BENCHMARKING EXECUTION IMPROVEMENTS OF AN APPLE MACINTOSH COMPUTER UPGRADED WITH A MATH COPROCESSOR

by

ROBERT WAYNE MOSS

B. S., Kansas State University, 1985

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

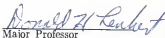
Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1988

Approved by:



Major Professor

A11208 232034

Table of Contents

List of Figures . . . iii

Introduction . . . 5

Methodology . . . 6

Benchmark Descriptions . . . 8

Fast Fourier Transform Benchmark . . . 10
Fibonacci Benchmark . . . 12
Eratosthenes Benchmark . . . 13
Float Benchmark . . . 14
Savage Benchmark . . . 15
Dhrystone Benchmark . . . 17
Whetstone Benchmark . . . 19

Conclusions . . . 21

Bibliography . . . 23

Appendixes

Dhrystone listing . . . 25
Fibonacci listing . . . 35
Float listing . . . 37
Whetstone listing . . . 39
Savage listing . . . 45
Eratosthenes listing . . . 47
Fast Fourier Transform listing . . . 49

List of Figures

1. Timing Benchmarks 9

2. Fast Fourier Transform Benchmark 11

3. Additional Benchmarks 16

4. Dhrystone Performance 18

5. Whetstone Performance 20

Acknowledgements

I would like to thank Dr. Don Lenhert for the special interest he has for his students. I would also like to thank my dear Angie for being there when I needed her.

Introduction

The research objectives were to install an accelerator board in an Apple MacIntosh Plus computer system, benchmark the resulting speed improvements and analyze the results. System benchmarks were established by a variety of common programs including the Dhrystone and Whetstone.

The Motorola 68881 math coprocessor implements the complete *IEEE Standard for Binary Floating-Point Arithmetic*. In addition to the add, subtract, multiply and divide operations, the coprocessor implements a full set of trigonometric and transcendental functions. When used with the 68000 microprocessor, the 68881 is accessed as a memory-mapped peripheral device, not as a true coprocessor.

Methodology

Speed improvements resulting from adding a math coprocessor and a faster processor to a MacIntosh Plus computer were measured. The MacIntosh Plus comes with a 7.8336MHz 68000 processor and 1MB of RAM. An upgrade product, called TurboMax, available from MacMemory, Inc. adds a 16MHz 68000, 1MB of RAM (addressed without video wait states), and a socket for a 68881 which is addressed as a memory mapped peripheral because the 68000 does not support a true coprocessor. This product includes a patch for the MacIntosh SANE math library to support the 68881. The SANE library is an IEEE standard set of floating point operations implemented in firmware. Due to the overhead of the TurboMax library patch, floating point operations run slightly slower with the TurboMax when the 68881 is removed than on a regular MacIntosh.

The TurboMax has additional circuitry that allows all access of its local address space to be transparent to the actions of the mother board. Mother board RAM is effectively slowed down by the addition of wait states to refresh the video display. Programs present in daughter board RAM run considerably faster because this RAM is not accessed by the video circuitry on the mother board. The same circuitry allows the 68881 to be interfaced without complications from mother board.

In view of the relative differences in memory speed, all benchmarks were forced to run in the daughter board RAM and in the mother board RAM to assess the memory dependency effects of each benchmark. A 1MB

RAM disc was created in mother board memory to force the benchmark to run in daughter board RAM.

One benchmark expected to be highly dependent on floating point speed is the fast Fourier transform. The fast Fourier transform was executed for a variety of different transform sizes to determine the overall effect of the RAM speed. With larger vectors the RAM overhead may become more significant than the floating point overhead. Float and savage were also chosen to evaluate two indices of mostly floating point speed. Neither the float nor the savage programs compute anything useful. Float repeatedly performs floating point multiplications and divisions. Savage repeatedly performs a variety of transcendental functions.

Fibonacci and Eratosthenes were two programs chosen to evaluate the effects of mostly memory and processor speed improvements, because they rely solely on integer arithmetic. Fibonacci computes Fibonacci numbers and Eratosthenes computes prime numbers with the Eratosthenes sieve algorithm. In addition, the two more common benchmarks, Dhrystone and Whetstone, were used. Dhrystone is an index of overall computer speed, and is computationally balanced in several respects. One would expect the Dhrystones to improve with the use of faster memory, but not to be significantly affected by an increase in floating point performance. Whetstone is an index which depends heavily on floating point performance. The Whetstone benchmark program is balanced with respect to simple and transcendental floating point operations, but spends little time on other computations. Neither the Dhrystone or the Whetstone compute anything useful. Their statements are performed only to determine execution time.

Benchmark Descriptions

Several benchmarks were run to determine the effects of the upgrade in various configurations. Two systems were used in benchmarks:

- Standard Macintosh Plus system. This is a machine with a MC68000 processor running at 7.8336 MHz and with 1MB RAM.
- Macintosh Plus with TurboMax upgrade. This machine has a MC68000 processor running at 15.6712 MHz, 1MB standard RAM, 1MB RAM accessed with no video wait states, and a 16 MHz MC68881.

In view of the faster memory on the TurboMax board, two sets of tests were made, with all memory accesses falling in either slow or fast memory. By creating a 1MB RAM disk in lower memory, lower memory was consumed and applications were forced to run in the higher portions of faster memory. Test configurations were as follows:

- Standard Macintosh Plus system.
- Macintosh Plus with TurboMax upgrade, no 68881, slower memory (i.e low memory).
- Macintosh Plus with TurboMax upgrade, no 68881, faster memory (i.e high memory).
- Macintosh Plus with TurboMax upgrade, using 68881, slower memory (i.e low memory).
- Macintosh Plus with TurboMax upgrade, using 68881, faster memory (i.e high memory).

