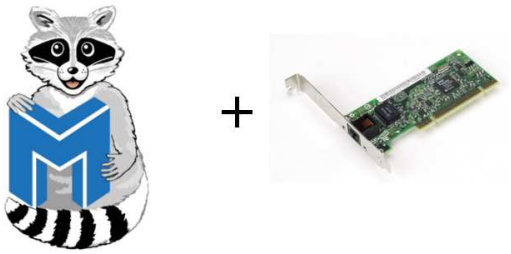
Implementing the Intel Pro/1000 on MINIX 3



Author:
Niek Q. Linnenbank

Supervisor:
Andrew S. Tanenbaum

Reviewed by:
Jorrit N. Herder

December 12, 2009

1843370

Abstract

This document describes the implementation and performance results of the Intel Pro/1000 driver for the MINIX 3 Operating System. In the first section we briefly describe the Intel Pro/1000 family of gigabit network adapters. In the second section we discuss the development environment we used, and the programs we developed. The third section describes the various performance measurements we executed and their results. Finally, the last section concludes.

Contents

1 Intel Pro/1000 Family

The Intel Pro/1000 is a family of gigabit network cards developed by Intel Corporation. In the official *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual* [1], Intel describes the Intel Pro/1000 as following:

" The PCI/PCI-X Family of Gigabit Ethernet Controllers are highly integrated, high-performance Ethernet LAN devices for 1000 Mb/s, 100 Mb/s and 10 Mb/s data rates. They are optimized for LAN on Motherboard (LOM) designs, enterprise networking, and Internet appliances that use the Peripheral Component Interconnect (PCI) and PCI-X bus.

The Ethernet controllers provide an interface to the host processor by using on-chip command and status registers and a shared host memory area, set up mainly during initialization. The controllers provide a highly optimized architecture to deliver high performance and PCI/CSA/PCI-X bus efficiency. "

2 Implementation

2.1 Development Environment

To develop an initial version of the device driver program for the Intel Pro/1000 [1] on MINIX [2] I used the QEMU processor emulator [3]. QEMU is especially useful when developing operating system or any of it's components, such as a device driver, because QEMU is an open source virtualization solution. This allows operating system developers to modify the code for debugging purposes, were appropriate. Additionally, QEMU provides an internal *GNU Debugger* [4] server for active debugging using a debugger client, and is able to emulate various CPU architectures which is very useful to port components to more architectures.

Fortunately, QEMU is able to emulate a Intel Pro/1000 gigabit card with the 82540EM chipset. The ability to easily modify the QEMU source code allowed me to turn on debug *printf()* calls in the **hw/e1000.c** source file in QEMU, so I could see every read and write interaction to the emulated hardware registers of the Intel Pro/1000, incoming and outgoing network packets and interrupts. Thanks to QEMU, I was constantly aware of the entire state of the emulated card. Obviously, on real hardware this can be much more difficult.

2.2 Programs

2.2.1 e1000

The **e1000** program is the device driver for the Intel Pro/1000 on MINIX 3. As described in section 2 I initially developed the driver under the QEMU processor emulator for the 82540EM chipset. After that I tested the **e1000** with two Intel Pro/1000 cards with the 82541PI and 82567LF chipsets. The driver did not work out-of-the-box on both chipsets, and it took a fair amount time to investigate the problems and update the driver appropriately:

- **82541PI**: needs the FCS flag set for transmit descriptors.
- **82567LF**: does not support the TXQE interrupt flag (Transmit Queue Empty). Use TXDW instead.
- **82567LF**: does not support the EERD register (EEPROM read). Use SPI instead.

The **e1000** driver was committed to the official MINIX *Subversion* repository [5] in revision 5730 by Ben Gras.

2.2.2 iperf

To test the performance of the Intel Pro/1000 and optionally existing network card drivers on MINIX, I ported the **iperf(1)** network benchmarking program [6]. Iperf uses the BSD sockets interface to send or receive TCP/UDP data. It has a lot of options and flags to tune the performance tests, including packet sizes, TCP window size, number of bytes to transfer and parallel client mode.

2.2.3 ethbench

For measuring the performance of the **e1000** driver or any other network device driver directly, I wrote a simple program called **ethbench**. **etchbench** mimicks the messaging protocol used by the **inet** server to measure the network driver performance. It sends or receives a given number of bytes to/from the driver by directly sending request messages and outputs the time it took to complete.

3 Performance

3.1 Test Environment

Machine	Laptop	HP150	Turtle2	DellXPS
CPU	Intel Core2Duo 1.8Ghz	AMD Athlon64 X2 4400+ 2.3Ghz	AMD Athlon XP 1800+ 1.5Ghz	Intel Core i7 920 2.6Ghz
Memory	1G	1G	512MB	8G
Bus	32-bit PCI	32-bit PCI	32-bit PCI	32-bit PCI-e
Onboard Card	BCM4401-B0 (100Mbit)	BCM5755 (1000Mbit)	-	82567LF (1000Mbit)

Table 1: Machines used in the test environment

Table 1 describes the machines we used for performance testing. Not all machines were available for performance testing at all times, as other people needed them. Therefore each of the following performance test results also describes which machines were used.

3.2 Minix Versions

The following measurements in figure 1 are performed using a direct link between the *Turtle2* and *Laptop* machines. We used the **urlget(1)** command on various MINIX versions to download a 100MB file filled with random bytes from the **vsftpd(8)** FTP server [7] on *Laptop* to the *Turtle2*. We used the *Intel Pro/100* network driver called **fxp** and the *Intel Pro/1000* network driver, the **e1000**. The results indicate that earlier versions of Minix have better network performance compared to recent versions. Overall the **e1000** performs little faster than the **fxp** on 100Mbit links. Note that the **e1000** was developed on MINIX *3.1.4* and *3.1.5*, and is not backwards compatible to earlier versions such as *3.1.3a* due to API changes in MINIX made over time.

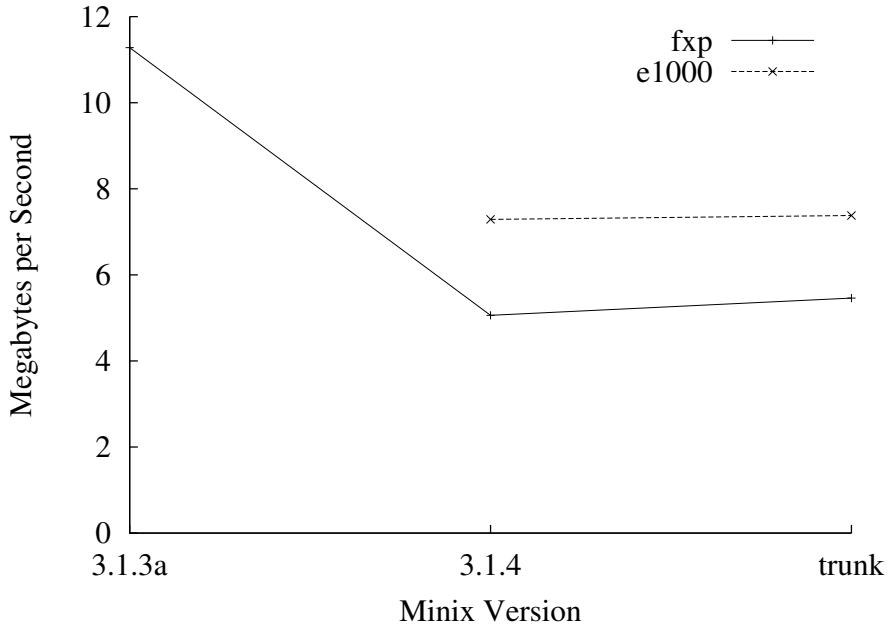


Figure 1: Performance of the urlget(1) command under various MINIX versions.

