Efficient Migration of Complex Off-Line Computer Vision Software to Real-Time System Implementation on Generic Computer Hardware

James Alexander Tyrrell, Justin M. LaPre, Christopher D. Carothers, Badrinath Roysam, *Member, IEEE*, and Charles V. Stewart, *Member, IEEE*

***Abstract*—This paper addresses the problem of migrating large and complex computer vision code bases that have been developed off-line, into efficient real-time implementations avoiding the need for rewriting the software, and the associated costs. Creative linking strategies based on Linux loadable kernel modules are presented to create a *simultaneous realization* of real-time and off-line frame rate computer vision systems from a single code base. In this approach, systemic predictability is achieved by inserting time-critical components of a user-level executable directly into the kernel as a *virtual device driver*. This effectively emulates a single process space model that is nonpreemptable, nonpageable, and that has direct access to a powerful set of system-level services. This overall approach is shown to provide the basis for building a predictable frame-rate vision system using commercial off-the-shelf hardware and a standard uniprocessor Linux operating system.**

Experiments on a frame-rate vision system designed for computer-assisted laser retinal surgery show that this method reduces the variance of observed per-frame central processing unit cycle counts by two orders of magnitude. The conclusion is that when predictable application algorithms are used, it is possible to efficiently migrate to a predictable frame-rate computer vision system.

***Index Terms*—Computer vision for surgery, Linux, open-source computing, ophthalmic surgery, real-time vision systems.**

I. Introduction

COMPUTER VISION algorithms have rapidly matured over the past decade, both in terms of sophistication and the range of realistic applications. We are particularly interested in algorithms for real-time frame-rate processing/analysis of image sequences (e.g., digital video) for use in guided surgical instrumentation. In these systems, a digital video camera is used to capture images of a surgical scene, at frame rates ranging from 15 to 200/s. These image sequences are analyzed in real-time to extract quantitative information that can be used to monitor the surgical procedure, perform spatial dosimetry, track structures, compensate for motion, detect hazards, and generate control signals for surgical tool guidance.

The present work is inspired by laser retinal surgery [1]–[4]. This procedure is widely used to treat the leading causes of blindness, including macular degeneration and diabetic retinopathy [5], using instruments that lack real-time guidance and control. For this and other reasons, the failure rate of these procedures is close to 50% [6]. A combination of accurate, responsive, and predictable computer vision aided guidance at frame rates can potentially improve the success rate.

Recent advances in fast and robust vision algorithms, and fast computing hardware make it possible to address the aforementioned needs [7]–[17]. However, researchers face a very practical barrier: Many vision systems are prototyped on software tools that were not designed expressly to operate in real-time implementations. This is further complicated by the fact that the software in many vision applications is unavoidably complex, relying heavily on team development, modern object-oriented programming methods, and leveraging provided by complex third-party software libraries. Code modification for the purpose of transitioning to real-time is either too expensive, error prone, impractical, or infeasible. Even if an expensive software rewrite is performed, one is faced with the problem of ensuring accuracy and consistency between separate code bases. This again is often impractical and inconsistent with modern software engineering principles. This last point is especially important when the vision algorithms themselves are in a constant state of refinement, which is often the case in a research setting. In summary, there is a compelling need to minimize (ideally, eliminate) the time and effort associated with migrating frame-rate vision systems to real-time implementations. Ideally, this migration would be simple enough to be considered "transparent." With this in mind, we propose a *rapid prototyping* solution to create a robust and predictable execution environment without the need to modify the vision code.

While a successful framework for transparently migrating off-line code to an equivalent real-time system has tremendous utility, it has been difficult prior to the advent of open source computing. Specialized operating systems (OSs)/environments were often necessary for achieving successful real-time performance. This was often made difficult by the "black box" nature of commercial or third-party OS and development environments. Each system must make certain tradeoffs between the real-time needs of a various target systems. As we have already mentioned, many vision systems contain code that was never intended to operate in real-time. Without prior knowledge, it is difficult to predict how these tradeoffs will affect

Manuscript received January 24, 2003; revised September 12, 2003. Portions of this work were supported by the National Science Foundation Experimental Partnerships under Grant EIA-0000417, by the Center for Subsurface Sensing and Imaging Systems under the Engineering Research Centers Program of the National Science Foundation (Award EEC-9986821), by the National Institutes for Health under Grant RR14038, and by Rensselaer Polytechnic Institute.

The authors are with Rensselaer Polytechnic Institute, Troy, NY 12180-3590 USA (e-mail: roysam@ecse.rpi.edu).

Digital Object Identifier 10.1109/TITB.2004.828883

a real-time system under different conditions. Environments built around an embedded model, typically characterized by lightweight code modules and a small memory footprint, are simply not appropriate for many vision systems that routinely need in excess of a gigabyte of random access memory (RAM). Complex event-driven real-time models may quickly obfuscate the basic need for highly predictable synchronous execution of a frame-rate vision system.

With the emergence of high-quality open-source community-developed OSs such as Linux, new options are available for the design and implementation of real-time vision systems. The present work is inspired and encouraged by the results of Hager and Toyama [10], Baglietto *et al.* [11], and Srinivasan *et al.* [17] using low-cost commercial off-the-shelf (COTS) computing platforms for real-time image processing applications, and builds upon our recent retinal image analysis algorithms.

The following sections describe the proposed methodology and lessons learned. Sections II-A–B describe the core methodology, in the context of the retinal application of direct interest. Section II-C summarizes previous and related work in the real-time community highlighting some of the strengths and deficiencies of various existing real-time frameworks from which motivate the present work. Section III provides an in-depth discussion on details of our proposed method.

II. Motivation and Approach

A. Motivation

At the core of a frame-rate vision system are generally three elements: a camera, a software component to perform image processing, and hardware to generate an external control signal. The interaction of these three components typically follows in a synchronous or *cyclic executive* manner that is initiated by the capture of an image by the camera and supporting hardware–firmware (e.g., frame grabber). If we turn our attention to the camera, we notice two things: 1) modern COTS video systems can deliver true hard real-time performance with minimal latency and jitter and 2) modern hardware design allowing direct memory access (DMA) and bus mastering essentially free the rest of the computer from processing overhead. Hence, it is now possible to capture frames in real-time and make them available in memory (RAM) for processing on a central processing unit (CPU) that is largely unburdened by the imaging subsystem and vice versa. We exploit these developments in order to establish efficient and predictable real-time performance. The mechanism that we propose is based on intelligent use of device drivers.

In the Linux OS, device drivers are needed as an intermediary to access a hardware device from user space. Device drivers reside in kernel space and are only accessible from a user-level process in a protected manner through the OS. In contrast, kernel-resident device drivers are free to access a number of important system-level services not directly available to a standard user-level process. This includes direct access to DMA, teletype serial interface, high-resolution timers, and access to other third-party device drivers installed on the machine. Device drivers can also share data across the user–kernel boundary via the standard ioctl() interface or direct memory mapping.

Achieving frame-to-frame predictability is a primary issue for frame-rate vision systems operating in real-time. To achieve this high degree of predictability, we must first remove all real-time threats associated with a modern multitasking OS. Fortunately, the Linux kernel provides a foundation for doing this by virtue of being nonpreemptive and not swapping kernel memory. This gives us the ability to emulate a single-process model directly in kernel by simply relinking the time-critical components of our object code into a *virtual device driver*. As the name implies, our approach is to create and use a standard Linux device driver without an associated physical device.

It is important to note that Linux does offer a similar capability by operating in single-user mode, e.g., Linux "S." Unfortunately, this execution mode is highly restrictive in terms of OS capabilities. For instance, there is no network support and certain hardware may not be accessible. What is particularly problematic from the standpoint of a vision system is that there is no graphics or graphical user interface (GUI) support in Linux S mode. This is not acceptable for clinical use where off-line monitoring in a GUI framework may need to coexist with a real-time executive. Also, interrupt handling cannot be handled effectively from outside kernel space. In contrast, our approach has the advantage of being simple while achieving good real-time predictability without restrictions on the operating mode of the host system. *In short, we achieve a real-time implementation by simply adding a new virtual device to the computer*.

As will be illustrated, the virtual device driver is basically an encapsulated single process space model installed in the kernel and invoked via a call from user space. Under this model, all real-time operations take place in the OS kernel under protection from real-time threats. Hence, instead of using asynchronous real-time processes or thread-level scheduling mechanisms that include context switches, translation-lookaside buffer misses, cache misses–flushes, and page swapping, the entire computer is viewed as a single process system tasked with the sole purpose of devoting as many CPU cycles as possible, for a specified duration, to the direct execution of our real-time code. Therefore, we propose a paradigm based on transparent migration of an off-line system to an equivalent online real-time system. *The key is leveraging the inherent real-time capabilities of a standard Linux OS obviating the need for real-time extensions or extensive code rewriting*.