The results of the benchmarks can be seen in Figure 1.

All times were computed by the executing programs from the difference between starting time and ending time as obtained with the standard unix C function : time(*tloc), which returns the time in seconds since January 1, 1970 00:00:00. The values come from the MacIntosh interrupt-driven real-time clock and are accurate to within 1/60th of a second. The approach excludes operating system overhead associated with launching and terminating programs from the measured time.

Figure 1 - Timing benchmarks (in seconds)

	Macintosh Plus	Macintosh Plus w TurboMax (no 68881) (Low Mem.)	Macintosh Plus w TurboMax (no 68881) (High Mem.)	Macintosh Plus w TurboMax (68881) (Low Mem.)	Macintosh Plus w TurboMax (68881) (High Mem.)
512 FFT	19	19	14	11	6
1024 FFT	41	43	31	23	12
2048 FFT	89	93	68	51	26
4096 FFT	194	203	148	110	56
8192 FFT	420	440	320	235	120
16384 FFT	758	816	569	626	352
Dhrystone	227	145	98	145	96
Whetstone	134	138	106	24	13
Float	125	124	104	39	20
Fibonacci	246	165	104	164	104
Eratosthenes	280	188	118	187	118
Savage	758	788	602	28	15
Dhrystones	881	1379	2040	1379	2083
Whetstones	7463	7246	9434	41667	76923
sec/Dhrystone	1/881	1/1379	1/2040	1/1379	1/2083
sec/Whetstone	1/7463	1/7246	1/9434	1/41667	1/76923

Fast Fourier Transform Benchmark

A fast Fourier transform (FFT) was chosen as one benchmark. Generally the FFT gives an indication of the floating point performance of a machine. This test also demonstrates that once one increases floating point performance, memory manipulations become dominant of the algorithms throughput. A common *decimation-in-time*, or *Cooley-Tukey*, algorithm is used.[1] The subroutine first performs a bit-reversal of a complex data sequence and then performs the Danielson-Lanczos algorithm which recursively evaluates the discrete Fourier transform.[2] The code is written in the C language and is optimized to all extents where it still follows the standards defined by Kernighan and Ritchie.[3] Optimizations include: assigning loop index variables to registers, using registers as pointers to access data rather than repeatedly computing vector indices, shift operators are used for integer multiplication, and assignment operators are used where a variable is being modified from an old value. It is interesting to note that this FFT algorithm performs many memory accesses for each arithmetic operation. The benchmarks demonstrate that the memory overhead becomes

---

[1] Ahmed, Nasir, and T. Natarajan. Dicrete-Time Signals and Systems. Reston: Reston Publishing Company Inc., 1983.

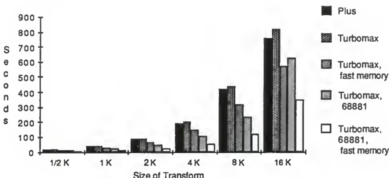
[2] Press, William H., et al. Numerical Recipes: the Art of Scientific Computing. Cambridge University Press, 1986.

[3] Kernighan, Brian W., and Dennis M. Ritchie. The C programming Language. Englewood Cliffs: Prentice-Hall Inc., 1978.

quite significant when the data size increases and when the floating point execution time is reduced.

Benchmarks of the FFT routine were run with the data vector size set at 512, 1024, 2048, 4096, 8192, and 16384 complex points. Figure 2 shows the relative execution times of the FFT with several system configurations. When floating point operations are done in firmware, the use of fast memory results in a 30% improvement in execution time. If the 68881 is used for floating point computations, the resulting improvement in execution time is closer to 54%. Note that when vectors larger than 8192 are used, the execution speed gains due to the math coprocessor are smaller than the gains from using faster memory. (see Figure 2.)

Figure 2 - Fast Fourier Transform Benchmark



Fibonacci Benchmark

Several other benchmarks were also run. These programs are all variations of programs from BYTE magazine's BYTEnet telecommunications service.[1] The variations are limited to changing code that is not portable into its equivalent in Lightspeed C. The modifications consist mostly of altering the procedures used to compute timing information.

The Fibonacci program computes the Fibonacci number of 24. The recursive definition of a Fibonacci number is:

```
fibonacci (x)
    if (x > 2)  then  fibonacci = fibonacci (x - 1) + fibonacci (x - 2)
                else  fibonacci = 1.
```

Because this algorithm calculates only integer values, the floating-point coprocessor does not affect the execution time. This is useful for benchmarking integer operations and effective processor throughput for simple calculations and recursive function calls. Running the program with the faster TurboMax memory configuration gives a 37% decrease in execution time from the slow TurboMax memory configuration. The slow TurboMax memory configuration decreases execution time by 33% over the original Macintosh. The fast TurboMax memory configuration decreases execution time by 58% over the original Macintosh configuration. (see Figure 3.)

---

[1] BYTE Editorial Staff. "High-Tech Horsepower," BYTE. vol. 12, no. 8 (July 1987), 101-108.

Eratosthenes Benchmark

The Eratosthenes sieve benchmarks resulted in improvements of execution time similar to the Fibonacci benchmarks. This program finds all the integer prime numbers through 8190. Because integer arithmetic is used the math coprocessor does not influence the results. This is useful for benchmarking integer math and effective processor throughput for simple calculations. Running the program with the faster TurboMax memory configuration gives a 37% decrease in execution time over the slow TurboMax memory configuration. The slow TurboMax memory configuration decreases execution time by 33% over the original Macintosh configuration. The fast TurboMax memory configuration decreases execution time by 58% over the original Macintosh configuration. (see Figure 3.) Both the Fibonacci and the Eratosthenes benchmarks indicate that there is no linear correlation between doubling the speed of the MC68000 and gains in execution speed. It is obvious that memory wait states cause a more serious degradation of system throughput than the CPU. It would be interesting to see the same benchmarks run with an effective RAM cache to minimize wait states. Because the Fibonacci computation is recursive, the compiler must take advantage of the cache for much gain in performance.