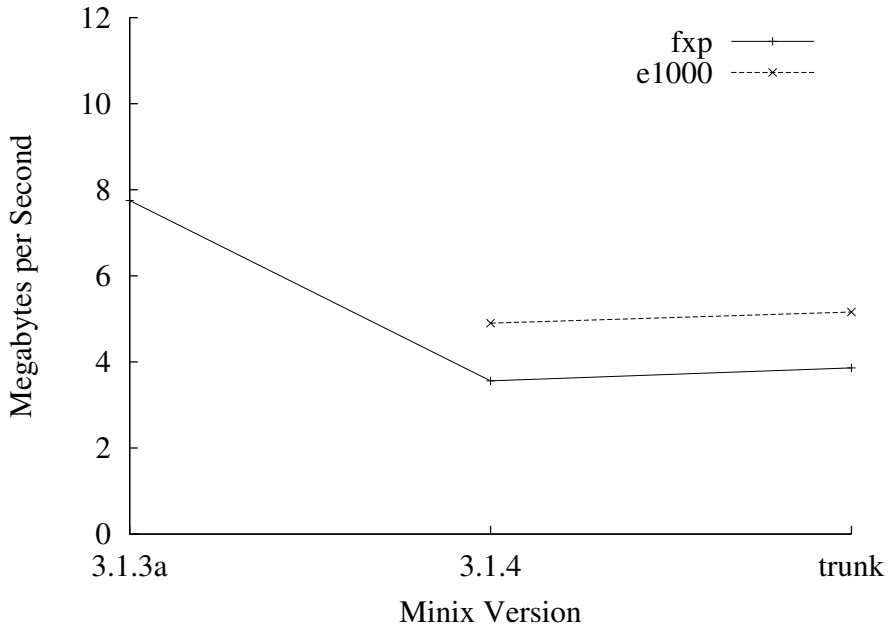


Figure 2: Performance of the ftp(1) command under various MINIX versions.

Figure 2 shows the performance results of a similar test using the **ftp(1)** command instead of **urlget(1)**. From the results we observe a slower performance when using **ftp(1)** compared to **urlget(1)**.

3.3 User Programs

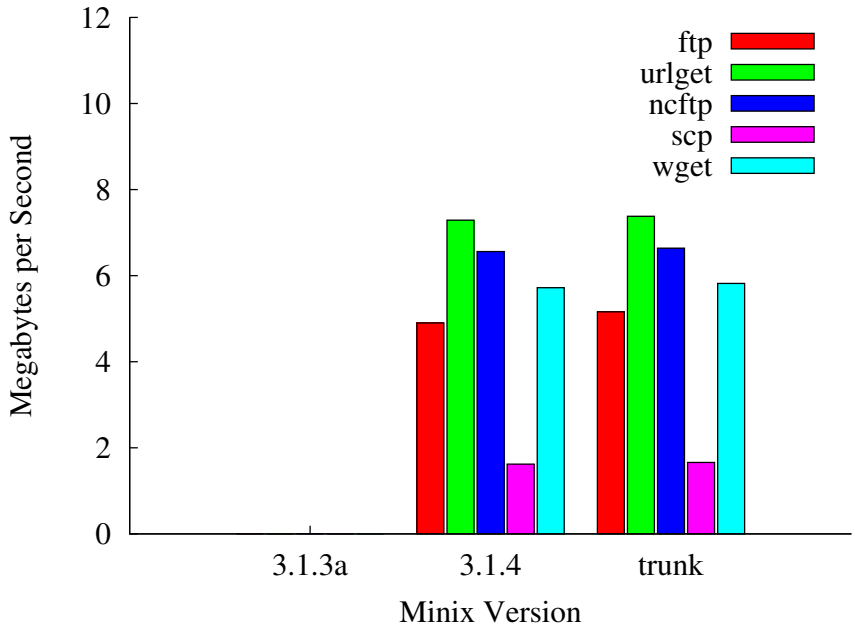


Figure 3: Performance of the Intel Pro/1000 with various user commands.

The slower performance of **ftp(1)** in figure 2 compared to **ncftp(1)** in figure 2 suggest that various user programs achieve different network performance. Figures 3 and 4 show the performance results of **ftp(1)**, **urlget(1)**, **ncftp(1)**, **wget(1)** and **scp(1)** on *Turtle2* when downloading a 100MB random file from the *Laptop* machine. The results confirm that various user programs seem to have different network performance on Minix. The fastest program overall is urlget(1), and is also the only one to achieve the full 100Mbit on Minix 3.1.3a.

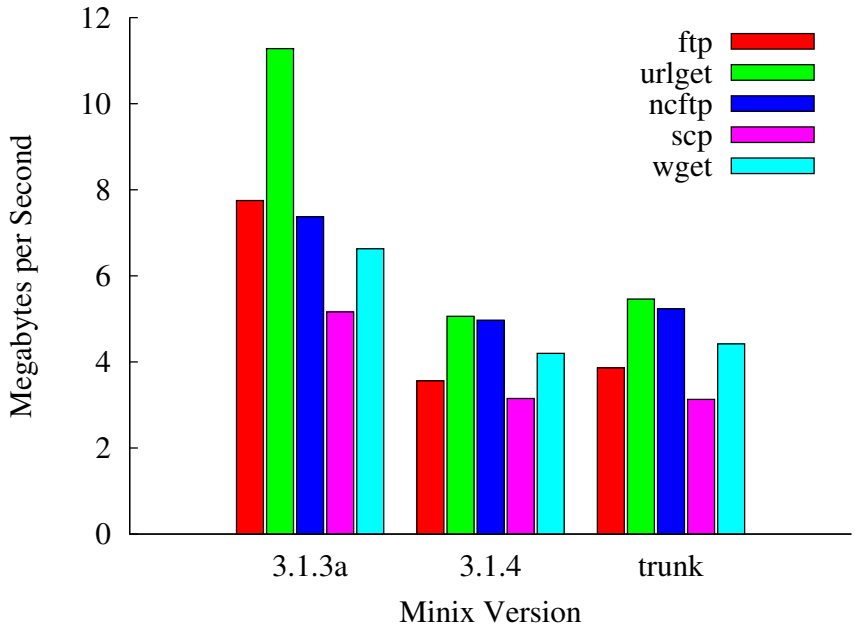


Figure 4: Performance of the Intel Pro/100 with various user commands.

Figure 4 shows similar performance results with various user programs using the **fxp** network driver compared to the **e1000** in figure 3.

3.4 Output Devices

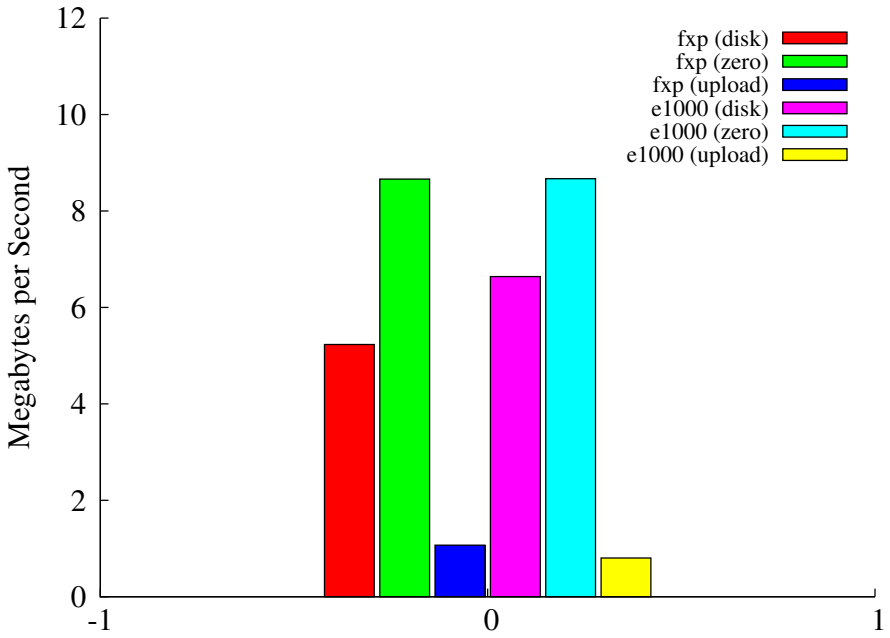


Figure 5: Performance of the ncftp(1) command.

