The Synthesis Kernel

Calton Pu, Henry Massalin and
John Ioannidis Columbia University

ABSTRACT: The Synthesis distributed operating system combines efficient kernel calls with a high-level, orthogonal interface. The key concept is the use of a code synthesizer in the kernel to generate specialized (thus short and fast) kernel routines for specific situations. We have three methods of synthesizing code: Factoring Invariants to bypass redundant computations; Collapsing Layers to eliminate unnecessary procedure calls and context switches; and Executable Data Structures to shorten data structure traversal time. Applying these methods, the kernel call synthesized to read */dev/mem* takes about 15 microseconds on a 68020 machine. A simple model of computation called a *synthetic machine* supports parallel and distributed processing. The interface to synthetic machine consists of six operations on four kinds of objects. This combination of a high-level interface with the code synthesizer avoids the traditional trade-off in operating systems between powerful interfaces and efficient implementations.

1. Introduction

A trade-off between powerful features and efficient implementation exists in many operating systems. Systems with high-level interfaces and powerful features, like Argus[10] and Eden,[2] require a lot of code for their implementation, and this added overhead makes the systems slow. Systems with simple kernel calls, like the V kernel,[8] Amoeba,[15] and Mach,[1] have little overhead and run fast. However, the application software then becomes more complex and slower because extra code is required to make up for the missing kernel functions. Our goal in developing the Synthesis distributed operating system is to escape from this trade-off. We want to provide a simple but high-level operating system interface to ease application development and at the same time offer fast execution.

To achieve our goal, we combine two concepts in Synthesis. The most important idea is the inclusion of a *code synthesizer* in the kernel. The code synthesizer provides efficiency through the generation of specialized code for frequently-executed kernel calls. For instance, when the programmer asks the operating system to open a file, special routines to read and write that specific file are returned. Through the generation of these frequently-executed system calls, Synthesis reduces operating system overhead. For example, typical Synthesis read routines average an execution path of 20 to 30 machine instructions. In contrast, the 4.3 BSD *read* call contains on the order of 500 lines of C code (Appendix B).

The second idea is an orthogonal interface called a *synthetic machine*. To a programmer, a synthetic machine presents a logical multi-processor with its own protected address space. Two reasons motivated this model of computation: to take advantage of general-purpose shared-memory multiprocessors, and to support the growing number of concurrent programs. The synthetic machine consists of three basic components: synthetic CPUs to run the program, synthetic memory to store the program

This research is partially supported by the New York State Center for Advanced Technology, Computer and Information Systems, NYSSTF CAT(87)-5.

and data, and synthetic I/O units to move data in to and out of the synthetic machine.

The synthetic machine interface and kernel code synthesizer are independent ideas that have a synergistic effect. Without the code synthesizer, even a sophisticated implementation of synthetic machines would be very inefficient. Each high-level kernel call would require a large amount of code with a long execution time. Instead, the kernel code synthesizer generates specialized routines to make kernel calls short. Without a high-level interface, more layers of software would be needed to provide adequate functionality. The synthetic machine supplies high-level system calls to reduce the number of layers and the associated overhead. In Synthesis, application programmers will enjoy a high-level system interface with the efficiency of a "lean and mean" kernel.

To test these ideas, we have designed and implemented a prototype system, with a simplified kernel code synthesizer that implements a subset of the synthetic machine interface. Encouraged by positive results of the prototype, which confirmed our expectations on its performance, we are now implementing the full version of Synthesis.

In Section 2, we describe how the code synthesizer generates and optimizes code in the kernel. In Section 3, we summarize the synthetic machine interface and illustrate the power of the interface with an emulation of the UNIX system using synthetic machines. We outline the current Synthesis prototype in Section 4, including some measurements to illustrate the efficiency gained with synthesized code. Section 5 compares Synthesis with related work, and Section 6 concludes with a summary of progress.

2. *Kernel Code Synthesizer*

Typical operating system kernel routines maintain the system state in data structures such as linked lists. To perform its function, a kernel routine finds out the system state by traversing the appropriate data structures and then takes the corresponding action. In current operating systems, there are few short cuts to reach frequently-visited system states, which may require lengthy data structure traversals.

