Threads and Input/Output in the Synthesis Kernel

Henry Massalin Calton Pu

Department of Computer Science
Columbia University,
New York, NY 10027

calton@cs.columbia.edu

November 3, 1995

Abstract

The Synthesis operating system kernel combines several techniques to provide high performance, including kernel code synthesis, fine-grain scheduling, and optimistic synchronization. Kernel code synthesis reduces the execution path for frequently used kernel calls. Optimistic synchronization increases concurrency within the kernel. Their combination results in significant performance improvement over traditional operating system implementations. Using hardware and software emulating a SUN 3/160 running SUNOS, Synthesis achieves several times to several dozen times speedup for UNIX kernel calls and context switch times of 21 microseconds or faster.

1 Introduction

Synthesis is an operating system kernel for a parallel and distributed computational environment. We have three major goals in the design and implementation of Synthesis:

1. high performance,
2. self-tuning capability to dynamic load and configuration changes,
3. a simple, uniform and intuitive model of computation with a high-level interface.

In this paper, we focus on the aspects of the Synthesis kernel implementation that supports threads and input/output. To achieve very high performance, we combine *kernel code synthesis* [5], which decreases kernel call overhead through specialization, and *reduced synchronization*, which decreases kernel thread synchronization overhead.

We have introduced the principles of code synthesis [5], which makes the Synthesis kernel fast for several reasons. First, frequently executed Synthesis kernel calls are "compiled" and optimized at run-time using ideas similar to currying and constant folding. For example, when we open a file for input, a custom-made (thus short and fast) read routine is returned for later read calls. Second, frequently traversed data structures are sprinkled with a few machine instructions to make them self-traversing. For example, the CPU dispatching, including context-switches, is done by the ready queue this way (for details see Figure 3). In this paper, we describe the synergy from combining code synthesis with the other kernel implementation techniques. To make the paper self-contained, we summarize kernel code synthesis and other aspects of background information in Section 2.

In traditional OS's, the kernel call and dispatching/scheduling overhead overshadows the kernel synchronization cost. Therefore, we see traditional kernels using powerful mutual exclusion mechanisms such as semaphores. However, in Synthesis we have used kernel code synthesis to trim kernel calls and context switches. The next bottleneck turned out to be kernel internal synchronization cost, given that the Synthesis kernel is highly parallel. Our answer to this problem consists of methods that reduce synchronization in the Synthesis kernel, described in Section 3.

To illustrate the new possibilities for performance improvements introduced by these techniques, we will describe two kinds of objects supported by the Synthesis kernel, threads and I/O. Our discussions on threads in Section 4 and I/O in Section 5 are relevant to uniprocessor and multiprocessor systems. The distribution aspects of Synthesis are beyond the scope of this paper.

All the performance improvement techniques follow from one software engineering principle, called the *principle of frugality*, which says that we should use the least powerful solution to a given problem. Since we carefully separate the kernel implementation from the interface specification, the principle of frugality has been applied throughout the system. Both kernel code synthesis and reduced synchronization are good examples. In Section 6 we present measurement data to show the effectiveness of these techniques.

2 Synthesis Background

2.1 Synthesis Model of Computation

The Synthesis model of computation is conceptually a von Neumann machine with threads of execution, memory protection boundaries, and I/O devices. To support parallel and distributed computing, the threads of execution form a directed graph, in which the nodes are threads and the arcs are data flow channels. This graph model and other support for parallel and distributed computation will be described in more detail in another paper [4].

Synthesis threads are threads of execution, like UNIX processes. Some threads never execute user-level code, but run entirely within the kernel to provide additional concurrency for some kernel operations. Threads execute programs in a quaspace (*qua*si *a*ddress *space*), which also store data. Finally, I/O devices move data between threads, including files and messages.

On one physical node, all the Synthesis quaspaces are subspaces of one single address space, defined by the CPU architecture (e.g., with a 32-bit microprocessor we have a 32-bit address space). The kernel blanks out the part of the address space that each quaspace is not supposed to see. Since they are parts of the same address space, it is easy to share memory between quaspaces by setting their address mappings. The current implementation of the kernel does not support virtual memory.

2.2 Kernel Code Synthesis

The idea of kernel code synthesis has been introduced in a previous paper [5]. In Synthesis, we have a code synthesizer in the kernel to generate specialized (thus short and fast) kernel routines for specific situations.

We have three methods to synthesize code. The Factoring Invariants method bypasses redundant computations, much like constant folding. The Collapsing Layers method eliminates unnecessary procedure calls and context switches, both vertically for layered modules and horizontally for pipelined threads. The Executable Data Structures method shortens data structure traversal time when the data structure is always traversed the same way.

2.3 Basic Kernel Components

To describe the Synthesis kernel implementation in concrete terms, we first summarize its basic components. The Synthesis kernel can be divided into a number of collections of procedures and data. We call these collections of procedures *quajects* that encapsulate hardware resources, like Hydra objects [7]. For this paper the most important quajects are threads and I/O device servers. Threads are an abstraction of the CPU. The device servers are abstractions of I/O devices. Except for the threads, quajects consist only of procedures and data. Events such as interrupts start the threads that animate the quajects and do work. The quajects do not support inheritance or any other language features.

