# A Study of Dynamic Optimization Techniques: Lessons and Directions in Kernel Design

*Calton Pu* and *Jonathan Walpole*

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
P.O. Box 91000
Portland, OR 97291-1000

{calton,walpole}@cse.ogi.edu

Technical Report No. OGI-CSE-93-007

**Abstract**

The Synthesis kernel [21, 22, 23, 27, 28] showed that dynamic code generation, software feedback, and fine-grain modular kernel organization are useful implementation techniques for improving the performance of operating system kernels. In addition, and perhaps more importantly, we discovered that there are strong interactions between the techniques. Hence, a careful and systematic combination of the techniques can be very powerful even though each one by itself may have serious limitations. By identifying these interactions we illustrate the problems of applying each technique in isolation to existing kernels. We also highlight the important common under-pinnings of the Synthesis experience and present our ideas on future operating system design and implementation. Finally, we outline a more uniform approach to dynamic optimizations called incremental partial evaluation.

# 1 Introduction

Historically, measures of throughput have formed the primary basis for evaluating operating system performance. As a direct consequence, many operating system implementation techniques are geared towards optimizing throughput. Unfortunately, traditional approaches to improving throughput also tend to increase latency. Examples include the use of large buffers for data transfer and coarse-grain scheduling quanta. This approach was appropriate for the batch processing model of early computer systems. Today's interactive multimedia computing environments, however, introduce a different processing model that requires different performance metrics and implementation techniques.

The new computing model is one in which data is transferred in real-time between I/O devices, along a pipeline of system and application-level computation steps. In this interactive environment, applications are primarily concerned with "end-to-end" performance, which is determined not only by operating system throughput, but also by the magnitude and variance of the latency introduced at each step in the pipeline. Reducing and controlling end-to-end latency, while maintaining throughput, in this "pipelined" environment, is a key goal for next-generation operating systems.

In contrast to the totally throughput-oriented implementation techniques of conventional operating systems, the Synthesis kernel sought to investigate dynamic optimization techniques that would provide lower and more predictable latency as well as improving throughput. In particular, Synthesis incorporates dynamic code generation to reduce the latency of critical kernel functions, and software feedback to control the variance in latency introduced by the operating system's resource scheduling policies.

Our experience with Synthesis showed these dynamic optimization techniques to be interesting and useful in their own right. However, the more important kernel design lessons we learned from the project relate to the interactions between the techniques used in Synthesis, and their relationship to more traditional kernel design approaches. By focusing on those lessons this paper makes the following contributions: (1) it discusses the interaction between fine grain modularity, dynamic code generation and software feedback, (2) it identifies the difficulties in applying each of these techniques in isolation to traditional operating systems, and (3) it explains the implementation-related limitations of Synthesis and outlines how we plan to overcome them in our new research project.

The paper is organized as follows. Section 2 motivates the need for dynamic optimization techniques by outlining some key performance challenges for next generation operating systems. Two of these techniques, dynamic code generation and software feedback, are summarized in sections 3 and 4 respectively, together with their advantages and problems. Section 5 discusses a more uniform methodology to address the problems in Synthesis. Section 6 outlines related work and Section 7 concludes the paper.

# 2 Performance Challenges

The advent of interactive multimedia computing imposes strict new requirements on operating system performance. In particular, next-generation operating systems must support the processing of real-time data types, such as digital audio and video, with low end-to-end latency and high throughput. The emerging model of computation is one in which real-time data enters the system via an input device, passes through a number of kernel and application processing steps, and is finally presented, in real-time, at an output device. In this environment, system performance is determined in large-part by the throughput and total end-to-end latency of this pipeline.

As multimedia applications and systems become more complex, the number of steps in the pipeline will increase. It is important that the addition of new steps in the pipeline does not cause significant increases in the end-to-end latency or decreases in throughput. This problem is a key challenge for operating system designers.

If operating systems implement data movement by buffering large amounts of data at each pipeline stage, and process it using correspondingly large CPU scheduling quanta, then adding pipeline elements will lead to undesirable increases in end-to-end latency. An alternative approach is to implement data movement and processing steps at a fine granularity, perhaps getting finer as the number of pipeline steps increases. This approach has traditionally not been taken because it does not allow operating system overhead, incurred during operations such as context switching, data transfer, system call invocation, and interrupt handling, to be amortized over large periods of useful work. Rather than focusing on amortizing these costs at the expense of end-to-end latency, we suggest that next generation operating systems must resolve the problem directly, by reducing the cost of these fundamental operating system operations.

The need for new design approaches is exacerbated by the trend towards microkernel-based operating systems. Such systems implement operating system functionality as a collection of coarse-grain server modules running above a minimal kernel. While this structuring approach has many well-publicized advantages, current implementations of it tend to lead to an increase in the cost of invoking operating system functions in addition to increasing the number of expensive steps in the pipeline.

Finally, the necessary process of emulating existing monolithic operating systems above microkernel-based operating systems makes the problem even worse. Current approaches to implementing emulation, such as redirecting system calls to user-level emulation libraries before invoking operating system function, introduce additional latency for kernel calls [16]. This in turn leads to unwanted increases in end-to-end latency for interactive real-time applications. Again, there are many important and well-accepted reasons for supporting emulation, e.g., utility and application software compatibility. What is needed are new implementation techniques to support it more efficiently.

In summary, next-generation operating systems, which are likely to be more modular and have multiple emulated interfaces, must provide support for very low-overhead data movement over long pipelines and control-flow transfer through many layers of software. This requirement has a major impact on the implementation of key operating system functions such as buffer management, interrupt handling, context switching, and system call invocation. They must also provide predictable real-time resource scheduling to support multimedia applications. Each of these areas has been well explored within the bounds of traditional kernel implementation approaches. The Synthesis kernel, however, departs from traditional approaches by making extensive use of the following two *dynamic optimization* techniques:

- dynamic code generation - to reduce the latency of common kernel functions, and
- software feedback - for adaptive resource scheduling with predictable variance in latency.