The results in figure 5 show the performance measurements of the **ncftp(1)** command. We used the *Turtle2* with **ncftp(1)** and the *Laptop* machine with **vsftpd(8)** again. From the results in figure 5 we observe that neither the **fxp** nor the **e1000** network device driver was able to achieve the full 100Mbit. Another observation is that downloading data and simultaneously writing the received data to disk has an significant performance impact compared to throwing received data away in */dev/zero*. Also note that uploads are significantly slower than downloads.

3.5 Network Protocols

The following performance tests measure the performance of TCP and UDP traffic with various packet sizes using the **iperf(1)** network benchmarking tool. CPU utilization is measured with the dhrystone benchmark framework using the **bm_exec** helper program. Also see appendixes **??** and **??** for the **dhrystone** and **bm_exec** source code. We measured the network performance of the **e1000** using a direct ethernet connection between the *Turtle2* and the *HP150* machines.
For a fair CPU utilization comparison we disabled any hardware offloading optimizations on FreeBSD and Linux, as the **e1000** driver on MINIX currently does not (nor can) implement hardware offloading optimizations. We also disabled *Symmetric Multi Processor* functionality under all systems, as **dhrystone** is not designed for multicore systems and would otherwise report false numbers. We used **nice(1)** to give **iperf(1)** a higher priority than the dhrystone process. On MINIX this did not work as expected: the **iperf(1)** process eventually received the same low priority as the dhrystone process. Therefore we had to measure the CPU and network performance seperately on MINIX.

3.5.1 TCP Uploads

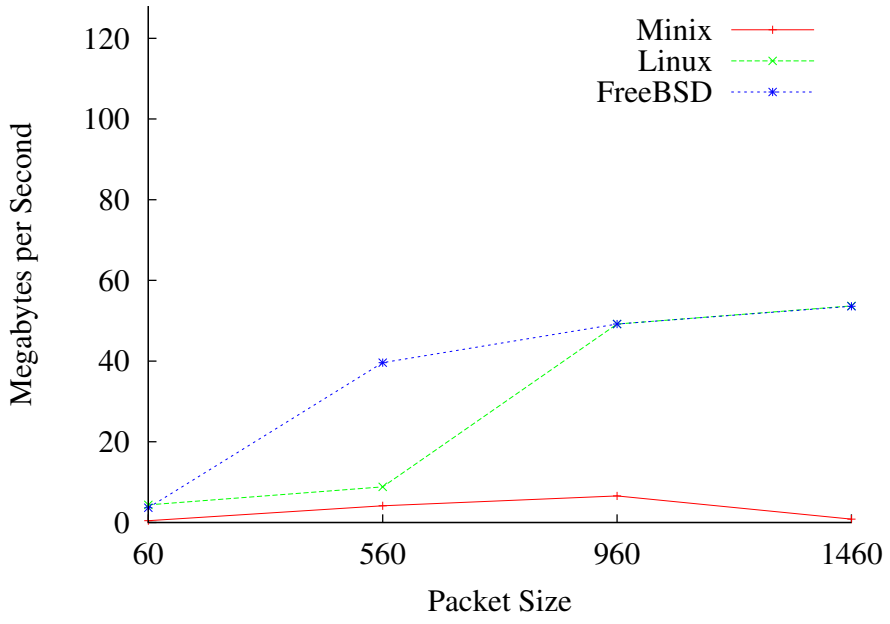


Figure 6: Performance of TCP uploads.

The results in figure 6 indicate that the maximum performance of the *Intel Pro/1000* is limited by the *PCI* bus. We experimented with the onboard *BCM5755* on the *HP150* and achieved a maximum of 90MB per second with UDP uploads. The *BCM5755* is connected to the PCI-e bus. Figure 6 also shows that TCP uploads on MINIX are very slow, even for large packet sizes. Linux and FreeBSD show similar performance results.

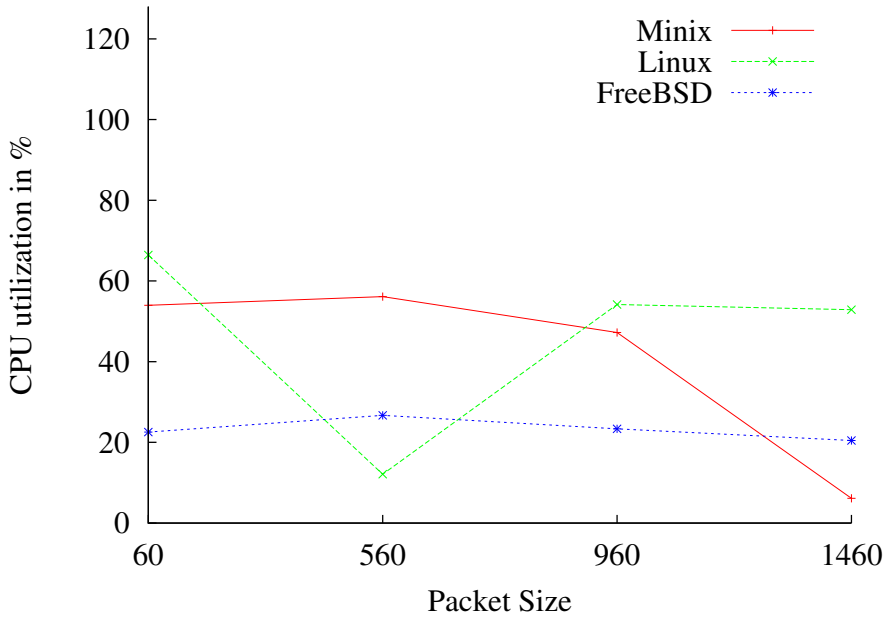


Figure 7: CPU utilization of TCP uploads.

Figure 7 shows that CPU utilization of MINIX drops for larger packet sizes. Linux and FreeBSD have a more constant CPU utilization, except for 560-byte packets under Linux.

3.5.2 TCP Downloads

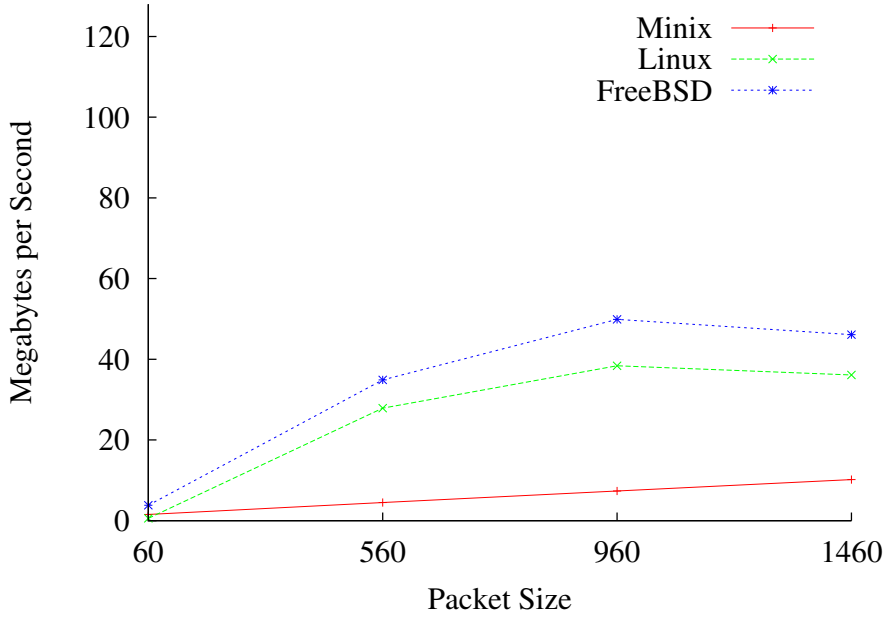


Figure 8: Performance of TCP downloads.

