numbers, and Infinities. Further details are given in [3] and [4].

On the IMS T414 the rational approximations are computed using fixed-point arithmetic rather than floatingpoint. The IMS T414 has a 'fractional multiply' instruction which multiplies two 32-bit numbers together, treating each as a fraction between +1 and -1; the normal 'add' instruction will add such fractions. As a result of this the multiply and add, needed in each stage of a polynomial evaluation, will execute in under 3.5 μS; if floating-point arithmetic were used these operations would take about seven times as long.

However the performance of the IMS T800 FPU is such that the multiply and add stage of a floating-point polynomial takes only 0.9 μS, so the library for this processor evaluates the rational approximations using floating-point arithmetic. Of course this library may be used on the IMS T414, producing identical results to those which would be obtained on an IMS T800, because of the equivalence of the floating-point software and hardware.

The importance of the speed of elementary function evaluation to the overall Whetstone performance figure is indicated by the proportion of time spent evaluating them, as indicated in the following table:

Processor :	**IMS T414**		**IMS T800**	
Floating-point format :	Single	Double	Single	Double
Trigonometric functions	26%	34%	23%	29%
Standard functions	13%	17%	21%	23%
Total	39%	51%	44%	52%

Table 2: Percentage of total execution time

These percentages would probably be lower on a processor with special hardware for speeding up elementary function evaluation. Neither the IMS T414 nor the IMS T800 have any such special hardware, since including it would have compromised some other aspect of performance, so the speed and accuracy of elementary function evaluation is a good test of these processors. This is considered more fully in the next section, and timings for the individual functions are given in section C.

3 The Savage Benchmark

3.1 Speed and accuracy of elementary functions

The Savage benchmark is a benchmark of elementary function evaluation only. It is actually named after its creator [5], although it is indeed quite a vicious test of an unsuspecting function library! It tests both speed and accuracy; in occam it is:

```
#USE "dblmath.lib"
REAL64 a :
SEQ
  a := 1.0(REAL64)
  time ? start.time

  SEQ i = 0 FOR 2499
    a := DTAN(DATAN(DEXP(DALOG(DSQRT(a*a))))) + 1.0 (REAL64)

  time ? finish.time
```

If the function subroutines were exact the final value of a would be 2500.0, so the difference from this figure is a measure of their accuracy. However it is important to note that the format (in this case IEEE double-precision) enforces a fundamental limitation no matter how carefully the functions are evaluated. The minimum error that can be achieved using double-precision

floating-point is $1.177 * 10^{-9}$, and it can be seen from the table in section A.2 that the occam function library produces a result which is very close to this figure. Some implementations give results more accurate than this, by using 'extended double precision' (80 bits) to evaluate the expression, only rounding to double-precision when the store into a is done.

Some results from this benchmark are given in section A.2. It is certainly not typical of application programs, but it does give some indication of performance on elementary function evaluation only.

4 The Dhrystone benchmark

4.1 String manipulation performance

The Dhrystone [6] is a synthetic benchmark designed to test processor performance on 'systems programs'. In fact it has a number of flaws which seriously limit its usefulness as a guide to performance on 'typical' programs. Unfortunately its use has become widespread, with results published on the USENET, and manufacturers reporting their performance in terms of 'Dhrystones per second'. It was originally published in Ada, but the most widely used version is a translation into C, distributed over USENET.

As the construction of the Dhrystone is fully explained in the original publication, our discussion of the benchmark is limited to its drawbacks. The two principal flaws are the omission of any significant looping from the program and the inclusion of character string operations.

Whilst the Dhrystone's major advantage over many small benchmarks is that is does not consist of just a single loop, it suffers from the drawback that it does not do any significant amount of looping. This is unsound because most programs do contain loops and code executed within them will often account for most of the execution time. Also, when generating code for loops, a good compiler will seek to minimise the time to execute the loop repeatedly, possibly at the expense of more loop initialisation. Furthermore, research shows [7] that the code found within loops differ from code outside of loops; for example, most accesses to subscripted variables occur within loops.

The second major drawback of the Dhrystone that it uses strings, even though the only dynamic statistics in [6] show no use of strings (although the static statistics from the same source do show use of strings). In addition, the use of strings causes a large number of other problems with the benchmark. There are too many to consider in detail, so we will just look at the most significant.