Each of these techniques has been described in detail in our earlier papers [22, 23, 28]. Therefore, with a brief introduction of the ideas, the following sections focus on the key lessons we learned from their application in Synthesis.

# 3 Dynamic Code Generation

## 3.1 The Techniques and Uses

Dynamic code generation is the process of creating executable code, during program execution, for use later during the same execution. The primary advantage of creating code dynamically, rather than at compile time, is that more information about the ultimate execution context is available to the code generation and optimization process. Consequently, more efficient code can be obtained.

The primary concern is that dynamic code generation is an on-line process carried out during run-time, in contrast to off-line compile-time code generation. Hence, only carefully selected low overhead optimization techniques can be applied, since the cost of run-time code generation may outweight its benefits. The code generation techniques used in Synthesis are divided into three groups: *factoring invariants*, *collapsing layers*, and *executable data structures*.

Factoring invariants is a special case of partial evaluation, that applies optimization techniques analogous to constant folding and constant propagation. The main difference is that Synthesis bypasses costly run-time data structure traversals in addition to constant folding. For a more efficient implementation of factoring invariants, pre-compiled templates that have already been optimized are used whenever possible. A good example of factoring invariants is the file system `open` call, which returns a critical path of a few dozen machine instructions that are used later by the calling thread to read/write that specific file [28]. In this case, the invariants are the thread requesting access, the file descriptor, and the file usage parameters.

Collapsing layers addresses the performance problem introduced by the increasingly popular abstract layered interfaces for systems software. Normal implementations fall into a difficult trade-off: either they implement each level separately for efficiency (resulting in untenable development and maintenance costs) or they compose lower layers to implement a high level (with heavy overhead at high levels). Collapsing layers is analogous to a combination of in-line macro expansion with constant folding. When a high-level function calls a lower level procedure, the code is expanded in-line. This inlining eliminates unnecessary barriers (the source of most data copying), allowing controlled and efficient data sharing by all the layers. An example of collapsing layers is the networking protocol stack [24]. A virtual circuit can allocate message buffer space at the top level and share it with the lower levels without additional copying. The Unix emulator in Synthesis also uses collapsing layers to reduce kernel call emulation cost [21].

Executable data structures are data structures (usually with fixed traversal order) optimized with embedded code to reduce interpretation overhead. Although this technique only saves a few instructions at a time, the savings are significant when the total number of instructions executed during each traversal step is small. This technique is especially useful when the work done on each data element is also small compared to the traversal cost. The Synthesis run queue, composed of thread table elements, is an example of an executable data structure [20]. At thread creation time, each element is optimized to reduce context switch cost. The pointer to the next thread, for example, serves as the destination of a jump instruction to eliminate address load overhead.

### 3.2 Performance Benefits

Many of the performance measurements on Synthesis were made on an experimental machine (called the Quamachine). Although the measured numbers on the Quamachine represent the compounded effects of custom software (the Synthesis kernel) and hardware, an effort was made to compare Synthesis performance fairly with that of an existing operating system kernel. We summarize here a comparison reported earlier [21].

The Quamachine was fitted with a Motorola 68020 CPU running at 16 MHz and memory speed comparable to a SUN-3/160, which has a 68020 processor at 16.67 MHz. Test programs were compiled under SUNOS 3.5 and the same executable run on both the SUN and the Quamachine with a partial Unix emulator on Synthesis. A validation program establishes that the two machines have comparable hardware (note that the test environment actually favors SUNOS performance). Figure 1 illustrates the performance improvements for pipe and file access obtained using Synthesis when running the same executable on equivalent hardware.[1] Reading and writing a pipe, 1 byte at a time, shows very high SUNOS overhead relative to Synthesis (56 times). Note, however that Synthesis also improves on SUNOS performance when reading and writing a pipe 4 kilobytes at a

[1] Figure extracted from Table 1 of [21].

| Test Program Description | SUN runtime | Synthesis Emulator | Speed Ratio | Synthesis throughput | Loops in prog |
|---|---|---|---|---|---|
| Validation | 20.3 | 21.42 | 0.95 | | 500000 |
| R/W pipe 1 Byte | 10.0 | 0.18 | 56. | 100KB/sec | 10000 |
| R/W pipe 4 KB | 37.9 | 9.64 | 3.9 | 8MB/sec | 10000 |
| R/W RAM file | 20.6 | 2.91 | 7.1 | 6MB/sec | 10000 |

*Execution time for complete programs, measured in seconds. Each program repeatedly executes the specified system calls (the left column). The validation program contains only user level memory location references.*

Figure 1: Comparison of Synthesis and SUNOS I/O Performance

time (almost 4 times).

A more recent experiment [20] illustrates the relative I/O latency for Synthesis and two widely used commercial operating systems. The Synthesis window system on the Sony 1860 NEWS workstation can finish `cat /etc/termcap` in 2.9 seconds, while X Windows (BSD Unix) takes 23 seconds and NextStep (a derivative of Mach) with similar hardware takes 55 seconds. Since dynamic optimization breaks down barriers between the kernel and server, this is not intended to be a direct comparison between systems. Nevertheless, while such high-level benchmarks do not isolate the specific benefits of each individual optimization (for example, the window system uses both dynamic code generation and software feedback, explained in Section 4.1), they do demonstrate the potential power of the combination of techniques used in Synthesis. Furthermore, sections 3.3 and 4.2 show not only that the various optimization techniques used in Synthesis are inter-dependent, but also that the interactions among them are very important. Hence, it is not appropriate, or particularly informative, to measure them in isolation.