Most quajects are implemented by combining a small number of building blocks. Some of the building blocks are well known, such as monitors, queues, and schedulers. The others are simple but somewhat unusual: switches, pumps and gauges. As we shall see in Section 5, all of Synthesis I/O is implemented with these building blocks. The quaject interfacer (see below) uses optimization techniques such as Collapsing Layers to combine these building blocks into kernel quajects.

The unusual building blocks require some explanation. A switch is equivalent to the C switch statement. For example, switches direct interrupts to the appropriate service routines. A pump contains a thread that actively copies its input into its output. Pumps connect passive producers with passive consumers. A gauge counts events (e.g., procedure calls, data arrival, interrupts). Schedulers use gauges to collect data for scheduling decisions.

Each building block may have several implementations. Applying the principle of frugality, we use the most economical implementation depending on the usage. For example, there are several kinds of queues in the Synthesis kernel. Semantically, we have the usual two kinds of queues, the synchronous queue which blocks at queue full or queue empty, and the asynchronous queue which signals at those conditions. For each kind, we have two implementations: dedicated queues and optimistic queues. Dedicated queues use the knowledge that only one producer (or consumer) is using the queue and omit the synchronization code. Optimistic queues accept queue insert and queue delete operations from multiple producers and multiple consumers. The optimistic queue is described in detail in Section 3.2.

Quajects such as threads are created by the quaject creator, which contains three stages: allocation, factorization, and optimization. The allocation stage allocates memory for the quaject and all associated synthesized procedures. The factorization stage uses Factoring Invariants to substitute constants into the quaject's code templates. The optimization stage then improves the final code with specialized peephole optimizations.

The quaject interfacer starts the execution of existing quajects by installing them in the invoking thread. The quaject interfacer has four stages: combination, factorization, optimization, and dynamic link. The combination stage finds the appropriate connecting mechanism (queue, monitor, pump, or a simple procedure call). Factorization and optimization (the same as quaject creator) clean up the connecting code. Then the dynamic link stage stores the synthesized code's entry points into the quajects.

3 Reduced Synchronization

3.1 Overview

We have three methods to reduce synchronization cost: Code Isolation, Procedure Chaining, and Optimistic Synchronization. Each method shortens the execution path in a somewhat different way. Informally speaking, Code Isolation and Procedure Chaining can be thought of as synchronization avoidance techniques. If absolutely unavoidable we use Optimistic Synchronization.

Code Isolation uses kernel code synthesis to separate and isolate fragments of data structure manipulation programs. The separation eliminates unnecessary synchronization if each fragment operates on its own piece of data. For example, each thread has a Thread Table Entry (TTE, equivalent to the `proctable` in UNIX). Naive procedures that traverse the Thread Table to modify a TTE would have to lock the table. However, in Synthesis each thread updates its own TTE exclusively. Therefore, we can synthesize short code to manipulate the TTE without synchronization.

Procedure Chaining avoids synchronization by serializing the execution of conflicting threads. Instead of allowing concurrent execution that would have complicated synchronization problems, we chain the new procedure to be executed to the end of the currently running procedure. For example, the currently executing thread handles interrupts in Synthesis. A signal arriving in the middle of interrupt handling is potentially dangerous: the **`kill`** signal may terminate the interrupt handling prematurely. Therefore, we chain the procedure invoked by the signal to the end of the interrupt handler. Procedure Chaining is implemented efficiently by simply changing the return addresses on the stack.

Optimistic Synchronization assumes that interference between threads is rare, so we should shorten the normal non-interfering case. The idea of optimistic validation is to go ahead and make the changes, without locking anything. A check at the end of the update tests whether the assumption of non-interference remains true. If the test fails, we rollback the changes and retry. Using Optimistic Synchronization we have implemented an optimistic queue, which we describe now.

3.2 Optimistic Queues

The queue manipulation example for optimistic synchronization is important because most of the Synthesis kernel data structures are queues. Also, some of the control structures, such as chained interrupt and signal handlers, are implemented as queues of pointers to the routines. In other words, once we can synchronize queue operations without locking, most of the Synthesis kernel will run without locking.

Although all the queues have the usual put-item (Q_put) and get-item (Q_get) operations, we classify them according to their operations environment. We have four kinds of queues: single-producer and single-consumer (SP-SC), multiple-producer and single-consumer (MP-SC), single-producer and multiple-consumer (SP-MC), multiple-producer and multiple-consumer (MP-MC).

The simplest case, SP-SC (figure 1), gives the basic idea of all four queues: when the queue buffer is neither full nor empty, the consumer and the producer operate on different parts of the buffer. Therefore, synchronization is necessary only when the buffer becomes empty or full. The synchronization primitives are the usual primitives, say busy wait or blocking wait.