The first problem comes from the method of construction of the benchmark, which was to ensure that the distribution of operators and operands matched that found in 'typical' programs. Unfortunately, the operators and operands seem to have been treated independently, and as a result, the statement

```
if String_Par_In_1 > String_Par_In_2
```

occurs in the Ada original. This may look inoffensive but when a translation into, for example, C occurs the result is

```
if (strcmp( StrParI1, StrParI2) > 0)
```

which involves a very suspicious looking call to a library routine. As very little computation is performed in the benchmark this may be very significant. The amount of time taken to perform the comparison will, in fact, depend on the two strings being compared. In the Dhrystone the strings used are:

```
"DHRYSTONE PROGRAM, 2'ND STRING"
```

and

```
"DHRYSTONE PROGRAM, 1'ST STRING"
```

which match for the first 19 characters! The overall result of this is that, with a straightforward implementation of strcmp the only loop of any significance has been introduced by accident rather than by design.

The second problem is that the program contains a string assignment, which also becomes more blatant when the program is translated. In the Dhrystone as originally published, written in Ada, the strings in the program were declared to be 30 characters long. This means that a processor with the ability to copy data in blocks would be able to do the assignment very efficiently. When the translation to C takes place the translator has to make a choice; either the strings are converted into C strings, or they are changed into a structure. The former is more natural whilst the latter is more in keeping with the original program. The effect of this is, again, that a seemingly small part of the benchmark contributes significantly to the overall result.

One final point that should be noted is that the Dhrystone program, although intended to represent a typical 'system program', is actually extremely small, which again may make the results misleading.

The best known version of the Dhrystone benchmark is that in C, distributed on the USENET. It is a fair translation of the Ada except that it uses C-strings rather than fixed-sized byte arrays. The consequences of this alteration have already been discussed.

For some time an erroneous version of the Dhrystone was circulated on the USENET. When making comparisons of performance it is essential to check that the Dhrystone figure is for the correct Version of the benchmark, known as version 1.1 by the USENET community. Figures for his erroneous version would be substantially higher than figures for the correct version. In particular the figures given in [8] are for the erroneous version.

The occam version attempts to be as close to the Ada as possible. There are some problems with this which were tackled as follows. The first difficulty is that the Ada Dhrystone uses structures, which occam does not support. The occam Dhrystone simulates structures using arrays, with the byte array (string) being 'punned' onto several words of the array. The second problem is that occam does not provide dynamic storage allocation which is used for allocating the structures. The occam Dhrystone uses an array of structures instead (this is of no significance to performance as the allocation of the structure is not timed as part of the benchmark). There are some other minor changes which have been necessitated such as re-ordering the declaration of procedures as in occam they must be declared before they are used.

The source of the occam version of the Dhrystone benchmark is given in section B.2.

5 Conclusion

The Whetstone benchmark is one of the most respected and widely used measures of performance on 'scientific' applications, even though it does not address important aspects of such computations, and overemphasises others. The IMS T414 and IMS T800 microprocessors are very well suited to such applications, and this is reflected in their Whetstone performance, shown in section A.1.

The Savage benchmark only measures performance on elementary functions, but is quite widely used in the microcomputing world. Although Transputers have no special hardware for elementary functions, in order to maximise performance on more common operations [4], they perform extremely well, as can be seen from the results in section A.2.

Thus the IMS T414 surpasses all other single-chip processors in performing numerical calculations with software, and outperforms many processor /co-

processor combinations. The IMS T800 is the world's fastest microprocessor, superior even to multi-chip sets and bit-slice machines.

The Dhrystone is also widely used, even though it is essentially useless as an indicator of performance on real programs. The table in section A.3 shows that Transputers give a high figure on this benchmark, but this is of relatively little significance. It is interesting to note that at least one recent 32-bit microprocessor has special hardware for processing strings; not surprisingly its projected Dhrystone figure is extremely high. However only programs that only process strings are likely to realise this promised performance. Transputers have not been optimised to 'pass' a particular benchmark; they are general-purpose processors delivering high performance on all applications.

References

[1] A Synthetic Benchmark, Curnow H.J., and Wichmann B.A., Computer Journal 19 no. 1, February 1976.

[2] Formal Methods Applied to a Floating Point Number System, Barrett G., Oxford University Computing Laboratory Technical Monograph PRG-58 1987.

[3] Transputer Development System Manual, INMOS Limited, Prentice Hall 1988.

[4] Technical Note 6: IMS T800 Architecture, INMOS Limited, Bristol, U.K. INMOS 1986.

[5] Dr. Dobb's Journal, Savage B., September 1983, p120.

[6] Dhrystone: a synthetic systems programming benchmark, Reinhold P. Weicker, Communications of the ACM, Vol. 27, Number 10, October 1984.

[7] An Empirical Analysis of FORTRAN programs, Robinson and Torsun, Computer Journal 19 no. 1, February 1976.

[8] The 80386: A High Performance Workstation Microprocessor, Intel Corporation, 1986, Order number: 231776-001.