Float Benchmark

Float is a benchmark of floating point speed which performs repetitive multiplications and divisions. This is useful for benchmarking floating point operations and effective processor throughput for simple calculations. Because the Macintosh floating library is contained in ROMs which require more than double the access time of the RAM, calls to these routines decrease system throughput due to wait states. The result of this is that the original Macintosh ran only slightly slower than the upgraded version. The speed improvements due to the 68881 are 68% with the slow TurboMax memory model and 81% with the faster TurboMax memory model over the original Macintosh configuration. When floating point operations are done in firmware, the use of fast TurboMax memory results in a 16% improvement in execution time over the slow TurboMax memory configuration. If the 68881 is used for floating point computations, the resulting speed improvement is 48% over the slow TurboMax memory configuration.

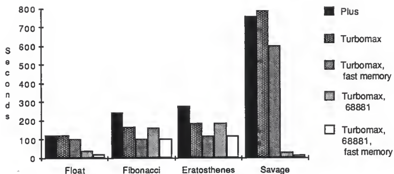
Savage Benchmark

Savage is another benchmark of floating point speed. This algorithm repetitively performs the following expression:

a = tan (atan (exp (log (sqrt (a*a))))) + 1.0

This is useful for benchmarking floating point math because the 68881 implements transcendental functions. The performance gains from this benchmark are similar to those produced by the float benchmark, but because transcendental functions require much more execution time than simple floating point operations when implemented in software, the results are more dramatic. The original configuration MacIntosh ran slightly faster than the upgraded version using firmware floating point operations. Due to the 68881, the execution time decreased 96% with the slow TurboMax memory model and 98% with the faster TurboMax memory model over the original MacIntosh. When floating point operations are done in firmware, the use of fast TurboMax memory results in a 23% decrease in execution time over the slow TurboMax memory configuration. When the 68881 is used for floating point computations, the use of fast TurboMax memory results in a 46% decrease in execution time over the slow TurboMax memory configuration. Benchmarks such as these demonstrate the improvements that a math coprocessor can attain with highly intensive math applications. Using the 68881, the execution time of the savage benchmark decreases by 98% from the time it takes to run on a standard MacIntosh. (see Figure 3.)

Figure 3- Additional Benchmarks



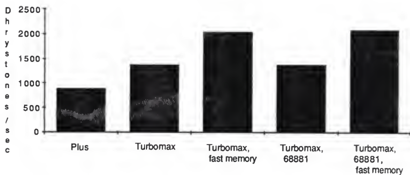
Dhrystone Benchmark

The Dhrystone is an index used for overall computer speed, and does not depend heavily on floating point speed. One pass of the program consists of a Dhrystone. The main loop of the program is executed 50000 times to find the average Dhrystones / second. The program contains statements distributed as follows:

assignments	53%
control statements	32%
procedure, function calls	15%

The program is balanced with respect to statement type, operand type (for simple data types), and operand access (operand global, local, parameter, or constant). It is readily apparent that the presence of the math coprocessor does not significantly affect the benchmark times. Because the program is also rather memory dependent, the use of faster memory significantly impacted performance. To be consistent with the previously presented benchmarks, performance of the Dhrystone is measured in seconds per Dhrystone. The TurboMax configuration with slow memory produced a 36% decrease of execution time relative to the original MacIntosh. The TurboMax configuration with faster memory produced a 58% decrease in execution time relative to the original MacIntosh. (see Figure 4.)

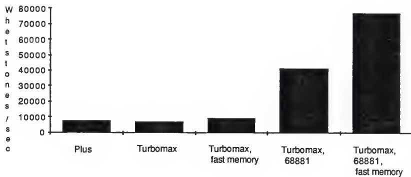
Figure 4 - Dhrystone Performance



Whetstone Benchmark

The Whetstone benchmark is used as an index to compiler optimization and floating point performance. The program executes ten million Whetstone instructions and computes an average for Whetstones / second. This benchmark resulted in improvements in execution time similar to the savage benchmark. The original configuration Macintosh ran slightly faster than the upgraded version. To be consistent with the previously presented benchmarks, performance gains with the Whetstone are reported in seconds per Whetstone. The decrease of execution time from the original MacIntosh due to the 68881 is 82% with the slow TurboMax memory model and 90% with the faster TurboMax memory model. The decrease of execution time from the original MacIntosh is 21% with the faster TurboMax memory model with floating point operations done in firmware. (see Figure 5.)

Figure 5 - Whetstone Performance



Conclusions

A TurboMax accelerator board was installed in an Apple MacIntosh Plus computer system. Resulting speed improvements were benchmarked and analyzed. Speed improvements were measured due to a math coprocessor, a faster processor and memory accessed without video wait states. Several programs were used to generate indices of floating point speed, memory and processor speed improvements, and overall computer speed.

It was determined that doubling the processor speed alone produced little increase in speed. When the faster processor was coupled to memory that did not have the video wait state overhead, the speed improvements became significant. When the MacIntosh was upgraded with the math coprocessor, benchmarks involving many floating point calculations had an effective execution gain of up to 90%. But in most benchmarks, this large increase was not observed due to the other overhead of the programs. Of course when the math coprocessor was used, the gains due to using faster memory were even more visible. Once the floating point operations were improved to a limit, memory accesses become the most significant performance limiting factor. So, in this case, an improvement in effective memory access speed will result in a more dramatic gain in execution speed.

