

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

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

Rensselaer Polytechnic Institute, 110 8th Street, Troy, New York 12180-3590, USA.

ABSTRACT

This paper presents a collection of techniques, and lessons learned in the context of efficient migration of a large and complex computer vision code base developed off-line into an equivalent real-time implementation under a standard open-source operating system (Linux). Using creative linking strategies, it is possible to create a robust environment based on loadable kernel modules that enables *simultaneous realization* of real-time and off-line frame rate computer vision systems from a single code base.

Using our approach we demonstrate hard real-time predictability of a complex frame-rate vision system using commercial-off-the-shelf (COTS) hardware and a standard uni-processor Linux OS. The approach is simple; one can maximize systemic predictability by simply placing 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 non-preemptable, non-pageable and that has direct access to a powerful set of system level services.

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

Key words: Computer Vision for Surgery, Real-time Systems, Open-Source Computing, Linux, and Ophthalmic Surgery.

Correspondence: Badrinath Roysam, Professor, JEC 7010, 110 8th Street, Rensselaer Polytechnic Institute, Troy, New York 12180-3590, USA. Phone: 518-276-8067, Fax: 518-276-8715, Email: roysam@ecse.rpi.edu.

A. 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 the surgical scene, at frame rates typically ranging from 15 - 200/sec. These image sequences are analyzed in real-time to extract quantitative information. This can be used to monitor the surgical procedure, perform spatial dosimetry, track structures, compensate for motion, detect hazards, and generate control signals for surgical tools.

The present work is inspired by laser retinal surgery [1,2,3,4]. 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 and responsive computer vision aided guidance at frame rates can potentially improve the success rate.

Recent advances in both algorithms and COTS hardware have made it possible to meet these requirements [7,8,9,10,11,12,13,14,15,16]. However, we contend that this creates a unique problem in the biomedical and computer vision communities as many vision systems are predicated on software that was not designed expressly to operate in real-time. 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. Any code modification for the purpose of transitioning to real-time may not be feasible.

Additionally, ensuring accuracy and consistency between separate code bases is often impractical and is not consistent 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. Ideally, this migration would be simple enough to be considered “transparent”. Therefore, we propose a *rapid prototyping* solution to create a robust *hard real-time* execution environment

without the need to modify any 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/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 operating system and development environments. Each system must make certain trade-offs 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 trade-offs will affect 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 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 operating systems 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 Hagar and Toyama [10], Baglietto et. al. [11], and Srinivasan et. al. [27] using low cost COTS 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. We begin with a simple statement of our methodology. Then, an overview of previous work in the real-time community will highlight some of the strengths and deficiencies of various existing real-time frameworks from which we draw motivation for our work. Finally, an in depth discussion will focus on details of our proposed method.

B. Motivation and Approach

B.1 Motivation

At the core of a frame-rate vision system are generally three elements: a camera, a software component to perform image processing and some type of 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 DMA and bus mastering essentially free the rest of computer from any processing overhead. Hence, we are able to capture frames in real-time and make them available in RAM for processing on a CPU that is virtually unaffected by the imaging subsystem and vice versa. It is our intention to exploit these facts in order to establish efficient and predictable real-time performance. A key step in this direction is to understand the purpose and operation of Linux device drivers.

In order to access a hardware device from user space one needs a device driver. On Linux, device drivers reside in kernel space and are only accessible from a user level process through careful monitoring by the OS. However, while in kernel space 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, TTY serial interface, high-resolution timing 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 multi-tasking OS. Fortunately the Linux kernel provides a foundation for doing this by virtue of being non-preemptive and not swapping kernel memory. This gives us the ability to emulate a single process model directly in kernel by simply re-linking the time-critical components of our object code into a *virtual device driver*. Where the name implies that we are creating a standard Linux device driver without a physical device.

Before continuing any further it is important to mention that Linux itself 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's particularly problematic from the standpoint of a vision system is that there is no graphics or GUI support in Linux S mode. This is not acceptable for

clinical use where off-line monitoring in a GUI framework may need to co-exist 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 any restrictions on the operating mode of the host system. *In short, we are essentially achieving real-time by simply adding a new device to the computer.*

As will be illustrated, our 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 operation takes place in 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 (TLB) / 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 controlled duration, to the direct execution of our real-time code. *Therefore, we propose a real-time paradigm based on transparent migration of offline-code to an equivalent real-time system that uses the inherent real-time capabilities of a standard Linux OS without the need for proprietary real-time extensions or extensive code re-writing.*

B.2 Real-Time Retinal Image Analysis

The time-critical object code that is to be installed in kernel must be capable of executing predictably at or above the desired frame rates of the target system. 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 our experiments to be presented later. Our intention is also to convey the fact that our code base is highly complex with significant memory and processing demands; in short we feel it is a prototypical frame-rate vision system.

By relying on sophisticated registration algorithms we are able to *spatially reference* a local point in a single image frame to a global point in a pre-operative retinal mosaic. Figure 1(a) is an example of a digital retinal image (1024x1024 pixels) acquired using a standard clinical instrument known as a fundus camera or biomicroscope [17]. 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. It is usually stable and can be used as a source of spatial landmarks. The overall goal is

to use such image features to map the entire retina from diagnostic pre-operative images Figure 1(c), and then to utilize this map to estimate the location of the surgical beam in real-time during surgery. These same features can also be used to detect and quantify retinal changes, for example before and after surgery. It is also within this global coordinate system that the surgeon can map out the treatment area prior to the procedure. In short, we have the ability to detect the position of the laser anywhere on the retina relative to this treatment area from a single image frame.

In order to perform registration we generate a 12 parameter quadratic transform that maps image coordinates to global coordinates in the pre-operative retinal mosaic. 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 found using a robust M-estimator [18] 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) [19]. Of course proper initialization is required to avoid convergence to a local minimum.