### 3.3 Interaction With Other Ideas

Although dynamic code generation is intuitively appealing, it is not naively applicable in any operating system kernel. Several conditions must be met for dynamic code generation to have a high payoff. The first necessary condition is an encapsulated kernel, i.e., an abstract kernel interface that hides implementation details. Dynamic code generation wins when pieces of the kernel can be replaced with more specialized versions. The scope for this type of dynamic replacement is severely restricted when kernel data structures are visible at the user level, since computations are often specialized by replacing data structures. The core Unix file system kernel calls such as `read` and `write` are good examples of an abstract interface, but `nlist`, which examines the name list in an executable directly, is not. This is the first important lesson from Synthesis.

**Lesson 1** *An abstract kernel interface is essential for any substantial performance optimization based on dynamic code generation.*

This requirement is in contrast to conventional operating system kernel design approaches in which direct access to kernel data structures is viewed as a short cut and a low overhead way to obtain system information. The prevalence of this approach in monolithic operating system kernels makes an extensive application of dynamic code generation very difficult. For example, Unix `/dev/kmem` and MVS Control Vector Table make it impossible to optimize context switch without breaking a large number of system utilities.

The second necessary condition is a fine-grain modular organization of the kernel. Typically, dynamic code generation manipulates encapsulated objects and small independent code fragments with specific function. Note that this level of modularity is orthogonal to the modularity introduced by most microkernels where modularity is defined by microkernel and server boundaries. Individually, microkernels and their servers are significantly smaller than a monolithic operating system, however these modules are still too large and complex for the purposes of dynamic code generation. In particular, when data structures are shared among many functions within a server it becomes difficult to specialize individual functions, independently of the other functions. In other words, the shared data structure creates a dependency between the implementations of the functions that share it. Consequently, dynamic code generation gains effectiveness and applicability as the granularity of kernel modules is refined. This is the second important lesson from Synthesis.

**Lesson 2** *Fine-grain modularity within the kernel significantly increases the scope for performance optimization.*

This approach to kernel structure takes the evolution from monolithic systems to microkernels one step further. It also explains the difficulty in applying dynamic code generation extensively to microkernels modularized solely at server and kernel boundaries. The internal dependencies in such coarse-grain modules limit the potential benefits of applying dynamic code generation.

Lessons one and two led to the "objectification" of the Synthesis kernel [27, 20]. In the current version of Synthesis, the kernel is composed of small encapsulated modules called quajects. For example, queues, buffers, threads and windows are considered basic quajects since they support some kernel calls by themselves. Composite quajects provide high level kernel services such as a file system. The implementation of quajects in Synthesis does not rely on language support such as type checking or inheritance. Nevertheless, particular attention was paid to the interface between quajects as well as the kernel interface, which is completely encapsulated and operational. This allows several specialized kernel routines to run under the same kernel call interface. Although the Synthesis implementation is minimally sufficient for the degree of fine-grain modularity required for dynamic code generation, Section 5.2 discusses the kind of language support needed for a fine-grain modularization of kernels.

### 3.4 Important Questions

The Synthesis kernel has shown that dynamic code generation can produce significant performance improvements [21, 23, 20]. In this sense, the Synthesis project was useful as a proof of concept for the application of dynamic code generation in operating systems. However, the focus on dynamic code generation required hand-coded optimizations written in macro-assembler. The important issues of high level programming language support for dynamic code generation and the definition of a clear programming methodology were left out of Synthesis. This section discusses some of the difficulties associated with this *ad hoc* implementation of dynamic code generation. Section 5 discusses more recent research that focuses on incorporating dynamic code generation into a more integrated and well defined systems programming approach.

The lack of high level programming language support for dynamic code generation and interface description introduces a number of difficulties which impact issues such as portability, debuggability, and correctness. For example, since the current approach does not allow invariants to be described explicitly, it becomes difficult to reason about the validity of an optimized piece of code which has been generated dynamically by factoring invariants. A key problem is that the code generator must ensure that the invariants used for optimization hold for the duration of code execution. If any invariant is violated, the situation must be corrected by re-synthesizing code. Without support for explicit descriptions of invariants, such consistency checks become implicit in the code and make program maintenance, porting, and debugging more difficult. A similar problem arises when optimization parameters and goals are not described explicitly. The general difficulties outlined above appear in different concrete situations in Synthesis. For instance, Synthesis pays careful attention to cache management, particularly instruction and data cache consistency when generating code dynamically. However, cache-related invariants and optimization parameters remain completely implicit in the kernel code.

While we believe that systems based on dynamic code generation can be portable, it is clear that Synthesis' current implementation of dynamic code generation makes it difficult to preserve performance optimizations when porting. On the one hand, the extensive use of macro-assembler has allowed the kernel to be ported to a family of machines (Synthesis has been ported from an early 68010 to 68020 and then Sony's workstation with 68030). On the other hand, however, machine specific optimizations remain implicit in the kernel code. This is an insidious problem because the performance gains due to these optimizations are easily lost when porting to different machines. Note that this is a problem in the current implementation, not an inherent limitation of dynamic code generation. In section 5 we discuss how this problem can be addressed in future systems.

Finally, debugging dynamically generated code is a well recognized problem. However, there is an important distinction between our approach to dynamic code generation and traditional self-modifying code. In Synthesis, once generated, an execution path remains unchanged, i.e., code is not updated in place. Technically, when a fragment of code is generated or updated, it is not immediately put in the execution path. In addition to programming/debugging, this is a precaution taken to avoid performance penalties due to instruction cache invalidation. From this perspective, debugging dynamically generated code is similar to debugging object-oriented systems where objects may be created at run-time. Other projects in the operating system domain, such as the $x$-kernel, have similar characteristics.

Within the limitations of a kernel written in macro-assembler, Synthesis offers significant debugging aids. The Synthesis kernel contains a symbol table as part of its dynamic linking module, which is used to allow symbolic references to memory addresses. The kernel also contains a powerful monitor that supports breakpoints, instruction tracing, a disassembler, and an on-line compiler for a subset of C. Although the Synthesis kernel was not production quality software, several talented project students were able to understand it, modify it, and extend it using the kernel monitor [24]. Nevertheless, from a software engineering point of view, the problem of debugging executable code for which no source code exists remains a challenge.