Measurements made using the TurboMax upgrade board have demonstrated improvements in performance for normal processing tasks as well as dramatic improvements in the speed of floating point operations. The amount of performance enhancement is largely dependent on the

computational characteristics of the benchmarking program. Fibonacci, Eratosthenes and Dhrystone benchmarks showed the most dramatic improvements with the use of only the faster memory and the processor on the TurboMax board. Float, savage and Whetstone, which are primarily floating point oriented, demonstrated additional gains due to the math coprocessor. The execution time for the fast Fourier transform was highly dependent on transform size. The addition of the coprocessor had the most effect for small vectors. When using large vectors, a significant decrease in execution time came both from faster memory and the coprocessor since memory overhead increases with vector size.

Bibliography

Ahmed, Nasir, and T. Natarajan. Discrete-Time Signals and Systems. Reston: Reston Publishing Company Inc., 1983.

BYTE Editorial Staff. "High-Tech Horsepower," BYTE. vol. 12, no. 8 (July 1987), 101-108.

Chernicoff, Stephen. Macintosh Revealed: Programing the Toolbox. Vol. I. Berkeley: Hayden Book Company, 1985.

Chernicoff, Stephen. Macintosh Revealed: Programing With the Toolbox. Vol. II. Berkeley: Hayden Book Company, 1985.

Kernighan, Brian W., and Dennis M. Ritchie. The C programming Language. Englewood Cliffs: Prentice-Hall Inc., 1978.

MacMemory. TurboMax Owner's Manual. MacMemory Inc., 1987

Motorola. MC6881 Floating-Point Coprocessor User's Manual. 1st ed. Motorola Inc., 1985.

Press, William H., et al. Numerical Recipes: the Art of Scientific Computing. Cambridge University Press, 1986.