B. Real-Time Retinal Image Analysis

The time-critical object code that is to be installed in kernel must be capable of executing within the time bounds of the target system's desired frame rates. In order to explore the real-time feasibility of our proposed methodology, we introduce such a system. A brief overview of this system is given here in order to establish a context for the experiments to be presented later. A more detailed description is deferred to the Appendix. Our intention below is also to convey the fact that our code base is highly complex with substantial memory and processing demands; in short, we feel it is a prototypical frame-rate vision system.

In this work, we have experimented with two computer vision applications, both related to laser retinal surgery. In these appli-

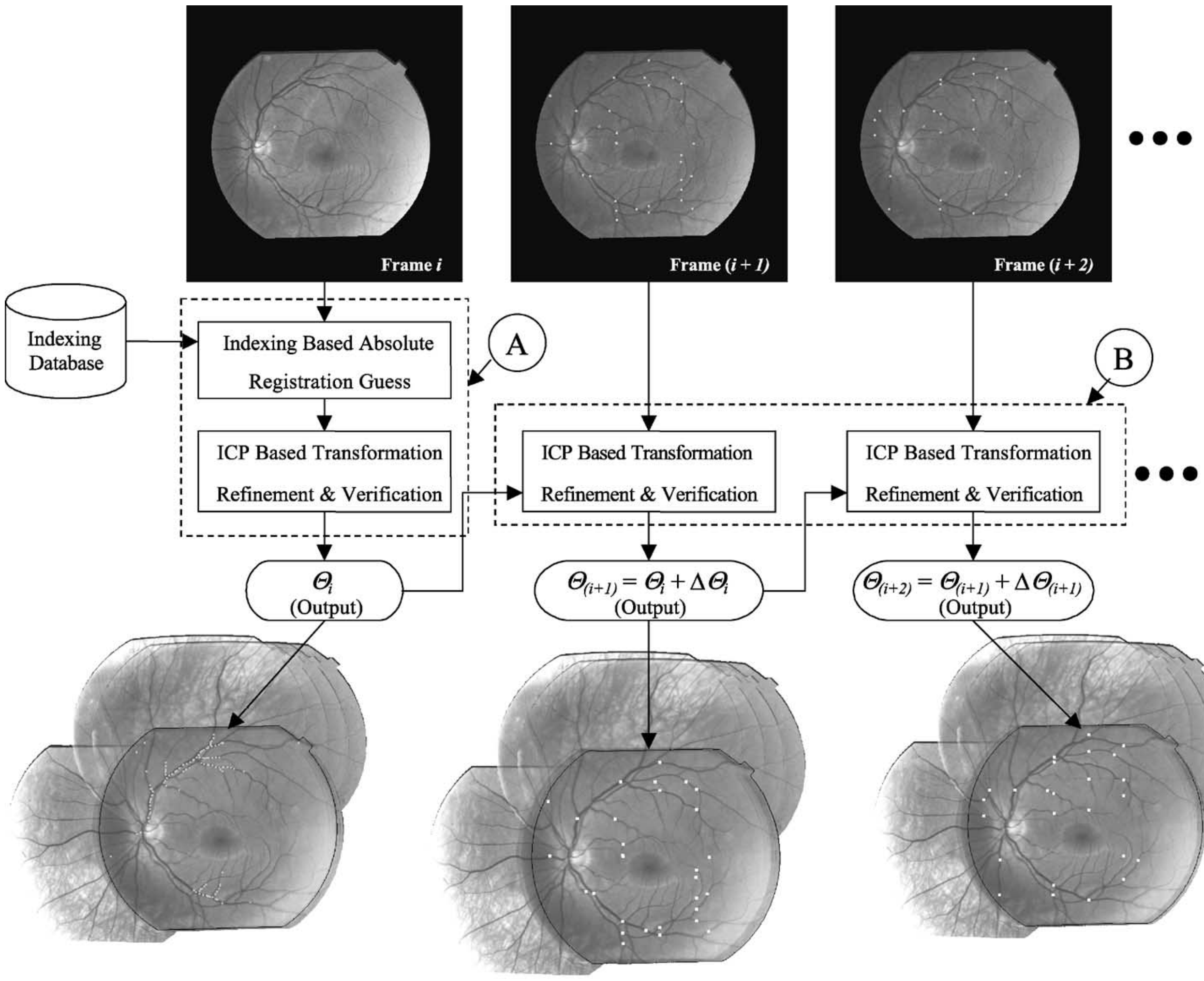

Frame i
Frame $(i+1)$
Frame $(i+2)$
Indexing Database
Indexing Based Absolute Registration Guess
A
B
ICP Based Transformation Refinement & Verification
ICP Based Transformation Refinement & Verification
ICP Based Transformation Refinement & Verification
Θ_i (Output)
$\Theta_{(i+1)} = \Theta_i + \Delta\Theta_i$ (Output)
$\Theta_{(i+2)} = \Theta_{(i+1)} + \Delta\Theta_{(i+1)}$ (Output)

Fig. 1. Illustrating the retinal spatial mapping, referencing, and tracking applications of interest. Prior to surgery, a spatial map of the retina is constructed. It consists of mosaics (lower row), an indexing database, and surgical plans. Spatial referencing (Box A) is the technique of registering each on-line image frame captured by a camera onto the spatial map. Correct registration is indicated in this figure by overlaying vascular features detected in the image frame over the corresponding mosaic illustration in the lower row. Tracking (Box B) is the technique of rapidly registering an image frame to the previous frame assuming small motions. For simplicity, this figure omits mechanisms for handling various types of errors and exceptions.

cations, it is of interest to locate specific points on the retina with the highest achievable accuracy and reliability for the purpose of guiding a surgical laser, monitoring the surgery, detection of errors, and performing beam shutoffs whenever appropriate. The retinal images on the upper row in Fig. 1 were captured using a digital mega-pixel video camera (1024×1024 12-bit pixels) mounted on a standard clinical instrument known as a fundus camera or biomicroscope [18]. This represents a partial angle (30°–60°) flat projectional view of the complete curved retina. The branched structure in this image is the retinal vasculature. The vasculature is usually stable and can be used as a source of spatial landmarks. These landmarks (features) are used to generate a spatial mosaic and map for the entire retina from a series of preoperative diagnostic images.

The first application of interest, termed "spatial referencing," is a method for absolute registration of an image frame captured by a digital camera to a preconstructed spatial map of the human retina. This method is summarized in Appendix A. The second application, termed "robust tracking," enables rapid registration of successive image frames within a predictable computational budget. This latter algorithm is summarized in Appendix B. Fig. 1 illustrates the roles of these two algorithms. In this illustration, three successive image frames from the digital camera are presented in the uppermost row. The first frame is registered using the absolute spatial referencing algorithm. Since the second and third frames undergo small motions relative to the first, they are registered using the robust tracking algorithm presented in Appendix B. The spatial referencing algorithm is highly complex and unpredictable due to the opportunistic manner in which it reduces pixel processing. Therefore, it is only invoked when needed. For instance, when the first frame of a sequence is obtained following a fadeout, or after a large motion or a registration failure. In contrast, the robust tracking algorithm is very efficient and predictable. In short, these algorithms and data structures are capable of rapidly estimating the position of the surgical laser anywhere on the retina for each image frame.

Several factors contribute to the size and complexity of these applications. In order to perform highly precise registration, we use a 12-parameter quadratic transformation model to map

TABLE I
SOURCE CODE PROFILE SHOWING RELATIVE SIZE OF MAJOR SOFTWARE COMPONENTS THAT WE LINK INTO A KERNEL MODULE.

Library	Static libraries	Object modules	Static Size (MB)
VXL	17	1298	27
RETL	8	160	15
PUBL	9	67	3

The VXL library is a third party standard C++ library. The RETL library is the Rensselaer tracing library, and PUBL is a public RPI vision library. In addition to a static code size of ~45 MB, we add a 300-MB data segment to the final device driver in the form of a static buffer. We have experienced little difficulty loading our modules on a system with 1 GB of RAM.

image coordinates to global coordinates in the preoperative retinal mosaic [7]. The use of a quadratic transform is necessary to mitigate the projective distortions resulting from the retinal curvature combined with a weak perspective camera. This transform is estimated by a robust M-estimator [19] over a set of closest point correspondences between an image and the mosaic. The estimate is found by employing a procedure called iteratively reweighted least squares (IRLS) [20]. In order to achieve fast computation in the face of large data volume, the spatial referencing method relies on extensive precomputed data structures that trade storage in favor of speed. All of these algorithms are complex in their implementation, utilizing object-oriented team programming effort, third-party libraries, and have substantial static and run-time memory requirements. Table I profiles the static size of the object code for the spatial referencing software system. In addition to a static code size of ~45 MB, this code typically requires roughly 300 MB of dynamically allocated memory at run time.