# 4 Software Feedback

## 4.1 The Technique, Uses, and Benefits

Feedback mechanisms are well known in control systems. For example, phase-locked loops implemented in hardware are used in many applications including FM radio receivers. The intuitive idea of feedback systems is to remember the recent history and predict the immediate future based on the history. If the input stream is well behaved and the feedback memory sophisticated enough to capture the fluctuations in the input, then the feedback system can "track" the input signals within specified limits of stability (maximum error between predicted and actual input) and responsiveness (maximum elapsed time before error is reduced during a fluctuation).

Most control systems work in a well-understood environment. For example, the frequency modulation in FM radio transmission is very regular. In fact, a truly random input stream cannot be tracked by any feedback system. For a specified degree of stability and responsiveness, the complexity of a feedback system depends on the complexity of the input stream. The more regular an input stream is, the less information the feedback mechanism needs to remember.

Software implementations of feedback mechanisms are used in Synthesis to solve two problems: fine-grain scheduling [22] and scheduling for real-time I/O processing.

A serious problem in the SUNOS adaptive scheduling algorithm [3] is the assumption that all processes are independent of each other. In a pipeline of processes, this assumption is false and the resulting schedule may not be good. Fine-grain scheduling was introduced in Synthesis to solve this problem. In Synthesis, a producer thread is connected to a consumer thread by a queue. Each queue has a small feedback mechanism that watches the queue's content. If the queue becomes empty, the producer thread is too slow and the consumer thread is too fast. If the queue becomes full, the producer is too fast and the consumer too slow. A small scheduler (specific to the queue) then adjusts the time slice of the producer and consumer threads accordingly. A counter that is incremented when queue full and decremented when queue empty shows the accumulated difference between producer and consumer. Large positive or negative values in the counter suggest large adjustments are necessary. The goal of the feedback-based scheduler is to keep the counter at zero (its initial value). Since context switches carry low overhead in Synthesis, frequent adjustments can adapt to the varying CPU demands of different threads.

Another important application of software feedback is to guarantee the I/O rate in a pipeline of threads that process high-rate real-time data streams, as in next-generation operating systems supporting multimedia (section 2). A Synthesis program [20] that plays a compact disc simply reads from `/dev/cd` and writes to `/dev/speaker`. Specialized schedulers monitor the data flow through both queues. A high input rate from CD will drive up the CPU slice of the player thread and allow it to move data to its output buffer. The result is a simple read/write program and the kernel takes care of CPU and memory allocation to keep the music flowing at the 44.1 KHz CD sampling rate, regardless of the other jobs in the system. Because of the adaptiveness of software feedback, the same program works for a wide range of sampling rates without change to the schedulers.

In addition to scheduling, another example of software feedback application is in the Synthesis window system mentioned in Section 3.2. It samples the virtual screen 60 times a second, the number of times the monitor hardware draws the screen. The window system only draws the parts of the screen that have changed since the last hardware update. If the data is arriving faster than the screen can scroll it, then the window bypasses the lines that "have scrolled off the top". This helped reduce the `cat /etc/termcap` run-time from a 3.4 seconds calculated cost (multiplying the text length by unit cost) to the actually measured 2.9 seconds.

## 4.2 Interaction With Other Ideas

One of the fundamental problems with feedback mechanisms in general is that their complexity and cost increases with the complexity in the input signal stream. For this reason, it is generally not a good idea to provide a single implementation of a general-purpose feedback algorithm with a wide range of applicability, because it will be too expensive for the majority of input cases that are relatively simple. Furthermore, in the few cases where the input stream is even more complex than anticipated, the algorithm breaks down anyway.

Synthesis uses dynamic code generation to synthesize a simple and efficient software feedback mechanism that is specialized for each use. In addition, dynamic code generation is not limited to the simplification of the feedback mechanism. It also dynamically links the feedback into the system. For example, when monitoring the relative progress of processes in a pipeline, a counter-based mechanism can be used to monitor queue length. However, the fine-grain scheduler still needs to adjust the time slices of neighboring threads based on this information. Dynamic code generation links the local scheduler directly to the producer and consumer thread table entries, avoiding the thread table traversal overhead when adjustments are desired. This is the third important lesson from Synthesis.

**Lesson 3** *Software feedback requires dynamic code generation to be practical.*

Given the difficulties with applying dynamic code generation (lessons 1 and 2 in section 3.3), it is easy to see that it would be difficult to apply software feedback extensively in existing systems.

The success of software feedback in Synthesis also depends heavily on the fine-grain modular structure of the kernel. Each of the kernel's fine-grain modules (quajects) has a relatively simple input domain. This simplicity allows the feedback mechanism to be small and efficient. For example, one software feedback mechanism is used for each queue in a pipeline to manage two directly related threads, instead of using a global scheduler to control many threads. Software feedback is much more difficult to apply to relatively coarse-grain modules, such as microkernels and their servers, because the input stream for each coarse-grain module is considerably more complex than those of Synthesis quajects. This is the fourth important lesson from Synthesis.

**Lesson 4** *Fine-grain modularity enables the application of simple software feedback mechanisms.*

### 4.3 Important Questions

Although software feedback mechanisms have been used successfully in Synthesis, many important research questions remain unanswered. First, our approach was entirely experimental. Unlike in control theory where feedback mechanisms are well understood and their behavior characterized in detail, the theoretical foundations of software feedback have yet to be established. The reason for postponing the theoretical work is that the applicability of classical analysis is restricted to relatively simple domains, such as linear systems. In systems software, small embedded systems may be amenable to such analysis. General operating system environments, in contrast, are subject to unpredictable input fluctuations and thus classical analysis (or theoretical work with similar constraints) is of limited value.