A Comparative benchmark results

A.1 Whetstone

The following tables compare the performance figures of the transputers with other processors and processor /co-processor combinations for both the single and double precision Whetstone benchmarks. Some of the figures may have been superseded since these tables were compiled, but they are adequate for illustrative purposes.

System	**Thousands of Single-precision Whetstones per Second**
IMS T800-30 (projected)	6800
IMS T800-20	4548
WE 32200/32206-24	2800
INTEL 80386 + 80387	1860
VAX 11/780	1083
MVII	925
SUN-3	860
NS 32332/32081	728
IMS T414-20	704
NS 32032 and 32081	390
INTEL 286/287	300
IBM RT-PC + FPA	200
IMS T212-20	181
INTEL 8086 + 8087	178
MC 68000	13
IBM RT-PC	12

System	**Thousands of Double-precision Whetstones per Second**
IMS T800-30 (projected)	4400
IMS T800-20	2932
INTEL 80386 + 80387	1730
MVII	925
SUN-3	790
VAX 11/780	715
IMS T414-20	161
INTEL 8086 + 8087	152

The figures for the IMS T414-20 were obtained by running the program on an IMS T414B-20 (50 nS cycle time), with 150 nS cycle time external memory. Note that running the program on a slower system, such as are provided by INMOS for hosting the development system, will give a lower figure. The

IBM RT-PC	software only
IBM RT-PC + FPA	with NS32081 floating-point chip, in 'direct mode'
IMS T212-20	20 MHz, using product occam compiler
IMS T414-20	20 MHz, using product occam compiler
IMS T800-20	20 MHz, using product occam compiler
IMS T800-30	30 MHz, scaled from -20 result
INTEL 8086 + 8087	8 MHz
INTEL 286/287	10 MHz
INTEL 386/387	20 MHz
MC 68000	10 MHz, assembler coded software floating-point
MVII	MicroVAX II with FPA, running MicroVMS
NS 32032 and 32081	10 MHz
NS 32332 and 32081	15 MHz
SUN-3	MC 68020 (16 MHz) and MC 68881 (12.5 MHz)
WE 32200/32206-24	24 MHz
VAX 11/780	8MB memory, FPA, running under UNIX 4.3BSD

Table 3: Systems used for the benchmarks

figures for the IMS T800-20 were obtained by running the program on an IMS T8000-20 (50 nS cycle time). Figures for the faster version (30 MHz) were then obtained by straightforward scaling.

The figure for the IMS T212-20 was obtained by running the program on an IMS T212-20 (50 nS cycle time), with 100 nS cycle time external memory, using the technique of section D.

Our sources for the other figures are as follows:

IBM RT-PC	IBM FIT Personal Computer Technology, SA 23-1057, IBM 1986
INTEL 8086 + 8087	Sun-3 Benchmarks (Sun Microsystems, inc)
INTEL 286/287	Sun benchmark document
INTEL 386/387	Doug Rick, 80387 Marketing Manager
MC 68000	Published figure
MVII	Sun Benchmark document
NS 32032 and 32081	Ray Curry, National Semiconductor, via USENET
NS 32332 and 32081	Ray Curry, National Semiconductor, via USENET
SUN-3	Sun published data
WE 32200/32206-24	Electronics, December 18, 1986
VAX 11/780	John Mashey at MIPS Computer Systems, via USENET

A.2 Savage

System	CPU,FPP	MHz	Language	Time	Error
IMS T800		30.0	Occam	0.3	1.2E-9
IMS T800		20.0	Occam	0.4	1.2E-9
Sun-3/160	68020,68881	16.7	Sun 3.0 F77	0.4	2.0E-12
HP 9000/320	68020,68881		Pascal	0.7	2.8E-7
VAX 8600			Fortran 77	0.9	1.8E-8
DMS	8086,8087		Turbo Pascal	3.8	1.1E-9
Zenith Z-248	80286,80287	8.0	Fortran 77	4.5	1.2E-9
IMS T414		20.0	Occam	6.3	1.2E-9
IBM PC-AT	80286,80287	6.0	Turbo Pascal	7.4	1.2E-9
Sun-3/160	68020	16.7	Sun 3.0 F77	21.5	3.1E-7
IMS T212		20.0	Occam	21.9	1.2E-9
Turbo-Amiga	68020	14.3	Absoft F77V2.2B	21.9	1.8E-7

Information in this table (except for the Transputer figures) was supplied on USENET on 16th December 1986 by Al Alburto et al. The Transputer figures were obtained using the product occam compiler and libraries. The time for the IMS T800-30 was obtained by scaling the -20 result.