C. Real-Time Computing Background

From a computational standpoint, the combination of techniques presented in the previous section enables spatial referencing at extremely high speeds, approaching frame rates notwithstanding the high data rates and complexity. This forms the necessary but insufficient basis for building a real-time spatial referencing system for ophthalmic applications. Still needed is a real-time OS (RTOS) that combines high throughput and low latency responsiveness to provide a predictable environment for meeting real-time deadlines.

Choosing an appropriate RTOS requires understanding the characteristics of the target real-time application. Real-time applications are generally characterized as being *hard* or *soft* as described by their relative time sensitivity to a real-time *deadline*. A hard real-time application becomes invalid when a deadline is not met. By contrast, soft real-time applications can tolerate more latency and the deadline constraint is less critical. This work focuses on *hard* real-time frame-rate vision systems.

The scheduling demands of a real-time application are an important factor when classifying the nature of a real-time application. One of the simplest scheduling models is *cyclic executive* or *frame based* execution [21], [22]. Applications of this type are characterized as being synchronous, often based on periodic execution of logically sequential tasks. This type of real-time system requires a trivial scheduling mechanism and is unlikely to benefit from complex parallel hardware configurations or multithreading. Applications that require truly preemptive process/thread based scheduling to respond to asynchronous inputs are defined as being *event driven* [21], [22]. Real-time frame-rate vision systems are naturally characterized as being cyclic executive. Hence, this is the target model for the proposed methodology.

D. Previous Work

Maeda [23] demonstrated the efficiency gains associated with executing type-safe user-level programs directly in the kernel space of a standard Linux OS. The idea is to eliminate the overhead associated with a transition across the protection boundary separating the user and kernel process space. This is an interesting approach that is based on type-safe-assembly language extensions to user-level object code [24]. These extensions are designed to protect the integrity of the OS in the presence of unstable or nefarious programs while at the same time greatly reducing systemic overhead in applications that must frequently access low-level services. Importantly, this approach is consistent with our previously stated goals of maintaining a common development and real-time testing environment on a single platform utilizing a common code base. Unfortunately, the language extensions used to make the user-level code “type safe” include array bounds and other memory checking operations that introduce overhead that is unacceptable for our purposes. In reality, this type of methodology is probably best suited for *soft real-time* applications such as multimedia and communications systems that require frequent access to specialized system-level services.

A similar approach to the one proposed here was developed by Srinivasan *et al.* [17]. Their approach uses COTS hardware components and a standard Linux OS to create a *firm* real-time execution environment. The notion of a firm real-time environment applies to time-critical real-time components that must unavoidably rely on nondeterministic OS services typically found on a timesharing OS. Again, this approach is better suited for multimedia and communication applications and is not a truly hard real-time method. An interesting aspect of this work however, is the introduction of a real-time priority-based scheduling mechanism into a standard Linux OS. This is a common approach for achieving true *hard* real-time performance from a standard Linux OS.

RT-Linux [25] and TimeSys Linux [26] are proprietary Linux variants that offer an abstract view of the Linux kernel that can be configured dynamically to create a real-time framework without compromising the integrity of the standard Linux OS. In short, these two systems promise to offer true hard real-time performance without complex specialization of the existing Linux kernel. This is a powerful extension to what is already a well-suited OS for real-time development.

Both of these Linux RTOS variants are very appealing from the standpoint of transparent migration. The real problem is that they introduce a layer of abstraction into a standard Linux OS that allows for micro-controlled scheduling of events. This is necessary for event driven applications but has no real use in our application. Even more problematic is that the kernel must be made preemptable to handle asynchronous behavior. This forces kernel modules to be reentrant which raises serious compatibility issues for existing hardware device drivers. Of particular note is the fact that RTLinux is built around the use of kernel modules implying that the steps that are described in this paper have essentially already been taken. Unfortunately, it then imposes a mutually exclusive existence constraint between real-time executives and a standard Linux OS. This is much more extreme than our method of simply using standard Linux artifacts to achieve a real-time implementation.

III. Transparent Migration Methodology

A. Overview of the Method

The main components of the proposed methodology are listed below:

1) encapsulate the application algorithms into a loadable kernel module (LKM);
2) design a virtual device driver that emulates a single process model in kernel;
3) register device driver with Linux OS enabling user-level access.

The key to our approach for ensuring systemic predictability is embedding the image processing system into the Linux kernel. In this environment, it is possible to eliminate OS-level scheduling and interrupt overheads as well as mitigating the uncertainties introduced by a shared memory environment. The use of a device driver is a natural approach to achieving the necessary real-time performance while still allowing user- and kernel-level interaction. Under this model, all time-critical processing is deferred to a kernel-level process accessed directly from user space as a device driver. When the time-critical processing is completed, the system returns to user space. From the standpoint of a frame-rate vision system, this implies that our device driver must directly interact with any necessary hardware components, such as video frame grabbers, from within the context of a kernel process. This is essential because real-time execution cannot be guaranteed if any processing is to be done outside of kernel mode. Fortunately, as noted earlier, Linux device drivers are intended to work in this manner.

B. LKMs

In this section, we describe LKMs and how they are used to create a virtual device driver that emulates a single process model in kernel space. Typical examples of LKMs are device drivers but can include any functionality that might need to be shared by multiple processes. The idea of an LKM is to dynamically add executable object code directly to the Linux kernel while the system is running. LKMs are essentially no different than relocatable object modules created by a standard compiler like GNU's C compiler (GCC). They can be written in C or C++ and there are in fact surprisingly few restrictions on the nature and size of the code. We routinely create LKMs in excess of 300 MB (refer to Table I for an overview of our code base size). The main difference is that LKMs must include two functions $\mathrm{initModule}()$ and $\mathrm{cleanupModule}()$. Each function is guaranteed to be called exactly once.

The function $\mathrm{initModule}()$ serves as the entry point into the module and is called when a kernel module is loaded via the Linux insmod utility [27]. The insmod utility is used to add modules to the kernel while the system is running. After loading a module using insmod, functions and data in that module become part of the Linux kernel space. The Linux kernel is monolithic in the sense that all modules (including the kernel itself) share a single kernel address space. This means that functions and data in one module are accessible from another. In addition, kernel functions can be invoked from a user-level process. These features of the Linux kernel model are important for the development of our real-time prototype.

C. Kernel Module Insertion

The process of taking large-scale user-level software and realizing it as a kernel module is relatively straightforward provided one adheres to some constraints. Aside from the hazards resulting from careless use, a potential problem is that the Linux kernel is a restricted process space and does not provide much of the functionality that user-level processes expect in order to execute.

Specifically, there are four key areas where user- and kernel-level processes differ:

1) dynamic memory allocation;
2) device input–output (I/O);
3) global variables;
4) stack management/usage.

The first three differences can all be handled using the linker operating directly on the object code. The last issue is more restrictive and in some cases can only be reconciled if certain conditions are already met by the existing object code.

Under Linux, kernel modules do not have a module-specific heap and stack segment. This implies that kernel modules cannot allocate dynamic memory the way a user-level process can. However, Linux does provide two specific kernel variants of the memory allocation system called $\mathrm{malloc}()$. The first is called $\mathrm{kmalloc}()$, which allocates physically contiguous blocks. This is ideal for our purposes but it becomes unreliable as memory gets fragmented. The other is called $\mathrm{vmalloc}()$, which allocates memory from the kernel's virtual map. In fact, this is the function used by insmod to load a kernel when it cannot be placed in physically contiguous real memory. These allocation functions have two problems: 1) they both allocate memory blocks whose sizes are in powers of two and 2) handling memory allocation requests during real-time execution is a potential source of uncertainty.