To argue the corretness of these queues, we need to show that these queues do not lose any items being put in or generate any items that has already been taken out. To avoid lost updates in the SP-SC queue, we use a variant of Code Isolation. Of the two variables being written, Q_head is updated only by

```
next(x):
    if(x == Q_size-1) return 0;
    else return x+1;

Q_get(data):            Q_put(data):
    t = Q_tail;             h = Q_head;
    if (t == Q_head)        if (next(h) == Q_tail)
        wait;                   wait;
    data = Q_buf[t];        Q_buf[h] = data;
    Q_tail = next(t);       Q_head = next(h);
```

Figure 1: SP-SC Queue

the producer and Q_tail only by the consumer. To avoid taking out an item repeatedly, we update Q_head at the last instruction during Q_put. Therefore, the consumer will not detect an item until the producer has finished.

The difference between the SP-SC queue and the MP-SC queue reduces to a single compare-and-swap instruction at the end plus the retry loop, to ensure the synchronization of multiple producers. (Larger critical sections may require more sophisticated synchronization.) A more interesting queue (shown in Figure 2) implements atomic inserts of many items (up to the size of the queue). Now we have two problems to consider: the multiple producer synchronization, solved by the compare-and-swap, and the atomic insert of multiple items, which we explain now.

To minimize the synchronization among the producers, each of them increments atomically the `Q_head` pointer by the number of items to be inserted, "staking a claim" to its space in the queue. The producer then proceeds to fill the space, at the same time as other producers are filling theirs. But now the consumer may not trust `Q_head` as a reliable indication that there is data in the queue. We fix this with a separate array of flag bits, one for each queue element. As the producers fill each queue element, they also set a flag in the associated array indicating to the consumer that the data item is valid. The consumer clears an item's flag as it is taken out of the queue.

To give an idea of relative costs, the current implementation of MP-SC has a normal execution path length of 11 instructions (on the MC68020 processor) through Q_put. In the case where two threads are trying to write an item to a sufficiently empty queue, they will either both succeed (if they attempt to increment Q_head at different times), or one of them will succeed as the other fails. The thread that succeeds consumes 11 instructions. The failing thread goes once around the retry loop for a total of 20 instructions.

```
AddWrap(x,n):
    x += n;
    if(x >= Qsize) x -= Qsize
    return x;

SpaceLeft(h):
    t = Q_tail;
    if(h >= t)  return t-h-1+Q_size;
    else  return t-h-1;

Q_put(data,N):
    do {
        h = Q_head;
        h1 = AddWrap(h,N);
    } while(Spaceleft(h) >= N
           && cas(Q_head,h,h1) == FAIL);
    for(i=0; i<N; i++) {
        Q_buf[ AddWrap(h,i) ] = data[i];
        Q_flag[ AddWrap(h,i) ] = 1;
    }
```

NOTE: cas(v,old,new) [compare-and-swap] performs the following operation atomically:
If(v == old) v = new; return OK; else return FAIL;

Figure 2: MP-SC Queue [Multiple Insert]

4 Threads

4.1 Synthesis Threads

Synthesis threads are light-weight processes. Each Synthesis thread (called simply "thread" from now on) executes in a context, defined by the TTE. The thread state is completely described by its TTE (see figure 3) containing: the register save area; the vector table, which points to four kinds of procedures (thread-specific system calls, interrupt handlers, error traps and signal vectors); the address map tables; and the context-switch-in and context-switch-out procedures.

Kernel code generated for a thread goes into a protected area to avoid user tampering. The kernel procedure bodies that make up part of the thread are:

- the `signal, start, stop, step` and `destroy` thread calls;
- the customized I/O system calls, synthesized by `open` (see Section 5);
- the synthesized interrupt handlers, such as for queue buffering (see Section 5.4);
- the specialized error trap handlers and the signal-me procedures (see Section 4.3).

Figure 3: Thread Context

When a Synthesis thread makes a kernel call, we say that the thread is executing in the kernel mode; this is in contrast to having a kernel server process run the kernel call on the behalf of the client thread. The `trap` instruction switches the thread into the supervisor state and makes the kernel quaspace accessible in addition to the user quaspace. Consequently, the kernel call may move data between the user quaspace and the kernel quaspace. Since the other quaspaces are outside the kernel quaspace, were the thread to attempt access to an illegal address, the thread will take a bus-fault exception, even in the kernel mode.

If a thread is not running, it is *waiting*. A waiting thread is blocked for some event or resource. Each resource has its own waiting queue. For example, a thread waiting for CPU is sitting in the ready queue; when the thread blocks for characters from a `tty` driver, it is chained to the `tty` driver queue. Spreading the waiting threads makes blocking and unblocking faster. Since we have eliminated the general blocked queue, we do not have to traverse it for insertion at blocking or to search it for deletion at unblocking. A waiting thread's unblocking procedure is chained to the end of the interrupt handling, so each waiting queue has reduced synchronization due to Code Isolation.

