Building Performance Measurement Tools for the MINIX 3 Operating System

Rogier Meurs

August 2006

Contents

1 INTRODUCTION

The efficiency of computer programs has been always a big theme in the history of computing. In the early days, programs were written in the machine's native instruction set to squeeze every bit of performance out of the CPU. Nowadays most programs are written in higher level languages which impose an overhead but ease the job of programming. Some languages are based on program execution by an interpreter, which slows down execution even more. However, the programming language is not the only factor influencing the speed of execution. A simple programming error or the wrong choice of data structure or sort algorithm could cause a program to execute orders of magnitudes slower then needed. To detect the existence or even find the location of such an error in a program is not always a trivial task.

1.1 MEASURING PERFORMANCE

In order to provide programmers with means to measure (at runtime) where a program spends its time, so called profilers came into existence. These profilers, which are usually integrated in the operating system or compiler, conceptually split up the code of a program into several pieces and measure how much CPU time is spent in each piece. The individual pieces can be functions, lines of code, address ranges in the program text, etc. The resulting measurements together form the 'profile' of a program, hence the name profiler.

Most measuring tools enable the profiling of user processes. When integrated into a compiler, these assume the availability of operating system functionality. For example, the Gprof functionality of the GCC compiler uses system calls for file operations to write its results to a file. Some operating systems provide a profil system call to allow basic address range profiling of user programs but not the operating system itself.

Profiling an operating system is a challenge. Because of the privileged mode the operating system runs (partly) in, the profiling functionality has to be running there as well. The profiler may not have all needed functionality (like writing the results to a file) available at all times when running inside an operating system. There is a risk that the profiler influences its own measurements because it is part of what it is measuring. The performance of operating systems plays a mayor role; it directly affects the performance of programs that use its services. For an operating system developer, a profiler that measures the performance of the operating system is therefore a useful tool.

This thesis describes the implementation of two different methods of profiling the MINIX 3 operating system: *statistical profiling* and *call profiling*.

1.2 MINIX 3

The MINIX 3 operating system is based on a microkernel (as opposed to a more common monolithic kernel). Having this type of kernel means that as little code as possible runs in the high-privileged mode of the CPU (this mode is also referred to as kernel-space). Most of the code runs in less privileged mode (user-space). The different components communicate by passing messages. An advantage of this approach is that the privileged part of the operating system code remains small in size and simple, and is therefore easier to keep bug-free. Bugs in user-space parts of the operating system are not able to bring the whole system down. A disadvantage is that there is a performance penalty because message passing implies an overhead (building and copying messages) where a monolithic kernel just has one address space where any line of code of the operating system has full control over. For example, in MINIX 3, the file server (FS), process server (PM) and other operating system components (including the device drivers) run as user-space processes that request the kernel to perform privileged actions for them. In a monolithic kernel, all parts of the operating system have high privileges and are able to perform the privileged actions themselves.

System calls in MINIX 3 are implemented as follows: the system call ends up in a user-space process that is related to the call (FS for file system related calls, PM for process related calls). The user-space process handles the call and if necessary, does a call to the kernel to inform it about changes or to request it to (for instance) copy data from one process to another. So, beneath the system calls there is a separate layer of calls. These calls are the *kernel calls*.

When reading this thesis it is important to understand that the following parts of MINIX 3 run in kernel-space: *KERNEL*, *CLOCK*, *IDLE* and *SYSTEM*. They run in the same address space but are actually different processes with their own CPU contexts and are separately scheduled. *KERNEL* (also known as the kernel task) is the part of the kernel that contains code for the lowest levels of interrupt handling, process scheduling and IPC (Inter Process Communication). *CLOCK* (the clock task) contains functionality related to the system timer; *IDLE* does nothing more than run when no other process is runnable and *SYSTEM* (the system task) contains the code for the kernel calls.

For more information about MINIX 3 I refer to its website http://www.minix3.org/ and the book *Operating Systems: Design and Implementation, Third Edition* by Andrew S. Tanenbaum and Albert S. Woodhull.

2 STATISTICAL PROFILING

2.1 INTRODUCTION

Statistical profiling is a method of profiling based on interrupting the operating system at a certain frequency to take a snapshot (sample). Such a snapshot denotes which process was active at the moment of interruption and where in its program execution it was. With this method of profiling, only a subset of all executions is registered. However, since the time intervals between snapshots are identical, the result represents all executions that occur often enough to be seen by the profiler. In this chapter I tell something about the timer used for profiling, a high level description of the design is given, the implementation is described in detail, and the results are discussed.

2.2 IN SEARCH OF A TIMER

To supply the frequency, a timer that is completely independent of existing processing is preferred. There is the system timer, but since this timer is already used by the operating system for scheduling purposes, it is not really independent. Another timer is preferred.

2.2.1 i8259 Timers

Let's have a brief look at the hardware behind the system timer. The timer is generated by the 8253/8254 PIT (Programmable Interval Timer). The 8253/8284 was a separate chip in the first IBM PC's, but now it is integrated somewhere on the motherboard. Conceptually it is still there though and it has three timers called Counter 0, 1 and 2. Counter 0 is connected to the first interrupt line (IRQ 0) of the i8259 PIC (Programmable Interrupt Controller) and is used by MINIX 3 as the system timer. Counter 1 is used for DRAM refresh and Counter 2 generates the frequency needed for the PC speaker. Unfortunately, Counter 2 is not connected to the PIC, otherwise it might have been a candidate (not having the PC speaker available during profiling would be a small price to pay for a suitable timer). Conclusion: the 8253/8254 timers are not suitable for our purpose.

2.2.2 CMOS Real-Time Clock

There is another timer in the IBM PC; it is the CMOS "Real-Time Clock". This timer is connected to the motherboard battery and keeps track of time. It is also connected to the first interrupt line on the secondary interrupt controller (IRQ 8), and it is programmable. It can generate interrupts ranging in frequency from 2 Hz to 8 KHz, in increasing powers of 2. This clock is an unused, independent timer which makes it suitable for statistical profiling. There is a drawback: since this clock is used to keep track of system time we run the risk of having unwanted side-effects on that time. As we later find out, these effects exist but they are a relatively small nuisance and do not affect profiling.

2.3 HIGH-LEVEL DESCRIPTION

A rough description of how statistical profiling in MINIX 3 works follows. The user can start and stop statistical profiling utilizing a user program. When the user starts profiling, the program allocates memory for profiling data and does a system call passing the location of the allocated memory as well as the location of an information structure ("info struct") that allows the kernel to supply feedback about profiling. The system call is handled by a user-space system process, as all system calls are in MINIX 3. This handler does not do a lot more then call the kernel. The kernel sets up the CMOS timer used for profiling. From now on the kernel will write profiling samples consisting of process name and program counter (also known as instruction pointer) to the memory allocated by the user program. It maintains the number of bytes written in a local info struct. The kernel call and system call return and the user program continues running in the background as a daemon process. The user can now run whatever he wants: profiling is turned on and will register the effects on the system. When the user wants to stop profiling, he runs the profile command again with the stop parameter. A system call is done which is again passed to the kernel. The kernel stops profiling and copies its local info struct to user memory. The profiling data in memory is written to a file by the daemonized user program. There is a separate Perl tool that the user runs to analyze the raw data file and present the results in a formatted manner.

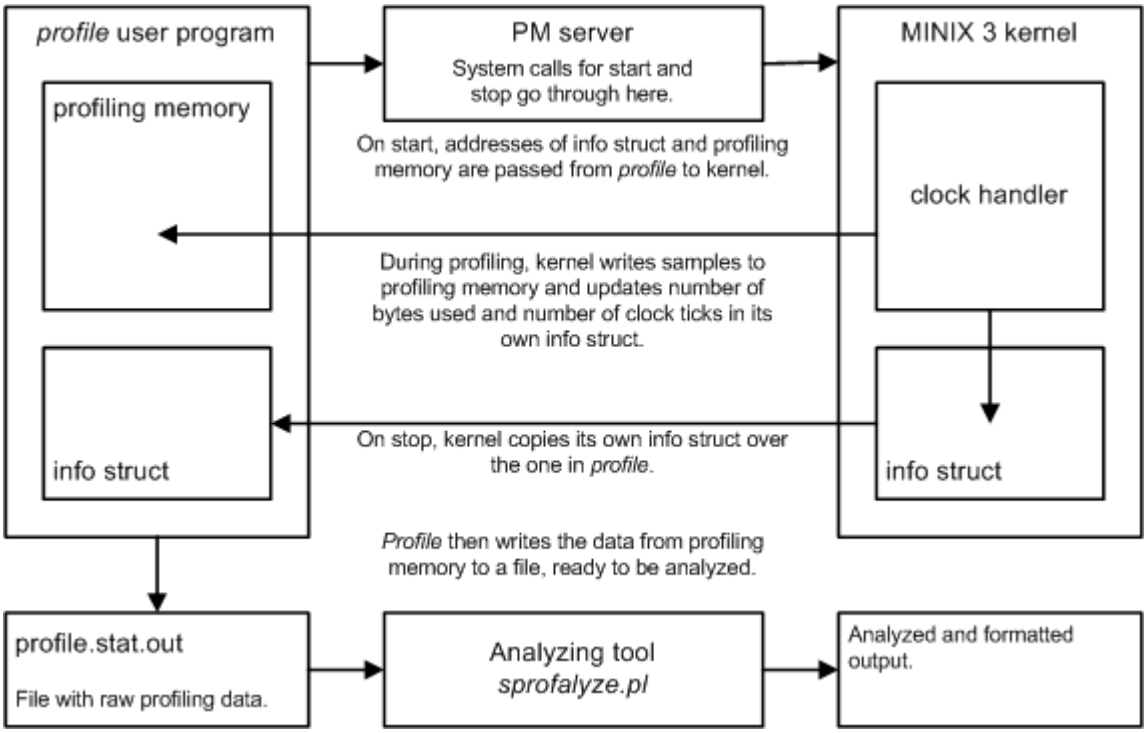

Figure 2-1. Overview of data flows in statistical profiling

The next paragraphs give a detailed description of statistical profiling in detail, split up in three levels: user-space, kernel-space and application level.

2.4 WORK DONE IN USER-SPACE

2.4.1 The SPROFILE System Call

A system call *SPROFILE* was added to MINIX 3 to enable user programs to start and stop profiling. Just like for existing system calls, a library function was created (in this case located in *lib/other/* which is part of the C library) as a wrapper around the call. This hides the message passing from the user program by making it as simple as calling *sprofile*. When start of profiling is requested, the caller supplies pointers to a chunk of memory available for profiling data and to an info struct (defined in *include/minix/profile.h*) which is used by the kernel to supply feedback when profiling is stopped.

```
struct sprof_info_s {
    int mem_used;
    int total_samples;
    int idle_samples;
    int system_samples;
    int user_samples;
} sprof_info_inst;
```

Figure 2-2. Info struct to be copied to from kernel to user program.

The system call ends up in the PM server, which does a few things before calling the associated kernel call *SPROF*: it adds the endpoint of the calling process to the message parameters and does a check on the supplied pointers variables to see if the supplied locations are actually in the user program's address space. In case stopping of profiling was requested, no pointers need to be supplied.

2.5 WORK DONE IN KERNEL-SPACE

2.5.1 The SPROF Kernel Call

A kernel call *SPROF* was added to start and stop profiling in the kernel. When it is called to start profiling, in *system/do_sprofile.c* the following is done. There is a check if profiling is already running, in which case an error is returned. The pointers to the info struct and to the memory available for profiling are translated to physical addresses and stored in local variables (variables related to profiling are in *kernel/profile.h*).

```
EXTERN int sprofiling;                  /* whether profiling is running */
EXTERN int sprof_mem_size;              /* available user memory for data */
EXTERN struct sprof_info_s sprof_info;  /* profiling info for user program */
EXTERN phys_bytes sprof_data_addr;      /* user address to write data */
EXTERN phys_bytes sprof_info_addr;      /* user address to write info struct */
```

Figure 2-3. Variables related to profiling, in the kernel.

The kernel uses its own struct with profiling info, of which the variables are reset. It is instantiated from the same declaration, so the kernel can just write it into the user's address space when profiling stops and supply the user with this information. The CMOS clock is started by calling *init_cmos_clock* and a variable is set to indicate that profiling is now running. During profiling, the interrupt handler for the CMOS clock will do the actual work. When profiling is requested to be stopped it is checked if profiling is indeed running. The CMOS clock is then stopped by calling *stop_cmos_clock*. Finally the local info struct is copied to the user program using the physical address calculated when profiling started.

2.5.2 Profiling using the CMOS Timer Interrupt

Functionality related to the CMOS clock is in *kernel/profile.c* (this file is listed in appendix C). Programming the CMOS is done by mapped I/O using existing *inb*/*outb* library functions. In order to read/write a register first a byte identifying the register must be written into an index register, and then the data register must be read or written. Reading/writing the data register after setting the index register is required. A description of the functions related to the CMOS clock now follows. *Init_cmos_clock* registers an interrupt handler in the kernel for the CMOS timer interrupts. It then programs the CMOS timer's frequency and enables its interrupts. *Cmos_clock_stop* disables the CMOS timer interrupts and deregisters the interrupt handler in the kernel. *Cmos_clock_handler* is the interrupt handler that runs on every tick. The handler first checks if profiling is actually turned on and if enough profiling memory is available. If this fails, it just returns. A check is done if writing a new profiling sample would overflow available profiling memory. If this is the case a variable in the info struct is set to -1 so the kernel itself and, at a later stage, a user program, will know about this. Depending on the process that was interrupted by the CMOS timer interrupt, a few things could now be done. If the system was idle or if the current process is a user process, the appropriate counter in the info struct is increased, but no sample is written. If the current process is a system (MINIX 3) process, an additional check is done whether the process is actually runnable before its name and program counter (the sample) are written to the profiling memory.

```
struct {
    char name[8];
    int pc;
} sprof_sample;
```

Figure 2-4. What a profiling sample looks like.

Writing a sample to user memory is done using the physical address calculated when profiling started. After the sample is written, the *mem_used* counter in the info struct is increased appropriately, to be used by the kernel itself when writing the next sample and later on by a user program to see how much memory was used. Finally, a counter for the total amount of samples in the info struct is increased to keep track of the number of CMOS clock ticks. At the end of the interrupt handler a data register of the CMOS is read. This is necessary to make the next interrupt happen.

2.6 WORK DONE AT THE APPLICATION LEVEL

2.6.1 Control Tool: profile

In *commands/profile/* are some user tools to control profiling and analyze raw profiling data. The user program *profile* was written in C and is used to start and stop profiling and have the profiling data collected by the operating system written to a file. When profiling is started using the appropriate command line parameter(s), the following is done: the program forks and allocates memory for the profiling data, a system call *SPROFILE* is done providing the operating system with the following: the location and size of the allocated memory, the location of a info struct (as mentioned earlier, this struct is defined in *include/minix/profile.h*) and the requested sample frequency. The user is informed that profiling has started (or an error is given if applicable), and the program detaches itself from the console. A named pipe is opened and a string of fixed length is written to it. From the user's perspective, the program has exited but the program is still running in the background, blocking on the write to the named pipe. The user can now run anything he likes. The whole operating system is profiled from now on and the profiling samples are written by the operating system to the memory provided by the user program.

When the user wants to stop profiling, he runs the profile program indicating that profiling should stop. The program does the *SPROFILE* system call to have profiling stopped in the operating system. It then opens the named pipe set up by the process that started profiling and reads the same fixed amount of characters as the starter process wrote. The starter process unblocks because of this and knows profiling has now stopped. It checks the info struct to see if there was enough memory for the profiling data (if there was not, the kernel set it to -1). If this was found OK, it writes to a file a header containing several numbers found in the info struct, namely the counts of idle, user, system and total ticks. It then starts writing the profiling data from memory to the file. Output for the user regarding progress or errors is directed to the named pipe. The stopper process just prints out what it receives through the pipe and therefore acts as a console for the starter process. When the starter process is done writing the data file, it closes the file handle and the named pipe and exits. Upon reading EOF on the named pipe, the stopper process knows everything is done and it exits as well. The user now has a file with raw profiling data.