The fundamental idea of kernel code synthesis is to capture frequently visited system states in small chunks of code. Instead of traversing the data structures, we branch to the synthesized code directly. In this section, we describe three methods to synthesize code: Factoring Invariants, Collapsing Layers, and Executable Data Structures.

2.1 Factoring Invariants

The *factoring invariants* method is based on the observation that a functional restriction is usually easier to calculate than the original function. Let us consider a general function:

$$F_{big}(p1, p2, \ldots, pn)$$

By factoring out the parameter *p1* through a process called currying,[14] we can obtain an equivalent composite function:

$$[F^{create}(p1)](p2, \ldots, pn) \equiv F_{big}(p1, p2, \ldots, pn)$$

F^{create} is a second-order function. Given the parameter *p1*, F^{create} returns another function, F_{small}, which is the restriction of F_{big} that has absorbed the constant argument *p1*:

$$F_{small}(p2, \ldots, pn) \subset F_{big}(p1, p2, \ldots, pn)$$

If F^{create} is independent of global data, then for a given *p1*, F^{create} will always compute the same F_{small} regardless of global state. This allows $F^{create}(p1)$ to be evaluated once and the resulting F_{small} used thereafter. If F_{small} is executed m times, generating and using it pays off when

$$Cost(F^{create}(p1)) + m*Cost(F_{small}(p2, \ldots, pn)) <$$

$$m*Cost(F_{big}(p1, \ldots, pn))$$

As the "factoring invariants" name suggests, this method resembles the constant folding optimization in compiler code generation. The analogy is strong, but the difference is also significant. Constant folding in code generation eliminates static code. In contrast, Factoring Invariants skips dynamic data structure traversals in addition to eliminating code.

As an example, we can use UNIX *open* as F^{create} and *read* as F_{small}, and the file name as the constant parameter *p1*. Constant global data are the process id, address of kernel buffers, and the device that the file resides on. F^{create} consists of many small procedure templates, each of which knows how to generate code for a basic operation such as "read disk block" or "process TTY input." The parameters passed to F^{create} determine which of these procedures are called and in what order. The final F_{small} is created by filling these templates with addresses of the process table and device registers.

2.2 Collapsing Layers

The *collapsing layers* method is based on the observation that in a layered design, separation between layers is a part of specification, not implementation. In other words, procedure calls and context switches between functional layers can be bypassed at execution time. Let us consider an example from the layered OSI model:

$$F_{big}(p1,p2,\ldots,pn) =$$

$$F_{applica}(p1,F_{present}(p2,F_{session}(\cdots F_{datalnk}(pn)\cdots)))$$

$F_{applica}$ is a function at the Application layer that calls successive lower layers to send a message. Through in-line code substitution of $F_{present}$ in $F_{applica}$, we can obtain an equivalent flat function by eliminating the procedure call from the Application to the Presentation layer:

$$F^{flat}_{applica}(p1,p2,F_{session}(\cdots)) \equiv$$

$$F_{applica}(p1,F_{present}(p2,F_{session}(\cdots)))$$

The process to eliminate the procedure call can be embedded into two second-order functions. $F^{create}_{present}$ returns code equivalent to $F_{present}$ and suitable for in-line insertion. $F^{create}_{applica}$ incorporates that code to generate $F^{flat}_{applica}$.

$$F^{create}_{applica}(p1,F^{create}_{present}(p2,\cdots),\ F^{flat}_{applica}(p1,p2,\cdots))$$

This technique is analogous to in-line code substitution for procedure calls in compiler code generation. In addition to the elimination of procedure calls and maybe context switches, the

resulting code typically exhibits opportunities for further optimization, such as Factoring Invariants and elimination of data copying.

By induction, $F^{create}_{present}$ can eliminate the procedure call to the Session layer, and down through all layers. When we execute $F^{create}_{applica}$ to establish a virtual circuit, the $F^{flat}_{applica}$ code used thereafter to send and receive messages may consist of only sequential code. The performance gain analysis is similar to the one in section 2.1.

2.3 Executable Data Structures

The *executable data structures* method is based on the observation that some data structures are traversed in some preferred order. Therefore, adding the executable code to the data structures to make them *self-traversing* may decrease the traversal overhead.