Appendixes

Dhrystone

```
/*****************************************************************************
 *     Dhrystone benchmark in C.                                             *
 *                                                                           *
 *     Modified from posting on BYTEnet (617) 861-9764, BYTE magazine        *
 *     to be compatible with Lightspeed C.                                   *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   The following program contains statements of a high-level programming  *
 *     language (C) in a distribution considered representative:            *
 *                                                                           *
 *     assignments                 53%                                       *
 *     control statements          32%                                       *
 *     procedure, function calls   15%                                       *
 *                                                                           *
 *  100 statements are dynamically executed.  The program is balanced with  *
 *     respect to the three aspects:                                         *
 *             - statement type                                              *
 *             - operand type (for simple data types)                        *
 *             - operand access                                              *
 *                    operand global, local, parameter, or constant.         *
 *                                                                           *
 *  The combination of these three aspects is balanced only approximately.   *
 *                                                                           *
 *     The program does not compute anything meaningful, but it is           *
 *     syntactically and schematically correct.                              *
 *                                                                           *
 *****************************************************************************/

#include "sane.h"                    /* ADDED */
#include "stdio.h"                   /* ADDED */
#include "math.h"                    /* ADDED */
```

```
#include "storage.h"                /* ADDED */
#include "strings.h"                /* ADDED */


/* Accuracy of timings and human fatigue controlled by next two lines */
#define LOOPS  200000


char   Version[] = "1.1";


#define      structassign(d, s)   d = s


typedef      enum {Ident1, Ident2, Ident3, Ident4, Ident5} Enumeration;


typedef int    OneToThirty;
typedef int    OneToFifty;
typedef char CapitalLetter;
typedef char String30[31];
typedef int    Array1Dim[51];
typedef int    Array2Dim[51][51];

struct  Record
{
      struct Record        *PtrComp;
      Enumeration          Discr;
      Enumeration          EnumComp;
      OneToFifty           IntComp;
      String30             StringComp;
};

typedef      struct Record        RecordType;
typedef      RecordType *         RecordPtr;
```

```
typedef     int         boolean;

#define     TRUE        1
#define     FALSE       0


extern      Enumeration       Func1();
extern      boolean     Func2();

main()
{
        Proc0();
        exit(0);
}

/*
 * Package 1
 */
int         IntGlob;
boolean     BoolGlob;
char        Char1Glob;
char        Char2Glob;
Array1Dim   Array1Glob;
Array2Dim   Array2Glob;
RecordPtr   PtrGlb;
RecordPtr   PtrGlbNext;

Proc0()
{
        OneToFifty          IntLoc1;
        register OneToFifty IntLoc2;
        OneToFifty          IntLoc3;
        register char             CharLoc;
        register char             CharIndex;
        Enumeration               EnumLoc;
```

```
	String30		String1Loc;
	String30		String2Loc;
	extern char		*malloc();

	long			time();
	long			starttime;
	long			benchtime;
	long			nulltime;
	register unsigned long	i;

	starttime = time( (long *) 0);
	for (i = 0; i < LOOPS; ++i);
	nulltime = time( (long *) 0) - starttime; /* Computes o'head of loop */

	PtrGlbNext = (RecordPtr) malloc(sizeof(RecordType));
	PtrGlb = (RecordPtr) malloc(sizeof(RecordType));
	PtrGlb->PtrComp = PtrGlbNext;
	PtrGlb->Discr = Ident1;
	PtrGlb->EnumComp = Ident3;
	PtrGlb->IntComp = 40;
	strcpy(PtrGlb->StringComp, "DHRYSTONE PROGRAM, SOME
STRING");
	Array2Glob[8][7] = 10;

/*****************
-- Start Timer --
*****************/
	starttime = time( (long *) 0);
	for (i = 0; i < LOOPS; ++i)
	{

		Proc5();
		Proc4();
		IntLoc1 = 2;
		IntLoc2 = 3;
```

```
        strcpy(String2Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
        EnumLoc = Ident2;
        BoolGlob = ! Func2(String1Loc, String2Loc);
        while (IntLoc1 < IntLoc2)
        {
                IntLoc3 = 5 * IntLoc1 - IntLoc2;
                Proc7(IntLoc1, IntLoc2, &IntLoc3);
                ++IntLoc1;
        }
        Proc8(Array1Glob, Array2Glob, IntLoc1, IntLoc3);
        Proc1(PtrGlb);
        for (CharIndex = 'A'; CharIndex <= Char2Glob; ++CharIndex)
                if (EnumLoc == Func1(CharIndex, 'C'))
                        Proc6(Ident1, &EnumLoc);
        IntLoc3 = IntLoc2 * IntLoc1;
        IntLoc2 = IntLoc3 / IntLoc1;
        IntLoc2 = 7 * (IntLoc3 - IntLoc2) - IntLoc1;
        Proc2(&IntLoc1);
    }

/*****************
-- Stop Timer --
*****************/

    benchtime = time( (long *) 0) - starttime - nulltime;
    printf("Dhrystone(%s) time for %ld passes = %ld\n",
        Version,
        (long) LOOPS, benchtime);
    printf("This machine benchmarks at %ld dhrystones/second\n",
        ((long) LOOPS) / benchtime);

}

Proc1(PtrParIn)
register RecordPtr  PtrParIn;
```

```
{
#define     NextRecord  (*(PtrParIn->PtrComp))

        structassign(NextRecord, *PtrGlb);
        PtrParIn->IntComp = 5;
        NextRecord.IntComp = PtrParIn->IntComp;
        NextRecord.PtrComp = PtrParIn->PtrComp;
        Proc3(NextRecord.PtrComp);
        if (NextRecord.Discr == Ident1)
        {
                NextRecord.IntComp = 6;
                Proc6(PtrParIn->EnumComp, &NextRecord.EnumComp);
                NextRecord.PtrComp = PtrGlb->PtrComp;
                Proc7(NextRecord.IntComp, 10, &NextRecord.IntComp);
        }
        else
                structassign(*PtrParIn, NextRecord);

#undef      NextRecord
}

Proc2(IntParIO)
OneToFifty  *IntParIO;
{
        register OneToFifty         IntLoc;
        register Enumeration                EnumLoc;

        IntLoc = *IntParIO + 10;
        for(;;)
        {
                if (Char1Glob == 'A')
                {
                        --IntLoc;
                        *IntParIO = IntLoc - IntGlob;
                        EnumLoc = Ident1;
```

```
			}
			if (EnumLoc == Ident1)
				break;
		}
}

Proc3(PtrParOut)
RecordPtr	*PtrParOut;
{
	if (PtrGlb != NULL)
		*PtrParOut = PtrGlb->PtrComp;
	else
		IntGlob = 100;
	Proc7(10, IntGlob, &PtrGlb->IntComp);
}

Proc4()
{
	register boolean	BoolLoc;

	BoolLoc = Char1Glob == 'A';
	BoolLoc |= BoolGlob;
	Char2Glob = 'B';
}

Proc5()
{
	Char1Glob = 'A';
	BoolGlob = FALSE;
}

extern boolean Func3();

Proc6(EnumParIn, EnumParOut)
register Enumeration	EnumParIn;
```

```
register Enumeration      *EnumParOut;
{
        *EnumParOut = EnumParIn;
        if (! Func3(EnumParIn) )
                *EnumParOut = Ident4;
        switch (EnumParIn)
        {
        case Ident1: *EnumParOut = Ident1; break;
        case Ident2: if (IntGlob > 100) *EnumParOut = Ident1;
                     else *EnumParOut = Ident4;
                     break;
        case Ident3: *EnumParOut = Ident2; break;
        case Ident4: break;
        case Ident5: *EnumParOut = Ident3;
        }
}

Proc7(IntParI1, IntParI2, IntParOut)
OneToFifty  IntParI1;
OneToFifty  IntParI2;
OneToFifty  *IntParOut;
{
        register OneToFifty IntLoc;

        IntLoc = IntParI1 + 2;
        *IntParOut = IntParI2 + IntLoc;
}

Proc8(Array1Par, Array2Par, IntParI1, IntParI2)
Array1Dim  Array1Par;
Array2Dim  Array2Par;
OneToFifty  IntParI1;
OneToFifty  IntParI2;
{
        register OneToFifty IntLoc;
```

```
	register OneToFifty IntIndex;

	IntLoc = IntParI1 + 5;
	Array1Par[IntLoc] = IntParI2;
	Array1Par[IntLoc+1] = Array1Par[IntLoc];
	Array1Par[IntLoc+30] = IntLoc;
	for (IntIndex = IntLoc; IntIndex <= (IntLoc+1); ++IntIndex)
		Array2Par[IntLoc][IntIndex] = IntLoc;
	++Array2Par[IntLoc][IntLoc-1];
	Array2Par[IntLoc+20][IntLoc] = Array1Par[IntLoc];
	IntGlob = 5;
}

Enumeration Func1(CharPar1, CharPar2)
CapitalLetter	CharPar1;
CapitalLetter	CharPar2;
{
	register CapitalLetter	CharLoc1;
	register CapitalLetter	CharLoc2;

	CharLoc1 = CharPar1;
	CharLoc2 = CharLoc1;
	if (CharLoc2 != CharPar2)
		return (Ident1);
	else
		return (Ident2);
}

boolean Func2(StrParI1, StrParI2)
String30	StrParI1;
String30	StrParI2;
{
	register OneToThirty	IntLoc;
	register CapitalLetter	CharLoc;
```

```
	IntLoc = 1;
	while (IntLoc <= 1)
		if (Func1(StrParI1[IntLoc], StrParI2[IntLoc+1]) == Ident1)
		{
			CharLoc = 'A';
			++IntLoc;
		}
	if (CharLoc >= 'W' && CharLoc <= 'Z')
		IntLoc = 7;
	if (CharLoc == 'X')
		return(TRUE);
	else
	{
		if (strcmp(StrParI1, StrParI2) > 0)
		{
			IntLoc += 7;
			return (TRUE);
		}
		else
			return (FALSE);
	}
}

boolean Func3(EnumParIn)
register Enumeration	EnumParIn;
{
	register Enumeration	EnumLoc;

	EnumLoc = EnumParIn;
	if (EnumLoc == Ident3) return (TRUE);
	return (FALSE);
}
```