2.6.2 Analyzing Tool: sprofalyze.pl

In the same directory, the Perl program *sprofalyze.pl* is available to analyze raw profiling data files. As a first step, *sprofalyze.pl* reads the symbol tables of all system executables by means of the *nm* program. From the symbol tables, indexes are generated as hash tables using the Perl built-in hash data structure. An index is generated for each system executable, where function names are indexed by address. Of course, from the symbol tables only the start addresses of the functions can be found. The gaps between the start addresses are filled with entries for every possible program counter value. This turns out to be well worth it as can be seen in the following step.

The script starts reading the data file. As mentioned, there is a header line containing the total number ticks the CMOS clock made during profiling, as well as the number of ticks that happened during execution of system processes, user processes and during idle time. These numbers are read into variables to be used later for statistics generation. After the header, the actual profiling samples follow. The samples are processed one by one. The fixed sample length is read and the process name (regular ASCII) and program counter (integer in binary form) are decoded from it. The process name is used to decide which index to use from the indexes created from the symbol tables. This index is then accessed using the program counter. Since every possible program counter value is in the index, this step is fast: just one lookup in a hash table. If the index would contain only the start addresses of the functions, time consuming (search) logic would be needed to find the correct function for each profiling sample. At 8 KHz, a profiling run could create one million samples in a few minutes at worst, so quick lookups are useful. Indexing into the table using the program counter returns a function name. A counter representing this function is increased; this counter is in another Perl hash: for each system process a table keeps track of the number of samples attributed to each function. Once all samples are processed, the results are sorted; percentages are calculated and formatting takes place. High scoring functions are shown aggregated and on a per process basis. There is a command line argument to *sprofalyze.pl* that allows the user to provide a minimum percentage (default: 1%). Only functions and processes that used at least that percentage of operating system time are shown.

2.7 WHAT CAN AND WHAT CANNOT BE PROFILED

All drivers, servers and the system task are profiled by the statistical profiler. However, as profiling is done through an interrupt handler in the kernel task, the kernel task itself cannot be profiled. The reason is that *KERNEL* is not reentrant: no interrupt, including the profiling timer interrupt, is allowed to occur while kernel task code is running. This implies that interrupt handling, scheduling and IPC code is not profiled. Note: *SYSTEM* which handles the kernel calls and also runs in kernel-space *can* be profiled.

In all processes that are profiled, C functions (including library functions) and assembly language (identified by label instead of function name) are profiled.

2.8 PROFILING RESULTS

The results of a profiling run can be found in appendix C. They are analyzed below and in chapter 5.

2.8.1 High Scoring IPC Functions

In the results IPC functions score high in many processes. This is suspicious because these are efficient assembly language routines that should not take much CPU time, even when a lot of messages are processed by the system. To make sure samples are not incor-

rectly attributed to blocking processes, a check is done in the interrupt handler to make sure the current process is actually runnable. The check is still there, although this always turned out to be the case. The latter makes sense since an un-runnable process is not scheduled and therefore cannot be the current process. I also checked for the value of *k_reenter* to make sure the kernel task was not running something else which would be incorrectly attributed to the current process. This was found to be OK, because as mentioned earlier *KERNEL* is not reentrant.

So, what could be the cause of the high scoring IPC functions then? In order to find out more about the scoring of these functions, I put extra labels in *__send*, *__receive* and *__sendrec* in *lib/i386/rts/_ipc.s*. The code for *__send* after placing these extra labels (*__snd[1-9]*) is shown below.

```
__send:
        push    ebp
__snd1:
        mov     ebp, esp
__snd2:
        push    ebx
__snd3:
        mov     eax, SRC_DST(ebp)       ! eax = dest-src
__snd4:
        mov     ebx, MESSAGE(ebp)       ! ebx = message pointer
__snd5:
        mov     ecx, SEND               ! _send(dest, ptr)
__snd6:
        int     SYSVEC                  ! trap to the kernel
__snd7:
        pop     ebx
__snd8:
        pop     ebp
__snd9:
        ret
```

After profiling with these extra labels, it turned out the results were always coming from label 7: the one after the trap instruction. It became clear the assembly routines are indeed not causing the strange results by being big time consumers: it has something to do with the software interrupt that takes place.

2.8.2 Interrupt Delay

My theory is that the CMOS timer interrupt is often delayed by the IPC traps. According to this theory, the high scoring IPC routines would then actually stand for the time spent during the trap by *KERNEL* in functions like *mini_send*.

To test this theory, I temporarily changed the profiling interrupt handler: instead of writing profiling samples it read the CPU cycle counter, calculated the difference with the previous measurement and wrote it to profiling memory. This delivered a series of time differences between the interrupts, which should be similar if none are delayed. Since the CPU cycle counter is a steady way of measuring time differentials - its clock is never delayed - this might indicate if many CMOS timer interrupts were delayed. Some information about these measurements: they were done on an Intel Celeron D running at 2.67 GHz. The timer interrupt was set to 1 KHz. Therefore, the cycle counter difference between two interrupts is expected to be roughly 2670000000 / 1024 = 2607422 (~2.6 mil-

lion cycles per 1/1024th of a second). I ran the measurements for roughly 80 seconds (while running a busy *make* job). This generated 81732 measurements. I then took from the unsorted measurements every third one; this left me with 27244 results which would still be representative. (This was needed to fit the results in the program that created the graphs; admittedly quite a limitation for such a program, but it sufficed for most of my purposes.) The average of the 27244 measurements is 2599532, the average of the measurements minus the extremes (explained below) is 2599580.

The results are in the following diagram. The results were first sorted from small to large. On X-axis is the measurement number: smallest measurement on the left, largest measurement on the right. On the Y-axis the cycle difference can be found, in number of cycles.

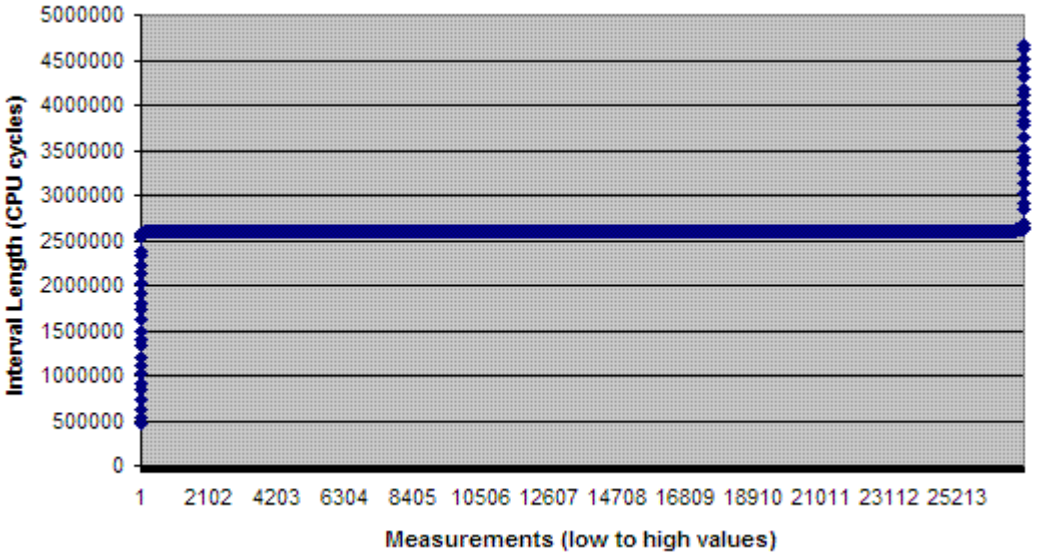

Figure 2-5. Time differences between CMOS timer interrupts.

As can be seen in the above diagram, except for some extreme values, most measurements have roughly identical values. However, the Y-axis of the diagram covers a very wide range to allow a few extreme values to be visible. For the next diagram, I took out the 30 lowest and 30 highest measurements; they account for 0.22% of all results. Now the Y-axis can zoom in to the remaining 99.78% of measurements.

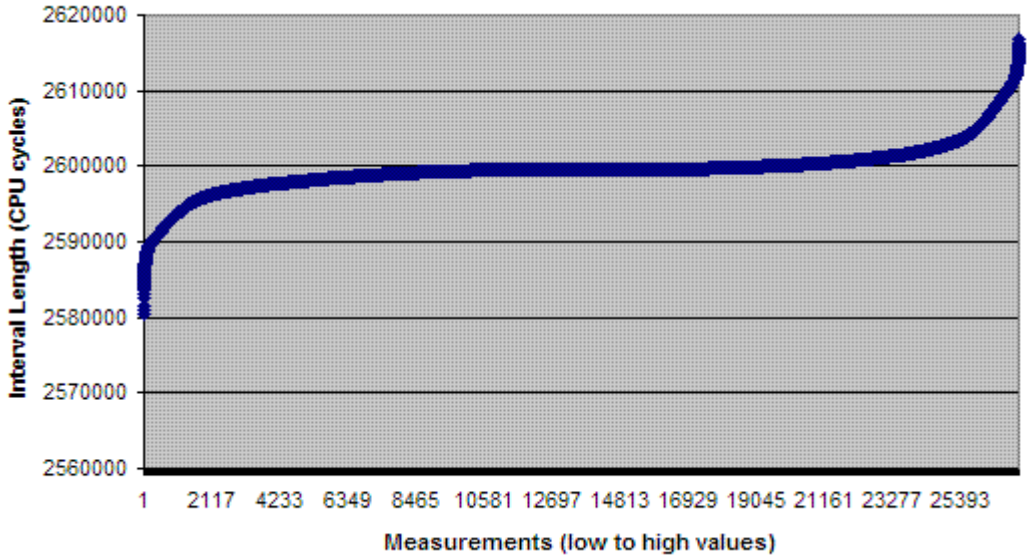

Figure 2-6. Time differences between CMOS timer interrupts without extreme values.

This diagram indicates that indeed a significant number of interrupts are delayed. This can be seen from the graph moving upwards at the right end where the intervals between interrupts are increasing.

The longer intervals on the right side are 'compensated' by the shorter intervals the left side. This can be explained as follows: assume successive interrupts *i*, *j* and *k* where *j* is delayed and *i* and *k* are on time. Then the interval between *i* and *j* is larger than normal because of *j*'s delay (*i* happens at a timer tick, *j* happens *after* the next tick). As another consequence of *j*'s delay, the interval between *j* and *k* is shorter then normal (*j* happens after a timer tick, *k* happens *at* the next tick). So, each delayed interrupt affects intervals before (longer than normal) and after (shorter than normal).

In my opinion, these measurements make interrupt delay a likely explanation for the high scoring IPC functions; in the rest of this document I will refer to this effect as the interrupt delay effect.

Finally, for completeness a histogram of the measurements (without the extremes) follows, as a more conventional way of showing the distribution.

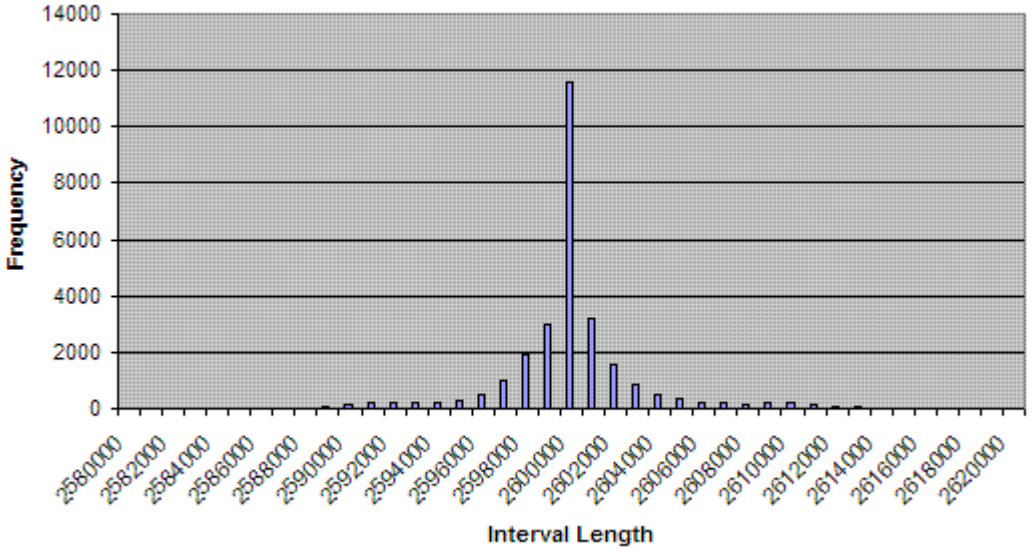

Figure 2-7. Histogram of time differences between CMOS timer interrupts (without extremes).

2.8.3 Profiling Runs on Simulator and Other CPU Models

I did profiling runs on an Intel Pentium III 600 MHz, an AMD Duron 1 GHz and an Intel Celeron D 2.67 GHz. Although the results were mostly similar, the scoring of IPC functions varied somewhat per CPU model. A possible explanation would be that the time it costs to trap is different from model to model. Cache size may have an effect on this. I also profiled in VMWare. In this simulator the IPC functions scored extremely high. I'm not sure if this is caused by very slow context switching because of the virtualization or for another reason.

2.9 SIDE-EFFECT OF USING THE CMOS CLOCK

Regularly after doing a profiling run, the system time will be off after the next reboot. Since the Real-Time Clock is used as profiling timer this is not really a surprise. The RTC is used to keep track of time in hardware. The operating system reads this time upon boot to set its own time variable. I expected that time would go slower or faster in the CMOS when its frequency was set lower or higher than its default, during profiling runs. In practice it would just be reset to 00:00:00 1 Jan 1970, a few days or years earlier or later then the current date or other corruption could be seen in the BIOS date/time. Therefore, the BIOS time should be checked after running statistical profiling; this is especially true if a tool like rdate for remote time synchronization is not run automatically after boot.

3 CALL PROFILING

3.1 INTRODUCTION

Call (path) profiling is a completely different approach to doing measurements. Instead of having a timer that takes snapshots periodically, function calls throughout the execution of a program are followed continuously. This allows more than just taking measurements on a function level: call paths can be tracked and saved. This could give more insights into the execution of a program. The subject of this chapter is to describe the implementation of call profiling in MINIX 3. A high level description of the design is given, the implementation is described in detail, and the results are presented.

3.1.1 Compiler-supported Call Profiling

MINIX 3 is usually built with the ACK (Amsterdam Compiler Kit) compiler. This compiler has a command line option -Rcem-p. When used, ACK creates function calls to *procentry* and *procexit* on entry and exit of every function call. The *procentry* and *procexit* functions should be written by the compiler user. I used the –Rcem-p functionality to have profiling code triggered. This code generates call paths and registers the number of calls and cycles spent for each unique call path.

3.1.2 Call Paths, Call and Cycle Attribution

When talking about a call path "a b c", it means execution started in *a*, *a* then called *b* and *b* then called *c*. To be able to differentiate where cycles were spent, the time attributed to a call path is the time spent *in the function at the end of the call path*. For example: call path "a b c" implies there are also paths "a" and "a b". The results could look like this:

calls	cycles	path
1	10	"a"
1	30	"a b"
2	20	"a b c"

The start of the execution path is *a* (usually the start is *main*). It was called exactly once. In its own body (outside function calls it did), it spent 10 cycles. The second path also has 1 as the number of calls, which means that *a* called *b* exactly once. There are 30 cycles attributed: these were spent in *b* when it was called by *a*. Possibly more cycles are spent in *b*, but only when another, different, call path exists where *b* is part of. "A b c" has two calls attributed to it; this means that from *b* in "a b", *c* was called twice. The 20 cycles are the total of time spent in *c* during these two calls.