On the experimental side, the scope of our contribution is restricted, so far, to the specific software feedback mechanisms developed in Synthesis. These mechanisms have been developed and optimized manually. Consequently, the design and implementation of new software feedback mechanisms for a different application is not an easy task. In addition, the software feedback mechanisms used in Synthesis have been tuned, and their parameters chosen, experimentally. There is no explicit testing of feedback stability or responsiveness for the cases input signal fluctuations exceed the specifications. For this reason, Synthesis feedback mechanisms tend to be conservative, having high stability even if this implies a somewhat slower response rate.

Another area of research that remained unexplored in Synthesis is the combination of software feedback with other approaches to guaranteeing levels of service and managing overload. For example, in systems where quality of service guarantees are important, resource reservation and admission controls have been proposed and used. During periods of fluctuating, medium to high, system load such approaches can become too conservative, resulting in excessive reservation of resources and unnecessary admission test failures. Software feedback, on the other hand, is a means for implementing *adaptive* resource management strategies that are very efficient during periods when resources are not saturated. In areas such as multimedia computing and networking where high data rates and strict latency requirements stress resources to the limits, efficiency and real-time service guarantees become critical.

## 5 A More Uniform Approach

### 5.1 A Next Generation Operating System Kernel

Synthesis showed that dynamic optimization techniques can be usefully added to the kernel developer's toolkit. Although dynamic code generation and software feedback borrow techniques developed in completely different areas, the former from compilers and the latter from control systems, they solve similar problems in an operating system kernel. Both gather information at runtime, one on state invariants and the other on program performance, to lower the execution overhead of the kernel. Both give the Synthesis kernel the ability to adapt to environmental changes: dynamic code generation provides coarse grain adaptation since invariants do not change often, and software feedback supports fine grain adaptation since it monitors changes continually. Both techniques are desirable in an operating system kernel, however, the problems enumerated in sections 3.4 and 4.3 remain to be solved.

Despite the apparent commonality in the underlying principles, the implementation and development of Synthesis fell short of defining a new kernel development methodology that applies these techniques in a uniform way. At the Oregon Graduate Institute, in collaboration with Charles Consel, we are developing a uniform programming methodology for next-generation operating system kernels, based on theoretical foundations in partial evaluation [12]. The approach, called *incremental partial evaluation* [13], applies partial evaluation repeatedly, whenever information useful for optimization becomes available in the system. Dynamic code generation in Synthesis can be seen as a concrete illustration of this general approach. Section 5.2 presents an overview of incremental partial evaluation.

To improve system adaptiveness we will make extensive use of software feedback for resource management. Instead of custom building each feedback mechanism, as was the case in Synthesis, we will construct a toolkit from which many software feedback mechanisms can be derived. Section 5.3 outlines the components of such a toolkit.

The commonality between each of these techniques, and their interaction with modular kernel design, is discussed in section 5.4. As mentioned in lessons 1 through 4, the ideas used in Synthesis are not easily applied to conventional kernel designs, especially not in isolation. Therefore, section 5.5 discusses the potential of our new approach to integrate techniques and hence, aid in their application to existing systems.

### 5.2 Incremental Partial Evaluation

Incremental partial evaluation can be divided into three parts: explicit invariant definition, incremental code generation, and dynamic linking. Kernel functions are defined in an abstract interface. At the top level of design, each function is implemented by a general algorithm, similar to traditional operating system kernels. The difference is that incremental partial evaluation hierarchically subdivides the input domain of the function by identifying and making explicit the invariants that lead to code optimization. For example, the creation of a thread and the opening of a file generate important invariants for the file `read` function when applied to that file in that thread. As these invariants become true at runtime, incremental partial evaluation uses them to incrementally optimize and generate code, a process called *specialization*. When the specialization process ends, the pieces are dynamically linked together and the execution path is ready for invocation.

Concretely, an operating system kernel using incremental partial evaluation is a hierarchical organization of multiple implementations for each function. At the top level we have the most general implementation for the entire input domain. At each level down the hierarchy, the implementations are for a subdomain of the input space and hence contain a simpler, faster execution algorithm. In order to achieve better performance on a specific architecture, the specialization at the lower levels can become increasingly architecture-dependent. This approach does not reduce portability, however, because the algorithms at the high level remain abstract and portable, i.e., the approach preserves the portability of operating system kernel code while allowing architecture-specific optimizations. Also, specializations that depend on particular architectures are clearly identified and isolated at the low levels of this implementation hierarchy.

Another important goal in incremental partial evaluation research is the application of automated specialization techniques, particularly on-line partial evaluation, to implement the hierarchy of multiple implementations. Automated specialization is usually abstract enough to be portable across different architectures. However, we do not rule out hand-written specialized implementations for two reasons. First, some critical paths in an operating system kernel may require hand tuning by the best programmer available. Second, new architectures may contain new instructions, memory mappings, and other facilities that existing automated procedures do not know about. For example, a simple but important function is data movement, commonly known as `bcopy`, which has several possible implementations, each with peculiar performance results for different situations. Therefore, we anticipate the usefulness of hand specialization for the foreseeable future.

A third goal in the incremental partial evaluation approach is to make synchronization primitives efficient *and* portable. Since we are building an operating system kernel for parallel and distributed systems, efficient synchronization is fundamental. In Synthesis, lock-free synchronization [20] was adopted and implemented with the compare-and-swap instruction. Since the compare-and-swap instruction is not available on all processor architectures, the portability of the synchronization mechanism is a serious question. We plan to adopt an abstract lock-free synchronization mechanism, such as transactional memory [17], and then use incremental partial evaluation to select an appropriate implementation of it using the facilities available on the target hardware platform.