We have two techniques for initializing the parameter search. The first compares vascular features between an image frame and the retinal mosaic. These vascular features are called landmarks and are determined by locating bifurcation points in the vasculature via vessel tracing. These bifurcation points are translated into quasi-invariant feature vectors [20,21,22] based on the angular relationships between branching vessels Figure 2(f). Using efficient search methodologies, likely candidates can be found in an off-line database extracted from the retinal mosaic. We can further combine multiple landmarks into *constellations* thereby potentially reducing the ambiguity of the search. For each potential match a local affine estimate is made around the landmarks. This serves as an initialization to a second estimation step using an image wide quadratic transform.

The problem is that the number of candidate landmark/constellation matches needed to successfully initialize the estimation problem is non-deterministic. Hence this stage of the algorithm is not appropriate for establishing hard real-time predictability. As a second method for initialization we simply use the transform from the previous frame combined with a sparse set of seed points extracted from the current frame. In this sense we are able to perform closed loop tracking as a side effect of the registration process. This method of feedback begins by estimating a transform using the landmark/constellation method. Then for successive frames we simply refine

this initial transformation by performing a fixed number of iterations of IRLS. If this refinement fails we simply re-start the process.

By fixing the number of seed points we are able to achieve very fast and predictable registration that successfully accounts for the projective distortions caused by excessive drift. This method is fast and stable. Contrary to other tracking methods based on matched filtering we can directly verify the overall registration error to sub-pixel accuracy. This technique is also completely automatic and very robust due to the fact that the features used to establish correspondences are not fixed between successive frames. In fact no attempt is ever made to compare features from adjacent frames that tend to be very noisy. Instead, only features from an on-line image and the pre-operative retinal mosaic are compared. For further discussion of these techniques please refer to the Appendices.

B.3 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 *hard 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 [23,24]. 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 multi-threading. Applications that require truly preemptive process/thread based scheduling to respond to asynchronous inputs are defined as being *event-driven* [23,24]. Real-time frame-rate vision systems are naturally characterized as being cyclic executive. Hence this is the target model of our proposed real-time methodology.

B.4 Previous Work

Maeda [25], 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 [26] extensions to user level object code. 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. Most 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 that introduces 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 we are proposing was developed by Srinivasan et. al. [27]. Their approach used 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 non-deterministic operating system services typically found on a timesharing system. Again this approach is highly suited for multimedia and communication systems but is not truly hard real-time. 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 common approach for achieving true *hard* real-time performance from a standard Linux

OS. RT-Linux [28] and TimeSys Linux [29] 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. What's worse is that the kernel must be made preemptable to handle asynchronous behavior. This forces kernel modules to be re-entrant 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 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 real-time.

C. Transparent Migration Methodology

C.1 Overview of the Method

The main components of methodology are listed below:

1. Encapsulate the application algorithms into a Loadable Kernel Module;
2. Design a virtual device driver that emulates a single process model in kernel;
3. Registering 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 operating system 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 finished 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 like a frame grabber 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 has been mentioned, Linux device drivers are intended to work in this manner.

C.2 Loadable Kernel Modules

In this section we describe loadable kernel modules (LKM) and how they are used to create a virtual device driver that emulates a single process model in kernel. Typical examples of LKMs are device drivers but can include any functionality that might need to be shared by multiple processes. The idea of a loadable kernel module is to dynamically add executable object code directly to the Linux kernel while the system is running. Loadable kernel modules are essentially no different than relocatable object modules created by a standard compiler like gnu's GCC compiler. 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 5 for an overview of our code base size). The main difference is that loadable kernel modules must include two functions `initModule()` and `cleanupModule()`. Each function is guaranteed to be called exactly once.

The function `initModule()` serves as the entry point into the module and is called when a kernel module is loaded via the Linux `insmod` utility. 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. Since the Linux kernel is monolithic, all modules (including the kernel itself) share a single kernel address space. This means 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 will turn out to be key components in the development of our real-time implementation.

C.3 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 mild 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 I/O;
3. Global variables; and
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. Linux does provide two specific kernel variants of the memory allocation system call `malloc()`. The first is called `kmalloc()`, which allocates physically contiguous blocks. This is ideal for our purposes but it becomes unreliable as memory gets fragmented. The other is called `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 can't be placed in physically contiguous real memory. These allocation functions have two problems: 1) they both allocate in blocks in powers of two 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 `malloc()` and `free()` that operate on a static buffer linked directly to the object code. In other words, we have circumvented the fact that kernel modules don't have a heap segment by simply inserting our own. The design of the modified `malloc` routine is described below, and key programming lines are provided in Figure (4).

Starting with the standard GNU `malloc` routine that is available in open-source form [33], 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, for sake of example call this file “`new_malloc.o`”. Having re-implemented `malloc`, we need to replace all instances of the standard version in the code. This is 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 change each reference to a certain 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:

```
ld -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 input/output 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.

Some other device I/O requests may present more of a problem but can generally be handled by choosing appropriate functions found in `/kernel/k.syms.c` and then performing the same function wrapping technique previously described. Fortunately, it is a reasonable assumption that a real-time system such as a system for surgical computer vision and control generally does not perform any

device I/O directly. Rather, such systems generally work in conjunction with specialized hardware/software. A common example is a digital image frame grabber device.

The use of global variables may present problems when creating loadable kernel modules. 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 subtler issue is relevant. When a user level process is created, the operating system 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.