A.3 Dhrystone

The following tables compare the performance of INMOS Transputers with other processors. The figure for the IMS T414 was obtained from an IMS B001 evaluation board, running an IMS T414B-20 with 3 cycle external memory. Note that running the program on a slower system, such as are provided by INMOS for hosting the development system, will give a lower figure. The other transputer figures were obtained by running the program on INMOS TRAMs.

It should be noted that Dhrystone figures, especially those quoted by manufacturers, are often invalid. Either they refer to the incorrect version 1.0 (and if no version is given, this is usually the case) or else they use optimising compilers, which are forbidden for this benchmark (frequently both). The figures above are believed to be free of such contamination. It is regretted that no such figure is currently available for the 80386, and so an old predicted figure is given instead.

System	Dhrystones per Second
IBM 3090/200	31250
IMS T800-30 (proj.)	13400
IMS T800-20	8956
IMS T212-20	8711
IMS T414-20	8193
VAX 8600	6423
Gould PN9080 Custom ECL	4992
Intel 386-16 (predicted)	4300
MC68020-17	3977
Intel 80286-9	1976
VAX 11/780	1650
MC68000-8	1136

B Source of the occam programs

B.1 Whetstone

This is the source of the occam version of the Whetstone benchmark. The output statements have been omitted, since they complicate the benchmarking process without affecting the results in any way. However the modules which are executed zero times have been included, since their omission would be a 'global optimisation' affecting the code-size. This is the single-precision version; the double-precision version is obtained by replacing all occurrences of REAL32 by REAL64, and all the library function calls by their double-precision versions.

```
PROC Whetstone (VAL [11]INT n, VAL INT iterations, INT time0, time1)

  #USE "snglmath.lib" -- this incorporates library code for the functions
  TIMER time :
  [4] REAL32 e1 :
  INT j, k, l :
  REAL32 t, t1, t2 :

  PROC p3 (VAL REAL32 xdash, ydash, REAL32 z)
    REAL32 x, y :
    SEQ
      x := t * (xdash + ydash)
```

```
    y := t * (x + ydash)
    z := (x + y) / t2
:
PROC p0 ()
  SEQ
    e1 [j] := e1 [k]
    e1 [k] := e1 [l]
    e1 [l] := e1 [j]
:
PROC pa ([4]REAL32 e)
  SEQ j = 0 FOR 6
    SEQ
      e[0] := (((e[0]    + e[1]) + e[2]) - e[3]) * t
      e[1] := (((e[0]    + e[1]) - e[2]) + e[3]) * t
      e[2] := (((e[0]    - e[1]) + e[2]) + e[3]) * t
      e[3] := ((((-e[0]) + e[1]) + e[2]) + e[3]) / t2
:
SEQ
  -- INITIALISE CONSTANTS
  t  := 0.499975(REAL32)
  t1 := 0.50025(REAL32)
  t2 := 2.0(REAL32)

  -- RECORD START TIME
  time ? time0

  -- MODULE 1 : SIMPLE IDENTIFIERS
  REAL32 x1, x2, x3, x4 :
  SEQ
    x1 :=  1.0(REAL32)
    x2 := -1.0(REAL32)
    x3 := -1.0(REAL32)
    x4 := -1.0(REAL32)
    SEQ i = 0 FOR n[0] * iterations
      SEQ
        x1 := (((   x1 + x2) + x3) - x4) * t
        x2 := (((   x1 + x2) - x3) + x4) * t
        x3 := (((   x1 - x2) + x3) + x4) * t
        x4 := ((((-x1) + x2) + x3) + x4) * t

  -- MODULE 2 : ARRAY ELEMENTS
  SEQ
    e1 [0] :=  1.0(REAL32)
    e1 [1] := -1.0(REAL32)
    e1 [2] := -1.0(REAL32)
    e1 [3] := -1.0(REAL32)
    SEQ i = 0 FOR n[1] * iterations
      SEQ
        e1[0] := (((e1[0]    + e1[1]) + e1[2]) - e1[3]) * t
```

```
      e1[1] := (((e1[0]    + e1[1]) - e1[2]) + e1[3]) * t
      e1[2] := (((e1[0]    - e1[1]) + e1[2]) + e1[3]) * t
      e1[3] := ((((-e1[0]) + e1[1]) + e1[2]) + e1[3]) * t

-- MODULE 3 : ARRAY AS PARAMETER
SEQ i = 0 FOR n[2] * iterations
  pa (e1)

-- MODULE 4 : CONDITIONAL JUMPS
SEQ
  j := 1
  SEQ i = 0 FOR n[3] * iterations
    SEQ
      IF
        j = 1
          j := 2
        TRUE
          j := 3
      IF
        j > 2
          j := 0
        TRUE
          j := 1
      IF
        j < 1
          j := 1
        TRUE
          j := 0

-- MODULE 5 : OMITTED IN ORIGINAL

-- MODULE 6 : INTEGER ARITHMETIC
SEQ
  j := 1
  k := 2
  l := 3
  SEQ i = 0 FOR n[5] * iterations
    SEQ
      j := (j * (k - j)) * (l - k)
      k := (l * k) - ((l - j) * k)
      l := (l - k) * (k + j)
      e1 [l - 2] := REAL32 ROUND ((j + k) + l)
      e1 [k - 2] := REAL32 ROUND ((j * k) * l)

-- MODULE 7 : TRIGONOMETRIC FUNCTIONS
REAL32 x, y :
SEQ
  x := 0.5(REAL32)
  y := 0.5(REAL32)
```

```
      SEQ i = 0 FOR n[6] * iterations
        SEQ
          x := t * ATAN ( (t2 * (SIN(x)*COS(x))) /
                              ((COS(x + y) + COS(x - y)) - 1.0(REAL32)) )
          y := t * ATAN ( (t2 * (SIN(y)*COS(y))) /
                              ((COS(x + y) + COS(x - y)) - 1.0(REAL32)) )

    -- MODULE 8 : PROCEDURE CALLS
    REAL32 x, y, z :
    SEQ
      x := 1.0(REAL32)
      y := 1.0(REAL32)
      z := 1.0(REAL32)
      SEQ i = 0 FOR n[7] * iterations
        p3 (x, y, z)

    -- MODULE 9 : ARRAY REFERENCES
    SEQ
      j := 1
      k := 2
      l := 3
      e1 [0] := 1.0(REAL32)
      e1 [1] := 2.0(REAL32)
      e1 [2] := 3.0(REAL32)
      SEQ i = 0 FOR n[8] * iterations
        p0 ()

    -- MODULE 10 : INTEGER ARITHMETIC
    SEQ
      j := 2
      k := 3
      SEQ i = 0 FOR n[9] * iterations
        SEQ
          j := j + k
          k := j + k
          j := k - j
          k := (k - j) - j

    -- MODULE 11 : STANDARD FUNCTIONS
    REAL32 x :
    SEQ
      x := 0.75(REAL32)
      SEQ i = 0 FOR n[10] * iterations
        REAL32 r2 :
        x := SQRT ( EXP (ALOG (x) /t1) )

    -- RECORD FINISH TIME
    time ? time1
:
```