Fibonacci

```
/*********************************************************************************
*       Fibonacci benchmark in C.                                                *
*                                                                                *
*       Modified from posting on BYTEnet (617) 861-9764, BYTE magazine           *
*       to be compatible with Lightspeed C.                                      *
*                                                                                *
*********************************************************************************/

#include "stdio.h"                      /* ADDED */
#include "unix.h"                       /* ADDED */
#include "sane.h"                       /* ADDED */
#include "math.h"                       /* ADDED */



#define NTIMES 100 /* number of times to compute Fibonacci value */
#define NUMBER 24 /* biggest one we can compute with 16 bits */


main()                  /* compute Fibonacci value */
{
   int i;
   unsigned value, fib();
   unsigned long time();                /* ADDED */
   long starttime;                      /* ADDED */
   long finishtime;                     /* ADDED */
   long nettime;                        /* ADDED */


   printf("Fibonacci\n");
   printf("%d iterations: ", NTIMES);
```

```
    starttime = time( (long *) 0);              /* ADDED */

    for (i = 1; i <= NTIMES; i++)
        value = fib(NUMBER);

    finishtime = time( (long *) 0);             /* ADDED */

    printf("Fibonacci(%d) = %u.\n", NUMBER, value);
    nettime = finishtime - starttime;           /* ADDED */
    printf("Time = %ld sec\n", nettime);        /* ADDED */

    exit(0);
}


unsigned fib(x)             /* compute Fibonacci number recursively */
int x;
{
    if (x > 2)
        return (fib(x - 1) + fib(x - 2));
    else
        return (1);
}
```

Float

```
/*************************************************************************************
*       Float benchmark in C.                                                        *
*                                                                                    *
*       Simple benchmark for testing floating point speed of c libraries             *
*       does repeated multiplications and divisions in a loop that is                *
*       large enough to make the looping time insignificant                          *
*                                                                                    *
*       Modified from posting on BYTEnet (617) 861-9764, BYTE magazine               *
*       to be compatible with Lightspeed C.                                          *
*                                                                                    *
*************************************************************************************/


#include "stdio.h"                          /* ADDED */
#include "unix.h"                           /* ADDED */
#include "sane.h"                           /* ADDED */
#include "math.h"                           /* ADDED */


#define CONST1   3.141597E0
#define CONST2   1.7839032E4
#define COUNT    10000

main()
{
        double      a, b, c;
        int         i;
        unsigned long time();               /* ADDED */
        long        starttime;              /* ADDED */
        long        finishtime;             /* ADDED */
        long        nettime;                /* ADDED */
```

```
    printf("Float\n");

    starttime = time( (long *) 0);        /* ADDED */

    a = CONST1;
    b = CONST2;
    for (i = 0; i < COUNT; ++i)
    {
            c = a * b;
            c = c / a;
            c = a * b;
            c = c / a;
            c = a * b;
            c = c / a;
            c = a * b;
            c = c / a;
            c = a * b;
            c = c / a;
            c = a * b;
            c = c / a;
            c = a * b;
            c = c / a;
    }

    finishtime = time( (long *) 0);          /* ADDED */

    nettime = finishtime - starttime;        /* ADDED */
    printf("Time = %ld sec\n", nettime);  /* ADDED */

    exit(0);

}
```