```
ld -Ur -static -o module.o gcc-lib-path/crtbegin.o [files] gcc-lib-path/crtend.o
```

This link line explicitly adds the gcc “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 `initModule()` and the call `__do_global_dtors_aux()` to `cleanupModule()`.

The last issue has to do with stack usage in kernel space. In Linux, 8K bytes (two memory pages on an 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 bytes of the 8KB address space. Thus, in total, the kernel has ~7KB of usable stack space to operate on behalf of the 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 ~7KB 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. Given this problem it is important that sound programming principles are employed. Specifically, one should avoid allocating large objects on the stack either as local variables or as function parameters. The use of recursion may also have to be handled carefully.

In applications where this constraint is too restrictive there are a variety of standard Linux patches available for reconfiguring the kernel stack size. Of course we've made every effort to avoid operating outside the system parameters established in a standard Linux OS and we've experienced little trouble in our work.

C.4 System Call Interface

Using the preceding techniques we are able to re-link existing object code into a relocatable kernel object module. 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 entries include `ioctl`, `read`, `write`, `lseek`, `open`, `flush`, etc. It is from within these functions that we call our vision code to invoke real-time processing. After loading with `insmod` an entry for the module can be found in `/proc/devices`. In this entry will be 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.

C.5 Handling Interrupts

Using the inherent properties of Linux we can use a virtual device driver to provide predictable real-time performance without modifying any code. At this point we have developed everything need 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()` 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). Admittedly 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. If hardware requirements on a host system demand, the solution is to selectively mask out all nonessential interrupts. In our work we've generally relied on the `cli()/sti()` method but masking is quite common, well supported and designed to be done from kernel space. For example, one might require temporarily masking out the mouse and keyboard if running X-Windows.

D. Experimental Results

Under this new configuration we can establish hard real-time predictability by protecting the time-critical components of our system from the real-time threats associated with a shared memory process space. Quite a remarkable fact given we've simply added a few link lines to our makefile and written a handful of `file_ops` functions to serve as entry points into our virtual device driver. To explore the effectiveness of our 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 PC with a 1 Ghz processor and 1 GB of RAM running X-Windows under Red Hat 7.1 kernel version 2.4.18.

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 while the landmark/constellation registration is performed in user space (Figure 5). 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 would typically consist of a binary signal where a failure to register the proper polarity every 33 ms as detected at the output would signal a missed deadline.

In our first experiment 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 variance due to the algorithm itself. In order to get meaningful results we want 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 1024x1024 12-bit gray scale images at ~30 frames/sec. 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 buffer switch 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. Each mode shares a common user level executable. This *driver program* calls landmark/constellation based spatial referencing (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 1. 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 many times to handle memory requests which leads to appreciable processing delays.