Figure 8 represents the performance results of TCP downloads with various packet sizes. FreeBSD achieves sligtly better performance than Linux, but MINIX does achieves far worse performance results.

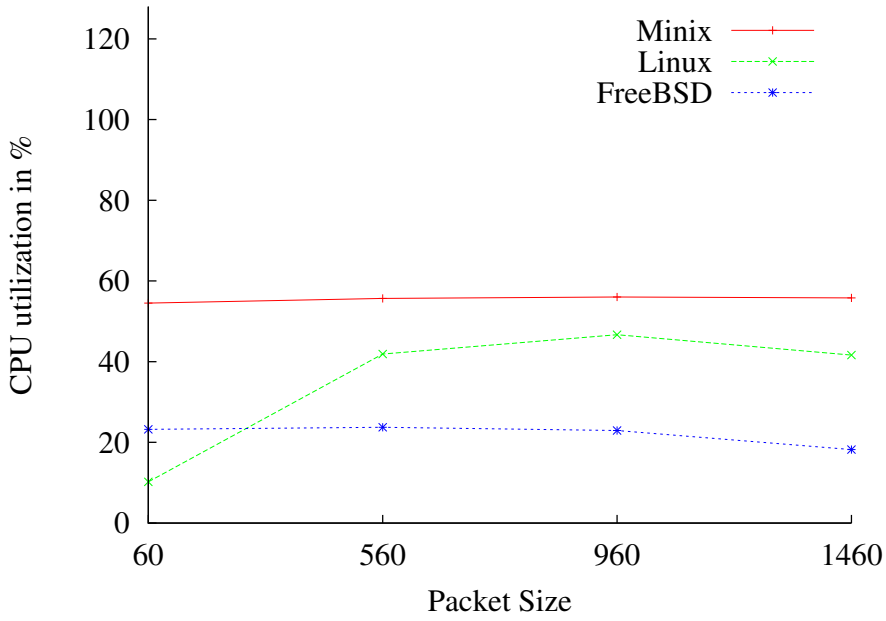


Figure 9: CPU utilization of TCP downloads.

Figure 9 illustrates the CPU utilization of TCP downloads. MINIX has the highest CPU utilization overall, and FreeBSD the least.

3.5.3 UDP Uploads

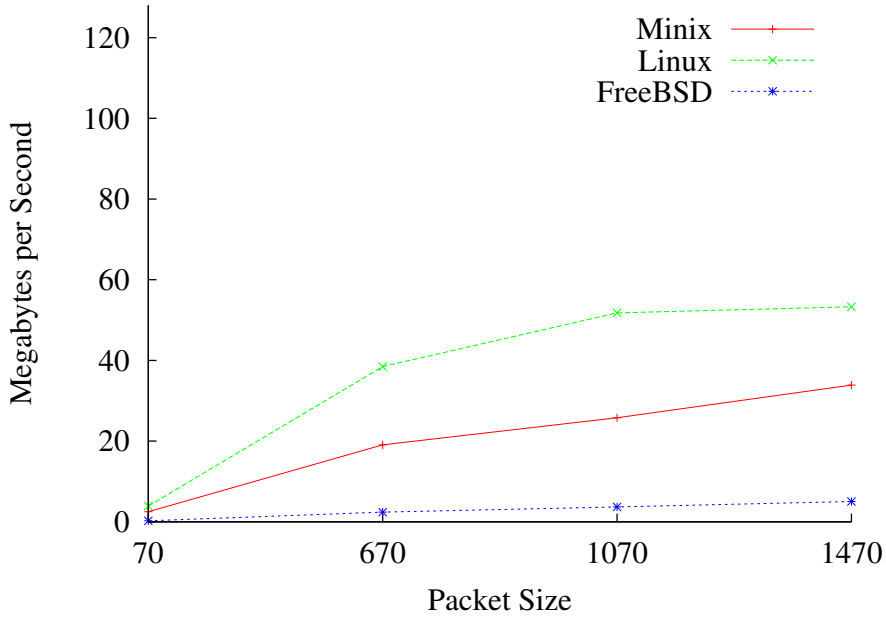


Figure 10: Performance of UDP uploads.

Figure 10 displays the performance of UDP uploads. Surpizingly MINIX is faster for UDP uploads with a maximum of 33.9 MB per second than FreeBSD with only 5.0 MB per second. It is unclear why FreeBSD has such slow performance for UDP uploads.

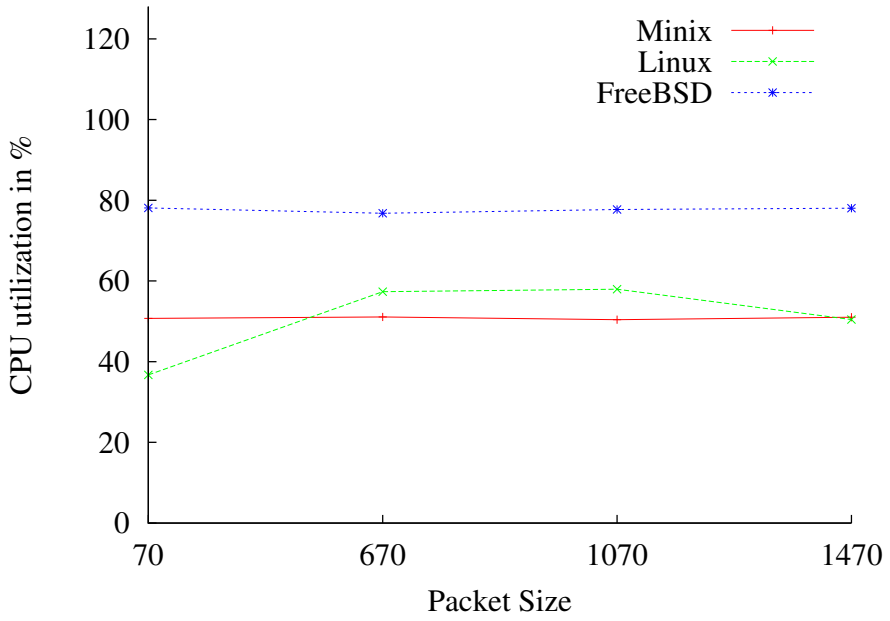


Figure 11: CPU utilization of UDP uploads.

Figure 11 shows the CPU utilization of UDP uploads. FreeBSD uses a constant 80 percent CPU followed by MINIX and Linux.