Whetstone

```
/****************************************************************************
*       Whetstone benchmark in C.                                          *
*                                                                          *
*       Used to test compiler optimization and floating point performance. *
*                                                                          *
*       Modified from posting on BYTEnet (617) 861-9764, BYTE magazine     *
*       to be compatible with Lightspeed C.                                *
*                                                                          *
****************************************************************************/

#include "sane.h"                   /* ADDED */
#include "stdio.h"                  /* ADDED */
#include "math.h"                   /* ADDED */
#include "unix.h"                   /* ADDED */

#define ITERATIONS 10

        double      xx1, xx2, xx3, xx4, x, y, z, t, t1, t2;
        double      e1[4];
        int         i, j, k, l, n1, n2, n3, n4, n6, n7, n8, n9, n10, n11;

main()
{
        unsigned long       time();         /* ADDED */
        long                starttime;      /* ADDED */
        long                finishtime;     /* ADDED */
        long                nettime;        /* ADDED */


        printf("\nWhetstone Benchmark\n\n");

        starttime = time( (long *) 0);      /* ADDED */
```

```
        /* initialize constants */
        t    =    0.499975;
        t1   =    0.50025;
        t2   =    2.0;

        /* set values of module weights */
        n1  =   0 * ITERATIONS;
        n2  =  12 * ITERATIONS;
        n3  =  14 * ITERATIONS;
        n4  = 345 * ITERATIONS;
        n6  = 210 * ITERATIONS;
        n7  =  32 * ITERATIONS;
        n8  = 899 * ITERATIONS;
        n9  = 616 * ITERATIONS;
        n10 =   0 * ITERATIONS;
        n11 =  93 * ITERATIONS;

/* MODULE 1: simple identifiers */

        xx1 = 1.0;
        xx2 = xx3 = xx4 = -1.0;

        for(i = 1; i <= n1; i += 1)
        {
                xx1 = ( xx1 + xx2 + xx3 - xx4 ) * t;
                xx2 = ( xx1 + xx2 - xx3 - xx4 ) * t;
                xx3 = ( xx1 - xx2 + xx3 + xx4 ) * t;
                xx4 = (-xx1 + xx2 + xx3 + xx4 ) * t;
        }

/* MODULE 2: array elements */
```

```
        e1[0] =  1.0;
        e1[1] = e1[2] = e1[3] = -1.0;

        for (i = 1; i <= n2; i +=1)
        {
                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 */

        for (i = 1; i <= n3; i += 1)
                pa(e1);

/* MODULE 4:  conditional jumps */

        j = 1;
        for (i = 1; i <= n4; i += 1)
        {
                if (j == 1)
                        j = 2;
                else
                        j = 3;
                if (j > 2)
                        j = 0;
                else
                        j = 1;
                if (j < 1 )
                        j = 1;
                else
                        j = 0;
        }
```

```
/* MODULE 5:  omitted */

/* MODULE 6:  integer arithmetic */

        j = 1;
        k = 2;
        l = 3;

        for (i = 1; i <= n6; i += 1)
        {
                j = j * (k - j) * (l -k);
                k = l * k - (l - j) * k;
                l = (l - k) * (k + j);
                e1[l - 2] = j + k + l;          /* C arrays are zero based */
                e1[k - 2] = j * k * l;
        }

/* MODULE 7:  trig. functions */

        x = y = 0.5;
        for(i = 1; i <= n7; i +=1)
        {
                x = t * atan(t2*sin(x)*cos(x)/(cos(x+y)+cos(x-y)-1.0));
                y = t * atan(t2*sin(y)*cos(y)/(cos(x+y)+cos(x-y)-1.0));
        }

/* MODULE 8:  procedure calls */

        x = y = z = 1.0;
        for (i = 1; i <= n8; i +=1)
                p3(x, y, &z);

/* MODULE9:  array references */

        j = 1;
```

```
        k = 2;
        l = 3;

        e1[0] = 1.0;
        e1[1] = 2.0;
        e1[2] = 3.0;

        for(i = 1; i <= n9; i += 1)
                p0();

/* MODULE10:  integer arithmetic */

        j = 2;
        k = 3;
        for(i = 1; i <= n10; i +=1)
        {
                j = j + k;
                k = j + k;
                j = k - j;
                k = k - j - j;
        }

/* MODULE11:  standard functions */

        x = 0.75;
        for(i = 1; i <= n11; i +=1)
                x = sqrt( exp( log(x) / t1));

        finishtime = time( (long *) 0);      /* ADDED */
        nettime = finishtime - starttime;    /* ADDED */


        printf("\nWhetstone runs in %ld seconds. %8.f whets/second\n",
                nettime, 100000.0*ITERATIONS/nettime);    /* MODIFIED */
```

```
	exit(0);
}

pa(e)
double		e[4];
{
	register int	j;

	j = 0;
lab:
	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;
	j += 1;
	if (j < 6)
		goto lab;
}


p3(x, y, z)
double x, y, *z;
{
	x  = t * (x + y);
	y  = t * (x + y);
	*z = (x + y) /t2;
}

p0()
{
	e1[j] = e1[k];
	e1[k] = e1[l];
	e1[l] = e1[j];
}
```

Savage

```
/***************************************************************************************
*      Savage benchmark in C.                                                          *
*                                                                                      *
*      Floating point speed and accuracy test.                                         *
*                                                                                      *
*      Modified from posting on BYTEnet (617) 861-9764, BYTE magazine                  *
*      to be compatible with Lightspeed C.                                             *
*                                                                                      *
***************************************************************************************/

#include "stdio.h"                          /* ADDED */
#include "unix.h"                           /* ADDED */
#include "sane.h"                           /* ADDED */
#include "math.h"                           /* ADDED */


#define     ILOOP       10000


main()

{
        int i;
        double a;
        unsigned long time();               /* ADDED */
        long starttime;                     /* ADDED */
        long finishtime;                    /* ADDED */
        long nettime;                       /* ADDED */
```

```
    printf("Savage\n");

    starttime = time( (long *) 0);         /* ADDED */

    a = 1.0;
    for (i = 1; i <= (ILOOP - 1); i++)
     a = tan(atan(exp(log(sqrt(a*a))))) + 1.0;

    finishtime = time( (long *) 0);        /* ADDED */

    nettime = finishtime - starttime;      /* ADDED */
    printf("Time = %ld sec\n", nettime);   /* ADDED */

    printf("a = %20.14e\n", a);
    exit(0);
}
```

Eratosthenes

```
/***************************************************************************************
*       Eratosthenes sieve prime number benchmark in C.                                *
*                                                                                      *
*       Modified from posting on BYTEnet (617) 861-9764, BYTE magazine                 *
*       to be compatible with Lightspeed C.                                            *
*                                                                                      *
***************************************************************************************/


#include "stdio.h"                      /* ADDED */
#include "unix.h"                       /* ADDED */
#include "sane.h"                       /* ADDED */
#include "math.h"                       /* ADDED */


#define TRUE   1
#define FALSE  0
#define size  8190

   char flags [size + 1];

main()
{
        int i, prime, k, count, iter;
       unsigned long time();            /* ADDED */
       long starttime;                  /* ADDED */
       long finishtime;                 /* ADDED */
       long nettime;                    /* ADDED */


       printf("Eratosthenes\n");

       printf ("500 iterations\n");
```

```
    starttime = time( (long *) 0);          /* ADDED */

    for (iter = 1; iter <= 500; iter++)          /* do program 500 times */
    {
            count = 0;                          /* prime counter */
            for (i = 0; i <= size; i++)          /* set all flags true */
                    flags [i] = TRUE;
            for (i = 0; i <= size; i++)
            {
                    if (flags [i])              /* found a prime */
                    {
                            prime = i + i + 3;          /* twice index + 3 */
                            for (k = i + prime; k <= size; k+= prime)
                                    flags [k] = FALSE;  /* kill all multiple */
                            count++;                    /* primes found */
                    }
            }
    }

    finishtime = time( (long *) 0);         /* ADDED */
    printf ("%d primes.\n", count);     /* primes found on 10th pass */


    nettime = finishtime - starttime;       /* ADDED */
    printf("Time = %ld sec\n", nettime);  /* ADDED */

    exit(0);

}
```