To overcome these limitations, we developed a novel kernel memory allocation function. As mentioned, the goal is to emulate a single process space directly in kernel. Hence, we introduce our own version of $\mathrm{malloc}()$ and $\mathrm{free}()$ that operate on a static buffer linked directly to the object code. In other words,

```
/* add these two declarations */
unsigned char g_static_buffer[MEM_SIZE];
int giOffset;

__ptr_t __default_morecore (ptrdiff_t increment)
{
  /* replaces call to sbrk */
  __ptr_t result = &g_static_buffer[giOffset];
  giOffset += increment;
  if (result == (__ptr_t) -1)
    return NULL;
  return result;
}
```

Fig. 2. Illustrating the technique for modifying the standard UNIX memory allocation system call "malloc." Note the g_static_buffer array which serves as our virtual data segment in kernel. Function fulfills virtual memory requests by simply returning a pointer from this array.

we have circumvented the fact that kernel modules do not have a heap segment by simply inserting a new allocation routine. The design of a special purpose malloc routine is described below, and key programming lines are provided in Fig. 2.

Starting with the standard GNU malloc routine that is available in open-source form [28], we locate the function morecore.c which contains a call to the function sbrk(). The sbrk() function, pronounced "S-Break," is used to dynamically reallocate the data segment of the calling process. Specifically it increments (or decrements) the *break address*, i.e., the address of the first location beyond the end of a process's data segment. The key artifice is to replace sbrk() with a simple pointer increment in our static buffer.

Next, we recompile the GNU-malloc source files and link them into a single new relocatable object module called "new_malloc.o" in the examples below. Having reimplemented malloc, we need to replace all instances of the standard version in the code. This can be done quite easily in one step during the linking phase of compilation using the wrap functionality provided by the linker. The idea is to substitute each reference to a chosen procedure in an object module's symbol table with a reference to a new procedure. Hence, we "wrap" the old procedure with the new one, using a UNIX command of the form

$$\mathrm{ld}\ -\mathrm{wrap\ malloc\ module.o\ new\_malloc.o}.$$

The end result is that we have created an emulated heap segment for our real-time kernel modules. From the standpoint of the Linux virtual memory system, this heap segment exists as a contiguous memory zone that can only be used by our real-time module.

The function wrapping technique used to substitute the kernel malloc routine can be used to resolve certain device I/O problems as well. Some simple I/O requests are trivially handled. For instance, calls to printf() or the C++ operator cout can be wrapped using the kernel variant printk(). Calls to printk() are mapped to a virtual device, usually /var/log/messages on Linux systems. They can also be effectively disabled using a null file.

Some other device I/O requests may present more of a problem but can generally be handled by choosing appropriate functions found in /kernel/ksyms.c and then performing the same function wrapping technique previously described. Fortunately, it is a reasonable assumption that the real-time vision and control system does not perform any device I/O directly. Rather, such systems generally work in conjunction with specialized hardware/software supported by a suitable device driver. A common example is a digital image frame grabber device.

The use of global variables may present problems when creating LKMs. Note that all kernel modules share the same address space. Under this model, global variables in an object module become global to the entire kernel address space meaning global variable names must be unique. This is generally not a major problem and can be resolved using name spaces. However, a more subtle issue is relevant. When a user-level process is created, the OS will invoke the constructors for each global and static object before the main() function is called. In kernel space, the constructors are not called. Fortunately, this problem can be solved using the linker by adding the following link line command and calls as described below:

$$\begin{aligned}&\mathrm{ld}\ -\mathrm{Ur}\ -\mathrm{static}\ -\mathrm{o\ module.ogcc}-\mathrm{lib}\\ &\qquad -\mathrm{path/crtbegin.o[files]\ gcc}-\mathrm{lib}-\mathrm{path/crtend.o}.\end{aligned}$$

This link line explicitly adds the GCC files (known as "stubs") crtbegin.o and crtend.o that are used to call constructors and destructors in a normal executable. The last step is to add the call _do_global_ctors_aux() in the initModule() function and the call _do_global_dtors_aux() to the cleanupModule() function.

A practical issue relates to stack usage in Linux kernel space. In Linux, 8 KB (two memory pages on an Intel IA32 architecture) of stack space are allocated for each kernel process. This space must be shared with the process control block [(PCB) i.e., struct task] which begins at the last ~700 B of the 8-KB address space. Thus, in total, the kernel has ~7 KB of usable stack space for each process.

This arrangement presents the potential for the process's kernel stack to grow directly down into the PCB, which would corrupt the process state and potentially other kernel data structures. Ultimately, stack growth beyond this ~7-KB limit will potentially cause the system to become unstable. This is a serious problem, especially since this is a run-time issue and it can be difficult to predict stack usage *a priori*. This problem can be avoided by the use of straightforward programming guidelines. Specifically, the programmer should avoid allocating large objects on the stack either as local variables or as function parameters. The use of recursion must also be handled carefully. In applications where this constraint is too restrictive, Linux patches are available for reconfiguring the kernel stack size. Of course we have made every effort to avoid operating outside the system parameters established in a standard Linux OS and we have experienced little trouble in our work to date.

D. System Call Interface

Using the preceding techniques, we are able to relink existing object code into a relocatable kernel object module that can effectively emulate a single process space model in kernel. In order to use the kernel module to perform time critical operations from user space, we need to link the module as a device driver. As mentioned, all kernel modules must include the functions initModule() and cleanupModule(). To qualify as a character device driver, the module must register with the OS via a call to register_chrdev() from within initModule() and also unregister_chrdev() in cleanupModule(). Next, the module must implement a set of functions in the file_operations structure (include/linux/fs.h) that is sent as an argument to register_chrdev(). Typical file operations include open, ioctl, read, and write. It is from within these functions that we call our computer vision code to invoke real-time processing. After loading with insmod, an entry for the module can be found in /proc/devices. This entry contains the name of the module and major version number. A simple call to mknod in the /dev directory providing the name and major version number will create the device entry. After following this procedure, the device driver can be accessed by simply opening the device by name from a user-level process and calling a set of functions that effectively wrap the core time-critical functionality.

E. Handling Interrupts

Using the inherent properties of Linux discussed above, we can use a virtual device driver to provide predictable real-time performance without modifying any of the application code. At this point, we have described everything needed to establish rapid prototyping of a real-time system. However, we cannot establish true hard real-time without addressing the issue of interrupts.

While it is true that a user- or kernel-level process cannot preempt our virtual device driver, hardware interrupts can and generally should cause preemption. Of course this is certainly a low threat priority as long as the host system is properly configured. Nonetheless, interrupt handling can still affect predictability. The simplest solution is just to call the functions cli() and sti() (clear and set interrupts, respectively) upon entry and exit from kernel space. This turns off all interrupts making a process sole owner of the CPU unless a hardware failure or other exception is generated (e.g., segmentation fault, divide by zero). This is an easy solution and most device drivers written to handle an interrupt request perform this operation at some point. There is, of course, the potential to render essential hardware like a frame grabber inoperable. In this case, the solution is to selectively mask out all nonessential interrupts. In our work, we have generally relied on the cli()/sti() method but masking is a common and well supported operation designed to be performed from kernel space. For example, one might require temporary masking of the mouse and keyboard if running in an X-Windows environment.

IV. Experimental Results

Using the virtual device driver configured as described above, we can establish hard real-time predictability by protecting the time-critical components of the system from the threats associated with a shared memory process space. To explore the effectiveness of the proposed methodology, we will compare the predictability of various kernel- and user-level configurations of a frame-rate vision system. In all our experiments, we used an Intel IA32 desktop computer with a 1-GHz processor and 1 GB of RAM running X-Windows under Red Hat Linux 7.1 kernel version 2.4.18.

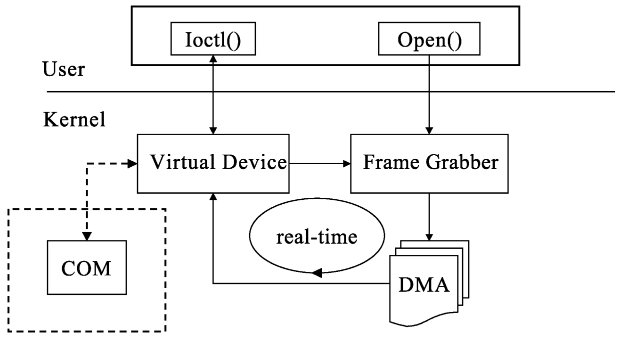


Fig. 3. System design for real-time tracking. Initially, the frame grabber is opened from user space placing the device into the Linux IRQ chain. Henceforth, the user process interacts directly with the virtual device driver that serves as a proxy for the frame grabber device driver. The circular arrow indicates the cyclic real-time executive that is protected in kernel space. In this conceptually simple arrangement, the virtual device driver simply calls the frame grabber's read() method directly. Also indicated is an interface to a serial port (COM) that can be used to control digital camera settings and other peripheral devices.