3.5.4 UDP Downloads

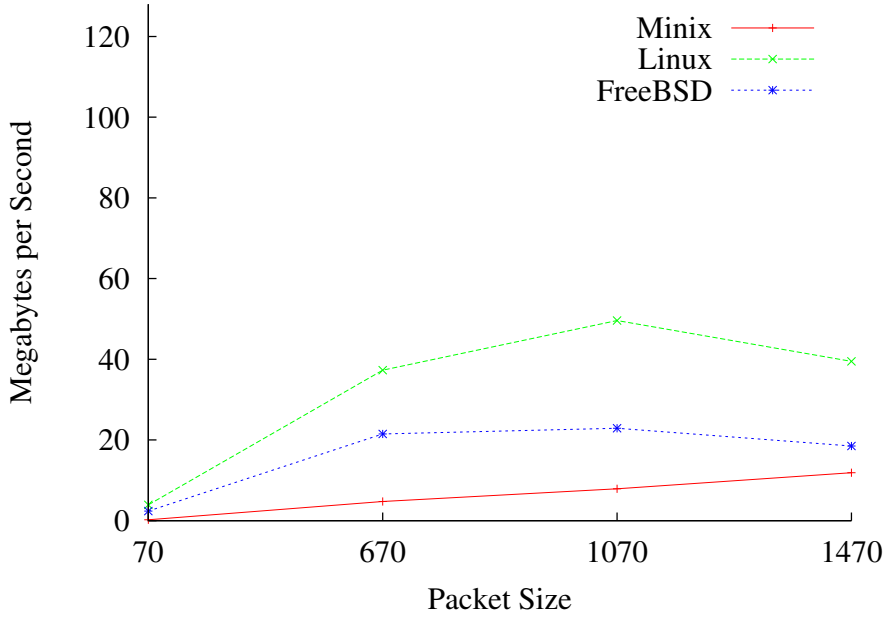


Figure 12: Performance of UDP downloads.

Figure 12 illustrates the performance of UDP downloads. Linux is the fastest followed by FreeBSD and MINIX. In constrast to the results in figure 10 FreeBSD is now faster than MINIX. MINIX only achieves a maximum of 11.6 MB per second in this case.

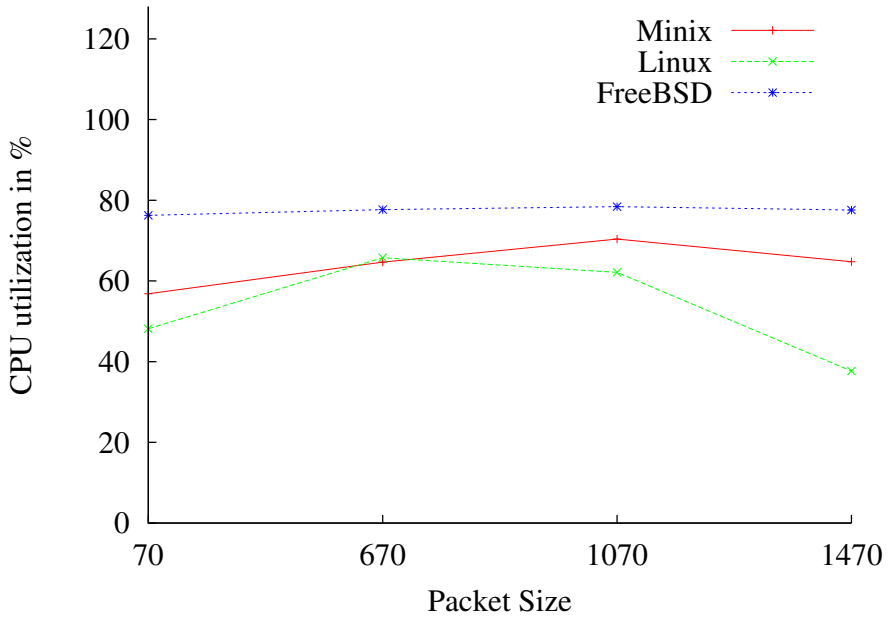


Figure 13: CPU utilization of UDP downloads.

Figure 13 shows the CPU utilization of UDP downloads. Again FreeBSD has the highest CPU usage followed by Linux and MINIX.

3.6 Direct Driver Benchmark

To directly measure the performance of the **e1000** without the **inet** server we used **ethbench** on the *DellXPS* machine connected with the *Turtle2*. The **ethbench** mimicks the protocol used by **inet** for sending and receiving ethernet packets. In the default **inet** only one packet is send and received by message. The **ethbench** program also tests the performance of sending and receiving multiple packets per message using a modified message format. Figure 14 shows the performance results of the **e1000** using the **ethbench** tool on the *DellXPS* machine for sending ethernet packets. These results show that MINIX is capable of operating at gigabit network speeds, and that handeling one packet per message is too inefficient for gigabit network drivers.

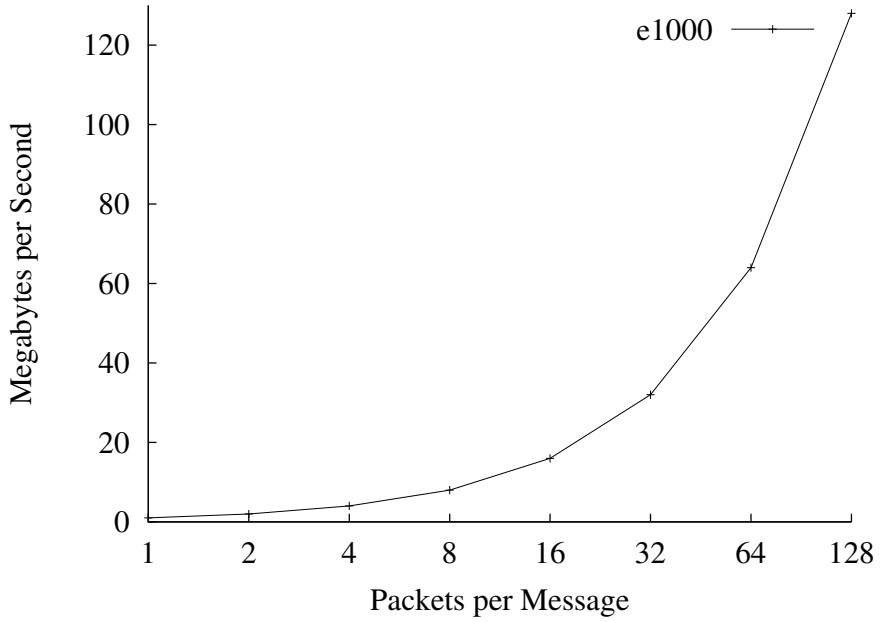


Figure 14: Performance of the e1000 measured with ethbench.

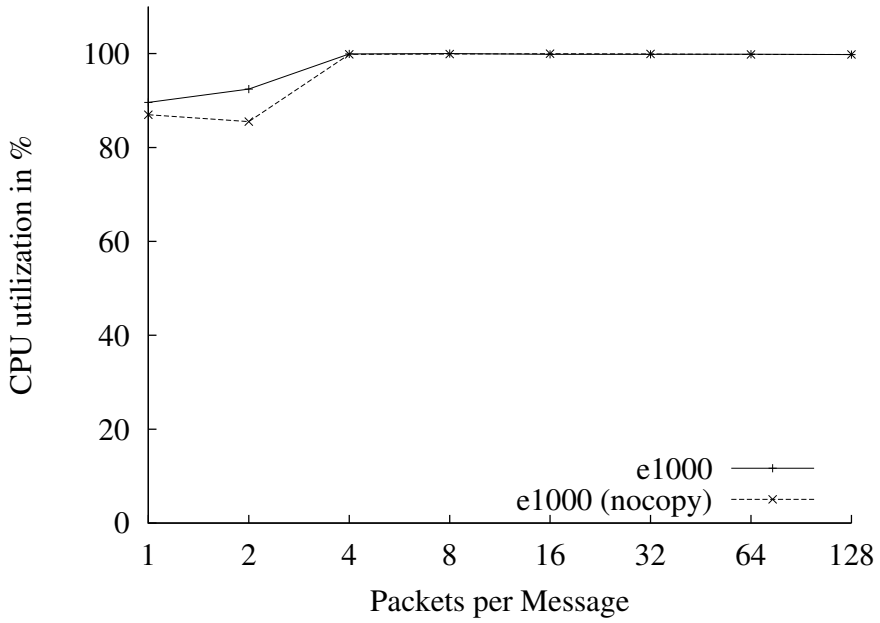


Figure 15: CPU usage of the e1000 measured with getidle().