We are in the process of defining high-level programming language support for incremental partial evaluation. To the kernel programmer, it will help by supporting modularity, strong typing, and well-defined semantics for automatic incremental partial evaluation, plus a systematic way to develop and maintain the hierarchy of multiple implementations. The necessary support includes the explicit definition of invariants, automated generation of guard statements to detect the breaking of an invariant, and support for the composition of specialized modules for dynamic linking.

A natural interface to specialized code in incremental partial evaluation would distinguish between abstract types and concrete types (as in Emerald [4]). Multiple implementations can be seen as the concrete types supporting the same abstract type. The invariants distinguish the concrete types and describe their relationship to each other in the implementation hierarchy. From this point of view, the hierarchy of multiple implementations is the symmetric reverse of inheritance. In a traditional object-oriented class hierarchy, subclasses inherit and generalize from a superclass by expanding its input domain. In dynamic optimization, each lower level in the implementation hierarchy specializes the implementation above it by restricting its input domain through some invariant.

### 5.3 Software Feedback Toolkit

Section 4.3 discussed the limitations of the Synthesis implementation of software feedback. To address these problems, we are developing a toolkit. This approach is analogous to the composition of elementary components, such as low pass, high pass, integrator, and derivative filters, in linear control systems. By composing these elements, the resulting feedback system can achieve a predictable degree of stability and responsiveness.

The software feedback toolkit is divided into three parts. First, interesting filters will be implemented in software. For example, low pass and band pass filters can be used in stabilizing feedback against "spikes" in the signal stream. Other filters can help in the scheduling of real-time tasks by incorporating the notion of priority, or value. Fine-grain schedulers can use a composition of these filters to achieve the desired stability and responsiveness given a well-behaved input stream.

To support this mode of construction, the toolkit provides a program that composes elementary filters to generate an efficient software implementation of a feedback mechanism, given a specification of the input stream. This program should run both off-line, to apply the full range of optimization techniques, as well as on-line, to support the regeneration of feedback mechanisms when the original specifications are exceeded. Note that this on-line version will utilize the same dynamic code generation techniques described earlier.

Finally, the toolkit will contain a set of test modules that observe the input stream and the feedback mechanism itself. When the input stream exceeds the specifications, or when the feedback is deemed unstable, a new feedback mechanism is generated dynamically to replace the "failed" one. Because a rigorous theoretical foundation for the application of software feedback does not exist yet, these tests are essential for protecting the overall stability of the system. Note that even

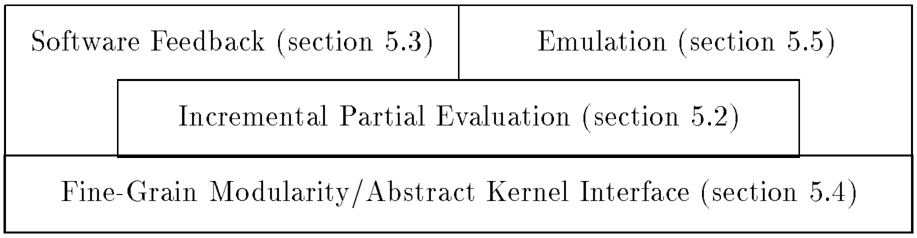


Figure 2: Techniques and Their Interaction

in control theory, composition, without testing, is guaranteed to work only for linear systems. In an open environment, such as a general-purpose operating system, there is no guarantee that the system behavior will remain linear, or bounded in any way. Therefore, even with a good theoretical understanding of the feedback mechanism, some form of test is necessary.

This toolkit should dramatically improve the ease with which software feedback mechanisms can be constructed and deployed in an operating system kernel. In this sense, the toolkit serves a similar role to the programming language support for incremental partial evaluation described above: both are tools that provide structured support for dynamic optimization.

### 5.4 Commonality Among Techniques

There are some striking similarities between the organization and use of the support mechanisms for software feedback and incremental partial evaluation. Both techniques achieve dynamic optimization by making assumptions about invariants. The test modules in the software feedback toolkit have an identical function to the invariant guards in incremental partial evaluation: both recognize when invariant-related assumptions are no longer valid. The effect of triggering both mechanisms is also similar: they both result in regeneration of new code. In the software feedback case a new feedback mechanism is generated to handle the new range of input stream values (which can be viewed as a new invariant). In the case of incremental partial evaluation a new code template is instantiated using new invariant-related information.

Despite these similarities, however, the two techniques are applicable in different circumstances. Where very fast response is needed, or where parameters change quickly, frequent code regeneration becomes too expensive. In these cases, a software feedback mechanism must be used that can adapt dynamically within the anticipated range of input stream values. Adaptation, via feedback, within this range is dramatically cheaper than adaptation, via code generation, outside the range. For this reason, software feedback is appropriate for highly volatile situations in which a small number of parameters change frequently but over a small range. In the less frequent cases where the input range is exceeded dynamic code generation is used to regenerate a new specialized feedback mechanism.

For other kernel modules involving infrequent parameter changes, dynamic code generation is more appropriate. Finally, if parameters are fixed or known prior to runtime, automatic or hand-coded off-line optimization techniques can be used. This wide range of optimization techniques has a common conceptual basis: the ability to identify invariants and localize the effects of implementation changes. Consequently, the requirements for an abstract kernel interface and fine-grain modular kernel structure are at the foundation of this approach. These dependencies are illustrated in Figure 2.