A typical application is to use a single call from user space into our virtual device driver to invoke a cyclic executive loop that performs real-time frame-rate processing. As an example, we have configured a device driver to perform tracking in kernel space while the landmark/constellation registration is performed in user space (Fig. 3). Once a frame is registered in user space, the transform is sent to the kernel at which point our device driver begins signaling the frame grabber to acquire the next image. This process continues until the tracking mechanism fails to register an image after which point control is given back to the invoking user process. After returning to user space, any relevant information can be retrieved via the ioctl interface. The resulting control sequence typically consists of a binary signal that must be set or reset by the real-time application every 33 ms. Failure to complete this computation, as detected by a process or hardware subsystem that is external to the application, represents a missed deadline.

In the first experiment described above, our sole interest is in establishing *systemic real-time predictability*. Hence, we will operate our frame-rate vision system multiple times on a single image effectively eliminating any timing variation due to the algorithm itself. In order to get results under realistic conditions, it is necessary to simulate actual run-time conditions. Therefore, we use a modified version of a prototype system to perform our tests. Currently, we have a prototype vision system that uses a Dalsa 1M30A digital video camera to capture 1024×1024 12-bit grayscale images at $\sim$30 frames/s. Using an ITI PC-Dig frame grabber, we memory map the captured images to RAM. These images are in raw format and must be converted via an explicit copy operation to a double buffer. Using this prototype system, we create a testing environment by simply suppressing the switching of buffers and placing a single resident image in one of the buffers. We operate on that image continuously and simply allow the camera to capture blank images.

In order to obtain timing results, we create two versions of the vision system, referred to as user mode and kernel mode, respectively. Each mode shares a common user-level executable known as the driver. This *driver program* calls the first computer vision application of interest to us—landmark/constella-

TABLE II
SUMMARY OF TIMING RESULTS EXPRESSED IN RAW CPU CYCLES ON A 1-GHz PROCESSOR. THE GOAL OF THIS EXPERIMENT IS TO DETERMINE THE SYSTEMIC PREDICTABILITY OF EACH CONFIGURATION BY PROCESSING A SINGLE IMAGE MULTIPLE TIMES. OBSERVE THE ROUGHLY TWO ORDERS OF MAGNITUDE REDUCTION IN THE STANDARD DEVIATION AND IN THE DIFFERENCE BETWEEN THE HIGHEST AND LOWEST READINGS (LAST COLUMN)

Execution Mode	Mean (Cycles)	St. Dev. (Cycles)	Maximum (Cycles)	Minimum (Cycles)	Max-Min (Cycles)
User mode	93,467,136	36,243,542	119,095,190	67,839,081	51,256,109
Kernel mode	70,042,109	418,591	70,338,097	69,746,120	591,977

TABLE III
SUMMARY OF TIMING RESULTS EXPRESSED IN RAW CPU CYCLES ON A 1-GHz PROCESSOR. IN THIS EXPERIMENT, WE ARE INTERESTED IN BOTH ALGORITHMIC AND SYSTEMIC PREDICTABILITY WHILE TRACKING A 500- FRAME IMAGE SEQUENCE. CLEARLY, THE KERNEL MODE IS MUCH MORE PREDICTABLE COMPARED TO THE USER MODE

Execution Mode	Mean (Cycles)	St. Dev. (Cycles)	Maximum (Cycles)	Minimum (Cycles)	Max-Min (Cycles)
User mode	3,511,707	17,354,746	331,489,486	1,500,398	329,989,088
Kernel mode	1,661,411	66,081	2,135,198	1,494,191	641,007

tion based spatial referencing (detailed in Appendix A). For user mode, we link spatial referencing directly into the final executable. To create the kernel mode, we modify the driver program to call a virtual device driver that encapsulates the user-level code.

Timing results using this testing configuration are summarized in Table II. The results show definitively that spatial referencing run in kernel mode displays much greater systemic predictability than the standard user-level configuration. It is interesting to note that the user-level configuration can potentially be slightly faster. This is likely explained by the differences in the way caching is handled in the two modes. What is perhaps more important is the existence of significant outliers in the user-level timings. These outliers are simply unacceptable in a real-time system and are likely caused by the memory management system itself. Initially, the heap segment of the user process must be increased several times to handle memory requests, which leads to appreciable processing delays.

Having demonstrated systemic real-time predictability on a single image frame under realistic run-time conditions, the next step is to run the system on an actual retinal image feed in real-time. This way we can measure the overall predictability of the system including both systemic and algorithmic predictability. In this second experiment, we will use the second application of interest—a 12-parameter tracking algorithm described in Appendix B. The spatial referencing algorithm described in Appendix A initializes this algorithm. The test data consists of a preloaded sequence of 500 retinal images captured at a resolution of 512×512 pixels, 30 frames/s, and 12 bits per pixel. This reduced image size is obtained by 2×2 binning of the charged-coupled device array in the Dalsa camera and is done to improve the signal-to-noise ratio. Again, we create a single user-level program to drive both a "kernel space" variant and a "user space" variant of the proposed system.

In the sequence of 500 images, we successfully tracked 436 frames by registering them onto a preoperative retinal mosaic. This is done with an average time of about 1.7 ms. (It is important to note that these times do not include seed point detection because in an actual system this will be done using a field programmable gate-array at the frame grabber). The ability to achieve such low processing time is essential. Since our camera runs at 30 frames/s, we have a maximum of 33 ms of processing time before the next frame. However, because the eye is in constant motion during frame integration, each successive image represents information that is potentially 33 ms old before processing even begins. From a control standpoint, this creates an inescapable risk that must be mitigated by minimizing the latency of our system.

The timing results in Table III show definitive improvement in predictability when using a kernel configuration. Again we note the extreme outliers in user mode that generally occur the first few times the algorithm is run. As before, we attribute this to the inordinate number of $\mathrm{sbrk}()$ calls made as the dynamic memory demands of the user-level process grow rapidly as processing begins. This is further exemplified in Fig. 4, where we ran the same experiment from the console without any system load or the overhead introduced by X-Windows. Recall that under our kernel implementation, the $\mathrm{sbrk}()$ call is emulated by a static buffer that is linked directly to the object module. The paging overhead incurred in user space as memory is allocated in noncontiguous blocks from the pageable memory pool contributes significantly to the observed variability in execution times. Admittedly, this effect may have more to do with the test than the system, i.e., we are potentially observing an uncertainty principle. Simply loading the 500 image frames into RAM significantly disrupts the memory access patterns of a program in user space. However, this example actually illustrates how nonpageable kernel memory allocation can significantly improve systemic predictability.

Since kernel modules have no stack or heap segment, they can be placed contiguously without generating memory fragments larger than a single page. Unfortunately kernel modules cannot always be loaded in this manner. However, from the standpoint of the kernel's virtual map, memory is contiguous and cannot be swapped. This is clearly an effective strategy for reducing real-time threats from paging that is evident empirically in our results. Although an improvement, this emulation of real memory presents a problem when using memory mapping or DMA operations that require real contiguous memory. In the Linux OS, as with most systems, the first 16 MB of real memory is reserved for DMA and it is reasonable to assume that most of this memory is available for our real-time needs.

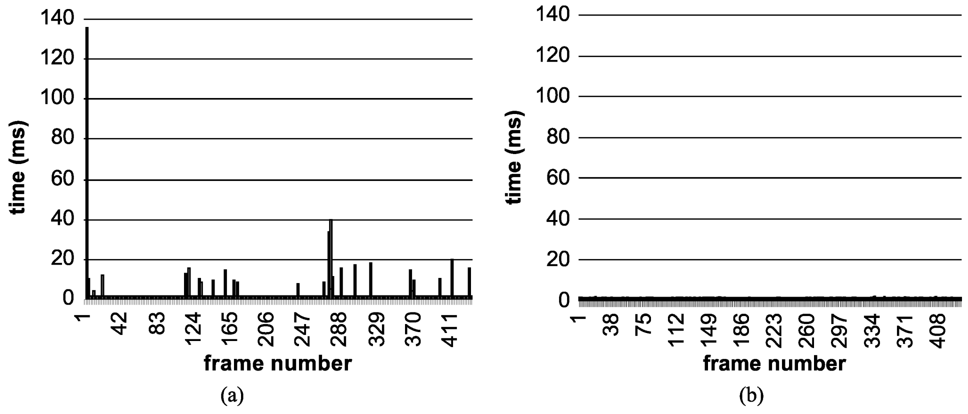


Fig. 4. Timing sequence for a 500-frame tracking experiment run from the console without a system load. User and kernel space timings are given in (a) and (b), respectively. Height of each bar indicates time in milliseconds required to successfully track each image frame. Note the high degree of variability throughout the timing sequence while in user mode. Contrast this with the flat profile while in kernel mode, indicating a dramatic improvement in predictability.