To accurately measure the system-wide CPU utilization we used the MINIX-specific *getidle()* function. It allows a user process to read the CPU percentage used by the IDLE task, which can be used to obtain the system-wide CPU utilization. Figure 15 shows the results we measured which indicate a very high CPU utilization.One possible explaination of the high CPU usage in figure 15 could be the time needed for copying packet buffers between the **ethbench** process and the **e1000**. We tested the effect on the CPU utilization by temporarily disabling copying operations. Figure 15 shows the CPU utilization results we measured and indicates that the copying operations slightly slow the system down, but are not the main CPU bottleneck. During the tests we observed a high CPU utilization in the *SYSTEM* task, but we cannot explain this behaviour without system profiling tools.

4 Conclusion

In this project we implemented a working device driver for the Intel Pro/1000 on the MINIX 3 operating system. We measured the performance of the **e1000** using various MINIX versions, user programs, network protocols and with a direct benchmark tool called **ethbench**. When using the **inet** server, Linux and FreeBSD overall perform better than MINIX. There are at least two problems which need to be solved in the current MINIX system, to fully support gigabit networks:

- Our measurements have shown that network performance under MINIX is very slow when measuring the BSD sockets interface (TCP,UDP) through **inet**. This indicates a problem in the **inet** server, or one of it's dependancies.
- Gigabit network drivers need more than one packet per message to operate on the full 1000Mbit speed.
- MINIX has a high CPU usage when transferring data at 1000Mbit speeds.

In this project I learned a lot about developing device drivers for the MINIX operating system, network cards and measuring performance. I discovered that hardware debugging is a painful but neccessary task. I also discoved that MINIX did not have a device driver programming tutorial for new developers to get started. Fortunately for me I already had some experience in device driver programming [8], but I contributed the first tutorial on programming device drivers on MINIX at the wiki: *Programming Device Drivers in MINIX* [9]. In my experience, not only was this project educationally useful but above all it was a very fun project!

References

[1] Intel Corporation. Pci/pci-x family of gigabit ethernet controllers software developer's manual. download.intel.com/design/network/manuals/8254x_GBe_SDM.pdf, March 2009.

[2] Andrew S. Tanenbaum. Minix 3. http://www.minix3.org, December 2009.

[3] Fabrice Bellard. Qemu emulator. http://bellard.org/qemu/, December 2009.

[4] Free Software Foundation. Gnu debugger. http://www.gnu.org/software/gdb/, December 2009.

[5] The MINIX Team. Minix 3 repository. https://gforge.cs.vu.nl/svn/minix/trunk/src, December 2009.

[6] The IPerf Team. Iperf. http://sf.net/projects/iperf, December 2009.

[7] The vsftpd Team. Secure fast ftp server for unix like systems. http://vsftpd.beasts.org/, December 2009.

[8] Niek Linnenbank. Free niek's operating system. http://www.freenos.org/, December 2009.

[9] Niek Linnenbank. Programming device drivers in minix. http://wiki.minix3.org/en/DevelopersGuide/DriverProgramming, December 2009.

A Ethbench Source Code

```
/**
 * Ethernet Driver Benchmark.
 *
 * @author Niek Linnenbank
 * @date November 2009
 */

#define _SYSTEM
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <minix/sysutil.h>
#include <minix/ds.h>
#include <minix/com.h>
#include <minix/ipc.h>
#include <minix/safecopies.h>
#include <minix/endpoint.h>

static char packet[1460];
static cp_grant_id_t vec_grant, io_grant;
static iovec_s_t iovec;
static int reading     = 0;
static int packet_size = sizeof(packet);
static u32_t tasknr    = 0;
static char *prog      = NULL;
static int bytes       = 0;
static u32_t this_proc = 0;

void next_packet(void)
{
    message msg;

    /* Fill in message. */
    msg.m_type = reading ? DL_READV_S : DL_WRITEV_S;
    msg.DL_PORT  = 0;
    msg.DL_COUNT = 1;
    msg.DL_GRANT = vec_grant;
    msg.DL_PROC  = this_proc;
    msg.DL_MODE  = DL_NOMODE;

    /* Attempt to send the I/O request. */
    if (asynsend(tasknr, &msg) != OK)
    {
        printf("%s: failed to send() to %d: %s\n",
                prog, tasknr, strerror(errno));
        exit(EXIT_FAILURE);
    }
}
```

```
int main(int argc, char **argv)
{
    message msg;
    int maximum = 1024*1024;
    struct timeval t1, t2;
    u32_t t;

    /* Check command-line arguments. */
    if (argc < 3)
    {
        printf("usage: %s DRIVER read|write [COUNT] [SIZE]\n");
        return EXIT_FAILURE;
    }
    else if (argc >= 4)
    {
        maximum = atoi(argv[3]);
    }
    if (argc >= 5)
    {
        if ((packet_size = atoi(argv[4])) > sizeof(packet))
        {
            printf("%s: SIZE to large. %d maximum.\n",
                    argv[0], sizeof(packet));
            return EXIT_FAILURE;
        }
    }
    prog    = argv[0];
    reading = argv[2][0] == 'r';

    /*
     * Lookup the driver task number.
     */
    if ((ds_retrieve_u32(argv[1], &tasknr)) != OK)
    {
        printf("%s: failed to lookup task %s\n", argv[0], argv[1]);
        return EXIT_FAILURE;
    }
    /* Lookup our own task number. */
    if ((ds_retrieve_u32("ethbench", &this_proc)) != OK)
    {
        printf("%s: failed to lookup ourselves\n", argv[0]);
        return EXIT_FAILURE;
    }
    printf("%s: %s is at %d. we are at %d\n",
            argv[0], argv[1], tasknr, this_proc);

    /* Allocate grants. */
    vec_grant = cpf_grant_direct(tasknr, (vir_bytes) &iovec,
                                 sizeof(iovec), CPF_READ|CPF_WRITE);
    io_grant  = cpf_grant_direct(tasknr, (vir_bytes) packet,
                                 packet_size, CPF_READ|CPF_WRITE);
    printf("%s: grants: %u / %u\n", prog, vec_grant, io_grant);

    /* Clear packet. */
```

```
memset(packet, 0, packet_size);

/* Fill I/O vector. */
iovec.iov_size  = packet_size;
iovec.iov_grant = io_grant;

/* Initialize the card. */
msg.DL_PORT = 0;
msg.DL_PROC = this_proc;
msg.m_type  = DL_CONF;
send(tasknr, &msg);

/* Measure #1. */
gettimeofday(&t1, NULL);

/*
 * Enter the main loop.
 */
while (1)
{
    /* Wait for a new message. */
    if (receive(ANY, &msg) != OK)
    {
        printf("%s: receive() failed: %s\n", prog, strerror(errno));
        return EXIT_FAILURE;
    }
    /* Check for notifications. */
    if (is_notify(msg.m_type))
    {
        switch (_ENDPOINT_P(msg.m_source))
        {
            case RS_PROC_NR:
                notify(msg.m_source);

            case PM_PROC_NR:
                exit(EXIT_SUCCESS);

            default:
                break;
        }
        continue;
    }
    /* Check for normal message. */
    switch (msg.m_type)
    {
        case DL_TASK_REPLY:
            if ((msg.DL_STAT & DL_PACK_SEND) ||
                (msg.DL_STAT & DL_PACK_RECV))
            {
                if (bytes < maximum)
                {
                bytes += packet_size;
                next_packet();
                }
```

```
                    else break;
                }
                continue;

            case DL_CONF_REPLY:
                printf("%s: %s initialized.\n", prog, argv[1]);

                /* Send initial I/O request. */
                next_packet();
                continue;
        }
        /* Measure #2. */
        gettimeofday(&t2, NULL);

        /* Calculate the total time in microseconds */
        t = ((t2.tv_sec  - t1.tv_sec ) * 1000000) +
             (t2.tv_usec - t1.tv_usec);

        /* Done. */
        printf("%s: %d bytes transferred in %u.%u seconds\n", prog,
                bytes, t / 1000000, t % 1000000);
    }
    return 0;
}
```