Given that we have established 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 experiment we will use the tracking mechanism initialized with landmark/constellation based spatial referencing as described in Appendix B. The test data consists of a pre-loaded sequence of 500 512x512 retinal images captured at 30 fps and 12 bits per pixel. This reduced image size is obtained by 2x2 binning of the CCD array in our Dalsa camera and is done to improve signal-to-noise. Again we create a user level program to drive a kernel and user space variant of the system.

In a sequence of 500 images we successfully track 436 frames by registering them onto a pre-operative retinal mosaic. This is done with an average time of about 1.7 ms. (It is important to note that these times don't include seed point detection because in an actual system this will be done in FPGA at the frame grabber.) Since our camera runs at 30 fps 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 inexorable risk that must be mitigated by minimizing the latency of our system. This average of 1.7 ms is significantly less than what we are able to achieve by simply translating a set of seeds around a small radius on the pre-operative retinal mosaic until the best match was found. Not only is this translation approach less efficient but also on a series of 1024 images it only tracks 61% of the images versus 82% for our method.

The timing results in Table 2 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 `sbrk()` calls made as the dynamic memory demands of the user level process jump as processing begins. This is further exemplified in Figure 6 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 `sbrk()` call is emulated by a static buffer that is linked directly to the object module. The difference in speed is most likely due to the paging overhead incurred in user space as memory must be allocated in non-contiguous blocks from the pageable memory pool. 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 is going to significantly disrupt the memory access patterns of a program in user space. However, this actually illustrates how non-pageable kernel memory allocation can significantly increase systemic predictability.

Since kernel modules have no stack or heap segment they can be placed contiguously without creating any memory fragmentation larger than a single page. Unfortunately kernel modules cannot always be loaded this way. However, from the standpoint of the kernel's virtual map memory is contiguous and further 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. Of course in Linux 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.

E. Conclusions and Discussion

We have designed a framework for establishing predictable frame-rate computing on a standard Linux operating system using COTS hardware. We make use of a virtual device driver to emulate a single process space directly in kernel without modifying any existing 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 non-standard kernel modifications.

Establishing real-time 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 don't 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 a degree of 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 non-preemptive nature of the Linux kernel, may greatly reduce the effort needed to introduce a degree of parallelism to an existing off-line code base. Our work using Linux kernel modules could be a key stepping stone towards such a design.

F. Acknowledgments