Let us consider the simplified example of the active job queue managed by a round-robin scheduler. Each element in the queue contains two short sequences of code: *stopjob* and *startjob*. The *stopjob* saves the registers and branches into the next job's *startjob* routine (in the next element in queue). The *startjob* restores the registers, installs the address of its own *stopjob* in the timer interrupt vector table, and resumes processing.

An interrupt causing a context switch will trigger the current program's *stopjob*, which saves the current state and branches directly into the next job's *startjob*. Note that the scheduler has been taken out of the loop. It is the queue itself that does the context switch, with a critical path on the order of ten machine instructions. The scheduler intervenes only to insert and delete elements from the queue.

2.4 Kernel Programmability

In the previous sections, we have described several innovative programming techniques to gain performance through code synthesis. Historically, the lambda functions in LISP have created S-expressions, which are executable in the LISP environment. However, new problems arose in our use of code synthesis on bare machine. Here, we summarize briefly the problems we have encountered and the approaches we have taken to solve them:

- Inflated kernel size due to code redundancy.
- Structuring of kernel and correctness of its algorithms.
- Protection of synthesized code.

One important concern in Synthesis is kernel size inflation due to the potential redundancy in the many F_{small} and F^{flat} programs generated by the same F^{create}. To solve this problem, F^{create} generates either in-line code or subroutine calls using a mechanism similar to threaded code.[4] Frequently invoked functions are expanded in-line into the user's synthetic machine and executed there. Rarely executed functions are stored in a common area, shared by all synthetic machines running threaded code. The decision of when to expand in-line is made by the programmer writing F^{create}.

Although the size of the code synthesizer depends heavily on the kinds of facilities supported by the kernel, we hope to minimize the cost paid for code specialization. For example, the Synthesis $F^{create}_{fileaccess}$ calls correspond to the UNIX *open* system calls, with similar data structures and algorithms. The main difference between them resides in the actions taken by the system calls; *open* fills in the UNIX data structures, and $F^{create}_{fileaccess}$ places machine op-codes into an array. Therefore, we expect the cost of $F^{create}_{fileaccess}$ to be comparable to that of the UNIX *open*.

The structure of Synthesis kernel is superficially similar to a traditional operating system. Kernel calls of the F^{create} type are invoked just like a normal system call. However, synthesized kernel calls created by F^{create} are invoked in the kernel mode (usually through *trap*) through a branch into the synthesized code. By construction, the synthesized kernel calls perform a subset of the actions of normal kernel calls. These subsets calculate the same results and cause the same side effects for each specific case.

Synthesized code is protected through memory management. Each address space has its own page table, and synthesized code is placed in protected pages, inaccessible to the user program. To prevent the user program from tricking the kernel into executing code outside the protected pages, the synthesized routines are accessed via a jump table in the protected area of the address space. Since the user program can only specify an index into this table, the synthesized routines are entered at the proper entry

points. This protection mechanism is similar to C-lists to prevent the forgery of capabilities.[16]

Synthesized routines run in supervisor state. The transition from user to supervisor state is made via a *trap* instruction. Thus, synthesized code can perform privileged operations such as accessing protected buffer pages. Just before returning control to the caller, the synthesized code reverts to the previous mode.

3. Synthetic Machines

3.1 Model of Computation

The synthetic machine is the unit of protection. Data in a synthetic machine are freely shared within it, but are carefully protected from access by other synthetic machines. Each synthetic machine runs one program, and has three kinds of components:

- synthetic CPUs (SCPU) to run the program,
- synthetic memory (SMEM) to store the program and data,
- synthetic I/O units (SIO) to move data into and out from the synthetic machine.

Each synthetic machine may have any number of SCPUs, SMEMs, and SIOs. Each SCPU is a thread of control scheduled on a physical CPU. Each SMEM is a segment of memory, accessible from all SCPUs of the same synthetic machine.

Examples of SMEM include program segments and shared memory between synthetic machines. Each SIO provides input or output to the synthetic machine. Examples of SIO are ordinary files, devices, and network communications.

An interesting example of a program running in a synthetic machine is a multiplexor supporting other synthetic machines to form a hierarchical structure similar to VM/370.[9] Child synthetic machines are scheduled as any other SCPU, but they may "sub-schedule" their own SCPUs with different scheduling algorithms. Similarly, their SMEMs are allocated by the parent synthetic machine. The SIO system calls for the child synthetic machine are synthesized from the parent's SIO system calls. Careful

application of the Collapsing Layers method decreases the cost of indirection and hierarchy in Synthesis.