3.2 HIGH-LEVEL DESCRIPTION

When the system is built using the –Rcem-p parameter, the system processes are linked with library functions *procentry* and *procexit*. Early after a profiled system process starts, it announces itself to the kernel. This allows the kernel to control profiling in the process through a shared data structure (the "control struct") and know where the profiling table of that process is. In the process, profiling is done by the *procentry* and *procexit* library functions. In *procentry* the call path for a new call level is built using the *name* parameter (the *name* parameter of *procentry* points to a string with the name of the function that is called). A slot in a hash table corresponding to the call path is found or inserted, and the number of calls for the call path is updated in its slot. A stack is utilized that has an entry for each function called but not yet exited. The entry at the top of the stack corresponds to the most recent function call. In *procentry* the stack is used to save the call path length, a pointer to the slot corresponding to the call path and the CPU cycle count for the current level. When returning to a lower call level, in *procexit* the call path is restored using the stack. The CPU cycle count is read and the difference with the count on the stack is added to the cycles attributed to this call path using the slot pointer on the stack. In order to attribute only time spent in a functions own body, time spent on higher call levels (also maintained in the stack) is subtracted from it.

The user can get the profiling tables of the profiled processed or have these tables reset using the same user program that is also used for statistical profiling. When the user wants to have the tables reset (the requested action is specified by a command line parameter), the program does a system call. The system call handler, which is as usual in a user-space system process, calls the kernel. The kernel writes a flag in the control struct of each profiled process. A profiled process sees this flag on its next entry of *procentry* and clears its table. When the user wants to get the profiling tables, the system is provided with pointers to user memory and an information structure ("info struct") using the same system call. The kernel then copies all the profiling tables from the profiled processes to the memory supplied by the user; it writes feedback about the number of bytes copied and possible errors to the local info struct which is then copied to the user program. The profiling data in memory is written to a file by the user program; it adds some information like the process names. There is a separate Perl tool that the user runs to analyze the raw data file and present the results in a formatted manner.

On the next page there is a data flow diagram to visualize this. The following paragraphs will describe in detail how call profiling was implemented.

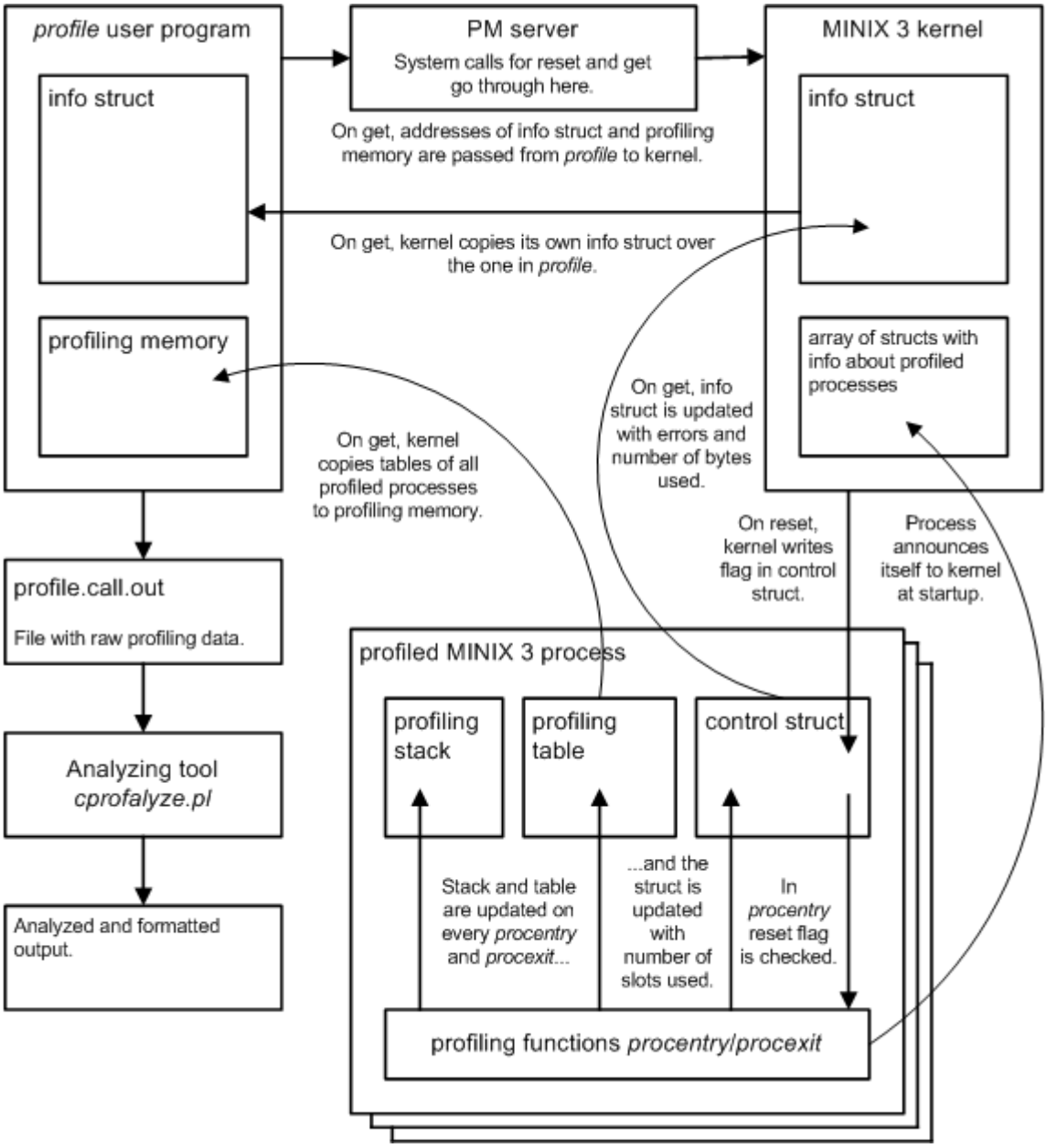

Figure 3-1. Overview of data flows in call profiling.

3.3 WORK DONE IN USER-SPACE

3.3.1 The CPROFILE System Call

Analogous to statistical profiling, a system call *CPROFILE* was added to enable user programs to control profiling. This allows user programs to get or reset profiling tables. When the profiling tables are requested, the caller supplies pointers to a chunk of memory and to an info struct which is used by the kernel to supply the number of bytes written to the memory and the errors that may have happened during profiling.

```
struct cprof_info_s {
    int mem_used;
    int err;
} cprof_info_inst;
```

Figure 3-2 Info struct to be copied from kernel to user program.

There is a library function *cprofile* for easy access to the system call. This call is handled by the PM server which does memory validity checks as well as adding the endpoint of the calling process to the message parameters.

3.4 WORK DONE IN KERNEL-SPACE

3.4.1 The PROFBUF and CPROF Kernel Calls

A kernel call *PROFBUF* was added. Profiled processes utilize it to inform the kernel about the location of their profiling table and control structure. When this call is made, the name, endpoint, physical address of control struct and profiling table are saved in an array. The array holds a struct for each process and is used by the kernel call that will be described next.

```
struct cprof_ctl_s {
    int reset;          /* kernel sets to have table reset */
    int slots_used;     /* proc writes nr slots used in table */
    int err;            /* proc writes errors that occurred */
} cprof_ctl_inst;
```

Figure 3-3. Control structure used in profiled processes.

A kernel call *CPROF* was added to allow a user program to get profiling results from the kernel, or have the profiling buffers reset. When *CPROF* is called (the kernel call is done through PM) in *system/do_cprofile.c* the following is done. On a *RESET* request, for each of the profiled processes it is checked whether the process is still alive and a flag is set in its address space using the physical address of its control struct. This address is in the array mentioned in the previous paragraph. The profiled process should now respond to the change of the flag and reset its table. On a *GET* request, the user program pointers to info struct and profiling memory are translated to physical addresses and stored in local variables. For each of the profiled processes, it control struct is copied to a local variable which is then read to see how much tables slots were used for profiling up to now. From the total number of table slots, it is calculated if the results of all processes together would fit in the profiling memory of the user program. The info struct is then updated with the result: -1 in case of insufficient memory, otherwise a positive number for the total number of bytes that will be copied to the user's memory. If there is enough memory available, the kernel copies for each profiled process to the user's memory the name of the process, the number of used slots, followed by all the used slots in its profiling table.

```
EXTERN int cprof_mem_size;                  /* available user memory for data */
EXTERN struct cprof_info_s cprof_info;      /* profiling info for user program */
EXTERN phys_bytes cprof_data_addr;          /* user address to write data */
EXTERN phys_bytes cprof_info_addr;          /* user address to write info struct */
EXTERN int cprof_procs_no;                  /* number of profiled processes */
EXTERN struct cprof_proc_info_s {           /* info about profiled process */
      int endpt;                            /* endpoint */
      char *name;                           /* name */
      phys_bytes ctl;                       /* location of control struct */
      phys_bytes buf;                       /* location of buffer */
      int slots_used;                       /* table slots used */
} cprof_proc_info_inst;
EXTERN struct cprof_proc_info_s cprof_proc_info[NR_SYS_PROCS];
```

Figure 3-4. Variables related to profiling, in the kernel.

3.5 WORK DONE IN LIBRARIES

3.5.1 Profiling Using Library Functions

The heart of call profiling is in the library functions *procentry* and *procexit*. They are in *lib/sysutil/profile.c* (the listing of this file is in appendix B). For processes that were compiled using –Rcem-p, ACK adds calls to *procentry* and *procexit* on entry and exit of every function call. By defining the functions in a library against which system processes are linked, they need to be put only in one place. Another advantage of the library approach is that there is absolutely no change needed to most parts of the operating system (especially the servers and drivers) except for adapting the Makefiles so that the –Rcem-p flag is used and to make sure the –lsysutil flag is in there for correct linking. To make this happen, a bunch of Makefiles were changed slightly. In these Makefiles, a variable $(CPROFILE) was added to the regular $(CFLAGS). By declaring *export CPROFILE=-Rcem-p* on the command line before building the system, this environment variable is in scope and applied during the *make* builds. An alternative would have been to use $(MAKEFLAGS) to have the *make* program propagate the flag instead of the shell.

3.5.2 The Procentry Library Function

Procentry first does a reentrancy check, the reason of which is explained in a later paragraph. Then, the Pentium CPU cycle counter is read and the result (a 64-bit number represented by two 32-bit unsigned integers) is stored in a local variable. The function that is used to read the counter is in a separate file *read_tsc.s* which is also in the *lib/sysutil/* directory. It is identical to *read_tsc* in *kernel/klib386.s*, but since the latter is kernel source code inaccessible to user-space processes I copied it to this location.

Using static variable *init* there is a check if *procentry* was called for the first time. If this is the case, *cprof_init* is called to initialize the variables used for profiling. Some information about these variables: *cpath* represents the current call path as a string of space separated function names. For example, if *foo* was called and *foo* called *bar*, the call to *procentry* on entry of *bar* will cause the call path to be set to “foo bar”. *Cpath_len* holds the

length of the call path string. A stack is maintained to store for each call level the call path length, the corresponding slot in a hash table, the CPU cycle counts taken at begin and end of *procentry* and the number of cycles that were spent at higher call levels. The size of the hash table and the announce number are then retrieved (these variables are explained further on). *Cprof_init* resets the stack and the hash table before returning.

Using the same static variable *init* that is used to check for the first run, it is now checked if this is the *announce run*. Announcing means that the profiler is going to announce its existence to the kernel. Since the profiling functions are implemented as library functions, the kernel has initially no idea which processes are being profiled. Therefore a function is called to have the process announce to the kernel the locations of its control struct and profiling hash table. This makes the kernel aware of the profiled process, and enables it to control its profiling.

Before continuing, *procentry* does a sanity check on a variable that indicates whether an error happened earlier on. Since this variable is in the control struct of which the location was passed to the kernel earlier, it is available there as well. The kernel passes the error information on to user programs when they want the profiling tables. A check is now done on another variable in the control struct to see if the kernel instructed to reset the profiling table, in which case this is done. Note that the stack is not reset: it must outlive resets to keep its integrity.

```
#define CPROF_CPATH_OVERRUN     0x1    /* call path overrun */
#define CPROF_STACK_OVERRUN     0x2    /* call stack overrun */
#define CPROF_TABLE_OVERRUN     0x4    /* hash table overrun */
```

Figure 3-5. Errors that may occur during profiling.

The variable that holds the top of the stack is increased by 1 to prepare for the new call level. Should the stack overflow, the error variable in the control struct is updated and the function returns. Now the CPU cycle count that was read at the start of *procentry* is saved on the stack. An overflow check similar to that for the stack is done for the call path string. The current call path length is saved on the stack before the new call path string and its length are updated using the *name* parameter that is passed to *procentry*. The *name* parameter is a pointer to a string containing the name of the called function.

The new call path string is run through a hash function to calculate the entry in the hash index that should point (directly or through chaining) to the slot for this string in the hash table. The hash algorithm used is ELF; I chose it because it seems a generally accepted algorithm with a reasonable good balance between distribution and calculation time. Using the hash index, a lookup is done in the hash table to see if this call path is already in there. If it is, the *calls* counter in the corresponding slot in the table is updated.

```
struct cprof_tbl_s {
    struct cprof_tbl_s *next;                  /* next in chain */
    char cpath[CPROF_CPATH_MAX_LEN];           /* string with call path */
    int calls;                                 /* nr of executions of path */
    u64_t cycles;                              /* execution time of path, in cycles */
} cprof_tbl_inst;
```

Figure 3-6 A hash table slot.

If the path is not in the hash table yet, these actions are taken: a check is done for table overflow, the first available slot in the table is appointed to this new unique call path, the *cpath* and *calls* fields of the slot are set to the current call path and the number 1 (since it is the first call for this path). The hash index (in case of no collision) or hash chain (in case of collision) is updated. The hash result (index entry for this call path) is stored on the stack. Finally, the CPU cycle counter is read again and saved on the stack.

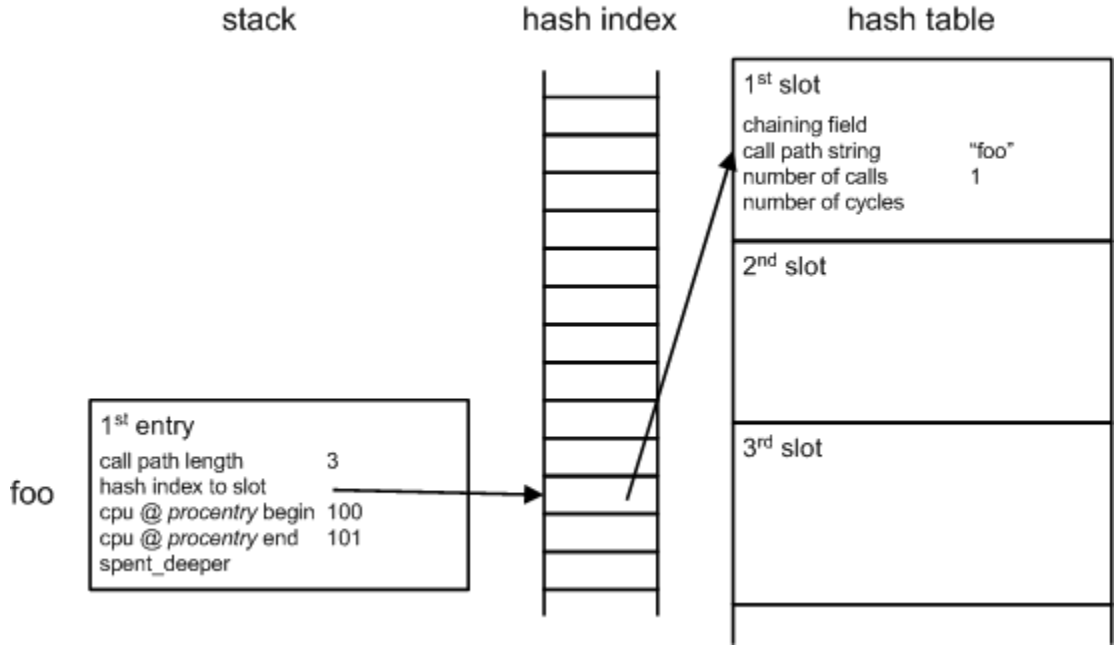

Figure 3-7. *Procentry* has just run when *foo* was called.