Various portions of this research was supported by the National Science Foundation Experimental Partnerships grant EIA-0000417, the Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the National Science Foundation (Award Number EEC-9986821), the National Institutes for Health grant RR14038, and by Rensselaer Polytechnic Institute. The authors would like to thank the staff at the Center for Sight, especially Dr. Howard L. Tanenbaum, and photographers Gary Howe and Mark Fish, for extensive image acquisition.

G. References

- [1] Monahan, P.N., Gitter, K.A., Eichler, J.D. and Cohen, G., "Evaluation Of Persistence Of Subretinal Neovascular Membranes Using Digitized Angiographic Analysis," *Retina-The Journal of Retinal and Vitreous Diseases*, 13(3):196-201, 1993.
- [2] Monahan, P.N., Gitter, K.A., Eichler, J.D., Cohen, G., and Schomaker, K., "Use Of Digitized Fluorescein Angiogram System To Evaluate Laser Treatment For Subretinal Neovascularization: Technique," *Retina-The Journal of Retinal and Vitreous Diseases*, 13(3):187-195, 1993.
- [3] Murphy, R., "Age-Related Macular Degeneration," *Ophthalmology*, 9:696--971, 1986.
- [4] Krauss, J. M. and Puliafito, C. A., "Lasers In Ophthalmology," in *Lasers in Surgery and Medicine*, 17:102-159, 1995.
- [5] Federman, J., editor. *Retina and Vitreous*, The C.V. Mosby Company, St. Louis, 1988.
- [6] Zimmergaller, I. E., Bressler, N. M., and Bressler, S. B., "Treatment of Choroidal Neovascularization – Updated Information from Recent Macular Photocoagulation Study Group Reports," *International Ophthalmology Clinics*, 35:37-57, 1995.
- [7] Can, A., Stewart, C. V., Roysam, B., and Tanenbaum, H. L., "A Feature-Based Algorithm for Joint, Linear Estimation of High-Order Image-to-Mosaic Transformations: Mosaicing the Curved Human Retina," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 3, March 2002.
- [8] Shen, H., Stewart, C. V., Roysam, B., Lin, G., and Tanenbaum, H. L., "Frame-Rate Spatial Referencing Based on Invariant Indexing and Alignment with Application to Laser Retinal Surgery," vol. 25, No. 3, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, March 2003.
- [9] Lin, G., Fritzsche, K. L., Stewart, C. V., Tanenbaum, H. L., and Roysam, B., "Predictive Scheduling Algorithms for Real-time Feature Extraction and Spatial Referencing: Application to Retinal Image Sequences," *IEEE Transactions on Biomedical Engineering*, April 2002.
- [10] Hager, G. and Toyama, K., "X Vision: A Portable Substrate for Real-Time Vision Applications," *Computer Vision and Image Understanding*, Vol. 69, No. 1, pp 23-27, January

1996.

- [11] Baglietto, P., Massimo, M., Migliardi, M., Zingirian, N., "Image Processing on High-Performance RISC Systems," Proceedings of the IEEE, Vol. 84, No. 7, July, 1996.
- [12] Polli, R. and Valli, G., "An algorithm for real-time vessel enhancement and detection," Computer methods and programs in Biomedicine, 52:1--22, 1997.
- [13] Sun, Y., Lucariello, R., and Chiaramida, S., "Directional low-pass filtering for improved accuracy and reproducibility of stenosis quantification in coronary arteriograms," IEEE Transactions on Medical Imaging, 14:242--248, June 1995.
- [14] Barrett, S. F., Jerath, M. R., Rylander, H. G. and Welch, A. J., "Digital tracking and control of retinal images," Optical Engineering, 1(33):150-159, Jan. 1994.
- [15] Barrett, S.F., Wright, C. H. G., Zwick, H., Wilcox, M., Rockwell, B.A., Naess, E., "Efficiently Tracking a Moving Object in Two-Dimensional Image Space", Journal of Electronic Imaging, Vol. 10 No. 3, July 2001, pp 1-9.
- [16] Markow, M.S., Rylander, H. G. and Welch, A. J., "Real-time algorithm for retinal tracking," IEEE Transactions on Biomedical Engineering, 40(12):1269--1281, December 1993.
- [17] Tyler, M., and Saine, P., *Ophthalmic Photography: Retinal Photography, Angiography, and Electronic Imaging*, Butterworth-Heinemann Medical, 2002.
- [18] Hampel, F.R., Rousseeuw, P. J., Ronchetti, E. N. and Stahel, W.A., *Robust Statistics: The Approach Based on Influence Functions*, John Wiley & Sons, 1986.