Fast Fourier Transform

```
/*****************************************************************************
 *   FFT benchmark in C.                                                     *
 *                                                                           *
 *    Fast Fourier Transform.                                                *
 *                                                                           *
 *****************************************************************************/

#include "stdio.h"
#include "unix.h"
#include "sane.h"
#include "math.h"


main()                          /* compute FFT */
{

        long i,nn,isign;
        float *data;

        unsigned long time();
        long starttime;
        long finishtime;
        long nettime;



        /* allocate array structure */
        data =(float *)NewPtr(sizeof(float)*32769);

        for (i = 1; i <= 16384; i += 2)
        {
                data[i] = sin((i*3.0/1000.0));
```

```
        data[i+1] = cos((i*3.0/2000.0));
};

starttime = time( (long *) 0);
fft(data,(long)512,(long)1);
finishtime = time( (long *) 0);
nettime = finishtime - starttime;
printf("FFT time for 512 points = %ld sec\n", nettime);


starttime = time( (long *) 0);
fft(data,(long)1024,(long)1);
finishtime = time( (long *) 0);
nettime = finishtime - starttime;
printf("FFT time for 1024 points = %ld sec\n", nettime);


starttime = time( (long *) 0);
fft(data,(long)2048,(long)1);
finishtime = time( (long *) 0);
nettime = finishtime - starttime;
printf("FFT time for 2048 points = %ld sec\n", nettime);


starttime = time( (long *) 0);
fft(data,(long)4096,(long)1);
finishtime = time( (long *) 0);
nettime = finishtime - starttime;
printf("FFT time for 4096 points = %ld sec\n", nettime);


starttime = time( (long *) 0);
fft(data,(long)8192,(long)1);
finishtime = time( (long *) 0);
nettime = finishtime - starttime;
```

```
	printf("FFT time for 8192 points = %ld sec\n", nettime);


	starttime = time( (long *) 0);
	fft(data,(long)16384,(long)1);
	finishtime = time( (long *) 0);
	nettime = finishtime - starttime;
	printf("FFT time for 16384 points = %ld sec\n", nettime);


	exit(0);
}


/************************************************************************************
*                                                                                   *
*	fft - Fast Fourier Transform                                                    *
*                                                                                   *
*************************************************************************************
*                                                                                   *
*	This subroutine is based upon a FORTRAN routine in:                             *
*                                                                                   *
*	Press, Flannery, Teukolsky, Vetterling, 1986,                                   *
*		"Numerical Recipes, The Art of Scientific Computing"                        *
*		(Cambridge)                                                                 *
*                                                                                   *
*                                                                                   *
************************************************************************************/


fft(data,nn,isign)
float *data;
long nn;
long isign;
```

```
{
	long			n,mmax,istep;
	double			wr, wi,wpr,wpi,wtemp,theta,theta_prod,tempr,tempi;
	register long		i,m,j;
	register float *datai, *dataj;


	n = nn << 1;
	j=1;

	for (i = 1, datai = data; i <= n; i++, i++, datai++, datai++)
	{
		if (j > i)
		{
			dataj = data + j;
			tempr = *dataj;
			tempi = *(dataj+1);
			*dataj = *datai;
			*(dataj+1) = *(datai+1);
			*datai = tempr;
			*(datai+1) = tempi;
		};
		m = n >> 1;
	  while ((m >= 2) && (j > m))
		{
			j -= m;
			m >>= 1;
		};
			j += m;
	};

/*	This is the Danielson-Lanczos section	*/
	mmax = 2;
	theta_prod = 6.28318530717959/isign;
	while (n > mmax)
```

```
    {
        istep = mmax << 1;
        theta = theta_prod / mmax;
        wpr = -2.0*ldexp(sin((0.5*theta)), (int) 2);
        wpi = sin(theta);
        wr = 1.0;
        wi = 0.0;
        for (m = 1; m <= mmax; m++, m++)
        {
            for (i = m, datai = data; i <= n; i += istep, datai += istep)
            {
                dataj = data + i + mmax;
                tempr = wr * *dataj - wi * *(dataj+1);
                tempi = wr * *(dataj+1) + wi * *dataj;
                *dataj = *datai - tempr;
                *(dataj+1) = *(datai+1) - tempi;
                *datai += tempr;
                *(datai+1) += tempi;
            };
            wtemp = wr;
            wr += wr*wpr - wi*wpi;
            wi += wi*wpr + wtemp*wpi;
        };
        mmax = istep;
    };
}
```

BENCHMARKING EXECUTION IMPROVEMENTS OF AN APPLE MACINTOSH COMPUTER UPGRADED WITH A MATH COPROCESSOR

by

ROBERT WAYNE MOSS

B.S., Kansas State University, 1985

AN ABSTRACT OF A MASTERS REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

This report covers the upgrade of an Apple MacIntosh Plus computer system with a Motorola 68881 math coprocessor, a faster processor and memory without video wait states. The MacMemory TurboMax product was used to perform the upgrade on the MacIntosh. The resulting speed improvements were benchmarked and the results were analyzed.

It was determined that doubling the processor speed alone produced little increase in speed. The speed improvements became significant when the faster processor was coupled to memory that did not have the video wait state overhead. By upgrading the MacIntosh with the math coprocessor, benchmarks involving many floating point calculations demonstrated an effective reduction in execution time of up to 98%.