B Dhrystone Source Code

```
/* dhrystone - benchmark program */

#define REGISTER
/*
 *
 * "DHRYSTONE" Benchmark Program
 *
 * Version: C/1.1, 12/01/84
 *
 * Date: PROGRAM updated 01/06/86, COMMENTS changed 01/31/87
 *
 * Author: Reinhold P. Weicker,  CACM Vol 27, No 10, 10/84 pg.1013
 * Translated from ADA by Rick Richardson
 * Every method to preserve ADA-likeness has been used,
 * at the expense of C-ness.
 *
 * Compile: cc -O dry.c -o drynr : No registers
 * cc -O -DREG=register dry.c -o dryr : Registers
 *
 * Defines: Defines are provided for old C compiler's
 * which don't have enums, and can't assign structures.
 * The time(2) function is library dependant; Most
 * return the time in seconds, but beware of some, like
 * Aztec C, which return other units.
 * The LOOPS define is initially set for 50000 loops.
 * If you have a machine with large integers and is
 * very fast, please change this number to 500000 to
 * get better accuracy.  Please select the way to
 * measure the execution time using the TIME define.
 * For single user machines, time(2) is adequate. For
 * multi-user machines where you cannot get single-user
 * access, use the times(2) function.  Be careful to
 * adjust the HZ parameter below for the units which
 * are returned by your times(2) function.  You can
 * sometimes find this in <sys/param.h>.  If you have
 * neither time(2) nor times(2), use a stopwatch in
 * the dead of the night.
 * Use a "printf" at the point marked "start timer"
 * to begin your timings. DO NOT use the UNIX "time(1)"
 * command, as this will measure the total time to
 * run this program, which will (erroneously) include
 * the time to malloc(3) storage and to compute the
 * time it takes to do nothing.
 *
 * Run: drynr; dryr
 *
 * Results: If you get any new machine/OS results, please send to:
 *
 * ihnp4!castor!pcrat!rick
 *
 * and thanks to all that do.
 *
```

```
 * Note: I order the list in increasing performance of the
 * "with registers" benchmark.  If the compiler doesn't
 * provide register variables, then the benchmark
 * is the same for both REG and NOREG.
 *
 * PLEASE: Send complete information about the machine type,
 * clock speed, OS and C manufacturer/version.  If
 * the machine is modified, tell me what was done.
 * On UNIX, execute uname -a and cc -V to get this info.
 *
 * 80x8x NOTE: 80x8x benchers: please try to do all memory models
 * for a particular compiler.
 *
 *
 * 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 meaningfull, but it is
 * syntactically and semantically correct.
 *
 */


#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

/* Accuracy of timings and human fatigue controlled by next two lines */
/*#define LOOPS 50000 */ /* Use this for slow or 16 bit machines */
/*#define LOOPS 500000 */ /* Use this for faster machines */
/*#define LOOPS (sizeof(int) == 2 ? 50000 : 1000000)*/

/* Seconds to run */
#define SECONDS 15

/* Compiler dependent options */
#define NOENUM /* Define if compiler has no enum's */
```

```
/* #define NOSTRUCTASSIGN */ /* Define if compiler can't assign structures*/


/* Define only one of the next two defines */
#define TIMES /* Use times(2) time function */
/*#define TIME */ /* Use time(2) time function */


#ifdef TIME
/* Ganularity of time(2) is of course 1 second */
#define HZ 1
#endif

#ifdef TIMES
/* Define the granularity of your times(2) function */
/*#define HZ 50 */ /* times(2) returns 1/50 second (europe?) */
/*#define HZ 60 */ /* times(2) returns 1/60 second (most) */
/*#define HZ 100  */ /* times(2) returns 1/100 second (WECo) */
#endif

/* For compatibility with goofed up version */
/*#undef GOOF */ /* Define if you want the goofed up version */


#ifdef GOOF
char Version[] = "1.0";
#else
char Version[] = "1.1";
#endif


#ifdef NOSTRUCTASSIGN
#define structassign(d, s) memcpy(&(d), &(s), sizeof(d))
#else
#define structassign(d, s) d = s
#endif


#ifdef NOENUM
#define Ident1 1
#define Ident2 2
#define Ident3 3
#define Ident4 4
#define Ident5 5
typedef int Enumeration;
#else
typedef enum {
  Ident1, Ident2, Ident3, Ident4, Ident5
} Enumeration;
#endif

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;

#ifdef NULL
#undef NULL
#endif

#define NULL 0
#define TRUE 1
#define FALSE 0

#ifndef REG
#define REG
#endif


#ifdef TIMES
#include <sys/times.h>
#endif

#ifndef _PROTOTYPE
#define _PROTOTYPE(fun, args) fun args
#endif

_PROTOTYPE(int main, (int, char **));
_PROTOTYPE(void prep_timer, (void));
_PROTOTYPE(void timeout, (int sig));
_PROTOTYPE(void Proc0, (int));
_PROTOTYPE(void Proc1, (RecordPtr PtrParIn));
_PROTOTYPE(void Proc2, (OneToFifty *IntParIO));
_PROTOTYPE(void Proc3, (RecordPtr *PtrParOut));
_PROTOTYPE(void Proc4, (void));
_PROTOTYPE(void Proc5, (void));
_PROTOTYPE(void Proc6, (Enumeration EnumParIn, Enumeration *EnumParOut));
_PROTOTYPE(void Proc7, (OneToFifty IntParI1, OneToFifty IntParI2,
OneToFifty *IntParOut));
_PROTOTYPE(void Proc8, (Array1Dim Array1Par, Array2Dim Array2Par,
OneToFifty IntParI1, OneToFifty IntParI2));
/*_PROTOTYPE(Enumeration Func1,(CapitalLetter CharPar1, CapitalLetter CharPar2));*/
_PROTOTYPE(boolean Func2, (String30 StrParI1, String30 StrParI2));
_PROTOTYPE(boolean Func3, (Enumeration EnumParIn));
```

```
_PROTOTYPE(Enumeration Func1, (int CharPar1, int CharPar2));


int main(int argc, char **argv)
{
  int sec = SECONDS;

  /* Allow user set the seconds, all other arguments are ignored simply */
  if (argc >= 2) {
    if ((sec = atoi (argv[1])) < 0) {
      fprintf (stderr, "Usage: dhyr [seconds]\n");
      return (1);
    }
  }

  Proc0(sec);
  return(0);
}


#if __STDC__
volatile int done;
#else
int done;
#endif

void prep_timer()
{
  signal(SIGALRM, timeout);
  done = 0;
}

void timeout(sig)
int sig;
{
  done = 1;
}

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


void Proc0(int sec)
{
  OneToFifty IntLoc1;
  REG OneToFifty IntLoc2;
```

```
  OneToFifty IntLoc3;
  REG char CharIndex;
  Enumeration EnumLoc;
  String30 String1Loc;
  String30 String2Loc;
  register unsigned long i;
  unsigned long starttime;
  unsigned long benchtime;
  unsigned long nulltime;
  unsigned long nullloops;
  unsigned long benchloops;
  unsigned long ticks_per_sec;
#ifdef TIMES
  struct tms tms;
#endif

#ifdef HZ
#define ticks_per_sec HZ
#else
  ticks_per_sec = sysconf(_SC_CLK_TCK);
#endif

  i = 0;
  prep_timer();

#ifdef TIME
  starttime = time((long *) 0);
#endif

#ifdef TIMES
  times(&tms);
  starttime = tms.tms_utime;
#endif

  alarm(1);
  while (!done) i++;

#ifdef TIME
  nulltime = time((long *) 0) - starttime; /* Computes o'head of loop */
#endif

#ifdef TIMES
  times(&tms);
  nulltime = tms.tms_utime - starttime; /* Computes overhead of looping */
#endif

  nullloops = i;


  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");
#ifndef GOOF
  strcpy(String1Loc, "DHRYSTONE PROGRAM, 1'ST STRING"); /* GOOF */
#endif

  Array2Glob[8][7] = 10; /* Was missing in published program */


/*****************
-- Start Timer --
*****************/
  i = 0;
  prep_timer();

#ifdef TIME
  starttime = time((long *) 0);
#endif

#ifdef TIMES
  times(&tms);
  starttime = tms.tms_utime;
#endif

  /* Original timer */
  /* alarm(SECONDS); */
  /* We use seconds specified by the user */
  alarm (sec);
  while (!done) {
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 --
*****************/


#ifdef TIME
  benchtime = time((long *) 0) - starttime;
#endif

#ifdef TIMES
  times(&tms);
  benchtime = tms.tms_utime - starttime;
#endif
  benchloops = i;

  /* Approximately correct benchtime to the nulltime. */
  benchtime -= nulltime / (nullloops / benchloops);

  /* We only use one value */
#ifdef _VU_BM_PROJECT_
  {
    /*
    unsigned long stones_per_second = benchloops * ticks_per_sec / benchtime ;
    write (STDOUT_FILENO, &stones_per_second, sizeof (unsigned long));
    */
    write (STDOUT_FILENO, &benchloops, sizeof (unsigned long));
  }
#else
  printf("Dhrystone(%s) time for %lu passes = %lu.%02lu\n",
Version,
benchloops, benchtime / ticks_per_sec,
benchtime % ticks_per_sec * 100 / ticks_per_sec);
  fprintf(stderr, "This machine benchmarks at %lu dhrystones/second\n",
benchloops * ticks_per_sec / benchtime);
#endif /* _VU_BM_PROJECT_ */
}


void Proc1(PtrParIn)
REG RecordPtr PtrParIn;
{
#define NextRecord (*(PtrParIn->PtrComp))


  structassign(NextRecord, *PtrGlb);
  PtrParIn->IntComp = 5;
  NextRecord.IntComp = PtrParIn->IntComp;
  NextRecord.PtrComp = PtrParIn->PtrComp;
  Proc3((RecordPtr *)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
}


void Proc2(IntParIO)
OneToFifty *IntParIO;
{
  REG OneToFifty IntLoc;
  REG Enumeration EnumLoc;


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


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


void Proc4()
{
  REG boolean BoolLoc;


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


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