In the example above, the first function that executed in the process was *foo*. When *foo* was entered, *procentry* created an entry on the stack. The new call path length was saved in it. The call path string itself, including the function name, is in *cpath* (not visible in the diagram) and not on the stack, as explained later. A hash index was calculated and saved. Also the CPU cycle counts at begin and end of *procentry* were saved (in reality, the cycle count numbers are much higher, of course). For this new call path, the first slot in the hash table was allocated. In the slot, the call path string was written and the number of calls was set to 1. In this case, (naturally) there is no collision; the chaining field is not used.

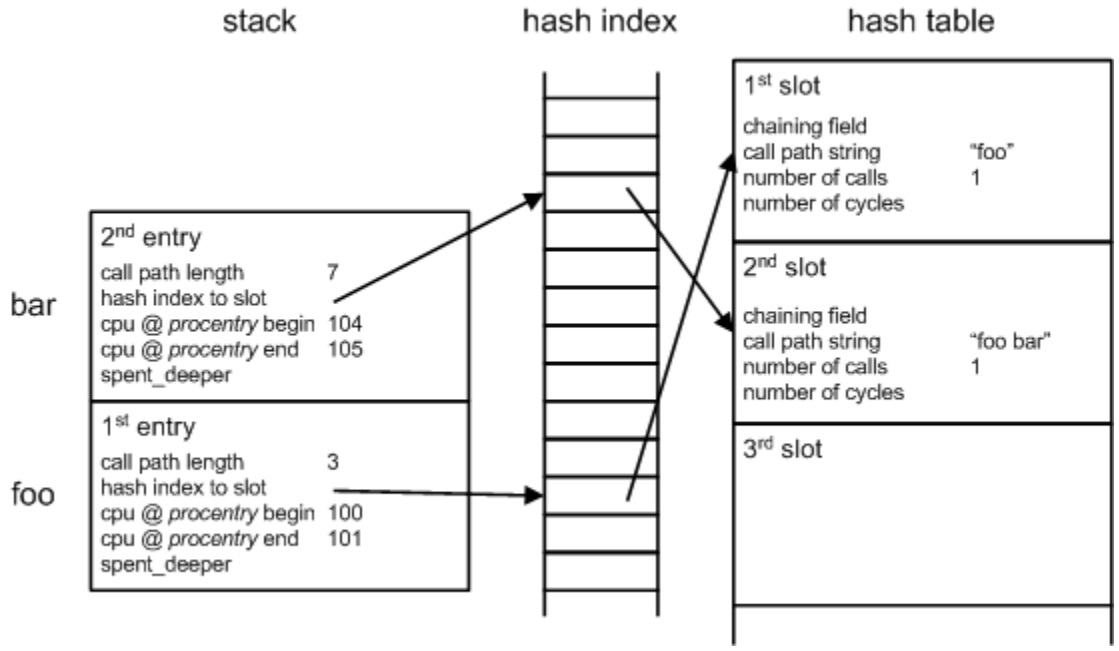

Figure 3-8. *Procentry* has just run when *foo* called *bar*.

In our example process, *foo* called *bar* during its execution. Just like for *foo*, *procentry* was run. It created a new stack entry and saved in it the new call path length, a hash index for this call path and the CPU cycle counts. A new slot on the hash table was allocated and the call path and number of calls (again 1, for a new path) were written to it.

3.5.3 The Procexit Library Function

So, *procentry* has run and a function in the profiled process was entered. The stack is updated with all the information for the currently running system function and there is a slot in the hash table that has the correct number of calls for this call path. What is missing is the cycle usage of the function. Naturally this will only become known when the function returns: when *procexit* is called. I will now describe what happens in *procexit*.

First a reentrancy check is done and then the CPU cycle counter is read into a local variable. Before continuing a check is done if errors occurred, just like in *procentry*. Remember that the CPU cycle counter was read at begin and end of *procentry*. This is also done in *procexit*. The cycle counter was read at the start, and will be read at the end. The reason for this is that *procentry* and *procexit* themselves take time, which we do not want to mess up the measurements. How does it work? In order to calculate the time taken up by a function call, the cycle counter is read at the end of *procentry* and at the beginning of *procexit* to have as little overhead of these calls as possible. Let's call the difference between these measurements the *small difference*. This number is used to attribute time to a certain call path, for example "foo". However, there may be deeper call levels, and this is extracted from the time attributed, because we need the time actually spent in the call path on the current level. So, the time spent in higher levels, which is maintained in the stack, is extracted from the attributed time. For instance: *foo* spends 25 cycles in its own body and calls *bar* which spends 100 cycles in its body. Call path "foo" is attributed 25

cycles; call path "foo bar" is attributed 100 cycles. There is a caveat here: when the time spent in deeper levels is extracted, it should be the time including the time spent in *procentry*/*procexit*, otherwise this overhead will turn up in the measurements after all. Therefore another difference is calculated, the *big difference*, which is the difference between the cycles measured at begin of *procentry* and end of *procexit* (almost, some 64-bit integer arithmetic cannot be avoided after the last reading in *procexit*). This is what happens in *procexit*:

First, the small difference is calculated, the *spent_deeper* variable related to the current call path is subtracted from it and the result is added to the *cycles* counter in the slot corresponding to the current call path. Then *spent_deeper* is cleared for the call level we are leaving (to start fresh should the call path return to this level). Finally the CPU cycle count is read for the second time and the big difference is calculated. The result is added to the *spent_deeper* variable in the entry on the stack below the current entry, so it can be subtracted later from the time spent at that lower level.

Meanwhile the call path string was updated by writing a null character at the appropriate position (the position was saved on the stack). This suffices since we're leaving a function and therefore the string can just be chopped at the proper length.

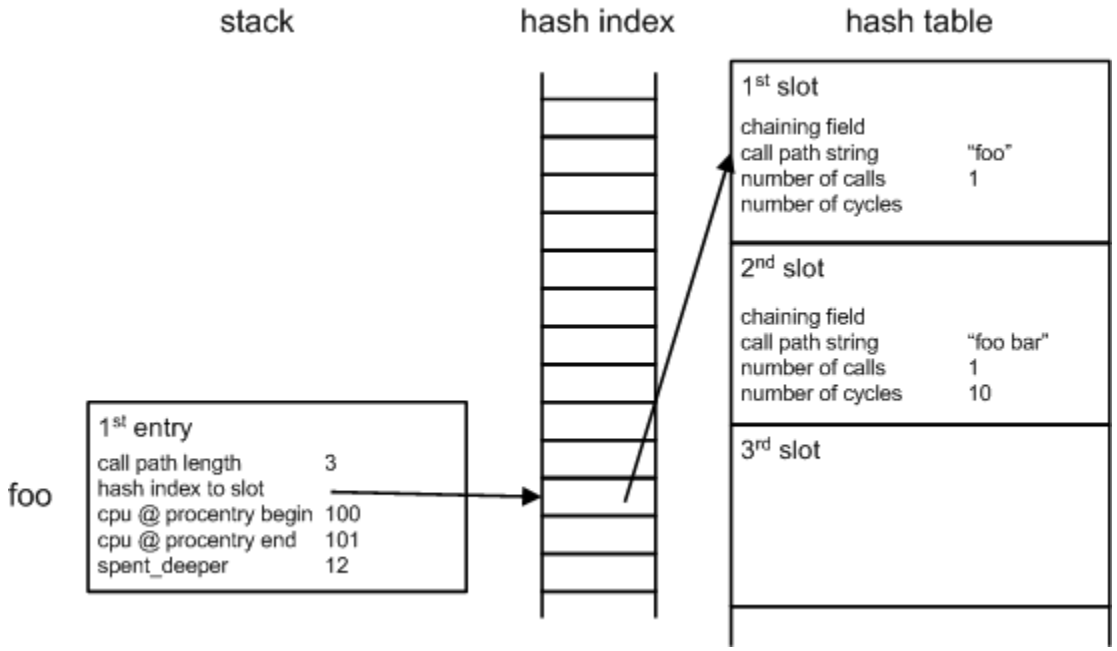

Figure 3-9. *Procexit* has just run when *bar* exited.

Our example function *bar* exited, which caused *procexit* to run. *Procexit* read the CPU cycle count (115; the *procexit* cycle read-outs are not visible in the diagram), subtracted from it the cycle count at the end of *procentry* (105, as saved on the stack). The result (10) was saved in the slot for "foo bar". At the end of its own execution, *procexit* also calculated the difference of a new cycle count read-out (116) with the cycle count at the start of *procentry* (104, as saved on the stack) and saved the result (12) in *spent_deeper* in *foo*'s stack entry. It then removed the stack entry for *bar*.

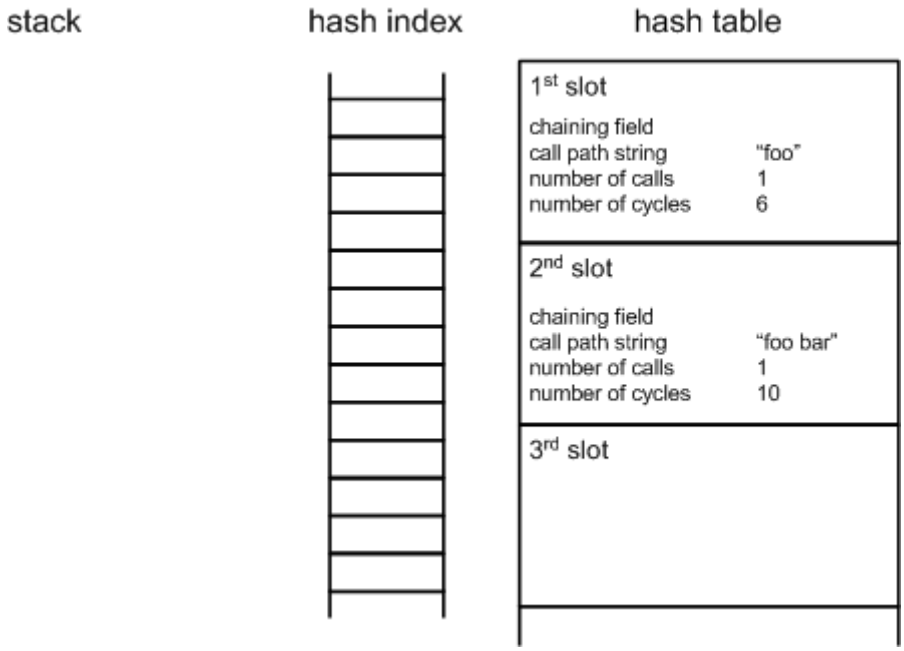

Figure 3-10. *Procexit* has just run when *foo* exited.

Example function *foo* exited, which caused *procexit* to run again. *Procexit* read the CPU cycle count (119), subtracted from it the cycle count at the end of *procentry* (101, as saved on the stack). *Spent_deeper* (from the stack) was then subtracted (18 – 12) and the result (6) was saved in the slot for "foo". Since the call level has reached zero (no running functions), there was no *spent_deeper* for a lower level that had to be updated. Finally, the stack entry for *bar* was removed.

3.5.4 The Call Path String

The call path could have been saved in components on the stack by storing for every call level the *name* parameter that points to a string with the function name. Instead, the call path is stored separately in character array *cpath* and its length in *cpath_len* (the latter *is* saved on the stack). This is done for performance reasons. Since on every call to *procentry* the call path would have to be constructed from (pointers to) function names on every entry on the stack, it makes sense to just keep the call path as a separate string to avoid this work. In *procentry* the string in *name* only has to be written to the call path character array at the right position to create the new call path; the old call path length is saved on the stack. In *procexit*, the call path is restored by simply chopping it at the length read from the stack.

Even though the measuring overhead of the profiling functions is eliminated for the most part, performance plays a factor. Right now the system is about 2-3 times slower doing system build *make* jobs with call profiling enabled. This is very acceptable but it should not be much slower.

3.5.5 Testing Overhead Elimination

I did some tests to check the small/big difference logic. I created functions *a*, *b*, *c* and *d*, all of which have an identical empty loop in their body to simulate some processing. In addition to the loop, *a* contains calls to *b* and *c*, and *b* contains a call to *d*. The code looks like this:

```
#define IT 10000

PRIVATE void a(){
	int i;
	for (i=0; i<IT; i++);
	b();
	c();
}

PRIVATE void b(){
	int i;
	for (i=0; i<IT; i++);
	d();
}

PRIVATE void c(){
	int i;
	for (i=0; i<IT; i++);
}

PRIVATE void d(){
	int i;
	for (i=0; i<IT; i++);
}
```

In *foo*, somewhere in a loop:

```
	…
    a();
	…
```

Since the profiler aims to attribute time spent in a functions own body, these functions should score roughly identical. I ran the test with single cycle count reads (at end of *procentry* and begin of *procexit*) and with the dual reads (as implemented). The call to *a* was placed in a loop in *foo*. There were two different runs in which the number of executions of *a* was almost -but not exactly- the same. So, one should look at the differences within a method (either single or dual read); the absolute differences between the methods cannot be used. Time is in milliseconds.

Single cycle count reads:

```
ms      path
1294    foo a
1237    foo a b
1195    foo a b d
1182    foo a c
```

As expected, the overhead of *procentry* and *procexit* accumulates mostly in call path "a" which has two call levels on top of it containing three functions calls (itself calling *b* and *c*, and *b* calling *d*). Call path "a b" suffers from the same accumulation although to a lesser extent since it has only one call level above it, containing one function call (itself calling *d*). Paths "a b d" and "a c" contain no more functions calls and are attributed the lowest times, according to expectations.

Dual cycle count read-outs:

```
ms      path
1246    foo a
1221    foo a b
1213    foo a b d
1212    foo a c
```

The times attributed to the functions are more similar. It seems that the dual reading has effect: the overhead of *procentry*/*procexit* is eliminated significantly.

3.6 PROFILING KERNEL-SPACE/USER-SPACE PROCESSES

There are some differences in the way the profiling library handles kernel-space processes and user-space processes. These are described in 3.6.1. In 3.6.2 and 3.6.3 some issues specific to profiling kernel-space are discussed.

3.6.1 Differences in Announcing and Table Sizes

As described earlier, a profiled process announces itself to the kernel. For the user-space processes this is done using the kernel call *PROFBUF*. The kernel, however, cannot do a kernel call. It has to manipulate the array of profiled processes directly. This posed a problem since the sysutil library is compiled separately from the system and at compile time it is unknown whether the library will be linked later on with a user-space process or the kernel. This could not be solved with some #if statements in the code.

The Solution is in Linking

I solved this by having *procentry* call a function *profile_register* to do the announcement. This function is defined in two places. It is in *lib/sysutil/profile_extern.c* as an additional library function; this one calls *PROFBUF*. It is also in *kernel/profile.c*; this one accesses the data structures directly. When user-space processes are linked with *procentry*/*procexit*

in the sysutil library, the linker will use *profile_register in lib/sysutil/profile_extern.c* because that is the only one known. However, when the kernel is linked the one from its own directory is chosen because the linker gives a locally defined function preference over one in a library. The same mechanism is used to solve two other problems.