V. Conclusions and Discussion

We have described a framework for establishing predictable frame-rate computing on a standard Linux OS using COTS hardware. We make use of a virtual device driver to emulate a single process space directly in kernel without modifying the application code. Our approach has the advantage of allowing rapid prototyping on a native system that allows full duplexing between kernel- and user-level code. The proposed methodology is simple, subject only to a mild set of constraints without requiring any nonstandard kernel modifications.

Establishing real-time operation in this manner promises to be both efficient and cost effective but also formulates an entirely new approach to real-time development of complex frame-rate vision systems. This approach places the focus on off-line algorithm development to achieve robust and efficient solutions to a particular vision problem that when coupled with our real-time execution environment results in the immediate realization of a predictable hard real-time application. Algorithm development is no longer bound by highly restrictive low-level implementations and can instead readily incorporate any software components that are known to be efficient and predictable. Shifting the focus to off-line algorithm development rather than a specialized real-time implementation in order to achieve efficiency and predictability is a significant departure from standard real-time design methodologies. The fact that this can be done using COTS components without the need for additional proprietary or specialized hardware/software further represents a substantially different approach to real-time development. Since the demonstrated effectiveness of this approach may in fact be unique to complex frame-rate vision systems, our results are even more important to this application domain.

The key aspect of frame-rate vision systems is that they do not generally require sophisticated event handling and asynchronous processing. This allows us to use very basic aspects of a standard Linux OS to establish predictable synchronous cyclic execution in real-time. From the standpoint of a general real-time solution, what is lacking is a sophisticated scheduling mechanism. With such a mechanism, it may be possible to introduce parallelism to an already existing off-line code base. The key is to carefully expose the off-line components that need to be run in parallel. Since we may not in general assume the off-line code is thread safe, this poses a number of potential problems. However, the techniques described in this work including function wrapping, interrupt masking, and the nonpreemptive nature of the Linux kernel, may greatly reduce the effort needed to introduce parallelism to an existing off-line code base. Our work using Linux kernel modules could be a key stepping stone toward such a design.

Appendix A

This appendix briefly summarizes the first major computer vision application of interest—spatial referencing [Fig. 1(Box A)]. In this application, each image frame from the camera is aligned to a preoperative mosaic map of the retina. The retinal vessels are the features used for registration that must be extracted at sufficient speed to permit frame-rate registration, and with sufficient adaptability to cope with illumination and patient variations. This is done using algorithms for fast exploratory vessel tracing [29]–[33] and real-time spatial prioritization [9].

These algorithms proceed in three stages. First, sparse vessel detection is performed over a sparse grid [Fig. 5(b)]. This step also estimates the local image statistics and noise levels. These positions, known as seeds, are refined and verified by testing for the existence of a pair of sufficiently strong two-dimensional antiparallel (opposite direction) edges [13] in a small region around each seed. Prioritizing the grid analysis by analyzing the angular patterns of seed points allows us to generate an early yield of landmarks and landmark constellations [9]. The second stage performs iterative tracing of the vasculature starting from seed points [Fig. 5(b)] and detection of points where traces meet or cross. The final stage refines these landmark points to subpixel accuracy using the algorithm of Tsai *et al.* [33].

The vessel traces and landmarks permit precise registration using algorithms that account for the unknown retinal curvature and the weakly perspective imaging geometry using a 12-parameter imaging model and robust hierarchical estimation procedures [7]. These pair-wise registrations are further processed to perform a joint registration of a set of (12–15) images to construct mosaic families with subpixel accuracy [7]. While these mosaics are independently useful as extended visualization tools, they are even more useful as a basis for *spatial mapping* of the retina. The retinal map is precomputed

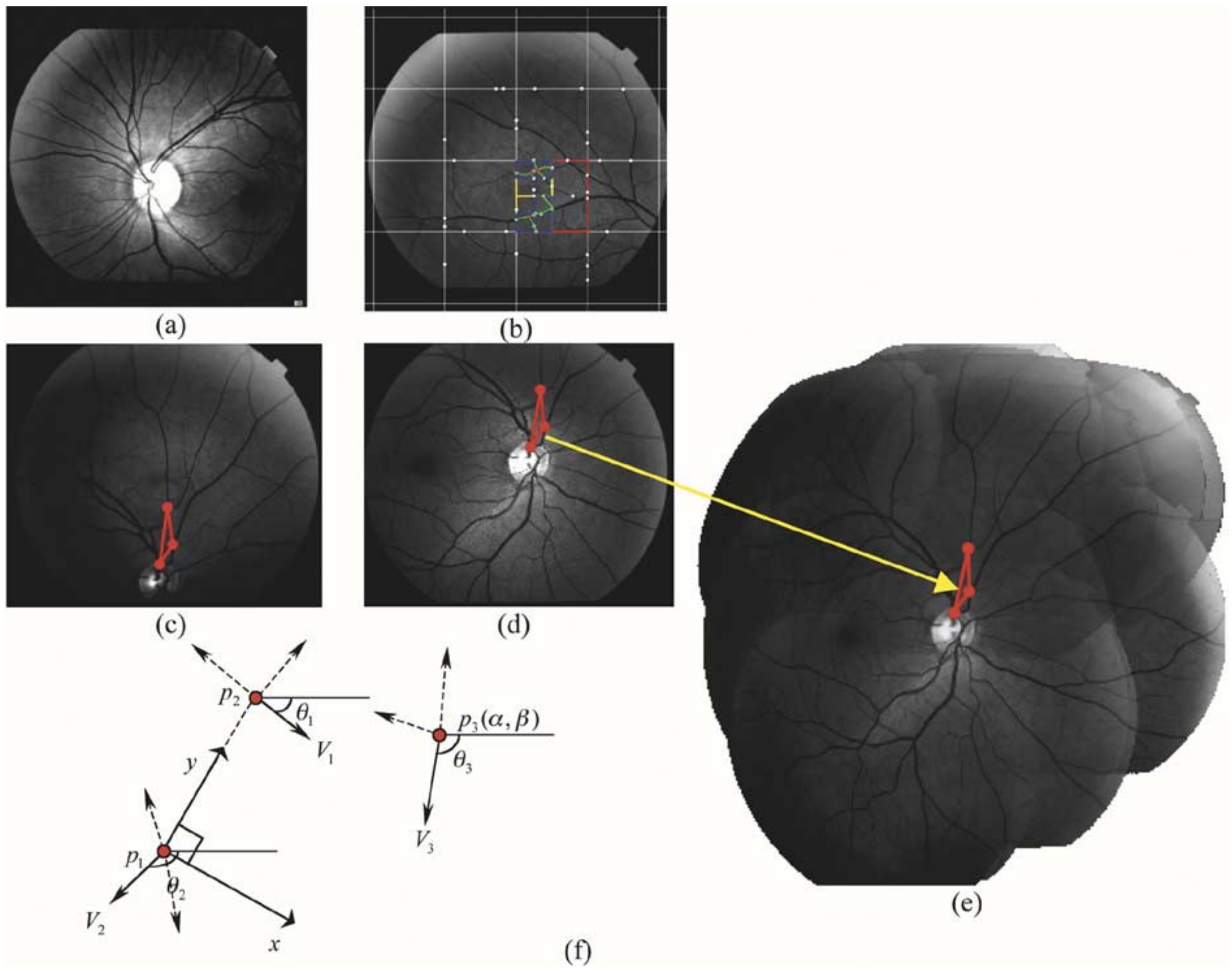


Fig. 5. Illustrating the quasi-invariant indexing-based approach to fast spatial referencing: (a) sample digital retinal image; (b) results of opportunistic extraction of a landmark constellation; (c–e) illustrating the same landmark constellation in two image frames and the mosaic, respectively; (f) illustrates for a constellation of three landmarks, the invariant feature vector (QIFV) consisting of five components (α, β, θ_1, θ_2, θ_3). This vector can be looked up rapidly in a precomputed hierarchical $k-d$ tree database of QIFVs.

and stored prior to laser retinal surgery, and is an enabling data structure for real-time spatial referencing.