The Synthesis kernel implements the root synthetic machine running on the real machine. The scheduler, memory management, and file system are part of the multiplexor program running in the root synthetic machine. Actual user programs run within child synthetic machines. The conceptual nesting of synthetic machines does not introduce run-time overhead because of code synthesis, in particular Collapsing Layers.

Many existing operating systems have entities corresponding to a synthetic machine. Some examples are the virtual machine in VM/370 and the UNIX process, which are similar to a synthetic machine with only one SCPU, or Mach tasks[1] and Distributed V teams,[7] which are similar to multiple synthetic CPUs. Although these systems share the same von Neumann model of computation, their kernel interfaces are less orthogonal than the synthetic machine interface that now we describe.

3.2 Synthetic Machine Interface

For each synthetic machine, the *create* kernel call is generative, synthesizing code for the executive kernel calls: *terminate*, *reconfigure*, and *query*. To destroy a synthetic machine, we call *terminate*. During a synthetic machine's lifetime, *reconfigure* changes its state, and *query* reads its state. The synthetic machine kernel calls are summarized in the second column of Table 1 (with the Synthetic Machine heading).

Analogous to synthetic machines, each component must be created before it can be used. The generative *create* kernel call returns the code synthesized for the executive kernel calls, which include *read* and *write* in addition to *terminate*, *reconfigure*, and *query*. The *read* kernel call moves data into a synthetic machine, while *write* moves data out from it. Table 1 contains a partial summary of the interface to synthetic components. During execution, if new situations require new code, the synthetic components may be resynthesized using the *create* kernel call with the REUSE option and a modified set of parameters.

	Synthetic Machine	Synthetic CPU	Synthetic Memory	Synthetic I/O
create (generative)	creates an SMACH	creates a thread of control	allocates memory	creates ports, opens files, allocates devices
terminate (executive)	kills an SMACH and all its components	kills a thread of control	frees memory	kills ports, closes files
reconfigure (executive)	resumes/suspen. an SMACH, changes its priority	resumes/suspen. SCPU, changes its priority, wait on event	changes protection, initiates sharing	lseek, changes protection
query (executive)	gets priority, gets SMACH id, gets uid	gets priority, gets state, gets SCPU id	gets size, starting address	gets device type, state, device pointer
read (executive)	unused	unused	unused	reads file, rec. messages, and any other input operation
write (executive)	unused	unused	unused	writes file, sends messages, and any other output operation

Table 1: Examples of Synthetic Machine Kernel Calls

To illustrate the use of these calls, we give an *SIO_create* example, which opens a file. Given the file name and a read/ write option, it returns the code for the executive calls. You call *read* to read from the file and *write* to write to it.

To find the file length, current seek position and other file attributes, you use *query*. The *reconfigure* call changes those attributes. Finally, *terminate* closes the file.

The synthetic machine interface is object-oriented. Synthetic machines and synthetic components are encapsulated resources, and users must ask them to perform the kernel calls defined in Table 1. Pioneer object-oriented systems such as Hydra,[16] Eden,[2] and Argus[10] have achieved performance adequate for a prototype, while Mach is comparable to BSD UNIX.[1] We believe that synthesized code will take the performance of Synthesis one step ahead of current systems. In Section 4.2, we justify our expectations with preliminary measurements.

3.3 Resource Sharing and Protection

To support the shared-memory model of parallel processing, all SCPUs within a synthetic machine share the same address space and thus, SMEMs can be used for communication and sharing between them. Semaphores control the synchronization between SCPUs. Currently, we are developing a high-level language to support concurrent programming with synthetic machines.

Efficient sharing between synthetic machines requires more care, since we enforce protection across their boundaries. SMEM may be shared between two cooperating synthetic machines. The original creator of the SMEM sends the appropriate routines (synthesized for sharing) to its peer synthetic machine through a protected SIO channel. The peer then uses the routines to map the SMEM into its own synthetic machine. The access routines given to the peer determine the access privileges.