When to Announce

For user-space processes, the announcement can take place on the first execution of *procentry*. For the kernel, this is not the case. The kernel code contains functions like *cstart* (the first C function in MINIX 3 after boot) and when it runs endpoints are not set up yet. Since the endpoint is needed for registration of a profiled process, the announcement is not possible yet. Therefore a function *profile_get_announce* is defined in *lib/sysutil/profile_extern.c* as well as in *kernel/profile.c* for separate linking. In case of user-space processes, it returns *CPROF_ANNOUNCE_OTHER*. This is defined as 1 in *include/minix/profile.h* so a user processes calls *profile_register* on the first run of *procentry*. In case of the kernel, *CPROF_ACCOUNCE_KERNEL* is returned. This is defined as 10000, so *profile_register* is called on *procentry*'s 10000^{th} run. The number was a rough guess that works well because it turns out that at this run the kernel has finished setting up its data structures. It might have sounded more logical if this was done from the kernel itself at the exact moment it is ready. However, that would have been very impractical to implement since the profiling data structures that are registered at announcement are private to the sysutil library. It made more sense to do it this way.

Profiling Table Size

The profiling table size depends on the size of a slot and the number of slots. The size of a slots is largely depended on *CPROF_CPATH_MAX_LEN* (in *include/minix/profile.h*), which must be large enough to fit the largest call path. At the moment it is defined as 256 bytes because some servers (notably FS and INET) have call path lengths that can exceed 200 characters. Since the number of unique call paths (which is the number of slots needed) can exceed 2000 in these processes, the table can easily reach 1 MB or more when sizing it with some additional headroom in mind.

For user-space processes this is not much of a problem, except that the boot image is loaded in extended memory and therefore much fit in 16 MB. Since there are only about eight processes in the boot image, this is still OK. For the kernel-space processes however there is a real problem: their image is loaded in lower memory and must therefore fit in 640 KB. To solve this, I let kernel and user-space processes have different table sizes. They are *CPROF_TABLE_SIZE_OTHER* (3000 slots) for user-space processes and *CPROF_TABLE_SIZE_KERNEL* (1500 slots) for the kernel. The different tables are declared in *lib/sysutil/profile_extern.c* and in *kernel/profile.c*. The function *profile_get_tbl_size* is defined twice as well so the profiling functions can find out the size of their table at runtime.

To illustrate the effect of call profiling on the size of the profiled processes, the output of *make hdboot* is shown below for a regular build and one with call profiling enabled.

```
  text     data      bss      size
 24208     3384    44396     71988  ../kernel/kernel
 21312     3116    93940    118368  ../servers/pm/pm
 41536     5232  5019704   5066472  ../servers/fs/fs
  6848      840    20388     28076  ../servers/rs/rs
  3280      464     1808      5552  ../servers/ds/ds
 27072     5696    48104     80872  ../drivers/tty/tty
  6144   574504     3068    583716  ../drivers/memory/memory
  5968      572    63280     69820  ../drivers/log/log
  7056     2412     1356     10824  ../servers/init/init
------   ------   ------   -------
143424   596220  5296044   6035688  total

  text     data      bss      size
 30560     5096   495392    531048  ../kernel/kernel
 25504     4360   952252    982116  ../servers/pm/pm
 47440     7492  5878016   5932948  ../servers/fs/fs
  9152     1116   878700    888968  ../servers/rs/rs
  5648      728   860120    866496  ../servers/ds/ds
 31728     7172   906416    945316  ../drivers/tty/tty
  8512   574752   861380   1444644  ../drivers/memory/memory
  8736     1028   921592    931356  ../drivers/log/log
  7056     2412     1356     10824  ../servers/init/init
------   ------   ------   -------
174336   604156 11755224  12533716  total
```

3.6.2 Kernel-Space Issue: Reentrancy

As mentioned in the previous paragraphs, the profiling functions *procentry*/*procexit* contain reentrancy checks. The checks are really there for when the system task is profiled. When a profiling function is running in this process, it is entirely possible that it is interrupted during its execution. The (software or hardware) interrupt will cause a handler to be run. If this handler is a C function, a call to *procentry* will be done, and *procexit* will be called when the handler returns. Possibly even more calls to the profiling functions happen, if the handler has function calls in it. The kernel-space processes all use the same profiling data structures and the profiling functions do not take reentrancy into account. This has a corrupting effect on the time calculation when interrupts cause a handler in *KERNEL* or *CLOCK* to run when *procentry* or *procexit* was running at that time on behalf of the system task.

To prevent this, lock variables are used to allow only one instance of the profiling functions to run at a time. The consequence is that some interrupt handlers are missed.

3.6.3 Kernel-Space Issue: The Call Path

Since the kernel-space processes (*KERNEL*, *CLOCK*, *IDLE* and *SYSTEM*) all run in one address space, they use the same profiling data structures. This means they have the same profiling stack, hash table, etc. As a result of this, the call paths generated are influenced by interrupts. The call paths are littered with interrupt handlers. Even though the full call paths are not so useful because of this, this does not need to be a problem. The paths in kernel-space are very short and if the analyzer script (to be discussed soon) is instructed to show the totals per function the results are still very useful.

3.7 WORK DONE AT THE APPLICATION LEVEL

3.7.1 Control Tool: profile

The user program in *commands/profile/* that was described earlier has options to control call profiling as well. When a reset of the profiling tables is requested, the program simply does a *CPROFILE* system call with the appropriate parameter and exits. When the user wants to get the profiling results the following is done. The program allocates memory for the profiling data. A system call *CPROFILE* is done which passes the operating system the following parameters: the location and size of the allocated memory and the location of an info struct. When *CPROFILE* returns, the user program reads the *mem_used* variable from the info struct. Just like with statistical profiling the value holds -1 when there was not sufficient memory, otherwise it holds the number of bytes used. If this was found OK, it writes to a file a header containing the value of *CPROF_CPATH_MAX_LEN* (which represents the length of the call path field in the hash table) and the value of *CPROF_PROCNAME_LEN* (fixed length reserved for process names) as defined in *include/minix/profile.h*. The reason for this is explained below. It then writes the profiling data from memory to the file and exits.

3.7.2 Analyzer Tool: cprofalyze.pl

The Perl program *cprofalyze.pl* is available to analyze raw profiling data files. This script starts with reading the header with call path and process name lengths. The reason the C program wrote them in the data file is that the Perl tool is not aware of the C language header file where these values are defined as macros. Changes to the macros affect the structure of the profiling tables. By having the C user program write them in the data file, the Perl tool is informed about them. This solution avoids hard coding of the same values in the Perl tool which would be redundant and could cause problems if the Perl tool was not updated after the macros were changed. Also, the Perl tool can process data files created by different operating system builds (with different values for these macros).

After the header, the actual profiling data follows. For each system process, the kernel has written the process name, the number of slots used and those actual slots. The process name and the number of slots are of fixed length. The length of the data following them depends on the used number of slots, but since this was just read the analyzer knows how much to read before for the next process data starts. The fields for each slot: unique call path, number of calls and cycles used are read into variables. There is one field that is not used: the hash table chaining field. It was only of use when the table was actively used during profiling. The reason that it was copied to the user program is simplicity: it would take more complexity to omit it during copying. The size of the field is only one word per slot: about 2% of the table. The chaining could have been done in a separate array, but this would have cost extra lookups during profiling. There are several command line options to this script that allow control over the output and the results are sorted and formatted according to the users wish before being outputted. The analyzer does not calculate percentages because most of the time they are not useful with blocking functions taking up so much time.

A mandatory parameter to *cprofalyze.pl* is the CPU clock speed. This is used to calculate milliseconds from the *cycles* field in the hash.

In addition to the data file header mentioned earlier, there is one more header in the data file for both call and statistical profiling. This is the first line of the file and it indicates whether the file contains data for statistical or call profiling. This allows the analyzer scripts to do a sanity check before processing and to suggest the correct analyzer script if the wrong one is used.

3.8 WHAT CAN AND WHAT CANNOT BE PROFILED

Contrary to the statistical profiler, the kernel task (*KERNEL*) can be profiled. This allows us to have a better understanding of time spent in the kernel.

Unfortunately due to the way call profiling works, blocking time (on I/O and sending/receiving of messages) is included in cycle attribution. The profiler in each profiled process has no notion of the state of the process.

The -Rcem-p option of ACK only works on C functions. There is functionality in the kernel that is written in assembly language (most of *SYSTEM*'s cycles are spent in it, as can be seen in the statistical profiling results), and this will not turn up in the profiler. Specifically these are the memory copying/setting, low-level message passing code and the *_hwint00-15* interrupt handlers. However, a lot can be learned already from the names of the C functions that call the assembly "functions". The time spent in assembly is attributed to the calling C function: to the profiler it looks as if the assembly cycles were spent inline and they are attributed as such. This makes up somewhat for the lack of assembly support. As a workaround, to point out the exact time spent in assembly language, the calls to it could be put in wrapping C functions with names like *asm_phys_copy*.

Library functions are not profiled because I got some strange effects when compiling them with –Rcem-p. Also library dependencies would be more complicated because the programs in *commands/* use them and require *Makefile* adaptation. My impression is that the system processes do not use library function a lot, though.

Although the TTY driver can be profiled like any other process, it shows strange behavior on the console when it is compiled with call profiling enabled. Terminals over the network are not affected. This is the reason that the $(CPROFILE) flag in *drivers/tty/Makefile* was taken out. It can be put back if profiling of TTY is needed.

3.9 PROFILING RESULTS

Some example results of profiling the kernel and FS are in appendix C. An analysis follows in chapter 5.

4 USAGE

4.1 USAGE AND REQUIREMENTS

Profiling is controlled using the *profile* program. This is its Usage: output:

```
Statistical Profiling:
  profile start [-m memsize] [-o outfile] [-f frequency]
  profile stop

Call Profiling:
  profile get   [-m memsize] [-o outfile]
  profile reset

   - memsize in MB, default: 64
   - default output file: profile.{stat|call}.out
   - sample frequencies (default: 6):
      3    8192 Hz          10     64 Hz
      4    4096 Hz          11     32 Hz
      5    2048 Hz          12     16 Hz
      6    1024 Hz          13      8 Hz
      7     512 Hz          14      4 Hz
      8     256 Hz          15      2 Hz
      9     128 Hz

Use [sc]profalyze.pl to analyze output file.
```

Before you can profile, you have to prepare the operating system. See the paragraphs below. To run the analyzer scripts, Perl must be installed. This is easily done using the *packman* installer program that is part of MINIX 3. No additional Perl modules are needed. For statistical profiling the *sprofalyze.pl* script is used. This is its Usage: output:

```
  sprofalyze.pl [-p percentage] file ...

    percentage  print only processes/functions >= percentage
```

And for call profiling the *cprofalyze.pl* script is used. This is its Usage: output:

```
  cprofalyze.pl <clock> [-f] [-aoct] [-i] [-n number] file ...

      clock  CPU clock of source machine in MHz (mandatory)
         -f  print totals per function (original order lost)
         -a  sort alphabetically (default)
         -o  no sort (original order)
         -c  sort by number of calls
         -t  sort by time spent
         -n  print maximum of number lines per process
         -i  when -[ao] used: print full paths
```

4.2 PREPARATIONS FOR STATISTICAL PROFILING

To enable statistical profiling, set *SPROFILE* in */usr/include/minix/config.h* to 1 and recompile the kernel (e.g. *make hdboot* in */usr/src/tools/*). Boot the new kernel. Do *make*

clean install in */usr/src/commands/profile/*. You should now be able to start and stop profiling.

Warning: the BIOS time may be wrong after doing a statistical profiling run! The BIOS time should be checked afterwards and corrected if needed.

The locations of all system executables are configured in *sprofalyze.pl*. When adding a driver or server to the operating system the location of the new executable should be added there. This allows the script to extract its symbol table and include it in system profiling.

4.3 PREPARATIONS FOR CALL PROFILING

To enable call profiling, set the *CPROFILE* flag in */usr/src/include/minix/config.h* (note, this is the *config.h* in the source tree). Do *export CPROFILE=-Rcem-p* in your shell to set this environment variable. Now build the operating system, including libraries, by doing *make fresh hdboot* in */usr/src/tools/*. Be aware that this will overwrite the *config.h* in */usr/include/minix/* with the one in the source tree! If you have a customized *config.h*, back it up before doing *make fresh*. Boot the new kernel. Do *make clean install* in */usr/src/commands/profile/*. You should now be able to reset and get profiling tables. The kernel prints a notice on the console if a get or reset action is taken; it includes the names of the profiled processes.

5 PROFILING RESULTS

All results presented here are based on profiling runs during *make clean install* in *commands/*.

5.1 KERNEL-SPACE PROCESSES

5.1.1 Time Distribution by Function

The following chart was made from the call profiler results listed in appendix C and represents *KERNEL*, *CLOCK* and *SYSTEM*. The clock task scored only 0.3% and is therefore in the other category.

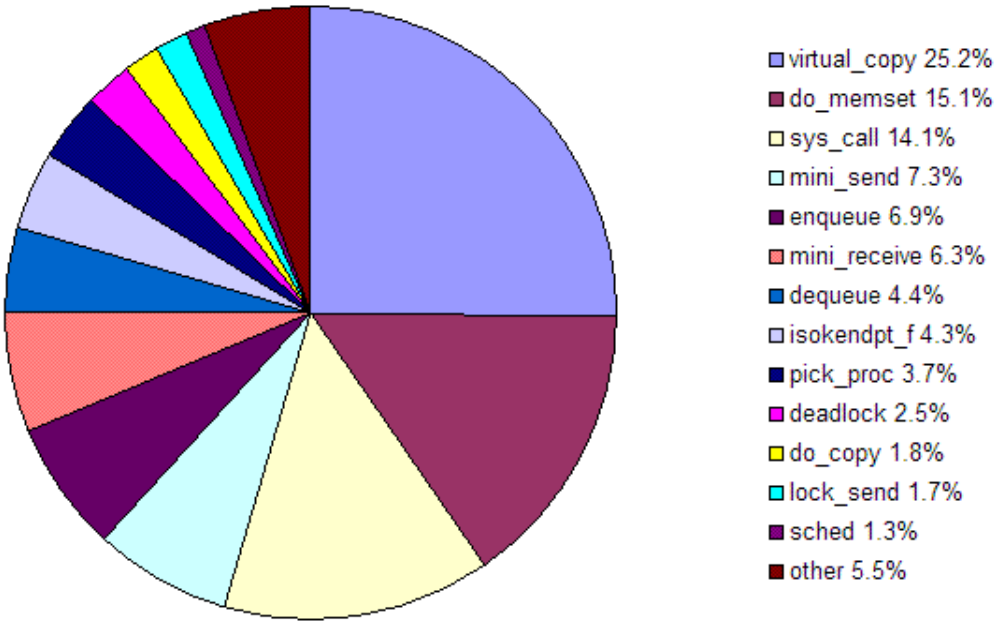

Figure 5-1. Distribution of time in kernel-space.

5.1.2 Time Distribution by Category

Using the function results from the previous paragraph, a distribution across certain categories of work is now made.

Memory Copying and Clearing

Virtual_copy (through *do_copy*), *do_copy* and *do_memset* together account for about **42%** of the time. This can be explained by the high number of process creations during

the *make* run. Since MINIX 3 uses segmented memory (instead of a more common virtual memory system), for each process creation, through forking, a full process image must be copied and cleared.

Note that this time is spent by *SYSTEM*, since the *do_** functions are kernel calls.

IPC

IPC functions *mini_send*, *mini_receive* and *lock_send* account to 15.3% of time spent. If *enqueue* and *dequeue* (by far, most calls to these are made by IPC functions) and *pick_proc* (only called by *enqueue* and *dequeue)* are included the total is about **30%**.

This time is spent by *KERNEL*.

Trapping

Time spent for handling software interrupts is the total of *sys_call* and *deadlock* (only called from *sys_call*): about **17%**.