4.2 Context Switches

Context switches are expensive in traditional systems like UNIX because they always do the work of a complete switch: save the registers in a system area, setup the C run-time stack, find the current proc-table and copy the registers into proc-table, start the next process, among other complications (summarized from source code [1]). A Synthesis context-switch is shorter for two reasons. First, we switch only the part of the context being used, not all of it. Second, we use executable data structures to minimize the critical path.

In two instances we can optimize context switch by moving data only when they are used. The first is the handling of floating point registers and the second is the MMU address space switch. Most of Synthesis threads do not use the floating point co-processor. If we were to save all the floating point co-processor information at each context switch, the hundred-plus bytes of information takes about 10 microseconds to save to memory, which is comparable to the 11 microseconds needed to do an entire context switch without the floating point (see Section 6.3 for more data). Since most threads will not use the floating point co-processor, we generate the default context switch code without it. When the thread executes its first floating point instruction, an illegal instruction trap happens. Then the Synthesis kernel resynthesizes the context switch procedures to include the floating point co-processor. This way, only users of the floating point co-processor will pay for the added overhead.

There is no "dispatcher" procedure in Synthesis. Figure 3 shows that the ready-to-run threads (waiting for CPU) are chained in an executable circular queue. A `jmp` instruction in each context-switch-out procedure of the preceding thread points to the context-switch-in procedure of the following thread. Assume thread-0 is currently running. When its time quantum expires, the interrupt is vectored to thread-0's context-switch-out procedure (`sw_out`). This procedure saves the CPU registers into thread-0's register save area (`TT0.reg`). The `jmp` instruction then directs control flow to one of two entry points of the next thread's (thread-1) context-switch-in procedure, `sw_in` or `sw_in_mmu`. Control flows to `sw_in_mmu` when a change of address space is required, otherwise control flows to `sw_in`. The context-switch-in procedure then loads the CPU's vector base register with the address of thread-1's vector table, restores the CPU general registers, and resumes execution of thread-1. The context switch takes about 11 microseconds (see Table 4). This is a striking example of what can be achieved with optimization through synthesized code.

4.3 Thread Operations

As a quaject, the thread supports several operations. Some of these operations are invoked by the hardware; the error trap handlers and the interrupt handlers fall into this category. Some of the operations are invoked by other threads; these are `signal`, `start`, `stop`, and `step`. We will introduce briefly these operations here and describe interrupt handling in Section 5.3.

In Synthesis (as in many other systems), a signal is an asynchronous software interrupt sent by a thread (or interrupt handler) to another thread. A synthesized `signal` system call (the signal-me procedure) in the receiving thread calls the signal handler procedure. To run the signal handler in user mode and user quaspace, the `signal` system call alters the general registers area of the receiving thread's TTE to make the receiving thread call the signal handler when activated.

Thread control and debugger support is implemented with three synthesized system calls: `stop`, `start`, and `step`. The `stop` system call suspends execution of a thread by removing the thread's TTE from the ready queue. The `start` system call puts the TTE back when invoked. The `step` system call causes a stopped thread to execute a single machine instruction and then stop again. The debugger runs as an asynchronous thread that shares the quaspace being debugged.

An error trap is a synchronous hardware interrupt generated by illegal operations such as referencing non-existent memory or dividing by zero. Like other hardware interrupts, error trap handlers run in kernel mode. Unlike other hardware interrupts, error traps are synchronous since they occur immediately after each illegal operation. Each thread may have its own error trap handlers. To allow arbitrarily complex error handling in user mode, we send an error signal to the interrupted thread itself. The error signal handler then runs in user mode (as described above). To send this error signal, the error trap handler copies the kernel stack frame onto the user stack, modifies the return address on the kernel stack to the user error signal procedure, and executes a return from exception which takes the thread into the user error signal procedure. Synthesized for each thread at creation time, these error trap handlers consume about 5 machine instructions, supporting efficient emulation of unimplemented kernel calls or machine instructions. The UNIX emulator used for performance measurement is implemented with traps.

4.4 Scheduling

Currently, the Synthesis scheduling policy is round-robin with an adaptively adjusted CPU quantum per thread. Instead of priorities, Synthesis uses *fine-grain scheduling*, which assigns larger or smaller quanta to threads based on a "need to execute" criterion. A detailed explanation on fine-grain scheduling is beyond the scope of this paper; the idea and its implementation in Synthesis are described in detail in another paper [3]. Here, we only give a brief informal summary.

In our directed graph model of computation (Section 2.1), a thread's "need to execute" is determined by the rate at which I/O data flows into and out of its quaspace. Since CPU time consumed by the thread is an increasing function of the data flow, the faster the I/O rate the faster a thread needs to run. Therefore, our scheduling algorithm assigns a larger CPU quantum to the thread. This kind of scheduling must have a fine granularity since the CPU requirements for a given I/O rate and the I/O rate itself may change quickly, requiring the scheduling policy to adapt to the changes.