To support the message-passing model of parallel and distributed processing, SIOs include network traffic. Sharing SIOs is similar to sharing SMEM, in that the creator of the SIO, e.g. a port for inter-process communication, also passes the access routines to the peer. Since the access routines are passed through the protected SIO channel, no forging is possible. This protection mechanism is more flexible than that achieved by capabilities with a constant number of bits, since these routines can implement any kind of control, for example, access control lists. Furthermore, the creator may reconfigure or resynthesize the SMEM or SIO in such a way as to invalidate the earlier access routines, thus revoking access rights already conceded.

4. Work in Progress

4.1 Target Hardware

For single-CPU systems, Synthesis is intended to run on a von Neumann-style CPU with memory management unit and large physical memory. The Synthesis prototype runs on an experimental machine based on a 68020 processor at 20 MHz with a 16-bit-wide bus.[3] For debugging and measurements, the prototype hardware provides single-step and real-time trace facilities. In addition, a ROM-based monitor contains an assembler, a disassembler, and a process manager, with a C-style interface.[11] Other commercially available machines are SUN-3, Macintosh II, and similar products. With multiple SCPUs in the same address space and SIOs to send/receive messages, Synthesis supports parallel machines of both shared-memory model and message-passing model.

For efficiency and lack of code synthesis support in high-level languages, we are using 68020 assembly language to write the first full version of Synthesis kernel. Our own assembler supports recursive calls to itself for translating static templates of code to be synthesized. Portability of application software is very important due to its volume and decentralized development. However, we believe that for the small kernel code efficiency is of paramount importance and no concessions should be made. We recognize that writing a Synthesis kernel for a different processor, say DEC's VAX family, may be a non-trivial experience. But an operating system should be defined by its interface and model of computation, not implementation. Until there is an optimizing compiler for a high-level language supporting code synthesis (LISP has too high run-time overhead), we plan to write a different set of programs to implement Synthesis for each type of hardware. Each implementation will emphasize performance, use the particular features of its own hardware, and maintain rigorous compatibility with the synthetic machine interface.

4.2 First Version of Software

The first version of Synthesis kernel is being written incrementally on top of a small kernel.[11] At the moment, the Factoring Invariants method has been used in all input and output devices (SIO), including terminal I/O and a RAM-based file system. In the round-robin scheduler, the Executable Data Structures method provides fast context switch between synthetic machines and synthetic CPUs.

We are designing the message passing kernel primitives to support distributed processing. At the core of the high-level message passing support is the optimization based on the Collapsing Layers method.

The first program we measured reads one character from the memory special-device file (equivalent to UNIX */dev/mem*). Since the only significant part of the program is the system call to read a byte, this program shows the promise of efficiency gained with synthesized code. A single example does not "prove" our approach, but it shows how far code synthesis can go.

In C, the program is:

```
#include <sio/SIO.h>
struct SIO_if *myfile, *SIO_create();
char buf[4];
int i;
    myfile = SIO_create(FILEACCESS, "/dev/mem", FA_RDONLY);
    for(i=100000; i--; )
        read(myfile, buf, 1);
    SIO_terminate(myfile);
```

A trace of the generated code running on the prototype is included in Appendix A.

The second program is similar to the first one.

```
char y[1000];
int i;
for(i=10000; i--; )
    read(myfile, y, 1000);
```

In this example, we have a more common situation where the overhead of a system call is amortized over a larger amount of useful work.

The numbers actually measured on the prototype system appear in the "Measured" column of Table 2. For comparison purposes, we translated the numbers measured from the prototype system into corresponding ones for a 68020-based machine running at 16 MHz, with a 32-bit data bus. This configuration is similar to SUN-3 and HP 9000/320 workstations. The translation was obtained by hand-counting the CPU cycles for such machines, and the results appear in the column titled "Corrected." The same programs were run on the HP 9000/320 workstation with the HP-UX 5.17, SUN-3/50 with the SUN-OS 3.2, and Masscomp MC5600 model 56S-02, with the RTU UNIX 3.1 operating system. The results appear in the same table, columns HP, SUN-3, and Masscomp.

program	Measured	Corrected	HP	SUN-3	Masscomp
Prog. 1	1.6 sec	1.2 sec	77 sec	48 sec	29 sec
Prog. 2	2.0 sec	2.0 sec	15 sec	4.9 sec	4.5 sec

Table 2: Measured Figures and Comparison

4.3 Prototype Experience

Of the three synthetic components, we have found SIO to benefit the most from synthesized code. Most of the improvement comes from the elimination of code that check parameters and states, since these remain the same from call to call. A concrete example, the *read* system call in 4.3 BSD, is included in Appendix B.