This time is spent by *KERNEL*.

What Remains

The remaining kernel-space time is spent in scheduling and in kernel calls other then *do_copy* and *do_memset*.

The time spent by *isokendpt_f* is spread over the categories above.

5.1.3 Conclusions

Time spent in MINIX 3 kernel-space when the system is busy running *make* jobs that compile and install lots of source code, can be clearly categorized into distinct types of work. Almost half goes into memory copying and clearing, almost a third is in message passing, a sixth is spent processing traps. The remaining 7% goes to scheduling, accounting/time keeping, kernel calls other then for memory work, etc.

Another categorization is also possible: about half is spent by the system task (memory work) and the other half is spent by the kernel task (the rest).

One thing is not measured here: the assembly language *_hwint** interrupt handlers. They are not called by C functions so they will not appear in the call profiling results. The results are based on the assumption that they do not use a lot of time.

5.2 ALL MINIX 3 PROCESSES

A distribution of the time spent by all MINIX 3 processes is not easily made. The call profiler is not helpful here since the blocking time included in the user-space processes makes it unsuitable. The statistical profiler might be useful, but it does not measure the kernel task and attributes IPC time that really belongs to the kernel task to other processes due to interrupt delay. Still, I find the distribution of time across all MINIX 3 processes important and therefore I will make an effort to approximate it.

5.2.1 Approximating

From the results of a statistical profiling run, the IPC time attributed to processes due to the interrupt delay effect is taken out. The percentages for each process' part of MINIX 3 time are then recalculated. In the previous paragraph, we found that system and kernel task spend roughly as much time. Using that knowledge, we will use for the kernel task time the same number as the system task and normalize the percentages. The following graph is based on these calculations, which can be found in detail in appendix D.

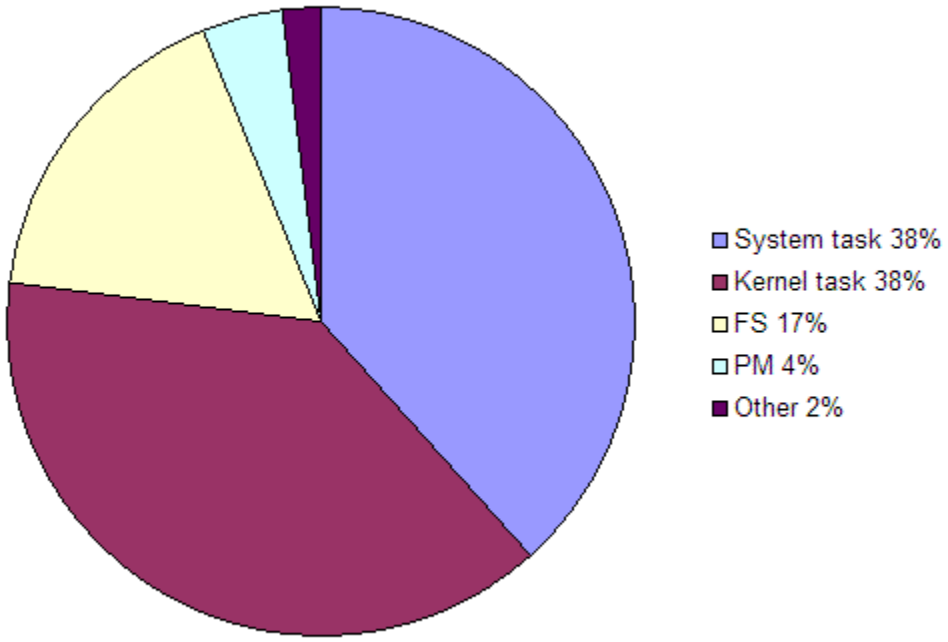

Figure 5-2. Approximation of time distribution of all MINIX 3 processes.

5.2.2 Message Processing Time

In support of the approximation done, notice in the last part of the calculations in appendix D that the IPC time taken out of the processes is 22% of MINIX 3 time. This is not far off from the 30% of kernel-space time that IPC takes as concluded in 5.1.3:
(38% + 38%) * 30% = 23%.

The distribution of the IPC time (as part of total MINIX 3 time) is as follows: FS 11.7%, system task 6.8% and PM 3.5%. This time is spent by the kernel task, on behalf of the processes mentioned.

We will now calculate the time spent by the CPU on MINIX 3 message handling, as part of everything that is running (MINIX 3 + user programs). From appendix C, statistical profiling results, we see that the balance MINIX 3 / user land ticks is 23 / 33. So the percentage of MINIX 3 message processing is 23 / (23 + 33) * 22% = 9%.

5.2.3 Conclusions

About three quarters of MINIX 3 time, when running the *make* jobs, is spent in kernel-space. So, even though the amount of code in kernel-space is small it gets executed a lot. FS takes the second place which is no surprise considering the file system operations required during the *make* jobs. PM takes the third place, this is also no surprise because its work is required for the many processes that are created and exited.

These are the times spent on message processing:

As part of kernel-space	30%
As part of MINIX 3	22%
As part of everything on CPU	9%

So, the performance penalty of MINIX 3 being based on messages seems to be around 10%. In my opinion this is a small price to pay, especially considering the increases in speed seen in hardware.

6 RECONCILATION

Although statistical and call profiler are very different in their approach, their measurements should be reconcilable. In fact, if their results are similar both profilers gain credibility. Unfortunately, a one-on-one comparison is not possible because of the following reasons:

1. The statistical profiler is more fine grained then the call profiler in the sense that it profiles assembly language and library functions, both of which the call profiler is not able to profile directly (instead the time is attributed to the C functions that called the assembly code/library functions).

2. The call profiler includes time spent blocking (on I/O and message sending and receiving), which is not the case with the statistical profiler.

3. The statistical profiler picks up an unknown part of the time spent by the kernel task on IPC functions due to interrupt delay (as discussed in 2.8.2).

Regarding point 1: in order to make comparison possible, I will analyze the functions reported by the call profiler for embedded assembly and library calls in an effort to reconcile them with statistical profiling results. However, since these assembly and library functions could have been called by different C functions, it is impossible to attribute them to a specific one. However, in most cases the ‘mapping’ is straightforward.

Regarding point 2: blocking functions in the call profiling results will be ignored.

Regarding point 3: I will take out high scoring IPC function(s) of the statistical results and recalculate the percentages to simulate non-existence of the interrupt delay effect.

I will try to reconcile the system task as well some user-space processes: FS and PM. All profiling runs were done during a *make clean install* in *commands/*. For call profiling results, percentages were calculated from the output of the analyzer program and used instead of the regular calls/cycles results.

6.1 RECONCILING THE SYSTEM TASK

Statistical Profiler				Call Profiler	
1: unaltered results of a profiling run 2: __receive set to 0% 3: percentages normalized					
	1	**2**	**3**		
_phys_co	43.1%	43.1%	50.9%	virtual_copy	55.7%
fill_sta	28.1%	28.1%	33.2%	do_memset	31.9%
__receiv	15.4%	0.0%	0.0%	do_copy	2.8%
_lock_se	3.3%	3.3%	3.9%	umap_local	1.9%
_isokend	2.7%	2.7%	3.2%	do_getinfo	1.5%
_clear_e	1.4%	1.4%	1.7%	clear_endpoint	1.4%
_sys_tas	0.8%	0.8%	0.9%	do_umap	1.2%
_virtual	0.7%	0.7%	0.8%	do_devio	0.9%
_do_copy	0.5%	0.5%	0.6%	do_newmap	0.9%
_do_vdev	0.5%	0.5%	0.6%	do_vdevio	0.6%
_umap_lo	0.4%	0.4%	0.5%	do_setalarm	0.2%
no0	0.4%	0.4%	0.5%	do_fork	0.2%
pc_small	0.4%	0.4%	0.5%	do_times	0.2%
_inb	0.3%	0.3%	0.4%	cause_alarm	0.1%
_do_geti	0.3%	0.3%	0.4%	clear_proc	0.1%
_alloc_s	0.3%	0.3%	0.4%	do_exec	0.1%
_do_newm	0.2%	0.2%	0.2%	do_irqctl	0.1%
_do_umap	0.2%	0.2%	0.2%	<0.1%	0.2%
_sdesc	0.2%	0.2%	0.2%		
_outb	0.1%	0.1%	0.1%		
_tmrs_cl	0.1%	0.1%	0.1%		
<0.1%	0.6%	0.6%	0.7%		
	100.0%	84.6%	100.0%		100.0%

Virtual_copy in *kernel/system.c* calls *_phys_copy* in *kernel/klib386.s*, so the biggest cycle-eater can be reconciled reasonably well being so close percentage-wise in the results of both profilers. The same is true for *do_memset* in *system/do_memset.c* which calls *_phys_memset* in *kernel/klib386.s* where *fill_start* is a label. It may seem strange that *fill_start* would have a higher percentage then *do_memset*, but keep in mind that *fill_start* can be reached in other ways then through *do_memset* (for example, through *do_exec*). This is true for many other functions and makes further reconciliation a daunting task. The other functions score much lower than the two top scorers though; together number 1 and 2 account for 80-90% of the CPU cycles used by *SYSTEM*.

How Call Profiling Results were Retrieved

In order to get the results for the system task only, in *kernel/* I compiled only *system.c* and *system/** with the –Rcem-p option. Also, *profile_register* in *kernel/profile.c* was changed slightly to have *SYSTEM* announce itself instead of *KERNEL*. I used the –f –t parameters in the analyzing script to show totals per function, sorted by time spent.

6.2 RECONCILING USER-SPACE PROCESSES

I have put quite some effort in trying to reconcile FS as well as PM results, and was not able to make a 'mapping'. Even though many functions end up roughly in the same class of cycle usage, many others do not.

The problem is that it is not possible to take the blocking time out of the call profiling results; this time is spread over many functions. These functions may spend time blocking as well as running, and the profiler does not differentiate between those states.

6.3 CONCLUSION

Directly comparing user-space processes is as good as impossible due to the very different nature of the profilers.

In the system task blocking functions played no role. This implied that the call profiling results were blocking time 'free' so the comparison to the statistical results (corrected for interrupt delay pollution) was possible. In this comparison, for the significantly scoring functions the results were almost identical.

7 CONCLUSIONS

7.1 WORK DONE

In this thesis I described the making of performance measuring tools for the MINIX 3 operating system. Two known methods of measuring program performance were implemented: statistical profiling and call profiling.

The actual profiling logic was implemented in kernel-space and user-space processes of the operating system. Data acquisition and analysis tools were implemented on the application level. Efforts were taken to minimize the impact of measuring on the results.

The work was done in such a way that measuring can be enabled or disabled from a single configuration file so it does not stand in the way of the operating system and does not affect performance when not enabled. In fact, when disabled the measuring code is not even compiled.

The results of both profilers were analyzed in chapter 5 and reconciled in chapter 6.

7.2 COMPARING THE PROFILERS

7.2.1 Advantages and Disadvantages of Statistical Profiling

The methods each have their advantages and disadvantages.

The statistical profiler is able to measure assembly and library functions. Its results are not polluted by time spent in blocking functions. It has hardly an impact on performance (the system runs only a few percent slower during a run).

However, the kernel task cannot be profiled. In addition the profiler suffers from interrupt delay pollution causing an unknown part of the time spent by the kernel task on IPC functions to show up in its results. It does not track whole call paths. It only sees functions that execute often enough to turn up in the results.

7.2.2 Advantages and Disadvantages of Call Profiling

The call profiler profiles full call paths. This allows the input to be used not for just measuring individual functions but also to gain insights into the execution paths of processes (for example, call graphs could be created from the results). It can profile all processes, including the kernel task, and it does not miss a single function call. There is no pollution from interrupt delay.

The disadvantages are that assembly and library function are not profiled; instead the time spent is included in the calling C functions. Blocking time (on I/O, sending and receiving messages) is included in the results which makes it hard to find out where time was actually spent running. The time spent in a function can only be measured when the function returns, this is a problem for functions that have not returned when the profiling results are retrieved (for example: the *main* function of a process). The system is two to three times slower during call profiling.

7.2.3 Overview of Differences

	Statistical Profiler	Call Profiler
Measures:		
Every single function call		•
Full call paths		•
Kernel task		•
Assembly code	•	
Library functions	•	
Hampered by:		
Blocking functions		•
Interrupt delay	•	
Functions that have not returned yet		•
Impact on performance	low	high

7.3 FINAL WORDS

The results analysis shows that during compiling and installing of many C source code files, MINIX 3 spends most of its time copying and clearing memory to support the creation of the many (compiler, linker, installer) processes. This is due to the segmented memory system that MINIX 3 uses and will probably be different when MINIX 3 changes to a virtual memory system. Of all the cycles spent (MINIX 3 + user land applications) about 10% of CPU goes to message processing.

The reconciliation of the profilers was not possible on all processes because of the fundamental differences between what is included in the results. Namely, blocking time is included in the results of the call profiler but it is not in the results of the statistical profiler. However, when comparing the only process that would allow a comparison because of the lack of time spent blocking and the availability of results in both profilers (the system task) the high scoring results were very similar. This seems to indicate that the measurements can be taken serious, but the user has to take in mind the characteristics of each profiler when weighing its results.

The call profiler shows its strength when measuring kernel-space processes because there it is not hampered by blocking functions. In user-space this profiler is useful to reveal execution paths and the number of times functions were called.

The statistical profiler shines when profiling user-space processes because the many blocking functions do not affect the results. The interrupt delay effect should be taken into account however.

Despite the shortcomings of each profiler, together they should be able to function as a useful tool in the further development of the MINIX 3 operating system.

8 FUTURE WORK

8.1 PORTABILITY

The CMOS clock used for statistical profiling is probably quite IBM PC specific. For a port to another platform, a timer should be available and the functions related to it, in *kernel/profile.c*, should be adapted. Depending on the frequencies available on the alternative timer, a few changes may need to be made to the user program as well to reflect them.

For call profiling, the Pentium cycle counter is used. Such a counter may not be available on non-x86 based CPU's. If such a counter is available, the assembly code to read it in *read_tsc.s* will have to be adjusted to make use of it.

Call profiling depends on the –Rcem-p option of ACK. A similar feature should be available when a different compiler is used.

8.2 EXTENDING TO USER PROGRAM PROFILING

Both statistical and call profiling works on MINIX 3 processes. It is imaginable that people will want to use the profilers for user programs as well (although users have other options like the Gprof functionality of GCC). Profiling user programs using the statistical and call profilers should be possible, but it would require at least the following changes to the code:

For statistical profiling the clock handler in *kernel/clock.c* needs to be adapted to take samples of non-MINIX 3 processes; the *sprofalyze.pl* script should be changed not to complain and exit when it encounter samples that are not accounted for in the list of executables configured within the script (all user programs will now be profiled, while the user probably only wants to see his own); the user has to add the location of his executable in the *sprofalyze.pl* script.

For call profiling, the user program should be compiled with ACK using the –Rcem-p option. In order to allow the process to announce itself to the kernel, a system call needs to be created that does the *PROFBUF* kernel call for the program.

8.3 IMPROVING THE STATISTICAL PROFILER

The statistical profiler could be improved by using a non-maskable interrupt instead of the RTC interrupt. This will yield the following improvements:

- the kernel task can be profiled
- profiler interrupts are not delayed

If call paths are not needed, the statistical profiler would then be the preferred profiler by far because it would have only advantages over the call profiler. Probably the kernel task does not need to be made reentrant for this since the profiling code does not interfere with any data structure of the kernel.

A statistical profiler for Linux, OProfile, uses hardware performance counters for the profiling clock. The interrupts are then generated by the CPU itself, and are non-maskable. It is a little bit more complicated because different CPU brands and models seem to be differently programmed.

Another option would be to look if the kernel task could be made reentrant some way (possibly just for profiling), for instance by not masking the CMOS timer.