```
void Proc6(EnumParIn, EnumParOut)
REG Enumeration EnumParIn;
REG 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;
}
}


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


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


void Proc8(Array1Par, Array2Par, IntParI1, IntParI2)
Array1Dim Array1Par;
Array2Dim Array2Par;
OneToFifty IntParI1;
OneToFifty IntParI2;
{
  REG OneToFifty IntLoc;
  REG 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;
{
  REG CapitalLetter CharLoc1;
  REG CapitalLetter CharLoc2;


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


boolean Func2(StrParI1, StrParI2)
String30 StrParI1;
String30 StrParI2;
{
  REG OneToThirty IntLoc;
  REG 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)
REG Enumeration EnumParIn;
{
  REG Enumeration EnumLoc;
```

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


#ifdef NOSTRUCTASSIGN
memcpy(d, s, l)
register char *d;
register char *s;
register int l;
{
  while (l--) *d++ = *s++;
}

#endif
```

C bm_exec Source Code

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/times.h>

#define KB * 1024
#define MB * 1024 KB
#define GB * 1024 MB
#define SECONDS 65
#define FILE_SIZE (1792 MB)
#define tick_per_second sysconf(_SC_CLK_TCK)
#define SEC_PER_EXP 60
#define NR_RUNS 1

struct time_info {
  clock_t t1;
  clock_t t2;
  double duration;
  unsigned long stones;
  unsigned long iterations;
};

struct avg_result {
  double duration;
  double throughput;
  double utilization;
  unsigned long iterations;
};

int pipe_fd[2];

static void
usage(char *prog_name)
{
    printf("Usage: %s <command>\n", prog_name);
}

static void
panic (const char *string)
{
  perror (string);

  exit (1);
```

```
}

static struct time_info
do_measurement (char *cmd, int sec)
{
  pid_t chr_pid, pid = 0;
  char number[12];
  char *argv[3];
  struct time_info t;
  struct tms tms1, tms2;
  off_t off;
  int off_index = 0;
  unsigned long io_count = 0;

  bzero (&t, sizeof (struct time_info));

  if (pipe (pipe_fd) == -1)
    panic ("Creating pipe failed");

  switch (chr_pid = fork ()) {
  case -1:
    panic ("fork () failed");

  case 0: /* Child */
    /* Close read end of the pipe */
    close (pipe_fd[0]);
    /* Redirect write end to standard output */
    dup2 (pipe_fd[1], STDOUT_FILENO);

    /* Use short length array for simplicity
     * We can control the legnth of command
     */
    snprintf(number, sizeof(number), "%d", sec);
    argv[0] = "./dryr";
    argv[1] = number;
    argv[2] = NULL;
    /* Exec dhrystone */
    if (execvp (argv[0], argv) < 0)
      panic ("Exec error");

  default: /* Parent */
    /* Close write end of the pipe */
    close (pipe_fd[1]);
    /* Redirect read end to standard input */
    dup2 (pipe_fd[0], STDIN_FILENO);

    if (cmd != NULL && !(pid = fork()))
    {
system(cmd);
        exit(0);
    }
    /* Collect the number of stones from the child */
    read (pipe_fd[0], &t.stones, sizeof (unsigned long));
    waitpid (chr_pid, NULL, 0);
```

```
    if (pid) waitpid(pid, NULL, 0);
    close (pipe_fd[0]);
  }

  return t;
}

int
main (int argc, char **argv)
{
  int sec = SECONDS;
  char *file, *op;
  unsigned long unit_size;
  char rw_flag /* read or write */, sr_flag /* sequetial or random */;
  char rb_flag /* raw or block */;
  int  fd, mode, rd_mode = O_RDONLY, wr_mode = O_WRONLY;
  struct time_info time_info_idle, time_info_io;
  struct avg_result avg_result_io;
  double rate_idle, max_stones_idle, expected, stones_io;
  double throughput, utilization;
  int i;

  if (NR_RUNS < 1) /* Kidding, run zero times */
    exit (0);

  if (argc != 2) {
    usage(argv[0]);
    exit(1);
  }

  /* Run dhrystone in an idle system */
  for (i = 0; i < 3; i++) {
    printf("measure #%d of dhrystone\n", i);

    time_info_idle = do_measurement (NULL, sec);
    max_stones_idle = time_info_idle.stones > max_stones_idle ?
                      time_info_idle.stones : max_stones_idle;
  }
  /* max_stones_idle /= i; */
  rate_idle = max_stones_idle / sec;
  printf("rate: %u\n", rate_idle);

  /* Now execute the command, with dhrystone running. */
  time_info_io = do_measurement(argv[1], sec);
  expected = rate_idle * sec;
  utilization = (expected - time_info_io.stones) / expected * 100.0;

  printf("stones: %f <-> %u rate: %f <-> %u utilization: %f\n",
  max_stones_idle, time_info_io.stones,
  rate_idle, time_info_io.stones / sec,
  utilization);

  return (0);
}
```

```
/* EOF */
```