- [19] Holland, P.W. and Welsch, R. E., "Robust regression using iteratively reweighted least-squares," Commun. Statist.-Theor. Meth., A6:813-827, 1977.
- [20] Binford, T. and Levitt, T., "Quasi-invariants: Theory and exploitation," In Proceedings of the DARPA Image Understanding Workshop, pages 819-829, 1993.
- [21] Beis, J. S. and Lowe, D. G., "Indexing without invariants in 3d object recognition," IEEE Transactions on Pattern Analysis and Machine Intelligence, 21(10):1000-1015, 1999.
- [22] Borgefors, G., "Distance transformations in digital images," CVGIP, 34(3):344-371, June 1986.
- [23] Buttazzo, Giorigio C., *Hard Real-Time Computing Systems-Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, 1997 pg. 109.

- [24] Laplante, P. A., editor. *Real-Time Systems Design and Analysis: An Engineer's Handbook*, 2nd ed., IEEE Press, 1996.
- [25] Maeda, T., "Safe Execution of User Programs in Kernel Mode Using Typed Assembly Language," Master's Thesis, University of Tokyo, 2002.
- [26] Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S. and Zdancewic, S., "TALx86: A Realistic Typed Assembly Language," In the *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25-35, Atlanta, GA, USA, May 1999.
- [27] Srinivasan, B., Pather, S., Hill, R., Ansari, F. and Niehaus, D., "A firm real-time system implementation using commercial off-the-shelf hardware and free software," In Fourth IEEE Denver, Colorado, June 1998.
- [28] Barabanov, M. and Yodaiken, V., "Introducing Real-Time Linux," *Linux Journal*, 34:19--23, 1997.
- [29] Lazenby, D., "Timesys Linux/RT (Professional Edition)," *Linux Journal*, September, 2000.
- [30] GNU-malloc, Available on-line at <http://www.mit.edu/afs/sipb/service/rtfm/src/gnu-malloc/>
- [31] Becker, D. E., Can, A, Tanenbaum, H. L., and Turner, J. N., Roysam, B., "Image Processing Algorithms for Retinal Montage Synthesis, Mapping, and Real-Time Location Determination," *IMIA Yearbook of Medical Informatics*, International Medical Informatics Association, Schattauer Press, Germany, (Bemmel et al., eds), pp. 433-446, 1999.
- [32] Stewart, C. V., Tsai, C-L, and Roysam, B., "A Dual Bootstrap Iterative Closest Point (ICP) Algorithm: Application to Retinal Image Registration," (accepted July 2003, to appear) *IEEE Transactions on Medical Imaging*, Special Issue on Medical Image Registration. (pre-print available at www.cs.rpi.edu/~stewart)

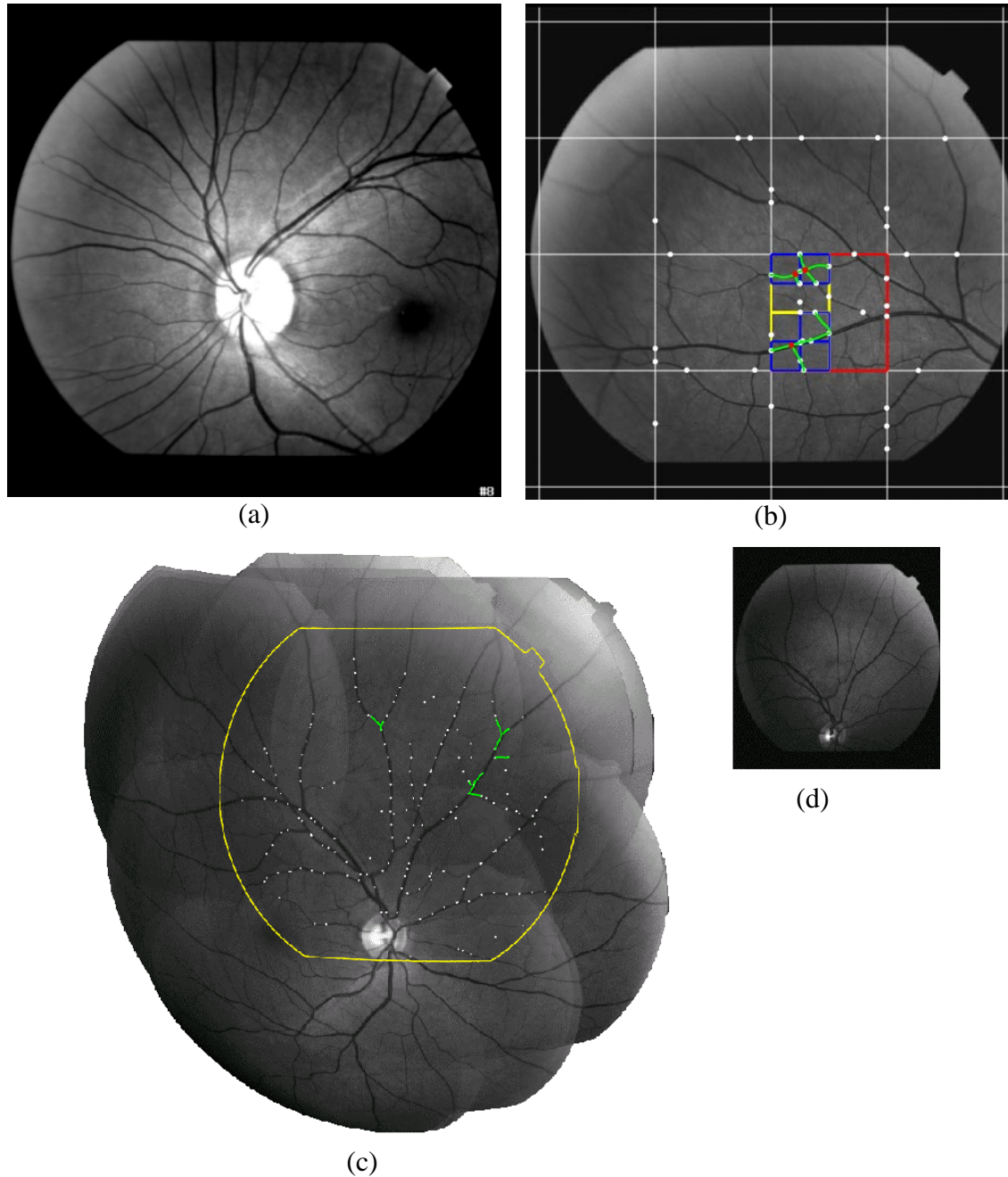


Figure 1. Illustrating the retinal spatial mapping and referencing application of interest. Spatial mapping is performed prior to surgery from diagnostic images (Panel (a)) to generate mosaics (Panel (b)), adding surgical plans and pre-computed data structures to support fast spatial referencing during surgery on-line. Spatial referencing is the problem of registering each image frame (Panel (d)) captured by a camera onto the spatial map. The yellow outline in Panel (c) illustrates this mapping *that must occur predictably in real-time, at frame rates*.

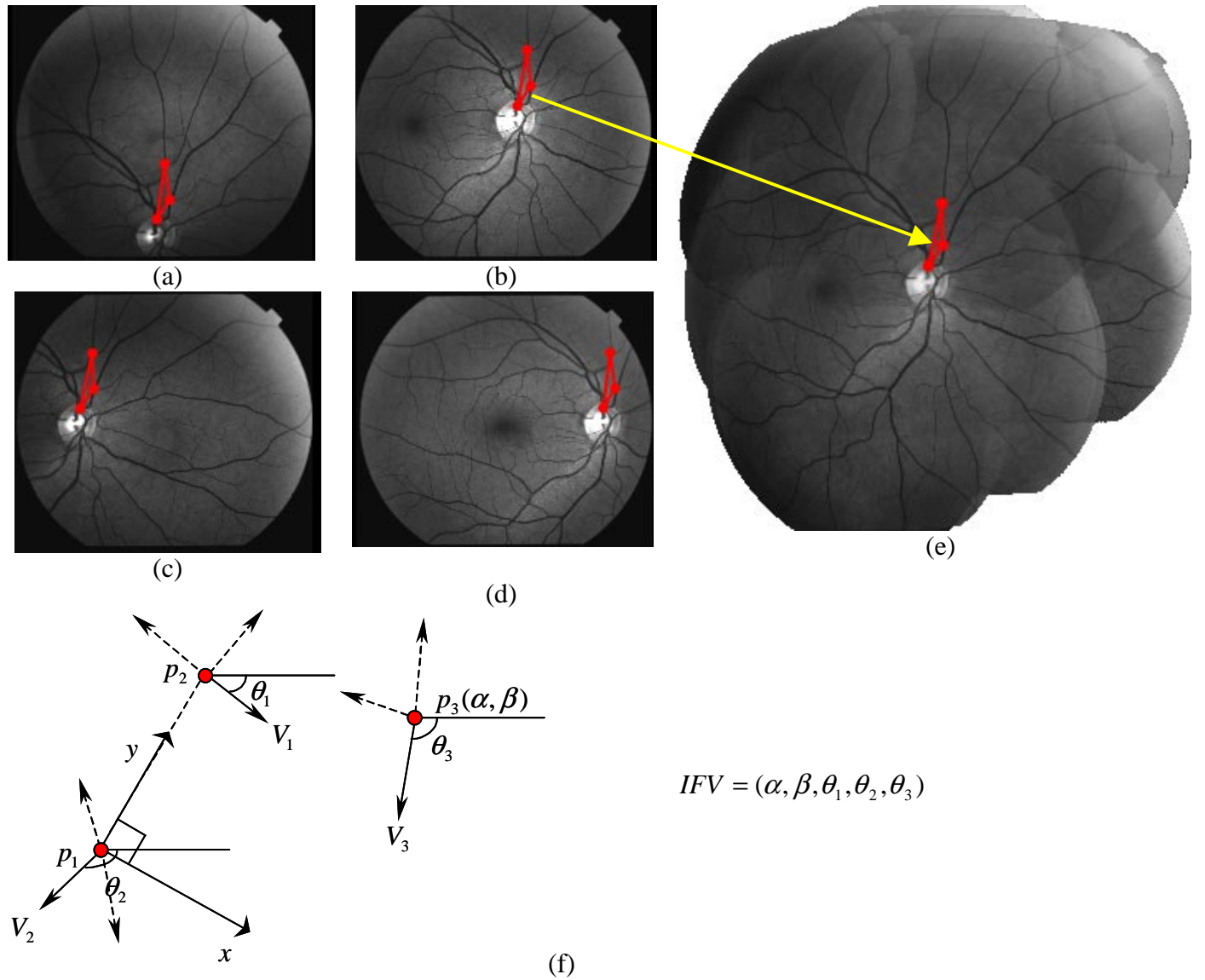


Figure 2: Illustrating the quasi-invariant indexing based approach to fast spatial referencing; Panels (a-d) are four sample image frames that must be registered to the spatial map shown in Panel (e); The red triangles are local constellations of vascular landmarks. For each local constellation, a feature vector can be computed that is invariant to similarity transformations. For a constellation of three landmarks, the invariant feature vector (QIFV) consists of five components $(\alpha, \beta, \theta_1, \theta_2, \theta_3)$ as illustrated in Panel (f). Landmarks p_1 and p_2 are used to construct a coordinate reference frame (x, y) . The coordinates of the third landmark p_3 in this frame are (α, β) . The orientations of closest vascular segments (V_1, V_2, V_3) relative to the (x, y) frame $(\theta_1, \theta_2, \theta_3)$ are also included in the QIFV. This vector can be looked up rapidly in a pre-computed hierarchical database of QIFVs.