The Whetstone benchmark is run at high priority to ensure that a 1 μS resolution timer is used.

The table n contains the number of iterations for each loop in the benchmark; these were calculated to make the benchmark equivalent to a 'typical' scientific application. This array of weights is an integral part of the benchmark, and if it is altered the results are not comparable with figures quoted in 'Whetstones'.

The actual number of iterations of each loop is the product of the table entry and the second parameter of the Whetstone procedure. If this is set to 10 then 1 million 'Whetstones' are performed.

B.2 Dhrystone

This is the source of the program run on an IMS T414B-20, compiled with the product occam compiler.

```
PROC Dhrystone(CHAN OF INT32 In, Out)

  -- Define constants etc for the Struct equivalent
  VAL NULL IS 0 :
  VAL Ident1 IS 1 :
  VAL Ident2 IS 2 :
  VAL Ident3 IS 3 :
  VAL Ident4 IS 4 :
  VAL Ident5 IS 5 :

  VAL PtrComp     IS 0 : -- 'pointer' to one of these records
  VAL Discr       IS 1 :
  VAL EnumComp    IS 2 :
  VAL IntComp     IS 3 :
  VAL StringComp  IS 4 : -- StringComp is subsequent 30 bytes

  VAL StringSize IS 30 :
  VAL StringWords IS 8 : -- allocate 30/4 + 1 = 8 words on an IMS T414

  VAL StructSize IS StringWords + 4 :

  [3][StructSize]INT Records : -- all the records required

  -- Global variable declarations
  [51]INT     Array1 :
  [51][51]INT Array2 :
  INT         IntGlob :
  BOOL        BoolGlob :
  BYTE        Char1Glob, Char2Glob :
```

```
INT         PtrGlb, PtrGlbNext :

-- array placement
PLACE Array1 AT (#800 / 4) : -- placement for an IMS T414 and IMS T800
PLACE Array2 AT (#800 / 4) + 51 :
Array2Glob IS Array2 :
Array1Glob IS Array1 :

INT FUNCTION Func1 (VAL BYTE CharPar1, CharPar2)
  INT Res :
  VALOF
    BYTE CharLoc1, CharLoc2 :
    SEQ
      CharLoc1 := CharPar1
      CharLoc2 := CharLoc1
      IF
        CharLoc2 <> CharPar2 -- true
          SEQ
            Res := Ident1
        TRUE
          Res := Ident2
    RESULT Res
:
BOOL FUNCTION Func2 (VAL [StringSize]BYTE StrParI1, StrParI2)
  BOOL Res :
  VALOF
    INT FUNCTION strcmp (VAL [StringSize]BYTE S1, S2)
      INT order :
      VALOF
        IF
          IF i = 0 FOR StringSize
            S1[i] <> S2[i]
              IF
                (INT S1[i]) > (INT S2[i])
                  order := 1
                TRUE
                  order := -1
          TRUE
            order := 0
        RESULT order
    :
    -- StrParI1 = "DHRYSTONE, 1*'ST STRING"
    -- StrParI2 = "DHRYSTONE, 2*'ND STRING"
    INT IntLoc :
    BYTE CharLoc :
    SEQ
      IntLoc := 1
      WHILE IntLoc <= 1 -- executed once
        IF
```

```
          Func1(StrParI1[IntLoc], StrParI2[IntLoc+1]) = Ident1
            SEQ
              CharLoc := 'A'
              IntLoc := IntLoc + 1
          TRUE
            SKIP
      VAL CharLoc.int IS INT CharLoc : -- because no `>' for BYTES
      IF
        (CharLoc.int >= (INT 'W')) AND (CharLoc.int <= (INT 'Z'))