The current prototype includes a UNIX-like hierarchical file system. In addition, several file systems will co-exist in Synthesis. Some file systems will consist of file servers running in synthetic machines, with well-known SIO ports for local and remote file service. Others, like the current hierarchical file system, may be incorporated into the kernel, accessible through a special type of

SIO. Even the file servers return synthesized code to speed up file access.

A music synthesizer program has been written to run as an application on the prototype system. The application consists of six SCPUs in three stages. First, a note sequencer running in one SCPU feeds a four-voice sampling synthesizer, one voice per SCPU. The voices from the second stage go to a summing program in a sixth SCPU, which sends its output to the digital-to-analog converter port. The music synthesizer runs in real-time with a 25 kHz sampling rate and has demonstrated the speed of our I/O operations and context switches.

We are developing a variant of the C language, called Lambda-C, to support code synthesis in a high-level language. Lambda-C will serve several purposes. First, we plan to build a portable version of Synthesis, written in this language. Second, we believe that code-generating programs make for efficient applications, not just operating systems. A high-level language such as Lambda-C will make these methods available to application programmers. Third, type-checking of synthesized code is non-trivial, and we need languages to support it.

5. *Comparison with Other Systems*

The main difference between Synthesis and other operating systems is in the combination of the synthetic machine interface and the kernel code synthesizer. No other operating system offers a high-level interface and the potential to generate efficient code.

UNIX[13] has evolved into a large system with Fourth Berkeley Distribution[12] and AT&T System V. Although the interface remains approximately the same in the many different variants of UNIX, the synthetic machine interface is more orthogonal. To the best of our knowledge, no UNIX system uses a kernel code synthesizer.

The V kernel[7] and Amoeba[15] are two examples of small distributed operating system kernels. They both encourage layers of software to be written on top of the kernel. Synthesis differs from both by the high-level synthetic machine interface, and the code synthesizer.

Mach[1] offers an object-oriented interface that is isomorphic to a specialized synthetic machine. A Mach task corresponds to a synthetic machine; Mach thread, an SCPU; Mach port, the network SIO; Mach messages, network SIO read and write; and Mach virtual memory, an SMem. The synthetic machine uses the same interface for all I/O activities, and child synthetic machines may be nested within a parent. As with other systems, Mach does not use a kernel code synthesizer.

Emerald[5,6] is an object-oriented, integrated language and system. Synthesis lacks the language support in Emerald, in particular the sophisticated typing system. In compensation, although Emerald objects may be constructed at run-time (in a way similar to synthesized code), its kernel calls are not synthesized.

6. *Conclusion*

We have combined two ideas in Synthesis. First, a kernel code synthesizer produces specialized and extremely efficient code for system calls. Second, an orthogonal, object-oriented, and high-level interface has been derived from a simple model of computation. This combination gives Synthesis unique advantages. The kernel code synthesizer reduces the high-level interface inefficiency problem. The high-level interface removes the slowdown due to multiple layers of software built on small kernels.

Efficiency derived from the code synthesizer has been demonstrated on a prototype system. An example is a specialized *read* system call, which takes about fifteen microseconds. In comparison, the HP-UX and Masscomp RTU systems running on similar hardware need a few hundred microseconds for an equivalent, non-specialized read call. We expect to do even better in our full system, with very efficient SIO kernel calls including file systems, network communications, and other devices.

We are implementing the full version of Synthesis for SUN-3 workstations and a 68020-based machine. After the kernel, we will design and implement language support, transaction processing, and real-time support, all taking advantage of synthesized code.

We believe that the unique combination of simplicity and efficiency makes Synthesis an excellent system to write and execute programs.