```

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

/* Allocate INCREMENT more bytes of data space,
and return the start of data space, or NULL on errors.
If INCREMENT is negative, shrink data space. */
__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;
}

```

Figure 4: Illustrating the technique for modifying the standard UNIX memory allocation system call “malloc”.

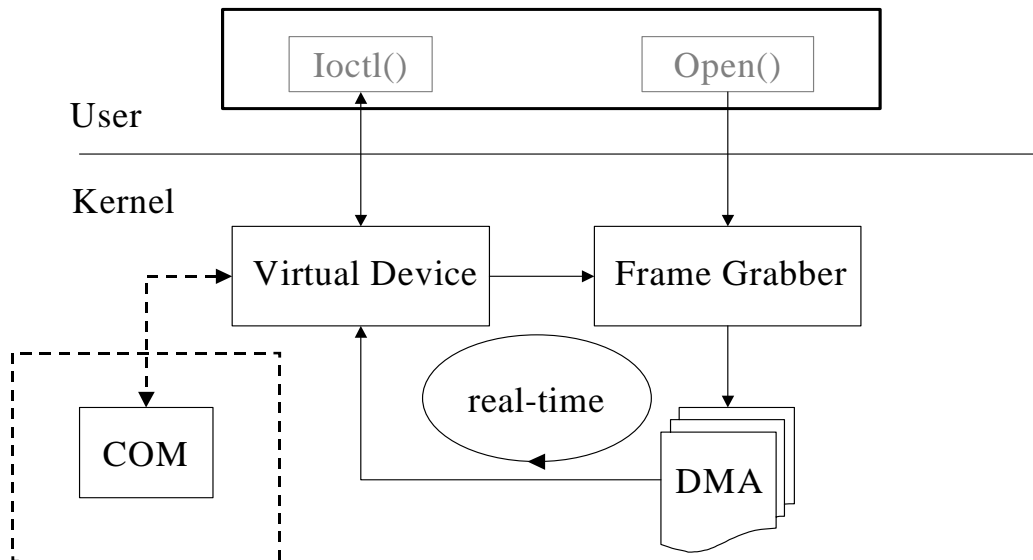


Figure 5. System design for real-time tracking. Initially, the frame grabber is opened from user space placing the device into the Linux IRQ chain. Henceforth, the user process interacts directly with our virtual device driver that serves as a proxy for the frame grabber device driver. Note the cyclic real-time executive protected in kernel space. This is an extremely simple arrangement, our virtual device driver simply calls the frame grabber’s `read()` method directly. Also shown is an optional interface to a serial port or similar device.

Table 1: 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 2: 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. User mode predictability has been severely diminished by virtual memory overheads.

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

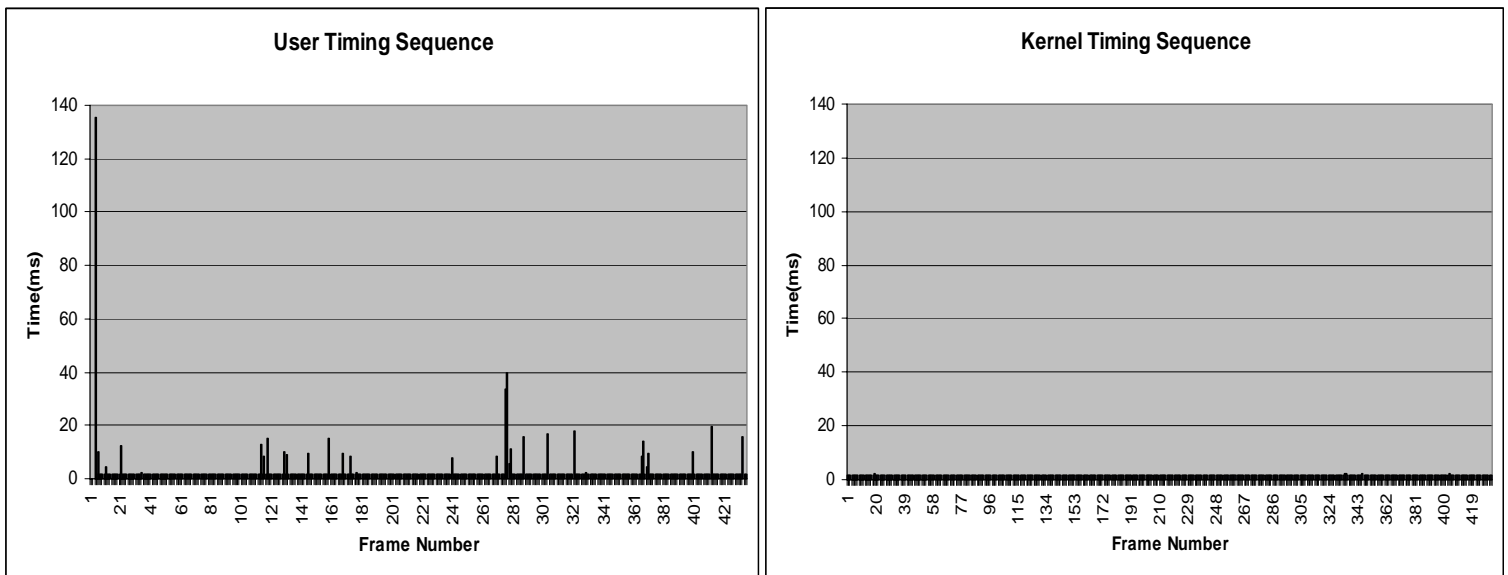


Figure 6. Timing sequence for 500 frame tracking experiment run from console without system load. Height of bar indicates time in milliseconds required to successfully track image frame.

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

Table 4. Source code profile showing relative size of major software components that we link into a kernel module. The VXL library is a 3rd 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’ve experienced little difficulty loading our modules on a system with 1 GB of RAM.

Appendix A

Extracting vascular landmarks by conventional image segmentation and feature extraction can be computationally expensive due to the sheer number of pixels (1M) in the image. Recently [8], we have described fast and adaptive exploratory algorithms for tracing the retinal vasculature, improving upon prior work [9]. They derive their performance from adaptively restricting computations to pixels representing the sparse vasculature and avoiding extensive pixel processing.

Vascular tracing and feature extraction algorithms proceed in three stages, controlled by a real-time spatial prioritization and scheduling algorithm [9]. First, the grid analysis and seed point detection stage surveys the image along a 2-D grid (Figure 1(b)), estimating the local image statistics and noise levels, and detecting pixels representing blood vessels. These positions, known as seeds, are refined and verified by testing for the existence of a pair of sufficiently strong 2-D anti-parallel (opposite direction) edges, using a set of directional kernels [13] in a small region around each seed. Algorithms have been described for prioritizing the grid analysis to generate an early yield of landmarks, and landmark constellations [9].

The second stage performs iterative tracing of the vasculature starting from seed points (Figure 1(b)) by repeatedly detecting and following anti-parallel edges forming a blood vessel boundary, and estimating vessel centerlines. The points where traces meet or cross are the vascular landmarks. The final landmark refinement stage uses a separate mathematical model for the bifurcation and crossover locations to refine their locations to sub-pixel accuracy by fitting lines to the traces near intersections and then finding the closest point to these lines.

The traces extracted from multiple retinal images can be used to construct accurate mosaics of the entire retina (Figure 1(c)). Recently, this group has published accurate registration 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 [18,19]. An important extension of pair-wise retinal image registration is joint registration of a set of (12-15) images to construct mosaic families with sub-pixel 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. Specifically, a spatial map of the retina consists of a set of mosaics, along with vascular traces, and an indexed hierarchical database, usually structured as a *k-d* tree, which organizes the spatial content of the retinal features into a form that can be searched rapidly. The retinal map is pre-computed and stored prior to laser retinal surgery, and is an enabling technology for real-time spatial referencing.

Spatial Referencing is the problem of estimating a spatial transformation that links *each* observed retinal image frame, illustrated in Figure 1(c), (size 1024x1024 pixels) to the pre-computed map during laser surgery, as illustrated in Figure 1(c-d). This method is an improvement over frame-to-frame tracking [14,15,16], which is subject to drift and undetected failure [31]. The core problem underlying spatial referencing is image registration, but the speed and accuracy requirements are extreme. This group has recently published methods [8,9] to meet these extreme requirements using a combination of extensive pre-computation and the use of quasi-invariant feature vectors. Pairs and triples of landmarks that are reasonably close to each other (within about 20% of the image width) are identified and formed into "landmark constellations". A vector of similarity invariants - geometric measurements that do not change under scale, rotation, and translation, is computed from the landmarks in each constellation (Figure (2)). These measures are quasi-invariant [20] because the similarity transformation is only approximate, and therefore the vectors are called quasi-invariant feature vectors (QIFVs). QIFVs computed from all constellations in all diagnostic images are stored in the spatial map using *k-d* trees, as in [21], for fast lookup during the on-line phase. So, the complete spatial map consists of a set of images, their traces, Euclidean distance maps [22] of these 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 apparently complex operation can be performed surprisingly fast by subsampling the vasculature and using a pre-computed digital distance map of the traces. Verified correspondence hypotheses produce crude 4-parameter similarity transformation estimates. They are refined in a series of steps that ultimately lead to an image-wide, 12-parameter transformation. At any point during this verification and refinement, if the alignment is too inaccurate, the hypothesis is rejected and another one is considered. Experiments show that the median number of matches tested is two.