Effective CPU time received by a thread is determined by the quantum assigned to that thread divided by the sum of quanta assigned to all threads. Priorities can be simulated and preferential treatment can be given to certain threads in two ways: we may raise a thread's CPU quantum, and we may reorder the ready queue when threads block and unblock. As an event unblocks a thread, its TTE is placed at the front of the ready queue, giving it immediate access to the CPU. This way we minimize response time to events. To minimize time spent context switching, CPU quanta are adjusted to be as large as possible while maintaining the fine granularity. A typical quantum is on the order of a few hundred microseconds.

5 Input/Output

In Synthesis, I/O means more than device drivers. I/O includes all data flow among hardware devices and quaspaces. Data move along logical channels we call *streams*, which connect the source and the destination of data flow. The details of the stream model of I/O will be described in a separate paper [4]. Here we describe how the streams are implemented using the building blocks described in Section 2.3.

5.1 I/O Device Servers

Physical I/O devices are encapsulated in quajects called device servers. Typically, the device server interface supports the usual I/O operations such as `read` and `write`. In general, `write` denotes data flow in the same direction of control flow (from caller to callee), and `read` denotes data flow in the opposite direction of control flow (from callee back to caller).

Each device server may have its own threads or not. A polling I/O server would run continuously on its own thread. An interrupt-driven server would block after its initialization. The server without threads wakes up when its physical device generates an interrupt.

High-level servers may be composed from more basic servers. At boot time, the kernel creates the servers for the raw physical devices. A simple example pipelines the output of a raw server into a filter. Concretely, the Synthesis equivalent of UNIX cooked `tty` driver is a filter that processes the output from the raw `tty` server and interprets the erase and kill control characters. This filter reads characters from the raw keyboard server through a dedicated queue. To send characters to the screen, however, the filter writes to an optimistic queue, since output can come from both a user program or the echoing of input characters.

The default file system server is composed of several filter stages. Connected to the disk hardware we have a raw disk device server. The next stage in the pipeline is the disk scheduler, which contains the disk request queue, followed by the default file system cache manager, which contains the queue of data transfer buffers. Directly connected to the cache manager we have the synthesized code to read the currently open files. The other file systems that share the same physical disk unit would connect to the disk scheduler through a monitor and switch. The disk scheduler then will redirect the data flow to the appropriate stream. With synthesized code, this pipeline has a very low overhead, shown by the measurements in Section 6.

5.2 Producer/Consumer

The implementation of the stream model of I/O in Synthesis can be summarized using the well-known producer/consumer paradigm. Each stream has a control flow that directs its data flow. There are three cases of producer/consumer relationships, which we shall consider in turn.

Perhaps the simplest case is an active producer and a passive consumer (or vice-versa). This case, called active-passive, has simple implementations. When there is only one producer and one consumer (single-single), a procedure call does the job. If there are multiple producers or consumers (multiple-single), we attach a monitor to the end with multiple participants to serialize their access.

But the normal producer/consumer problem has both an active producer and an active consumer. This case, called active-active, requires a queue to mediate the two. For the single-single case, an SP-SC queue suffices. For the multiple-single case, we may attach a monitor to the multiple end, resulting in MP-SC or MP-MC queues. Each queue may be synchronous (blocking) or asynchronous (using signals) depending on the situation.

The last case is a passive producer and a passive consumer. One example is the `xclock` program that has the clock producer ready to provide a reading at any time and a display consumer that accepts new pixels to be painted on the screen. In these cases, we use a *pump* quaject that reads (the clock time) from the producer and writes the information (new pixels) to the consumer. This works for multiple passive producers and consumers as well.

In summary, we have an efficient implementation for each case of the producer/consumer problem. Since the stream model of I/O can be easily described as a composition of producers and consumers through the three building blocks (switches, monitors, and queues), we have shown the generality of the Synthesis implementation. In practice, composing a new device server with these building blocks is straightforward.

5.3 Interrupt Handling

At the heart of an I/O device server is the interrupt handler. Interrupt processing combines some elements of procedure calls and others of context switches. Like a procedure call, an interrupt pushes the currently executing stack and the return address. When the interrupt handling finishes, the execution resumes from the interrupted instruction in the current thread. Like a context switch, an interrupt is unexpected and unrelated to the current thread. Furthermore, the interrupt handler temporarily changes the program counter and some general registers of the current thread, without receiving any arguments from or returning any results to the current thread.

Synthesis interrupt handling differs from some traditional OS's (such as UNIX) in that each thread in Synthesis synthesizes its own interrupt handling routine, as well as system calls. These customized interrupt handlers and system calls may run much faster than general-purpose equivalents. Two examples of synthesized interrupt handlers are the timer interrupt to context switch out the current thread (Section 4.2) and the analog to digital (A/D) buffered queue (Section 5.4).

One way to increase the concurrency in the kernel is to push the bulk of interrupt processing (e.g., a character arrives at `/dev/tty`, to be inserted into the queue) into a separate thread which is created by the interrupt handler. However, in most cases the separate thread is uneconomical, since normal interrupts require very little processing. For the simple cases the interrupt handler could run under the currently executing thread to avoid context switch. We only have to take care to save and restore the few registers that the interrupt handler will use. During the (short) interrupt processing, higher level interrupts may happen and as long as the interrupt handling is simple, the scenario repeats until eventually the highest level interrupt processing completes and returns to the next level. Ultimately the entire stack of interrupts is handled.