          IntLoc := 7 -- not executed
        TRUE
          SKIP
      IF
        CharLoc = 'X'
          Res := TRUE -- not executed
        strcmp(StrParI1, StrParI2) > 0
          SEQ -- not executed
            IntLoc := IntLoc + 7
            Res := TRUE
        TRUE
          Res := FALSE
    RESULT Res
:
BOOL FUNCTION Func3(VAL INT EnumParIn)
  BOOL Res :
  VALOF
    INT EnumLoc :
    SEQ
      EnumLoc := EnumParIn
      IF
        EnumLoc = Ident3
          Res := TRUE
        TRUE
          Res := FALSE
    RESULT Res
:
PROC P8([51]INT Array1Par, [51][51]INT Array2Par, VAL INT IntParI1, IntParI2)
  -- once; IntParI1 = 3, IntParI2 = 7
  INT IntLoc, IntIndex :
  SEQ
    IntLoc := IntParI1 + 5
    Array1Par[IntLoc]      := IntParI2
    Array1Par[IntLoc +  1] := Array1Par[IntLoc]
    Array1Par[IntLoc + 30] := IntLoc
    SEQ IntIndex = IntLoc FOR 2 -- twice
      Array2Par[IntLoc][IntIndex] := IntLoc
    Array2Par[IntLoc][IntLoc-1] := Array2Par[IntLoc][IntLoc-1] + 1
    Array2Par[IntLoc+20][IntLoc] := Array1Par[IntLoc]
    IntGlob := 5
```

```
:
PROC P7 (VAL INT IntParI1, IntParI2, INT IntParOut) -- thrice
  -- 1) IntParI1 =  2, IntParI2 =  3, IntParOut :=  7
  -- 2) IntParI1 =  6, IntParI2 = 10, IntParOut := 18
  -- 3) IntParI1 = 10, IntParI2 =  5, IntParOut := 17
  INT IntLoc :
  SEQ
    IntLoc := IntParI1 + 2
    IntParOut := IntParI2 + IntLoc
:
PROC P5() -- once
  SEQ
    Char1Glob := 'A'
    BoolGlob := FALSE
:
PROC P4() -- once
  BOOL BoolLoc :
  SEQ
    BoolLoc := Char1Glob = 'A'
    BoolLoc := BoolLoc OR BoolGlob
    Char2Glob := 'B'
:
PROC P3(INT PtrParOut) -- executed once
  SEQ
    IF
      PtrGlb <> NULL -- true
        PtrParOut := Records[PtrGlb][PtrComp]
      TRUE
        IntGlob := 100
    P7(10, IntGlob, Records[PtrGlb][IntComp])
:
PROC P6 (VAL INT EnumParIn, INT EnumParOut) -- once
  -- EnumParOut = Ident3, EnumParOut := Ident2
  SEQ
    EnumParOut := EnumParIn
    IF
      NOT Func3(EnumParIn) -- not taken
        EnumParOut := Ident4
      TRUE
        SKIP
    CASE EnumParIn
      Ident1
        EnumParOut := Ident1
      Ident2
        IF
          IntGlob > 100
            EnumParOut := Ident1
          TRUE
            EnumParOut := Ident4
```

```
      Ident3 -- this one chosen
        EnumParOut := Ident2
      Ident4
        SKIP
      Ident5
        EnumParOut := Ident3
:
PROC P2(INT IntParIO) -- executed once
  INT IntLoc, EnumLoc :
  BOOL Going :
  SEQ
    IntLoc := IntParIO + 10
    Going := TRUE
    WHILE Going -- executed once
      SEQ
        IF
          Char1Glob = 'A'
            SEQ
              IntLoc := IntLoc - 1
              IntParIO := IntLoc - IntGlob
              EnumLoc := Ident1
          TRUE
            SKIP
        Going := EnumLoc <> Ident1
:
PROC P1(VAL INT PtrParln) -- executed once
  [StructSize] INT NextRecTemp :
  SEQ
    NextRecTemp := Records[PtrGlb] -- must do this to avoid aliasing
    Records[PtrParln][IntComp] := 5
    NextRecTemp[IntComp] := Records[PtrParln][IntComp]
    NextRecTemp[PtrComp] := Records[PtrParln][PtrComp]