APPENDIX A – FILES ADDED/CHANGED

These files in MINIX 3 have been changed or added for the profiling project.

```
changed     commands/Makefile
new         commands/profile/Makefile
new         commands/profile/cprofalyze.pl
new         commands/profile/profile.c
new         commands/profile/sprofalyze.pl
changed     drivers/at_wini/Makefile
changed     drivers/bios_wini/Makefile
changed     drivers/cmos/Makefile
changed     drivers/dp8390/Makefile
changed     drivers/dpeth/Makefile
changed     drivers/floppy/Makefile
changed     drivers/fxp/Makefile
changed     drivers/lance/Makefile
changed     drivers/libdriver/Makefile
changed     drivers/log/Makefile
changed     drivers/memory/Makefile
changed     drivers/memory/ramdisk/proto
changed     drivers/pci/Makefile
changed     drivers/printer/Makefile
changed     drivers/random/Makefile
changed     drivers/rescue/Makefile
changed     drivers/rtl8139/Makefile
changed     drivers/sb16/Makefile
changed     drivers/ti1225/Makefile
changed     drivers/tty/Makefile
changed     drivers/tty/keymaps/Makefile
changed     include/ibm/interrupt.h
changed     include/minix/callnr.h
changed     include/minix/com.h
changed     include/minix/config.h
new         include/minix/profile.h
changed     include/minix/syslib.h
changed     include/unistd.h
changed     kernel/Makefile
changed     kernel/kernel.h
changed     kernel/main.c
new         kernel/profile.c                    partly listed in appendix B
new         kernel/profile.h
changed     kernel/system/Makefile
new         kernel/system/do_cprofile.c
new         kernel/system/do_profbuf.c
new         kernel/system/do_sprofile.c
changed     kernel/system.c
changed     kernel/system.h
changed     kernel/table.c
changed     lib/other/Makefile.in
new         lib/other/_cprofile.c
new         lib/other/_sprofile.c
changed     lib/other/syscall.c
changed     lib/syscall/Makefile.in
new         lib/syscall/cprofile.s
new         lib/syscall/sprofile.s
changed     lib/syslib/Makefile.in
new         lib/syslib/sys_cprof.c
new         lib/syslib/sys_profbuf.c
new         lib/syslib/sys_sprof.c
changed     lib/sysutil/Makefile.in
new         lib/sysutil/profile.c               listed in appendix B
new         lib/sysutil/profile_extern.c
new         lib/sysutil/read_tsc.h
new         lib/sysutil/read_tsc.s
```

```
changed     servers/ds/Makefile
changed     servers/fs/Makefile
changed     servers/fs/table.c
changed     servers/inet/Makefile
changed     servers/is/Makefile
changed     servers/pm/Makefile
new         servers/pm/profile.c
changed     servers/pm/proto.h
changed     servers/pm/table.c
changed     servers/rs/Makefile
changed     servers/sm/Makefile
```

APPENDIX B – SOURCE CODE

kernel/profile.c

Only the code concerning statistical profiling is shown.

```
/*
 * This file contains several functions and variables used for system
 * profiling.
 *
 * Statistical Profiling:
 *   The interrupt handler and control functions for the CMOS clock.
 *
 * Call Profiling:
 *   The table used for profiling data and a function to get its size.
 *
 *   The function used by kernel-space processes to register the locations
 *   of their control struct and profiling table.
*/

#include <minix/config.h>

#if SPROFILE || CPROFILE

#include <minix/profile.h>
#include "kernel.h"
#include "profile.h"
#include "proc.h"

#endif

#if SPROFILE

#include <string.h>
#include <ibm/cmos.h>

/* Function prototype for the CMOS clock handler. */
FORWARD _PROTOTYPE( int cmos_clock_handler, (irq_hook_t *hook) );

/* A hook for the CMOS clock interrupt handler. */
PRIVATE irq_hook_t cmos_clock_hook;

/*===========================================================================*
 *                            init_cmos_clock                                *
 *===========================================================================*/
PUBLIC void init_cmos_clock(unsigned freq)
{
  int r;
  /* Register interrupt handler for statistical system profiling.
   * This uses the CMOS timer.
   */
  cmos_clock_hook.proc_nr_e = CLOCK;
  put_irq_handler(&cmos_clock_hook, CMOS_CLOCK_IRQ, cmos_clock_handler);
  enable_irq(&cmos_clock_hook);

  intr_disable();

  /* Set CMOS timer frequency. */
  outb(RTC_INDEX, RTC_REG_A);
  outb(RTC_IO, RTC_A_DV_OK | freq);
  /* Enable CMOS timer interrupts. */
  outb(RTC_INDEX, RTC_REG_B);
  r = inb(RTC_IO);
  outb(RTC_INDEX, RTC_REG_B);
```

```
  outb(RTC_IO, r | RTC_B_PIE);
  /* Mandatory read of CMOS register to enable timer interrupts. */
  outb(RTC_INDEX, RTC_REG_C);
  inb(RTC_IO);

  intr_enable();
}

/*===========================================================================*
 *                              cmos_clock_stop                              *
 *===========================================================================*/
PUBLIC void stop_cmos_clock()
{
  int r;

  intr_disable();

  /* Disable CMOS timer interrupts. */
  outb(RTC_INDEX, RTC_REG_B);
  r = inb(RTC_IO);
  outb(RTC_INDEX, RTC_REG_B);
  outb(RTC_IO, r & !RTC_B_PIE);

  intr_enable();

  /* Unregister interrupt handler. */
  disable_irq(&cmos_clock_hook);
  rm_irq_handler(&cmos_clock_hook);
}

/*===========================================================================*
 *                              cmos_clock_handler                           *
 *===========================================================================*/
PRIVATE int cmos_clock_handler(hook)
irq_hook_t *hook;
{
/* This executes on every tick of the CMOS timer. */

  /* Are we profiling, and profiling memory not full? */
  if (!sprofiling || sprof_info.mem_used == -1) return (1);

  /* Check if enough memory available before writing sample. */
  if (sprof_info.mem_used + sizeof(sprof_info) > sprof_mem_size) {
        sprof_info.mem_used = -1;
        return(1);
  }

  /* All is OK */

  /* Idle process? */
  if (priv(proc_ptr)->s_proc_nr == IDLE) {
        sprof_info.idle_samples++;
  } else
  /* Runnable system process? */
  if (priv(proc_ptr)->s_flags & SYS_PROC && !proc_ptr->p_rts_flags) {
        /* Note: k_reenter is always 0 here. */

        /* Store sample (process name and program counter). */
        phys_copy(vir2phys(proc_ptr->p_name),
                (phys_bytes) (sprof_data_addr + sprof_info.mem_used),
                (phys_bytes) strlen(proc_ptr->p_name));

        phys_copy(vir2phys(&proc_ptr->p_reg.pc),
                (phys_bytes) (sprof_data_addr+sprof_info.mem_used +
                                        sizeof(proc_ptr->p_name)),
                (phys_bytes) sizeof(proc_ptr->p_reg.pc));

        sprof_info.mem_used += sizeof(sprof_sample);

        sprof_info.system_samples++;
  } else {
```

```
        /* User process. */
        sprof_info.user_samples++;
    }

    sprof_info.total_samples++;

    /* Mandatory read of CMOS register to re-enable timer interrupts. */
    outb(RTC_INDEX, RTC_REG_C);
    inb(RTC_IO);

    return(1);                                  /* reenable interrupts */
}

#endif /* SPROFILE */
```

lib/sysutil/profile.c

```
/*
 * profile.c - library functions for call profiling
 *
 * For processes that were compiled using ACK with the -Rcem-p option,
 * procentry and procexit will be called on entry and exit of their
 * functions.  Procentry/procexit are implemented here as generic library
 * functions.
 */

#include <lib.h>

#if CPROFILE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <minix/profile.h>
#include <minix/syslib.h>
#include <minix/u64.h>
#include "read_tsc.h"

#define U64_LO 0
#define U64_HI 1

PRIVATE char cpath[CPROF_CPATH_MAX_LEN];     /* current call path string */
PRIVATE int cpath_len;                       /* current call path len */
PRIVATE struct cprof_tbl_s *cprof_slot;      /* slot of current function */
PRIVATE struct stack_s {                     /* stack entry */
               int cpath_len;                /* call path len */
               struct cprof_tbl_s *slot;     /* table slot */
               u64_t start_1;                /* count @ begin of procentry */
               u64_t start_2;                /* count @ end of procentry */
               u64_t spent_deeper;           /* spent in called functions */
};
PRIVATE struct stack_s cprof_stk[CPROF_STACK_SIZE];  /* stack */
PRIVATE int cprof_stk_top;                           /* top of stack */
EXTERN struct cprof_tbl_s cprof_tbl[];               /* hash table */
PRIVATE int cprof_tbl_size;                          /* nr of slots */
PRIVATE struct cprof_tbl_s *idx[CPROF_INDEX_SIZE];   /* index to table */
PRIVATE struct cprof_ctl_s control;          /* for comms with kernel */
PRIVATE int cprof_announce;                  /* announce on n-th execution
                                              * of procentry */
PRIVATE int cprof_locked;                    /* for reentrancy */

_PROTOTYPE(void procentry, (char *name) );
_PROTOTYPE(void procexit, (char *name) );

FORWARD _PROTOTYPE(void cprof_init, (void) );
FORWARD _PROTOTYPE(void reset, (void) );
FORWARD _PROTOTYPE(void clear_tbl, (void) );


PUBLIC void procentry (name)
char *name;
{
  static int init = 0;
  unsigned hash = 0, i = 0, x = 0;
  unsigned long hi, lo;
  struct cprof_tbl_s *last;
  char c;
  u64_t start;

  /* Procentry is not reentrant. */
  if (cprof_locked) return; else cprof_locked = 1;
```

```
/* Read CPU cycle count into local variable. */
read_tsc(&start._[U64_HI], &start._[U64_LO]);

/* Run init code once after system boot. */
if (init == 0) {
      cprof_init();
      init++;
}

/* Announce once. */
if (init > -1 && init++ == cprof_announce) {
      /* Tell kernel about control structure and table locations.
      *
      * In user-space processes, the library function profile_register
      * will be used. This function does a kernel call (sys_profbuf) to
      * announce to the kernel the location of the control struct and
      * hash table. The control struct is used by the kernel to write
      * a flag if resetting of the table is requested. The location of
      * the table is needed to copy the information to the user process
      * that requests it.
      *
      * Kernel-space processes don't use the library function but have
      * their own implemention that executes logic similar to sys_profbuf.
      */
      profile_register((void *) &control, (void *) &cprof_tbl);
      init = -1;
}

/* Only continue if sane. */
if (control.err) return;

/* Check if kernel instructed to reset profiling data. */
if (control.reset) reset();

/* Increase stack. */
if (++cprof_stk_top == CPROF_STACK_SIZE) {
      printf("CPROFILE error: stack overrun\n");
      control.err |= CPROF_STACK_OVERRUN;
      return;
}

/* Save initial cycle count on stack. */
cprof_stk[cprof_stk_top].start_1._[U64_HI] = start._[U64_HI];
cprof_stk[cprof_stk_top].start_1._[U64_LO] = start._[U64_LO];

/* Check available call path len. */
if (cpath_len + strlen(name) + 1 > CPROF_CPATH_MAX_LEN) {
      printf("CPROFILE error: call path overrun\n");
      control.err |= CPROF_CPATH_OVERRUN;
      return;
}

/* Save previous call path length on stack. */
cprof_stk[cprof_stk_top].cpath_len = cpath_len;

/* Generate new call path string and length.*/
if (cprof_stk_top > 0)              /* Path is space separated. */
      cpath[cpath_len++] = ' ';
while ((c = *(name++)) != '\0')     /* Append function name. */
      cpath[cpath_len++] = c;
cpath[cpath_len] = '\0';            /* Null-termination. */

/* Calculate hash for call path string (algorithm: ELF). */
for (i=0; i<cpath_len; i++) {
       hash = (hash << 4) + cpath[i];
       if ((x = hash & 0xF0000000L) != 0) {
              hash ^= (x >> 24);
              hash &= ~x;
       }
}
hash %= CPROF_INDEX_SIZE;
```

```
  /* Look up the slot for this call path in the hash table. */
  for (cprof_slot = idx[hash]; cprof_slot != 0; cprof_slot = cprof_slot->next)
       if (strcmp(cprof_slot->cpath, cpath) == 0) break;

  if (cprof_slot)
       cprof_slot->calls++;   /* found slot: update call counter */
  else {
       /* Not found: insert path into hash table. */
       if (control.slots_used == cprof_tbl_size) {
              printf("CPROFILE error: table overrun\n");
              control.err |= CPROF_TABLE_OVERRUN;
              return;
       }
       /* Set values for new slot. */
       cprof_slot = &cprof_tbl[control.slots_used++];
       strcpy(cprof_slot->cpath, cpath);
       cprof_slot->calls = 1;

       /* Update index. */
       if (idx[hash] == 0) {
              /* No collision: simple update. */
              idx[hash] = cprof_slot;
       } else {
              /* Collision: update last in chain. */
              for (last = idx[hash]; last->next != 0; last = last->next);
              last->next = cprof_slot;
       }
  }
  /* Save slot on stack. */
  cprof_stk[cprof_stk_top].slot = cprof_slot;

  /* Again save CPU cycle count on stack. */
  read_tsc(&cprof_stk[cprof_stk_top].start_2._[U64_HI],
              &cprof_stk[cprof_stk_top].start_2._[U64_LO]);
  cprof_locked = 0;
}


PUBLIC void procexit (name)
char *name;
{
  u64_t stop, spent;

  /* Procexit is not reentrant. */
  if (cprof_locked) return; else cprof_locked = 1;

  /* First thing: read CPU cycle count into local variable. */
  read_tsc(&stop._[U64_HI], &stop._[U64_LO]);

  /* Only continue if sane. */
  if (control.err) return;

  /* Update cycle count for this call path. Exclude time spent in procentry/
   * procexit by using measurements taken at end of procentry and begin of
   * procexit (the "small" difference). This way, only the call overhead for
   * the procentry/procexit functions will be attributed to this call path,
   * not the procentry/procexit cycles.
   */

  /* Calculate "small" difference. */
  spent = sub64(stop, cprof_stk[cprof_stk_top].start_2);
  cprof_stk[cprof_stk_top].slot->cycles =
       add64(cprof_stk[cprof_stk_top].slot->cycles,
              sub64(spent, cprof_stk[cprof_stk_top].spent_deeper));

  /* Clear spent_deeper for call level we're leaving. */
  cprof_stk[cprof_stk_top].spent_deeper._[U64_LO] = 0;
  cprof_stk[cprof_stk_top].spent_deeper._[U64_HI] = 0;

  /* Adjust call path string and stack. */
```

```
  cpath_len = cprof_stk[cprof_stk_top].cpath_len;
  cpath[cpath_len] = '\0';

  /* Update spent_deeper for call level below. Include time spent in
   * procentry/procexit by using measurements taken at begin of procentry
   * and end of procexit (the "big" difference). This way the time spent in
   * procentry/procexit will be included in spent_deeper and therefore, since
   * this value is substracted from the lower call level, it will not be
   * attributed to any call path. This way, pollution of the statistics
   * because of procentry/procexit is kept to a minimum.
   */

  /* Read CPU cycle count. */
  read_tsc(&stop._[U64_HI], &stop._[U64_LO]);

  /* Calculate "big" difference. */
  spent = sub64(stop, cprof_stk[cprof_stk_top].start_1);
  cprof_stk_top--;                                      /* decrease stack */
  if (cprof_stk_top >= 0)         /* don't update non-existent level -1 */
        cprof_stk[cprof_stk_top].spent_deeper =
                add64(cprof_stk[cprof_stk_top].spent_deeper, spent);
  cprof_locked = 0;
}


PRIVATE void cprof_init() {
  message m;
  int i;

  cpath[0] = '\0';
  cpath_len = 0;
  cprof_stk_top = -1;
  control.reset = 0;
  control.err = 0;
  cprof_tbl_size = profile_get_tbl_size();
  cprof_announce = profile_get_announce();
  clear_tbl();

  for (i=0; i<CPROF_STACK_SIZE; i++) {
        cprof_stk[i].cpath_len = 0;
        cprof_stk[i].slot = 0;
        cprof_stk[i].start_1._[U64_LO] = 0;
        cprof_stk[i].start_1._[U64_HI] = 0;
        cprof_stk[i].start_2._[U64_LO] = 0;
        cprof_stk[i].start_2._[U64_HI] = 0;
        cprof_stk[i].spent_deeper._[U64_LO] = 0;
        cprof_stk[i].spent_deeper._[U64_HI] = 0;
  }
}


PRIVATE void reset()
{
  clear_tbl();
  control.reset = 0;
}


PRIVATE void clear_tbl()
{
  int i;

  /* Reset profiling table. */
  control.slots_used = 0;
  for (i=0; i<CPROF_INDEX_SIZE; i++) idx[i] = 0;     /* clear index */
  for (i=0; i<cprof_tbl_size; i++) {                 /* clear table */
        memset(cprof_tbl[i].cpath, '\0', CPROF_CPATH_MAX_LEN);
        cprof_tbl[i].next = 0;
        cprof_tbl[i].calls = 0;
        cprof_tbl[i].cycles._[U64_LO] = 0;
        cprof_tbl[i].cycles._[U64_HI] = 0;
```

```
  }
}

#endif /* CPROFILE */
```