Even though the thread running the simple interrupt handler can take care of recursive interrupts, signals may cause synchronization problems. We have two choices to handle a signal in the middle of an interrupt: either we create a new thread to finish the interrupt, or we delay the processing of the signal. Delaying the signal costs less, since it bypasses the creation of a new thread, and it does not degrade system performance significantly, since the current interrupt handling should be quick. We use Procedure Chaining to delay the signal, linking the signal processing routine to the end of interrupt handler.

Each Synthesis thread has its own vector table, which points to routines servicing hardware interrupts, error traps, and system calls. Although in principle each thread may have a completely different set of interrupt handlers, currently the majority of them are shared by all threads. System calls, however, are frequently customized for each thread. In particular, I/O operations such as `read` and `write` are synthesized by the `open` operation. As new quajects are opened (such as files, devices, threads, and others), the thread's system call vectors are changed to point to the synthesized procedures. At its creation, a thread's vector table is filled with a default set of system calls and error vectors that help debugging.

5.4 Optimizations

At boot time, the kernel uses Collapsing Layers to optimize the device servers. For example, instead of communicating to the raw `tty` through a pipe (as in the conceptual model) the cooked `tty` makes a procedure call to the raw `tty` to get the next character. This transforms a combination of active-passive producer/consumer pair into a procedure call. Down the pipeline, the cooked `tty` actively reads and the `tty` device itself actively writes, forming an active-active pair connected by an SP-SC optimistic queue.

Another optimization is the *buffered queue*. Usually, queue operations are cheap (a dozen instructions) compared to the processing time of each element in the queue. However, in the kernel we have cases of data movement that do very little work for each queue operation, thus the queue operations become the main overhead. Buffered queues use kernel code synthesis to generate several specialized queue insert operations (a couple of instructions); each moves a chunk of data into a different area of the same queue element. This way, the overhead of a queue insert is amortized by the blocking factor. For example, the A/D device server handles 44,100 (single word) interrupts per second by packing eight 32-bit words per queue element (hardware described in Section 6.1).

		====Raw Sun Data===				Sun	Synthesis		Synthesis
No	Descr.	user	sys	tot	watch	U+S	Emulator	Ratio	thruput
1	Compute	19.8	0.5	20	20.9	20.3	21.42	0.95	
2	R/W pipe 1	0.4	9.6	10	10.2	10.0	0.18	56.	100KB/s
3	R/W pipe 1024	0.5	14.6	15	15.3	15.1	2.42	6.2	8MB/sec
4	R/W pipe 4096	0.7	37.2	38	38.2	37.9	9.64	3.9	8MB/sec
5	R/W file	0.5	20.1	21	23.4	20.6	2.91	7.1	6MB/sec
6	open null/close	0.5	17.3	17	17.4	17.8	0.69	26.	
7	open tty/close	0.5	42.1	43	43.1	42.6	0.88	48.	

Table 1: Measured UNIX System Calls (in seconds)

6 Measurements

6.1 Environment

The current implementation of Synthesis runs on an experimental machine (called the Quamachine), which is similar to a SUN-3: a Motorola 68020 CPU, 2.5 MB no-wait state main memory, 390 MB hard disk, $3\frac{1}{2}$ inch floppy drive. In addition, it has some unusual I/O devices: two-channel 16-bit analog output, two-channel 16-bit analog input, a compact disc player interface, and a 2Kx2Kx8-bit framebuffer with graphics co-processor.

The Quamachine is designed and instrumented to aid systems research. Measurement facilities include an instruction counter, a memory reference counter, hardware program tracing, and a microsecond-resolution interval timer. The CPU can operate at any clock speed from 1 MHz up to 50 MHz. Normally we run the Quamachine at 50 MHz. By setting the CPU speed to 16 MHz and introducing 1 wait-state into the memory access, the Quamachine can closely emulate the performance of a SUN-3/160.

We also have written a UNIX emulator running on top of the Synthesis kernel, which is capable of servicing SUNOS kernel calls. In the simplest case, the emulator translates the UNIX kernel call into an equivalent Synthesis kernel call. Otherwise, multiple Synthesis primitives are combined to emulate a UNIX call. With both hardware and software emulation, we run the same object code on equivalent hardware to achieve a fair comparison between Synthesis and SUNOS.

All benchmark programs were compiled on the SUN 3/160, using `cc -O` under SUNOS release 3.5. The executable `a.out` was timed on the SUN, then brought over to the Quamachine and executed under the UNIX emulator. To validate our emulation, the first benchmark program is a compute-bound test of similarity between the two machines. This test program implements a function producing a chaotic sequence [2]. It touches a large array at non-contiguous points, which ensures that we are not just measuring the "in-the-cache" performance.