Acknowledgments

We would like to thank Perry Metzger, Mike Schwartz, and Eric Jul for their comments that improved the presentation of the paper. Gail Kaiser, Chip Maguire, Rajendra Raj, and Jonathan Smith helped with a previous version of the paper. Equally important, we would like to thank Ed Hee and Paul Kanevsky for their varied contributions to the project, and Al Lash for assistance in procuring laboratory facilities. AMD, Hitachi, Intel, Motorola, and Mupac contributed with hardware parts for the project.

Appendix A: Trace of Generated Code

```
        003F8000  7061              moveq   #00000061, d0
        003F8002  223C 000F 4240    move.l  #000F4240, d2
27----  003F8008  4E41              trap    #1
20 ↑    00000ACE  48E7 40E0    [1] movem.l <a2, a1, a0, d1>, -(sp)
 6 |    00000AD2  2078 0000    [2] move.l  SYSVARS, a0
 4 |    00000AD6  E989          [3] lsl.l   #4, d1
 6 |    00000AD8  D0FC FF3C     [4] lea     (PROCTABLE,a0,d1), a0
 7 |    00000ADC  2468 0004     [5] move.l  (FN_READ,a0), a2
 5 |    00000AE0  43E8 000C     [6] lea     (SEEKPOINTER,a0), a1
 7sys   00000AE4  2068 0008     [7] move.l  (MEMBASE,a0), a0
13call  00000AE8  4E92          [8] jsr     (a2)
 7 |    000010A0  D1D1          [9] add.l   (a1), a0
 6 |    000010A2  1010         [10] move.b  (a0), d0
 9 |    000010A4  5291         [11] addq.l  #1, (a1)
 7 |    000010A8  44FC 0000    [12] move.w  #0, cc
10 |    00000AEA  4E75         [13] rts
28 |    00000AEC  4CDF 0702    [14] movem.l (sp)+, <d1, a0, a1, a2>
 5 ↓    00000AF0  40D7         [15] move.w  sr, (a7)
21----  00000AF2  4E73         [16] rte
 4      003F800A  5381              subq.l  #1, d2
 9      003F800C  66FA              bne     003F8008
        003F8008  4E41              trap    #1
                  ⋮
```

Figure 1: Trace of Code Actually Measured

In Figure 1, we show the trace produced by the execution of the code synthesized for program 1 (Section 4.2). Instruction `[1]` saves the registers that will be used. Instruction `[2]` gets the address of the kernel data space; instructions `[3]` and `[4]` adds an offset to point into the SCPU's file table. The file table contains the address of the specialized read routine and its static data. The specialized read routine address is placed in register a2 `[5]`, and the two pieces of static data is extracted: the file seek position `[6]`, and the base memory address `[7]`. The specialized routine is called `[8]`, and the seek position is added to the base memory address `[9]`. The byte is read `[10]`, and the seek position is updated `[11]`. The "read OK" status code is signalled `[12]` and the function returns `[13]`. Finally the registers are restored `[14]`, the status saved `[15]`, and the system call exits `[16]`.

```
clocks  addr      op-codes         instruction
------  --------  ---------------  ----------------------------------
27----  003F8008  4E41               trap    1
28 ↑    000008D0  4EB0 1DA6 0000     jsr     ([SYSVARS],PROCTBL,d1*4)
   |              FF3C
16 |    003F2EDC  1030 05F1 003F     move.b  ([SEEKPOINTER]), d0
   |              2ED8
10 |    003F2EE4  52B9 003F 2ED8     addq.l  #1, SEEKPOINTER
 7 |    003F2EEA  44FC 0000          move.w  #0, cc
10 |    003FF2E8  4E75               rts
 5 ↓    000008D8  40D7               move.w   sr, (sp)
21----  000008DA  4E73               rte
```

Figure 2: Trace of Expected Code

We describe in Figure 2 the optimized code synthesis we expect to produce in the full version. We will now explain what is happening in these two versions of the read call, and this will help illustrate the effect that the full code optimizer will have once it is implemented. There are several places where the long code is less than optimal. The seek position and the base address should be kept with the specialized read function rather than in the file table. Doing so will eliminate the need for instructions `[6]` and `[7]`, as well as the save and restore of registers a0 and a1 When the full optimizer is implemented, instructions `[2]`, `[3]`, `[4]` and `[8]` can be collapsed into one of the 68020 memory indirect addressing modes. Finally, there is no need to keep both a base address and a seek pointer. A trivial modification of the seek routine (not shown here) allows us to use just one pointer. Figure 2 is the result of applying all these optimizations. The execution trace of a "vanilla" UNIX *read* is too long to be included here, but we summarize its actions in Appendix B.