Note that the interdependence between layers in Figure 2 is mutual. Not only are the optimization techniques dependent on modularity, they also offer the key to implementing highly modular

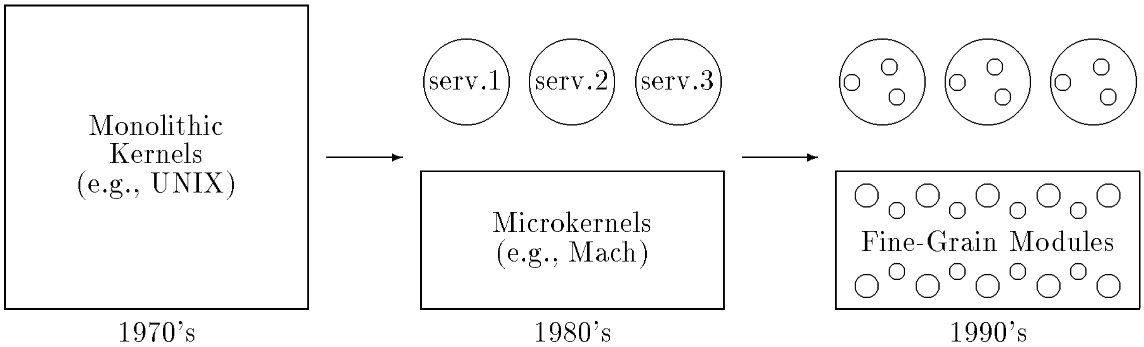


Figure 3: Evolution of Operating System Kernels

kernels efficiently. Without dynamic optimization, the overhead of a layered system design increases with the number of layers. The performance degradation of this approach tends to be worse than linear because layers tend to encapsulate functionality, hiding optimization-related information that could be utilized by higher layers.

Furthermore, a key tenet of systems design, "optimize for the most common case," breaks down since the growing complexity of the system at the top eventually defeats such optimizations implemented at the bottom. Note that dynamic optimization techniques allow an important extension of this principle. Rather than optimizing for a single case, dynamic optimization techniques allow a number of potentially common cases to be anticipated and the choice of a specialized implementation to be delayed until runtime, at which point it is possible to recognize the "actual case." If the actual case turns out to be one of the anticipated cases an optimized implementation is used. Otherwise, invariant guards or test modules cause a more general algorithm to be employed. A generalized tenet, therefore, is "optimize for many common cases." This tenet suggests multiple implementations, each for its own "common case." A corollary of the generalized tenet is "delay committing to a specific implementation as long as possible." This delay narrows down the number of possible cases, allowing the most appropriate implementation to be selected.

### 5.5 Integration Into Existing Systems

Figure 3 illustrates an evolution in operating system kernel structure from monolithic kernels, through coarse-grain modular kernels (including microkernels), to fine-grain modular kernels. The significance of this evolution is that it becomes progressively easier to apply Synthesis-style dynamic optimization techniques.

Moving from the left to the center of Figure 3 represents the introduction of encapsulated kernel and server interfaces in systems such as Mach [5], for example. This is the first requirement for applying dynamic optimization techniques. Moving from the center to the right of Figure 3 represents the introduction of fine-grain modularity within the kernel code at the level of objects or abstract data types. The most well known systems in this category are Choices [14] and Chorus [2], which use object oriented programming languages and techniques. Other systems in this category are discussed in [7, 6].

Even in the presence of fine-grain modularity, considerable work must be done to incorporate dynamic optimization techniques into a kernel. Selected modules must be rewritten using incremental partial evaluation and/or software feedback before being reintroduced in the original system. This approach can be extended to replace complete coarse-grain modules with new optimized implementations. Full encapsulation is essential to facilitate the reintroduction of these new implementations, as is the integration of the toolkit and partial evaluators that must participate at runtime.

This approach of evolving existing systems by systematically replacing encapsulated components opens the possibility for performance comparisons using the same underlying hardware and the same overlaying application software. In addition, the dynamically optimized modules can be used immediately by existing systems.

# 6 Related Work

Many of the individual optimization techniques and kernel structuring ideas discussed in this paper have been studied in other contexts. Coarse-grain modularity has been studied in the context of microkernel-based operating systems such as Mach [1], Chorus [33] and V [9]. Mach, for example, offers an encapsulated kernel, in which kernel resources and services are hidden behind a port/message-based interface [5]. The facilities that allow dynamic linking and loading of servers into the address space of a running Chorus kernel are also related to our research in that they allow a choice between different implementations of the kernel interface to made dynamically [16].

The use of object-oriented programming languages and design approaches has allowed operating systems such as Choices [8], Chorus [30], and Apertos [31] to utilize finer-grain modularity within their kernel code. The $x$-kernel [25] also offers relatively fine-grain modularity through its concept of micro-protocols. Its use of dynamic linking to compose micro-protocol modules is a good example of a dynamic optimization technique, i.e., the approach of dynamically constructing a protocol stack to better match the required execution context is closely related to the principles that underlie Synthesis.

Dynamic code generation has been used in a number of other research efforts. The Blit terminal from Bell Labs, for example, used dynamically optimized `bitblt` operations to improve display update speed [26]. Feedback systems have been discussed extensively in the context of control theory. Their application to system software has been focused in two areas: network protocols and resource management. In network protocols, feedback has been applied in the design of protocols for congestion avoidance [29]. In resource management, feedback has been used in goal-oriented CPU scheduling [15]. The principal distinction between Synthesis and these other research efforts is that Synthesis has applied these techniques extensively, and in careful combination, in the design of an operating system kernel.

As we start to emphasize a formal approach to dynamic optimization, existing partial evaluation work becomes more relevant. Dynamic optimization can benefit from off-line algorithms such as binding-time analysis [10] in practical systems [11]. Another related area of research on dynamic optimization is on reflection and meta-object protocols [32]. While most of the programming languages supporting meta-object protocols are interpreted, there are significant efforts focused on building an open compiler with customizable components [18]. An experiment to add reflection to $C^{++}$ [19] resulted in a recommendation to *not* modify $C^{++}$ to support reflection. However, much of this research could be useful in the support of the dynamic optimization techniques we envision.

# 7 Conclusion

The Synthesis project investigated the use of several interesting structuring and dynamic optimization techniques in the implementation of an operating system. Kernel structure was modular at a very fine granularity. Runtime optimization was based on two techniques, dynamic code generation and software feedback. Both of these dynamic optimization techniques depend heavily on the ability to encapsulate functionality at a fine granularity. Conversely, dynamic code generation is the key to building efficient implementations of highly modular operating systems, since it facilitates the collapsing of inter-module boundaries at execution time. Similar synergistic effects exist between software feedback and dynamic code generation: to be efficient and hence widely applicable, software feedback requires dynamic code generation.