    P3(NextRecTemp[PtrComp])
    -- NextRecTemp[PtrComp] = Records[PtrGlb][PtrComp] = PtrGlbNext
    IF
      NextRecTemp[Discr] = Ident1 -- it does
        INT IntCompTemp :
        SEQ
          NextRecTemp[IntComp] := 6
          P6(Records[PtrParln][EnumComp], NextRecTemp[EnumComp])
          NextRecTemp[PtrComp] := Records[PtrGlb][PtrComp]
          IntCompTemp := NextRecTemp[IntComp] -- to avoid aliasing
          P7(IntCompTemp, 10, NextRecTemp[IntComp])
      TRUE
        Records[PtrParln] := NextRecTemp
    Records[Records[PtrParln][PtrComp]] := NextRecTemp
:
PROC P0(INT32 out, VAL INT32 loops)
  TIMER TIME :
```

```
[StringSize]BYTE String1Loc, String2Loc :
INT IntLoc1, IntLoc2, IntLoc3 :
BYTE CharLoc :
INT EnumLoc :
INT StartTime, EndTime, NullTime :
VAL Loops IS 10 * (INT loops) :
SEQ
  -- initialisation
  -- initialise arrays to avoid overflow
  SEQ i = 0 FOR SIZE Array1Glob
    Array1Glob[i] := 0
  SEQ i = 0 FOR SIZE Array2Glob
    SEQ j = 0 FOR SIZE Array2Glob[0]
      Array2Glob[i][j] := 0
  PtrGlb := 1
  PtrGlbNext := 2
  -- initialise record 'pointed' to by PtrGlb
  Record IS Records[PtrGlb] :
  SEQ
    Record[PtrComp]  := PtrGlbNext
    Record[Discr]    := Ident1
    Record[EnumComp] := Ident3
    Record[IntComp]  := 40
    [4*StringWords]BYTE ByteBuff RETYPES
                        [Record FROM StringComp FOR StringWords] :
    [ByteBuff FROM 0 FOR StringSize]:=
                        "DHRYSTONE PROGRAM, SOME STRING"
  String1Loc := "DHRYSTONE PROGRAM, 1*'ST STRING"

  -- measure loop overhead
  TIME ? StartTime
  SEQ i = 0 FOR Loops
    SKIP
  TIME ? EndTime
  NullTime := EndTime MINUS StartTime

  TIME ? StartTime
  SEQ i = 0 FOR Loops
    SEQ
      P5()
      P4()
      -- Char1Glob = 'A', Char2Glob = 'B', BoolGlob = FALSE
      IntLoc1 := 2
      IntLoc2 := 3
      String2Loc := "DHRYSTONE PROGRAM, 2*'ND STRING"
      EnumLoc := Ident2
      BoolGlob := NOT Func2(String1Loc, String2Loc)
      -- BoolGlob = TRUE
      WHILE IntLoc1 < IntLoc2 -- body executed once only
```

```
            SEQ
              IntLoc3 := (5 * IntLoc1) - IntLoc2
              P7(IntLoc1, IntLoc2, IntLoc3)
              IntLoc1 := IntLoc1 + 1
          P8(Array1Glob, Array2Glob, IntLoc1, IntLoc3)
          -- IntGlob = 5
          P1(PtrGlb)
          SEQ Charlndex = INT 'A' FOR ((INT Char2Glob) - ((INT 'A')-1))
            -- twice
            IF
              EnumLoc = Func1(BYTE Charlndex, 'C')
                P6(Ident1, EnumLoc)
              TRUE
                SKIP
          -- EnumLoc = Ident1
          -- IntLoc1 = 3, IntLoc2 = 3, IntLoc3 = 7
          IntLoc3 := IntLoc2 * IntLoc1
          IntLoc2 := IntLoc3 / IntLoc1
          IntLoc2 := (7 * (IntLoc3 - IntLoc2)) - IntLoc1
          P2(IntLoc1)
      TIME ? EndTime

      out := INT32 ((EndTime MINUS StartTime) - NullTime)
  :
  PRI PAR -- to get high priority timer
    INT32 count, result :
    SEQ
      In ? count
      P0(result, count)
      Out ! result
    SKIP
:
```