Spatial referencing is the problem of registering a single observed retinal image frame to the precomputed map during laser surgery, as illustrated in Fig. 1(Box A). Such registration avoids the drift and undetected failure problems encountered in conventional frame-to-frame tracking methods [14]–[16]. In essence, spatial referencing is an image registration problem, but with extreme speed and accuracy requirements. This group has recently published methods [8], [9] to meet these extreme requirements using a combination of extensive precomputation and the use of quasi-invariant feature vectors (QIFV) [34]–[36]. Pairs and triples of landmarks that are reasonably close to each other (within about 20% of the image width) are formed into "constellations." A vector of similarity quasi-invariants—geometric measurements that are approximately invariant under scale, rotation, and translation—is computed from the constellation [Fig. 5(c)–(e)]. QIFVs computed from all constellations in all diagnostic images are stored in a hierarchical database built from $k-d$ trees [34] for fast lookup during the on-line phase. To summarize, the complete spatial map consists of a set of images, their features (traces, landmarks), the mosaic, Euclidean distance maps [29] of the traces, a set of pair-wise 12-parameter quadratic spatial transformations linking the images, and the $k-d$ tree indexing database.

The QIFV driven database lookup generates several hypotheses representing landmark correspondences with the spatial map. These hypotheses must be verified by computing a robust measure of alignment of the vascular traces between the real-time image frame and the stored map. This ordinarily complex operation can be performed surprisingly fast by subsampling the vasculature and using a precomputed digital distance map of the traces. Verified correspondence hypotheses produce crude four-parameter similarity transformation estimates. They are refined in a series of steps that ultimately lead to an image-wide 12-parameter transformation. If the estimated alignment is not sufficiently accurate, the hypothesis is rejected and another one is considered. Typically, just two to five such trials are sufficient.

Appendix B

This section describes the second application algorithm of interest—robust retinal tracking, indicated in Fig. 1 (Box B). Although the spatial referencing application described in Appendix A is extremely powerful and general, it has the disadvantage of having computation requirements that are not precisely predictable. The opportunistic nature of the algorithm means that the number of hypotheses under consideration cannot be predicted *a priori*. The algorithm described here offers the attractive combination of precise computational determinism, ability to verify results against the spatial map described above, and spatial alignment that is robust and precise for the most common case, namely small image motions. When this algorithm fails, the full spatial referencing algorithm is invoked as in [31], or the frame is rejected with the surgical laser disabled.

In contrast to conventional feature-based tracking algorithms [14]–[16] that attempt to correlate specific image features between frames, the present algorithm exploits the robust map-based verification and refinement step of the spatial referencing algorithm described in Appendix A. This approach provides much-needed robustness and verifiability, as well as freedom from the projective distortions caused by excessive drift.

The core of the spatial referencing algorithm is the iterative-closest-point (ICP) algorithm [30]—that searches for a transformation and a set of correspondences that minimize the alignment error between a set of reference points between two images. Given two sets of feature-point vectors ($P = \{\mathbf{p_i}\}$ in image I_1 and $Q = \{\mathbf{q_i}\}$ in image I_2), it finds the transformation parameters, and associated set of correspondences ($C \subset P \times Q$) that minimize an error norm of the following form:

$$E(\Theta; C) = \sum_{(p_i, q_i) \in C} \rho\left(\frac{d(M(\mathbf{p}_i; \Theta), \mathbf{q}_i)}{\sigma}\right)$$

where $M(\mathbf{p_i}; \Theta)$ is a 12-dimensional quadratic spatial transformation mapping an image point $\mathbf{p_i}$ in image I_i to a point corresponding to $\mathbf{q_i}$ in image I_2. In the above equation, $d(.)$ is a Euclidean distance measure, $\rho(\cdot)$ is the monotonically nondecreasing robust lost function proposed by Beaton and Tukey [20], and σ is the scaling parameter.

The ICP algorithm minimizes the above error measure when C is not known in an iterative fashion by alternately fixing C, minimizing $E(\theta; C)$, and re-estimating the point set C. Hence, the closest points are dynamic, changing until the algorithm eventually converges to a stable fixed point. In our method, we make use of a high-dimensional transform (12 parameters) and incorporate a robust loss function through an M-estimator, as well as utilizing a number of efficient data structures for identifying the closest points. We also incorporate a separate robust map-based verification step rather than simply relying on the minimum alignment error as determined by ICP.

The optimal transform parameters are estimated using the IRLS method [20]. To ensure that the algorithm converges to a global rather than a local minimum, this algorithm must be properly initialized. In our work, the verified transformation from the previous frame serves this purpose. In other words, the algorithm simply uses a set of extracted seed points distributed over a coarse grid to determine the closest points in the preoperative retinal mosaic. For small motions, this initialization will lead to a stable fixed point after a relatively few number of iterations (typically five). If more machine cycles are available, increasing the number of iterations showed excellent convergence properties even for frames that overlapped by as little as 30%. Large motions and other errors are readily detected based on alignment accuracy relative to the spatial map.

This algorithm is made predictable by always selecting a fixed subset of the most promising seed points from each frame. At least twelve points are needed to constrain the IRLS estimation. Doubling this number and adding another seed point (25 seeds) achieves a breakdown point of 50%, i.e., median alignment error that is robust to noise in half the seed points. Again, if more machine cycles are available, this number can be increased.

Acknowledgment

The authors would like to thank the staff at the Center for Sight, especially Dr. H. L. Tanenbaum, anonymous human subject volunteers, and photographers G. Howe and M. Fish, for extensive image acquisition assistance.

References

[1] P. N. Monahan, K. A. Gitter, J. D. Eichler, and G. Cohen, "Evaluation of persistence of subretinal neovascular membranes using digitized angiographic analysis," *Retina—J. Retinal and Vitreous Diseases*, vol. 13, no. 3, pp. 196–201, 1993.

[2] P. N. Monahan, K. A. Gitter, J. D. Eichler, G. Cohen, and K. Schomaker, "Use of digitized fluorescein angiogram system to evaluate laser treatment for subretinal neovascularization: technique," *Retina—J. Retinal and Vitreous Diseases*, vol. 13, no. 3, pp. 187–195, 1993.

[3] R. Murphy, "Age-related macular degeneration," *Ophthalmology*, vol. 9, pp. 696–971, 1986.

[4] J. M. Krauss and C. A. Puliafito, "Lasers in ophthalmology," *Lasers Surgery and Medicine*, vol. 17, pp. 102–159, 1995.

[5] J. Federman, Ed., *Retina and Vitreous*. St. Louis, MO: Mosby, 1988.

[6] I. E. Zimmergaller, N. M. Bressler, and S. B. Bressler, "Treatment of choroidal neovascularization—updated information from recent macular photocoagulation study group reports," *Int. Ophthalmology Clinics*, vol. 35, pp. 37–57, 1995.

[7] A. Can, C. V. Stewart, B. Roysam, and H. L. Tanenbaum, "A feature-based algorithm for joint, linear estimation of high-order image-to-mosaic transformations: mosaicing the curved human retina," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 24, pp. 412–419, Mar. 2002.

[8] H. Shen, C. V. Stewart, B. Roysam, G. Lin, and H. L. Tanenbaum, "Frame-rate spatial referencing based on invariant indexing and alignment with application to laser retinal surgery," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 25, pp. 379–384, Mar. 2003.

[9] G. Lin, C. V. Stewart, B. Roysam, K. L. Fritzsche, G. Yang, and H. L. Tanenbaum, "Predictive scheduling algorithms for real-time feature extraction and spatial referencing: Application to retinal image sequences," *IEEE Trans. Biomed. Eng.*, vol. 51, pp. 115–125, Jan. 2004.

[10] G. Hager and K. Toyama, "X vision: A portable substrate for real-time vision applications," *Comput. Vision and Image Understanding*, vol. 69, no. 1, pp. 23–27, Jan. 1996.

[11] P. Baglietto, M. Massimo, M. Migliardi, and N. Zingirian, "Image processing on high-performance RISC systems," *Proc. IEEE*, vol. 84, pp. 917–930, July 1996.

[12] R. Polli and G. Valli, "An algorithm for real-time vessel enhancement and detection," *Comput. Methods and Programs Biomed.*, vol. 52, pp. 1–22, 1997.

[13] Y. Sun, R. Lucariello, and S. Chiaramida, "Directional low-pass filtering for improved accuracy and reproducibility of stenosis quantification in coronary arteriograms," *IEEE Trans. Med. Imag.*, vol. 14, pp. 242–248, June 1995.

[14] S. F. Barrett, M. R. Jerath, H. G. Rylander, and A. J. Welch, "Digital tracking and control of retinal images," *Opt. Eng.*, vol. 1, no. 33, pp. 150–159, Jan. 1994.