By hand-coding these techniques in a prototype operating system kernel, Synthesis has shown that, when used in combination, they can be very powerful. However, the strong interactions and inter-dependencies between the techniques have inhibited the direct application of these positive results in other systems. An important research challenge, therefore, is to show how the techniques demonstrated in Synthesis can be incorporated into production quality operating system kernels.

This paper represents a concrete step in addressing this challenge. The interactions among the techniques are explained, their relationship to other kernel design approaches is discussed, and a potential migration path from existing encapsulated kernels is outlined. We also describe new technology that will facilitate the migration of dynamic optimization techniques into existing systems, specifically, a more uniform programming methodology based on incremental partial evaluation and a software feedback toolbox.

Our current research is focused on applying this new programming methodology, in an evolutionary way, to existing operating system kernels. We suggest that a new kernel programming approach, such as this, is key to meeting the stringent demands on kernel efficiency that arise in modern multimedia computing environments.

# 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. In *Proceedings of the 1986 Usenix Conference*, pages 93–112. Usenix Association, 1986.

[2] P. Amaral, R. Lea, and C. Jacquemot. A model for persistent shared memory addressing in distributed systems. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 2–12, Dourdan, France, September 1992.

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

[4] 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.

[5] D.L. Black, D.B. Golub, D.P. Julin, R.F. Rashid, R.P. Draves, R.W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and mach. In *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, Seattle, April 1992.

[6] L-F. Cabrera and E. Jul, editors. *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, Dourdan, France, September 1992. IEEE Computer Society Press.

[7] L-F. Cabrera, V. Russo, and M. Shapiro, editors. *Proceedings of the International Workshop on Object Orientation in Operating Systems*, Palo Alto, California, October 1991. IEEE Computer Society Press.

[8] R.H. Campbell, N. Islam, and P. Madany. *Choices*, frameworks, and refinement. *Computing Systems*, 5(3), Summer 1992.

[9] D. Cheriton. The V distributed system. *Communications of ACM*, 31(3):314–333, March 1988.

[10] C. Consel. Binding time analysis for higher order untyped functional languages. In *ACM Conference on Lisp and Functional Programming*, pages 264–272, 1990.

[11] C. Consel. Report on Schism'92. Research report, Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, USA, 1992.

[12] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[13] C. Consel, C. Pu, and J. Walpole. Incremental partial evaluation: The key to high performance, modularity and portability in operating systems. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, 1993. To appear.

[14] A. Dave, M. Sefika, and R.H. Campbell. Proxies, application interfaces, and distributed systems. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 212–220, Dourdan, France, September 1992.

[15] L. Georgiadis and C. Nikolaou. Adaptive scheduling algorithms that satisfy average response time objectives. Technical Report TR-RC-14851, IBM Research, August 1989.

[16] M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. A second-generation micro-kernel based unix: Lessons in performance and compatibility. In *Proceedings of the Winter Technical USENIX Conference '91*, Dallas, 1991.

[17] M. Herlihy and E.B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*, San Diego, May 1993. Full paper as DEC/CRL TR number CRL-92/07.

[18] J. Lamping, G. Kiczales, L. Rodriguez, and E. Ruf. An architecture for an open compiler. In *Proceedings of the International Workshop on New Models for Software Architecture '92*, pages 95–106, Tokyo, Japan, November 1992.

[19] P. Madany, P. Kougiouris, N. Islam, and R.H. Campbell. Practical examples of reificationand reflection in c++. In *Proceedings of the International Workshop on New Models for Software Architecture '92*, pages 76–81, Tokyo, Japan, November 1992. RISE, IPA, ACM SIGPLAN.

[20] H. Massalin. *Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Department of Computer Science, Columbia University, April 1992.

[21] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 191–201, Arizona, December 1989.

[22] H. Massalin and C. Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, Winter 1990. Special Issue on selected papers from the Workshop on Experiences in Building Distributed Systems, Florida, October 1989.

[23] H. Massalin and C. Pu. Reimplementing the Synthesis kernel. In *Proceedings of Workshop on Micro-kernels and Other Kernel Architecturs*, Seattle, April 1992. Usenix Association.

[24] Thomas Matthews. Implementation of tcp/ip for the Synthesis kernel. Master's thesis, Columbia University, Department of Computer Science, New York City, 1991.

[25] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[26] R. Pike, B. Locanthi, and J. Reiser. Hardware/software trade-offs for bitmap graphics on the blit. *Software—Practice and Experience*, 15(2):131–151, February 1985.

[27] C. Pu and H. Massalin. Quaject composition in the Synthesis kernel. In *Proceedings of International Workshop on Object Orientation in Operating Systems*, Palo Alto, October 1991. IEEE/Computer Society.

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

[29] K. K. Ramakrishnan and Raj Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Transaction on Computer Systems*, 8(2), May 1990.

[30] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the chorus distributed operating system. In *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–69, Seattle, April 1992.

[31] Y. Yokote, F. Teraoka, and M. Tokoro. A reflective architecture for an object-oriented distributed operating system. In *Proceedings of the 189 European Conference on Object-Oriented Programming*, pages 89–108, Nottingham, UK, July 1989. Cambridge Unversity Press.

[32] A. Yonezawa and B.C. Smith, editors. *Proceedings of the International Workshop on New Models for Software Architecture '92*, Tokyo, Japan, November 1992. RISE, IPA, ACM SIGPLAN.

[33] H. Zimmermann, J-S. Banino, A. Caristan, M. Guillemont, and G. Morisset. Basic concepts for the support of distributed systems: the Chorus approach. In *Proceedings of 2nd International Conference on Distributed Computing Systems*, July 1981.