This program is intended to be run on a single processor, with channel out mapped onto a hard link connected to another processor, running a process which outputs the number of loops to be performed (to improve the resolution of the timer) - typically 10000 - and then inputs the number of microseconds taken. A simple calculation turns this into a number of 'Dhrystones per second'.

C Elementary function performance

The table below gives the time taken to evaluate complete standard elementary functions on an IMS T800-20 and an IMS T414-20, each with 150 nS external RAM. Timings are given for both the case when the function code and the process workspace are in the on-chip RAM (for the IMS T800) and

when the code is stored in the external RAM (both processors). The figures for each function were derived from measurements taken for 8000 arguments chosen at random from the interval [0.0, 10.0], except for arcsine and arccosine where the points were drawn from the interval [-1.0, 1.0], and the double-precision hyperbolic functions, for which the points were drawn from [0.0, 20.0].

	IMS T800-20 ON-CHIP		**IMS T800-20** OFF-CHIP		**IMS T414-20** OFF-CHIP	
single-precision	mean	max	mean	max	mean	max
SORT	5.9	6.4	6.0	6.5	26.0	27.6
ALOG	22.5	22.9	27.4	27.9	131.1	141.4
ALOG10	25.7	26.1	31.4	31.9	145.2	155.3
EXP	22.0	22.2	26.7	27.0	120.6	126.8
SIN	16.2	16.8	19.2	19.9	146.7	169.6
COS	18.9	19.3	22.2	22.6	178.1	186.8
TAN	18.4	19.2	22.3	23.2	142.7	164.4
ASIN	17.0	22.2	19.8	25.3	105.1	145.7
ACOS	16.7	21.3	19.8	24.8	101.5	132.6
ATAN	18.5	21.9	22.6	26.4	125.6	161.7
SINH	26.6	28.7	32.7	35.3	149.8	167.6
COSH	26.2	26.7	31.9	32.6	155.2	166.1
TANH	23.4	28.3	28.6	34.6	137.3	175.6
double-precision	mean	max	mean	max	mean	max
DSQRT	12.2	12.9	12.2	13.0	204.0	212.8
DALOG	34.5	38.5	45.0	46.0	607.9	636.1
DALOG10	42.5	43.1	49.9	50.9	658.7	687.4
DEXP	39.8	40.4	47.0	47.7	512.9	538.5
DSIN	33.1	34.2	38.1	39.2	590.0	655.2
DCOS	29.4	29.9	33.7	34.2	671.9	700.0
DTAN	35.8	37.3	42.0	43.7	632.4	712.2
DASIN	33.1	41.7	37.0	45.7	587.0	758.7
DACOS	32.9	40.6	36.8	44.9	574.3	714.2
DATAN	31.3	35.7	36.6	41.7	565.6	701.8
DSINH	45.9	47.8	54.2	56.5	609.5	649.0
DCOSH	44.8	45.4	53.3	54.1	618.4	648.5
DTANH	44.2	47.4	52.8	56.8	623.6	686.4

Table 4: Timings in microseconds

No figures are given for the IMS T212, but as a rough guide, consider single-precision functions to take between 5 and 7 times as long as for an IMS T414.

D Benchmarking the IMS T212

It should be noted that obtaining benchmark figures for the IMS T212 is slightly more involved than for either the IMS T414 or the IMS T800. This is because the built-in timer has only 16 bits on this processor, as opposed to 32 on the other two processors, so consequently the clock 'wraps round' very much faster. In fact it does so faster than a benchmark program can be run, and so the run-time of the program cannot be obtained simply by reading the clock at the beginning and end of the run, as shown in the preceding listings.

The solution to this problem is to use another processor to perform the timing. Instead of reading the timer the program on the IMS T212 sends a message to another processor (an IMS T414 or an IMS T800) which responds by reading its own timer. The quoted benchmark results for the IMS T212 were obtained in this way.