Appendix B: UNIX Comparison

For comparison purposes, we have analyzed 4.3 BSD since its source was available. First we introduce some terms. A *file descriptor* is an indirect index into an entry in the system open file table. Each such "open file structure" includes the address of the first *inode* (which contain the actual data stored in the file) of that file and pointers to the set of functions used to manipulate this file type, which are either the *inode* or the *socket* operations.

Now, let us examine what happens when a file is *open*ed. First, *copen()* checks permissions and allocates an open file structure. Then *namei* interprets the path name taking care of special cases like symbolic links and mounted file systems. If the file exists, its first *inode* is returned. Otherwise, *maknode* is called to create the actual file. Then, the open file structure is filled out and the file descriptor leading to it is returned. The length of the source code for all this is of the order of 1000 lines of C.

Now, let us see how we can read from that open file. *read* massages its arguments and calls *rwuio*. The "open file structure" is examined to check for permissions, and validity checks are performed for the buffers. Then *ino_rw* (the generic function to read or write a file), is called indirectly. It does consistency checking, then calls another function (*rwip*). In the case of a regular or a block-device file, special processing is done to take advantage of the block I/O interface, then the block read (*bread*) function eventually invokes the device driver to do the actual operation. For character-device files, the device driver is invoked directly. Finally, the system call returns the number of characters read.

Just going through the code and not counting all the secondary procedures called, we counted more than 500 lines of C code. A great deal of that code consists of multiple validity tests, case statements etc. This is because a large variety of cases has to be handled by the same piece of code.

References

1. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, Mach: a new kernel foundation for UNIX development, *Proceedings of the 1986 USENIX Conference*, pages 93-112, USENIX Association, 1986.
2. G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, The Eden system: a technical review, *IEEE Transactions on Software Engineering*, SE-11(1):43-58, January 1985.
3. James Arleth, *A 68010 Multiuser Development System*, Master's thesis, The Cooper Union for the Advancement of Science and Art, New York City, 1984.
4. J. R. Bell, Threaded code, *Communications of ACM*, 16(6):370-372, June 1973.
5. A. Black, N. Hutchinson, E. Jul, and H. Levy, Object structure in the Emerald system, *Proceedings of the First Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 78-86, ACM, September 1986.
6. A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, Distribution and abstract types in Emerald, *IEEE Transactions on Software Engineering*, SE-12(12):65-76, January 1987.
7. D. Cheriton, The V kernel: a software base for distributed systems, *IEEE Software*, 1(2):19-43, April 1984.
8. D. Cheriton and W. Zwaenepoel, The Distributed V kernel and its performance for diskless workstations, *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 129-140, ACM/SIGOPS, October 1983.
9. H. M. Deitel, *An Introduction to Operating Systems*, Addison-Wesley Publishing Company, revised first edition, 1984.
10. B. H. Liskov and R. W. Scheifler, Guardians and Actions: linguistic support for robust, distributed programs, *Proceedings of the Ninth Annual Symposium on Principles of Programming Languages*, pages 7-19, January 1982.
11. H. Massalin, *A 68010 Multitasking Development System*, Master's thesis, The Cooper Union for the Advancement of Science and Art, New York City, 1984.
12. J. S. Quarterman, A. Silberschatz, and J. L. Peterson, 4.2BSD and 4.3BSD as examples of the UNIX system, *ACM Computing Surveys*, 17(4):379-418, December 1985.

13. D. M. Ritchie and K. Thompson, The UNIX Time-sharing System, *Communications of ACM*, 7(7):365-375, July 1974.

14. J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, 1977.

15. A. S. Tanenbaum and S. J. Mullender, *The Design of a Capability-Based Distributed Operating System*, Technical Report IR-88, Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam, November 1984.

16. W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, Hydra: the kernel of a multiprocessing operating system, *Communications of ACM*, 17(6):337-345, June 1974.

[*submitted Sept. 22, 1987; accepted Nov. 5, 1987*]