6.2 Comparing Synthesis with SUNOS

The purpose of making the Synthesis hardware and software emulate the SUN 3/160 is to compare Synthesis with SUNOS kernel calls. Since the executables are the same, the comparison is direct. In table 1 we summarize and compare the results of the measurements. The columns under "Raw SUN data" were obtained with the `time` command and also with a stopwatch. The SUN was unloaded during these measurements, as `time` reported more than 99% CPU available for them. The Synthesis emulator data were obtained by using the microsecond-resolution real-time clock on the Quamachine, rounded to hundredths of a second. These times were also verified with stopwatch (sometimes running each test 10 times to obtain a more easily measured time interval).

The source code for the programs numbered 1 to 7 are included in appendix A. Program 1 is the computation-intensive calibration function to validate the hardware emulation. The calibration program shows the Synthesis emulator to be roughly 5% slower than a SUN 3/160. Recently we learned that the SUN 3/160 runs actually at 16.7 MHz, which is about 5% faster than 16 MHz.

Programs 2, 3, and 4 write and then read back data from a pipe in chunks of 1, 1K and 4K bytes. They show a remarkable speed advantage (56 times) for single-byte read/write operations. This is due to a combination of synthesized kernel calls, which are very short, and fine-grain scheduling, which reduces the average queue operation costs. When the chunk grows to page size, the difference is still very significant (4 to 6 times). The generated code loads long words from one quaspace into registers and stores them back in the other quaspace. With unrolled loops this achieves the data transfer rate of about 8MB per second. Program 5 reads and writes a file (cached in main memory) in chunks of 1K bytes. This is the same program used in an earlier measurement of Synthesis [5] and shows some improvement in the current implementation of Synthesis.

We include the programs 6 and 7, which open/close `/dev/null` and `/dev/tty`, to show that Synthesis kernel code generation is very efficient. The open and close operations synthesize code for later read and write, yet they are 20 to 40 times faster than the UNIX open without code generation. Although this Synthesis file system is entirely memory-resident, the 10000 loops must have kept all the data pages in the SUNOS memory buffers, minimizing this difference. Table 2 contains more details of file system operations that are discussed in the next section.

6.3 Synthesis Kernel Calls

To obtain direct timings of Synthesis kernel call times (in microseconds), we use the Synthesis kernel monitor execution trace, which records in memory the instructions executed by the current thread. Using this trace, we can calculate the

operation	native time (usec)	Unix emulation (usec)
emulation trap overhead	-	2
open (/dev/null)	43	49
open (/dev/tty)	62	68
open (file)	73	85
close	18	22
read 1 char from file	9 (*)	10 (*)
read N chars from file	9+N/8 (*)	10+N/8 (*)
read N from /dev/null	6	8

(*) Data already in kernel queues or buffer cache.

Table 2: File and Device I/O (in microseconds)

operation	time (usec)
create	142
destroy	11
stop	8
start	8
step	37
signal	8 (thread to thread)

Table 3: Thread Operations (in microseconds)

exact kernel call times by counting the memory references and each instruction execution time. Tables 2 to 5 show the timings calculated for SUN emulation mode. (When running full speed at 50 MHz, the actual performance is about three times faster.)

In table 2 we have more file and device I/O operations. These operations are the native Synthesis file and device calls. A comparison of the native mode and the emulator mode shows the cost of UNIX emulation in Synthesis. Worth noting in Table 2 is the cost of `open`. The simplest case, `open (/dev/null)`, takes 49 microseconds, of which about 60% are used to find the file (hashed string names stored backwards) and 40% for code synthesis (read and write null). The additional 19 microseconds in opening `/dev/tty` come from generating real code to read and write. Finally, opening a file implies synthesizing more sophisticated code and buffer allocations (17 additional microseconds).

In table 3, we see that Synthesis threads are light-weight – less than 150 microsecond creation time. Of these, about 100 are needed to fill approximately 1KBytes in the TTE and the rest are used by code synthesis. The short time to start, stop, and step a thread makes it possible to trace and debug threads in a highly interactive way.

operation	time (usec)	
Full context switch	11	(*)
Full context switch	21	(with FP registers)
Partial context switch	3	
Block thread	4	
Unblock thread	4	

(*) If the thread does not use the Floating Point co-processor.

Table 4: Dispatcher/Scheduler (in microseconds)

operation	time (usec)	
Service raw TTY interrupt	16	
Service raw A/D interrupt	3	
Set alarm	9	
Alarm interrupt	7	
Chain to a procedure	4	(if no re-try)
Chain to a procedure	7	(with 1 re-try)
Chain (signal) a thread	9	(delayed interrupt)

Table 5: Interrupt Handling (in microseconds)

In table 4 we see the context switch times consumed by the dispatcher. Again we note that these timings are achieved with generated code (executable data structures, in this case). The separation between using and not using the floating point co-processor is to shorten the main critical path (explained in Section 4.2). Table 5 shows some timings for interrupt handling, alarm setting and handling, and signaling. For example, raw `tty` interrupt handling simply picks up the character.