In principle, a single constellation of landmarks can yield a complete and accurate alignment of an image frame and the stored spatial map. With this in mind, Lin et al. [9] have described an opportunistic strategy for coordinating the feature extraction computations (vessel tracing) and the spatial referencing computations (QIFV database lookup and verification) that minimizes the overall computations to the bare minimum. Figure 1(c-d) illustrates a typical result. Panel 1(b) illustrates a hierarchical strategy for finding seeds and tracing the vasculature with the goal of extracting a promising constellation of landmarks. The green lines in panels 1(b) and 1(c) indicate the full extent of the real-time vessel tracing that was actually necessary to align a 1024x1024 image frame with the spatial map. The white dots in Panel 1(c) represent the sub-sampled vascular traces used for verification of the alignment. The yellow outline in panel 1(c) is the outline of the real-time image frame that was registered to the spatial map.

Appendix B

From a real-time standpoint the problem with landmark/constellation based spatial referencing is that it is not predictable. The opportunistic nature of the algorithm prevents any real level of determinism. The amount of tracing and the number of hypotheses under consideration cannot be made exact. The solution is to incorporate a tracking mechanism that does a fixed amount of processing for each frame.

In contrast to many feature-based tracking systems in the literature [14,15,16] we make no attempt to directly correlate specific image features between frames. This is simply not robust

enough in our application. Instead we make clever use of the landmark/constellation based spatial referencing algorithm (Appendix A) that, as a side effect, tracks changes in the position of the retinal vasculature between successive image frames.

At the core of the spatial referencing algorithm is essentially the familiar iterative-closest-point (ICP) registration algorithm [32]. In short, ICP is a search process that finds a transformation that minimizes the alignment error between a set of reference points between two images. The alignment error is a function of the distance between the closest points in each image as measured at each step. Hence, the closest points are dynamic, changing until the algorithm eventually converges to a stable fixed point. Of course our method is more sophisticated than the default ICP algorithm. We make use of a high dimensional transform 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 verification step rather than simply relying on the minimum alignment error as determined by ICP.

Finding the optimal transform parameters via ICP requires an iterative minimization procedure like gradient descent, Newton's method or in our case iteratively reweighted least-squares (IRLS)[19]. This procedure must be properly initialized to prevent ICP from converging to a local minimum. It is essentially this initialization process based on landmark feature correspondence that makes spatial referencing non-deterministic. However, provided we have a single successful transformation between an image frame and the pre-operative retinal mosaic, we can immediately initialize our modified ICP search on the next frame. This ICP step simply uses a set of extracted seed points distributed over a coarse grid to determine the closest points in the pre-operative retinal mosaic. Provided the frame hasn't shifted too much this initialization will lead to a stable fixed point after a relatively few number of iterations. In this case we limit this number to 5 but we found that given an unlimited number of iterations this process showed excellent convergence properties given that two frames overlapped by as little as 30%.

Most importantly this operation is predictable by virtue of the fact that we always select a fixed subset of the most promising seed points from each frame. We need a minimum of 12 points to constrain the IRLS estimation problem. By doubling this number and adding another seed point we arrive at 25, giving a median alignment error that is insensitive to noise in half the seed points.

Since we are tracking based on the results of an image registration we have the ability to verify the accuracy of the resulting transformation. This is done by directly comparing the extracted seeds to the vascular structures we have identified in the pre-operative retinal mosaic. This is a claim tracking algorithms based on matched filtering cannot make. The process is also significantly faster than general match filtering and is much more robust to noise.

As mentioned, the fact that this algorithm performs tracking is really a side effect of the ICP registration. With each successive frame, ICP refines the initial transform in a closed loop system allowing the model parameters to track through transform space correctly compensating for any large-scale drift. This is especially important since the curvature of the retina combined with a weak perspective camera makes tracking by simple translation of a matched filter practically useless. In the event that the tracking is unsuccessful the entire process is re-initialized using spatial referencing.