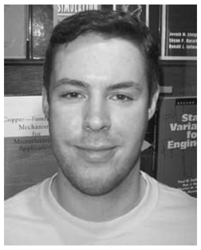
[15] S. F. Barrett, C. H. G. Wright, H. Zwick, M. Wilcox, B. A. Rockwell, and E. Naess, "Efficiently tracking a moving object in two-dimensional image space," *J. Elect. Imag.*, vol. 10, no. 3, pp. 1–9, July 2001.

[16] M. S. Markow, H. G. Rylander, and A. J. Welch, "Real-time algorithm for retinal tracking," *IEEE Trans. Biomed. Eng.*, vol. 40, pp. 1269–1281, Dec. 1993.

[17] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, “A firm real-time system implementation using commercial off-the-shelf hardware and free software,” presented at the 4th IEEE Real Time Technology and Applications Symp. (RTAS), Denver, CO, June 1998.
[18] M. Tyler and P. Saine, *Ophthalmic Photography: Retinal Photography, Angiography, and Electronic Imaging*. London, U.K.: Butterworth, 2002.
[19] F. R. Hampel, P. J. Rousseeuw, E. N. Ronchetti, and W. A. Stahel, *Robust Statistics: The Approach Based on Influence Functions*. New York: Wiley, 1986.
[20] P. W. Holland and R. E. Welsch, “Robust regression using iteratively reweighted least-squares,” *Commun. Statist.—Theor. Meth.*, vol. A6, pp. 813–827, 1977.
[21] R. E. Buttazzo and C. Giorigio, *Hard Real-Time Computing Systems—Predictable Scheduling Algorithms and Applications*. Norwell, MA: Kluwer, 1997, pp. 109–110.
[22] P. A. Laplante, Ed., *Real-Time Systems Design and Analysis: An Engineer’s Handbook*, 2nd ed. Piscataway, NJ: IEEE Press, 1996.
[23] T. Maeda, “Safe execution of user programs in kernel mode using typed assembly language,” Master’s Thesis, Univ. of Tokyo, 2002.
[24] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic, “TALx86: a realistic typed assembly language,” in *1999 ACM SIGPLAN Workshop Compiler Support for System Software*, Atlanta, GA, May 1999, pp. 25–35.
[25] M. Barabanov and V. Yodaiken, “Introducing real-time Linux,” *Linux J.*, vol. 34, pp. 19–23, 1997.
[26] D. Lazenby, “Timesys Linux/RT (professional edition),” *Linux J.*, no. 77es, Article 21, Sept. 2000.
[27] E. Siever, S. Spainhour, J. P. Hekman, and S. Figgins, *Linux in a Nutshell*, 3rd ed. Sebastopol, CA: O’Reilly Publishers, Aug. 2000.
[28] GNU-Malloc [Online]. Available: hhttp://www.mit.edu/afs/sipb/service/rtfm/src/gnu-malloc/
[29] D. E. Becker, A. Can, H. L. Tanenbaum, J. N. Turner, and B. Roysam *et al.*, “Image processing algorithms for retinal montage synthesis, mapping, and real-time location determination,” in *IMIA Yearbook of Medical Informatics*, D. Bemmel *et al.*, Eds, Germany: International Medical Informatics Association, Schattauer Press, 1999, pp. 433–446.
[30] C. V. Stewart, C.-L. Tsai, and B. Roysam, “The dual-bootstrap iterative closest point (ICP) algorithm with application to retinal image registration,” *IEEE Trans. Med. Imag.*, vol. 22, pp. 1379–1394, Nov. 2003.
[31] A. Can, H. Shen, J. N. Turner, H. L. Tanenbaum, and B. Roysam, “Rapid automated tracing and feature extraction from live high-resolution retinal fundus images using direct exploratory algorithms,” *IEEE Trans. Inform. Technol. Biomed.*, vol. 3, pp. 125–138, June 1999.
[32] H. Shen, B. Roysam, C. V. Stewart, J. N. Turner, and H. L. Tanenbaum, “Optimal scheduling of tracing computations for real-time vascular landmark extraction from retinal fundus images,” *IEEE Trans. Inform. Technol. Biomed.*, vol. 5, pp. 77–91, Mar. 2001.
[33] C.-L. Tsai, C. V. Stewart, H. L. Tanenbaum, and B. Roysam, “Model-based method for improving the accuracy and repeatability of estimating vascular bifurcations and crossovers from retinal fundus images,” *IEEE Trans. Inform. Technol. Biomed.*, vol. 8, June 2004.
[34] T. Binford and T. Levitt, “Quasiinvariants: Theory and exploitation,” in *Proc. DARPA Image Understanding Workshop*, 1993, pp. 819–829.
[35] J. S. Beis and D. G. Lowe, “Indexing without invariants in 3D object recognition,” *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 21, pp. 1000–1015, Oct. 1999.
[36] G. Borgefors, “Distance transformations in digital images,” *Comput. Vis. Graph. Image Process.*, vol. 34, no. 3, pp. 344–371, 1986.

James Alexander Tyrrell received the Sc.B. degree in geophysics/mathematics from Brown University, Providence, RI, in 1996 and received the M.S. degree in computer science from the Rochester Institute of Technology, Rochester, NY, in 2002.

Since 2002, he has been a Research Assistant in the Department of Electrical, Computer and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY. From 1997 to 2001, he worked in the R&D Department of Thomson Legal and Regulatory (West Group), Rochester, NY, where his research interests focused on natural language processing and machine learning. He is currently interested in real-time systems, robust estimation techniques, computer vision, and statistical signal processing.



Justin M. LaPre is a working toward the Ph.D. degree in computer science at Rensselaer Polytechnic Institute, Troy, NY.

His research interests include operating systems, especially Linux and Mac OS X kernel programming, networking, simulation, and computer architectures.

Christopher D. Carothers received the M.S. and Ph.D degrees both from the College of Computing at the Georgia Institute of Technology, Atlanta, in December 1996 and September 1997, respectively.

He is an Assistant Professor of Computer Science at Rensselaer Polytechnic Institute, Troy, NY. His research interest is in large-scale parallel distributed computations with emphasis on the modeling and simulation of network systems.

Dr. Carothers received a National Science Foundation CAREER Award in 2002 for his work on “Scalable, High-Performance, Network Simulations Using Reverse Computation.” He has won two best paper awards (1999 and 2003) for his research in this area and has published over 30 papers in the area of parallel and distributed simulation. He has also served as a Guest Editor for the Society for Computer Simulation (SCS) publications as well as serving on the program committee of numerous conferences and workshops including the Workshop on Parallel and Distributed Simulation (PADS) and the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS). He will also serve as the 2006 Proceedings Editor for the Winter Simulation Conference as well as the Program Chair for PADS 2005.

Badrinath Roysam (M’89) received the B.Tech degree in electronics engineering from the Indian Institute of Technology, Madras, India, in 1984, and the M.S. and D.Sc. degrees from Washington University, St. Louis, MO, in 1987 and 1989, respectively.

He has been at Rensselaer Polytechnic Institute, Troy, NY since 1989, where he is currently a Professor in the Electrical, Computer and Systems Engineering Department. He is an Associate Director of the Center for Subsurface Sensing and Imaging Systems (CenSSIS)—a multiuniversity National Science Foundation-sponsored engineering research center. He also holds an appointment in the Biomedical Engineering Department. His ongoing projects are in the areas of two-, three-, and four-dimensional biomedical image analysis, biotechnology automation, optical instrumentation, high-speed and real-time computing architectures, and parallel algorithms.

Dr. Roysam is an Associate Editor for the IEEE TRANSACTIONS ON INFORMATION TECHNOLOGY IN BIOMEDICINE. He is a member of the Microscopy Society of America, Society for Neuroscience, Society for Molecular Imaging, and the Association for Research in Vision and Ophthalmology.

Charles V. Stewart (M’88) received the B.A. degree in mathematical sciences from Williams College, Williamstown, MA, in 1982, and the M.S. and Ph.D. degrees in computer science from the University of Wisconsin, Madison, in 1985 and 1988, respectively.

Currently, he is a Professor in the Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY. During the 1996–1997 academic year, he spent a sabbatical at the GE Center for Research and Development in Niskayuna, NY. His research interests include medical image analysis, computer vision, robust statistics, and computational geometry.

Dr. Stewart is a Member of Sigma Xi and the ACM. In 1999, together with A. Can and B. Roysam, he received the Best Paper Award at the IEEE Conference on Computer Vision and Pattern Recognition.