Attentive readers would have noticed that our measurement figures are faster than traditional run-time library routines. For example, naive implementations of memory allocation, block copy, and string comparison would have slowed down our system considerably. In Synthesis, the memory allocation routine is an executable data structure implementing a fast-fit heap [6] with randomized traversal added. The block copy as used in `read` has been outlined in Section 6.2. The string comparison was mentioned as part of the `open` earlier in this section.

6.4 Kernel Size

The Synthesis kernel is written in 68020 assembly language, which is used as a fast prototyping language. This may sound peculiar, since usually people use high-level programming languages for fast prototyping. However, given the lack of support for efficient dynamic code synthesis in particular and efficient static code generation in general, we were unable to find a suitable compiler. We are actively pursuing the design and implementation of a high-level programming language for the development of the next-generation Synthesis.

A rough breakdown of the kernel shows about 3000 lines of code for the raw device drivers (TTY, disk, A/D and D/A, graphics), 1000 lines for the quaject creator and interfacer, 1000 lines for the templates used in code synthesis (e.g. queues, threads, files), 1000 lines for utilities and shared library (e.g. printf), and about 5000 lines for the kernel monitor with high-level debugging and programming tools. The whole kernel assembles to 64KBytes, of which 32KB are the kernel monitor.

There is some concern on kernel size when using code generation since many little functions can add up to a lot of memory. However, there are space advantages to code generation. While it is true that a synthesis kernel running several hundred threads each having many open files can use more memory than a UNIX system running a similar load, such heavily loaded systems are not normally seen. On a lightly loaded system, the static kernel size dominates any space allocated dynamically. This is where Synthesis excels. With 3 processes running, the Synthesis kernel occupies only 32K. As more threads are created and more files opened, the space requirements go up. However, the small static space required for the kernel means that you can run Synthesis on small, PC-like computers and embedded industrial controllers, two application areas that are unlikely to have much more than a few tens of threads running simultaneously. On the other hand, if you have a machine that *can* run 300 jobs concurrently, then you probably have the extra memory space to run them well.

7 Conclusion

We expect the techniques described here to be useful to operating system implementors. Specifically, we have used kernel code synthesis, optimistic synchronization, and fine-grain scheduling to increase OS kernel concurrency and efficiency in the implementation of the Synthesis kernel support for threads and input/output.

To achieve very high performance in Synthesis, we repeatedly apply the principle of frugality, which says that we should use the simplest abstraction and the cheapest implementation for each case. Given the level of abstraction of Synthesis kernel interface (all references to kernel data structures or algorithms eliminated), we can then use sophisticated algorithms to implement this interface. Although we use many different tricks to speed up the Synthesis kernel, their common theme is the simplification of the normal case, as dictated by the principle of frugality.

Kernel code synthesis shortens the normal execution path by binding the system state early; subsequent kernel calls simply jump into the generated routines, avoiding the system state traversal repetition. At code generation time, we also apply known compiler optimization techniques, such as constant folding and common sub-expression elimination. This is applied throughout Synthesis, including threads and input/output.

Reduced synchronization shortens the critical path by careful set-up and exception handling. For example, we have implemented queue operations using only Optimistic Synchronization. Since almost all of Synthesis kernel data structures are queues, the kernel basically runs without any inter-locking. We expect this to be especially important when we move Synthesis to a multi-processor, as it is designed for.

Combining kernel code synthesis and optimistic synchronization we have achieved very high performance compared to mature, commercial systems. For example, using a UNIX emulator running on a hardware emulator of SUN 3/160 to run the same binary executable, Synthesis performance (for I/O and threads) is several times or several dozen times better than SUNOS. Since optimistic synchronization is best suited for a multi-processor and fine-grain scheduling for a distributed system, we expect more performance gains when we run Synthesis on those environments.

8 Acknowledgments

This work is partially funded by the New York State Center for Advanced Technology on Computer and Information Systems under the grant NYSSTF CU-0112580, by the AT&T Foundation under a Special Purpose Grant, and by the National Science Foundation under the grant CDA-88-20754. We gladly acknowledge the hardware parts contributed by AT&T, Hitachi, IBM, and Motorola. Finally, very special thanks go to John Ousterhout, our "shepherd" SOSP program committee member, who helped shape both the style and the content of this paper.

References

[1] Anonymous et al.
SUNOS release 3.5 source code.
SUN Microsystems Source License, 1988.

[2] Douglas Hofstadter.
Gödel, Escher, Bach: an eternal golden braid.
Basic Books, 1979.

[3] H. Massalin and C. Pu.
Fine-grain scheduling.
In *Proceedings of the Workshop on Experience in Building Distributed Systems*, Asilomar, Calilfornia, October 1989.

[4] C. Pu and H. Massalin.
Model of computation in Synthesis.
Technical Report CUCS-383-88, Department of Computer Science, Columbia University, In preparation.

[5] C. Pu, H. Massalin, and J. Ioannidis.
The Synthesis kernel.
Computing Systems, 1(1):11–32, Winter 1988.

[6] C.J. Stephenson.
Fast fits.
In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 30–32, October 1983.

[7] 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.

A Measurement Programs