APPENDIX C – RESULTS

STATISTICAL PROFILING

Example output from *sprofalize.pl* on a data file created during *make clean install* in */usr/src/commands/*. After the distribution of ticks the aggregation of highest scoring functions is shown followed by the individual results of system task, FS and PM.

```
Building indexes from symbol tables: kernel ds fs inet is pm rs service at_wini
bios_wini cmos dp8390 dpeth floppy fxp lance log memory pci printer random rescu
e rtl8139 sb16_dsp sb16_mixer ti1225 tty.

Showing processes and functions using at least 0.8% time.

================================================================================
Data file: profile.stat.make_clean_commands.out
================================================================================

  System process ticks:      94358 ( 23%)
    User process ticks:     135400 ( 33%)          Details of system process
       Idle time ticks:     181570 ( 44%)          samples, aggregated and
                         ----------  ----          per process, are below.
            Total ticks:     411328 (100%)

--------------------------------------------------------------------------------
Total system process time                                          94358 samples
--------------------------------------------------------------------------------
  system _phys_co ******************************************************  23.8%
  system fill_sta **********************************                    15.0%
  system __receiv ******************                                     8.1%
      fs _get_ino **************                                         5.8%
      fs __receiv ************                                           4.8%
      fs __sendre ***********                                           4.6%
      fs   __send ***********                                           4.5%
      fs _search_ *******                                               2.8%
      pm __sendre *****                                                 2.0%
      pm    _main *****                                                 1.9%
  system _lock_se *****                                                 1.8%
      fs _get_blo ****                                                  1.4%
  system _isokend ****                                                  1.4%
      pm __receiv ***                                                   1.2%
      pm   __send ***                                                   1.0%
           <0.8% *********************************************          19.9%
--------------------------------------------------------------------------------
total                                                                 100.0%

--------------------------------------------------------------------------------
system                                          54.2% of system process samples
--------------------------------------------------------------------------------
_phys_co ********************************************************************  43.9%
fill_sta ****************************************                          27.7%
__receiv **********************                                           15.0%
_lock_se *****                                                             3.3%
_isokend ****                                                              2.7%
_clear_e **                                                                1.3%
_sys_tas **                                                                0.8%
   <0.8% ********                                                          5.3%
--------------------------------------------------------------------------------
system                                                                100.0%

--------------------------------------------------------------------------------
fs                                              34.2% of system process samples
```

```
-------------------------------------------------------------------------------
_get_ino ************************************************************  16.9%
__receiv ******************************************************       14.0%
__sendre ******************************************************       13.5%
   __send ****************************************************       13.3%
_search_ ********************************                              8.2%
_get_blo ****************                                              4.2%
 compare *********                                                     2.3%
_read_wr *********                                                     2.1%
_new_ico *******                                                       1.6%
_get_wor ******                                                        1.5%
  slword ******                                                        1.4%
_unsuspe *****                                                         1.1%
_read_ma ****                                                          1.0%
_parse_p ****                                                          0.9%
_get_sup ****                                                          0.9%
_select_ ****                                                          0.9%
   _main ****                                                          0.8%
_advance ****                                                          0.8%
__taskca ****                                                          0.8%
   <0.8% ****************************************************       13.8%
-------------------------------------------------------------------------------
fs                                                                   100.0%

-------------------------------------------------------------------------------
pm                                              9.3% of system process samples
-------------------------------------------------------------------------------
__sendre ************************************************************  22.1%
   _main ********************************************************       20.2%
__receiv ************************************                         12.9%
   __send ******************************                              10.3%
_get_fre ******************                                            6.7%
_find_sh ******************                                            6.2%
_pm_isok *******                                                       2.3%
_do_wait *****                                                         1.4%
_do_exec ****                                                          1.3%
__loadna ****                                                          1.2%
_pm_exit ****                                                          1.1%
 _adjust ***                                                           1.0%
_swap_in ***                                                           1.0%
_do_fork ***                                                           1.0%
__taskca ***                                                           1.0%
_get_wor ***                                                           1.0%
 _do_brk ***                                                           0.8%
   <0.8% ************************                                    8.5%
-------------------------------------------------------------------------------
pm                                                                   100.0%

-------------------------------------------------------------------------------
processes <0.8% (not showing functions)           2.3% of system process samples
-------------------------------------------------------------------------------
total                                          100.0%
```

CALL PROFILING

This notice is printed when cprofalyze.pl is run:

```
Notes:
- Calls attributed to a path are calls done on that call level.
    For instance: a() is called once and calls b() twice. Call path "a" is
    attributed 1 call, call path "a b" is attributed 2 calls.
- Time spent blocking is included.
- Time attributed to a path is time spent on that call level.
    For instance: a() spends 10 cycles in its own body and calls b() which
    spends 5 cycles in its body. Call path "a" is attributed 10 cycles,
    call path "a b" is attributed 5 cycles.
- Time is attributed when a function exits. Functions calls that have not
  returned yet are therefore not measured. This is most notable in main
  functions that are printed as having zero cycles.
- When "profile reset" was run, the actual resetting in a process happens
  when a function is entered. In some processes (for example, blocking
  ones) this may not happen immediately, or at all.
- Time is in milliseconds.
```

The following listings are the results of profiling runs during *make clean install* in */usr/src/commands/*, showing totals per function, sorted by time spent. The differences in absolute times between the runs are explained by the fact that for each run different sets of files were compiled with –Rcem-p, which affects performance.

All Kernel-Space Processes

These are the results of compiling everything in *kernel/* with the –Rcem-p option (measuring *KERNEL*, *CLOCK* and *SYSTEM*).

```
-------------------------------------------------------------------------------
kernel                                                          75 functions
-------------------------------------------------------------------------------
     calls         msecs  function
-------------------------------------------------------------------------------
    523333       2879.00  virtual_copy
      5519       1728.94  do_memset
   4425770       1614.96  sys_call
   3664081        840.24  mini_send
   3651605        787.03  enqueue
   3714055        716.77  mini_receive
   3657116        503.28  dequeue
   8830260        486.25  isokendpt_f
   7301952        420.09  pick_proc
   4423274        290.99  deadlock
    521428        202.22  do_copy
   1120596        194.75  lock_send
   3651605        144.33  sched
   1423496         92.32  umap_local
     79279         80.87  do_getinfo
    241035         68.72  do_umap
      4903         68.59  clear_endpoint
     82118         35.60  do_devio
     55283         34.52  do_newmap
      3417         33.90  do_vdevio
     55295         23.02  alloc_segments
     43906         18.72  mini_notify
     84725         18.53  do_times
     41410         16.57  lock_notify
    110703         12.87  sdesc
```

```
 23464         11.10  set_timer
 55296         10.59  init_codeseg
 23294         10.25  do_setalarm
174259         10.21  get_uptime
 16861          9.41  clock_handler
  4927          9.28  do_fork
  2179          9.21  do_irqctl
 16773          8.02  do_clocktick
 55407          7.76  init_dataseg
 22191          6.58  intr_handle
  5519          6.39  do_exec
 16861          5.93  load_update
  4903          4.53  clear_proc
  5330          3.60  generic_handler
 19223          2.59  cause_alarm
  7343          2.31  lock_enqueue
     2          1.89  do_cprofile
  4903          1.83  reset_timer
```

FS

An example showing the 35 most often occurring call paths in FS, sorted by number of calls.

```
--------------------------------------------------------------------------------
fs                                                                 1587 call paths
--------------------------------------------------------------------------------
     calls         msecs  path
--------------------------------------------------------------------------------
   1015798         40.65  main do_open common_open eat_path parse_path advance get_inode rw_inode new_icopy conv4
    778652         32.39  main do_stat eat_path parse_path advance get_inode rw_inode new_icopy conv4
    569243     224364.67  main get_work
    558091       8902.77  main reply
    416052         16.91  main do_close put_inode rw_inode new_icopy conv4
    290228         12.47  main do_open common_open eat_path parse_path advance get_inode rw_inode new_icopy conv2
    262291         14.72  main do_read read_write rw_chunk rahead get_block rm_lru
    262291         56.64  main do_read read_write rw_chunk rahead get_block
    258173         16.96  main do_read read_write rw_chunk put_block
    258173         36.01  main do_read read_write rw_chunk rahead get_block_size
    258173         69.25  main do_read read_write rw_chunk rahead
    258173       9181.87  main do_read read_write rw_chunk
    258117         54.04  main do_read read_write rw_chunk read_map
    222472         10.64  main do_stat eat_path parse_path advance get_inode rw_inode new_icopy conv2
    196742          7.96  main do_access eat_path parse_path advance get_inode rw_inode new_icopy conv4
    176803          8.18  main do_read read_write get_filp
    176803       4926.87  main do_read read_write
    176803         29.29  main do_read
    147090          5.92  main do_exit free_proc do_close get_filp
    147090         21.10  main do_exit free_proc do_close
    137484          7.09  main do_stat eat_path parse_path advance search_dir put_block
    137484          6.86  main do_stat eat_path parse_path advance search_dir get_block rm_lru
    137484         36.22  main do_stat eat_path parse_path advance search_dir get_block
    137484         13.02  main do_stat eat_path parse_path advance search_dir read_map
    123977          6.21  main do_open common_open eat_path parse_path advance search_dir put_block
    123977          5.71  main do_open common_open eat_path parse_path advance search_dir get_block rm_lru
    123977         27.36  main do_open common_open eat_path parse_path advance search_dir get_block
    123977         11.53  main do_open common_open eat_path parse_path advance search_dir read_map
    121458        278.13  main do_stat eat_path parse_path advance get_inode
    121187        261.47  main do_open common_open eat_path parse_path advance get_inode
    118872          5.72  main do_close put_inode rw_inode new_icopy conv2
    111655          9.15  main do_stat eat_path parse_path put_inode
    111655         42.56  main do_stat eat_path parse_path advance
    111655         14.44  main do_stat eat_path parse_path get_name
    111653         10.68  main do_stat eat_path parse_path advance search_dir forbidden
```

APPENDIX D – APPROXIMATION

The following tables are supporting material for paragraph 5.2.

Removal of IPC Time

1: unaltered results of a profiling run
2: IPC functions set to 0%
3: percentages normalized

	1	**2**	**3**		
					System Task
_phys_co	43.9%	43.9%	51.6%		
fill_sta	27.7%	27.7%	32.6%		
__receiv	15.0%	**0.0%**	**0.0%**		
_lock_se	3.3%	3.3%	3.9%		
_isokend	2.7%	2.7%	3.2%		
_clear_e	1.3%	1.3%	1.5%		
_sys_tas	0.8%	0.8%	0.9%	*SYSTEM's* part of MINIX 3 time	54.2%
<0.8%	5.3%	5.3%	6.2%	IPC part of SYSTEM taken out	15.0%
				Of MINIX 3 time this is	8.1%
	100.0%	85.0%	100.0%	SYSTEM's part of MINIX 3 time after	**46.1%**

_get_ino	16.9%	16.9%	28.5%		**FS**
__receiv	14.0%	**0.0%**	**0.0%**		
__sendre	13.5%	**0.0%**	**0.0%**		
__send	13.3%	**0.0%**	**0.0%**		
search	8.2%	8.2%	13.9%		
_get_blo	4.2%	4.2%	7.1%		
compare	2.3%	2.3%	3.9%		
_read_wr	2.1%	2.1%	3.5%		
_new_ico	1.6%	1.6%	2.7%		
_get_wor	1.5%	1.5%	2.5%		
slword	1.4%	1.4%	2.4%		
_unsuspe	1.1%	1.1%	1.9%		
_read_ma	1.0%	1.0%	1.7%		
_parse_p	0.9%	0.9%	1.5%		
_get_sup	0.9%	0.9%	1.5%		
select	0.9%	0.9%	1.5%		
_main	0.8%	0.8%	1.4%		
_advance	0.8%	0.8%	1.4%		
__taskca	0.8%	0.8%	1.4%	FS' part of MINIX 3 time	34.2%
<0.8%	13.8%	13.8%	23.3%	IPC part of FS taken out	40.8%
				Of MINIX 3 time this is	14.0%
	100.0%	59.2%	100.0%	FS' part of MINIX 3 time after	**20.2%**

__sendre	22.1%	**0.0%**	**0.0%**		**PM**
_main	20.2%	20.2%	36.9%		
__receiv	12.9%	**0.0%**	**0.0%**		
__send	10.3%	**0.0%**	**0.0%**		
_get_fre	6.7%	6.7%	12.2%		
_find_sh	6.2%	6.2%	11.3%		
_pm_isok	2.3%	2.3%	4.2%		
_do_wait	1.4%	1.4%	2.6%		
_do_exec	1.3%	1.3%	2.4%		
__loadna	1.2%	1.2%	2.2%		
_pm_exit	1.1%	1.1%	2.0%		
_adjust	1.0%	1.0%	1.8%		
_swap_in	1.0%	1.0%	1.8%		
_do_fork	1.0%	1.0%	1.8%		
__taskca	1.0%	1.0%	1.8%		
_get_wor	1.0%	1.0%	1.8%		
_do_brk	0.8%	0.8%	1.5%	PM's part of MINIX 3 time	9.3%
<0.8%	8.5%	8.5%	15.5%	IPC part of PM taken out	45.3%
				Of MINIX 3 time this is	4.2%
	100.0%	54.7%	100.0%	PM's part of MINIX 3 time after	**5.1%**

Total IPC time taken out, as percentage of MINIX 3 time	**26.3%**

Introduction of Kernel Task Time

1: unaltered results of a profiling run
2: results after taking out IPC time due to interrupt delay
3: setting kernel task time to system task time
4: percentages normalized and rounded
5: percentages normalized

1	**2**	**3**	**4**	
54.2%	46.1%	46.1%	38%	System task
0.0%	0.0%	46.1%	38%	Kernel task
34.2%	20.2%	20.2%	17%	FS
9.3%	5.1%	5.1%	4%	PM
2.3%	2.3%	2.3%	2%	Other
100.0%	73.7%	119.8%	100.0%	

IPC out		**5**	
0.0%	8.1%	6.8%	System task
0.0%	14.0%	11.7%	FS
0.0%	4.2%	3.5%	PM
0.0%	26.3%	22.0%	