Starfish: Efficient Concurrency Support for Computer Vision Applications

Robert LiKamWa and Lin Zhong
Rice University, Houston, TX
roblkw@rice.edu, lzhong@rice.edu

ABSTRACT

Emerging wearable devices promise a multitude of computer vision-based applications that serve users without active engagement. However, vision algorithms are known to be resource-hungry; and modern mobile systems do not support concurrent application use of the camera. Toward supporting efficient concurrency of vision applications, we report *Starfish*, a split-process execution system that supports concurrent vision applications by allowing them to share computation and memory objects in a secure and efficient manner. Starfish splits the vision library from an application into a separate process, called the Core, which centrally serves all vision applications. The Core shares library call results among applications, eliminating redundant computation and memory use. Starfish supports unmodified applications and unmodified libraries without needing their source code, and guarantees correctness to the applications. In doing so, Starfish improves both the performance and energy efficiency of concurrent vision applications. Using a prototype implementation on Google Glass, we experimentally demonstrate that Starfish reduces the time spent processing repeated vision library calls by 71% – 97%. When running two to ten concurrent face recognition applications at 0.3 frames per second, Starfish reduces CPU utilization by more than 42% – 80%. Notably, this keeps CPU utilization below 13%, even as the number of applications increases. This reduces system power consumption by 19% – 58%, as Starfish maintains a power consumption at approximately 1210 mW while running the concurrent application workloads.

Categories and Subject Descriptors

I.4.m [**Image Processing and Computer Vision**]: Miscellaneous; D.0 [**Software**]: General

Keywords

Computer vision; vision library; mobile computing; memoization

1. INTRODUCTION

In our envisioned future of *continuous mobile vision* [3], multiple vision applications continuously and concurrently run on mobile devices, analyzing camera frames to understand its environment without active user engagement, e.g., [12, 25]. A wearable device could simultaneously recognize faces to log social interactions, and detecting food items to record eating habits. The same device may also be tasked with remembering where a car is parked and detecting landmarks for rapid localization. We believe that it is highly likely that these applications will be developed by various parties each with their own specialization such as medical informatics, personal analytics, and social networking.

Unfortunately, modern mobile systems support only a single application's access to a camera, which is adequate for photo/video taking and interactive vision applications. Furthermore, camera devices and vision algorithms are known to be too power hungry for always-on operation [17, 18]. Thus, there is an urgent need to support concurrent vision applications efficiently. While hardware innovations are necessary [17, 20], this work approaches the above challenge in system software.

We hold two key insights. First, concurrent vision applications operate on the same root data: frames from the camera. Second, typical vision applications use a relatively small set of foundational algorithms, usually implemented with a standard library, e.g., OpenCV or FastCV. Thus, there is often significant redundancy in the library functions they execute and the objects they create. This presents an opportunity to share redundant library processing among multiple applications to perform simultaneous vision tasks with efficient system performance.

To exploit these insights, we design the *Starfish* software system that not only supports concurrent vision applications, but also allows them to share computation and memory objects securely and efficiently. Our design stems from two major decisions. (*i*) Starfish splits the vision library from an application into a separate process, called the Core, which centrally serves all applications. Essentially, Starfish converts a vision library call into a remote procedure call into the Core. (*ii*) The Core shares library call results among applications securely, eliminating redundant computation and memory use. Importantly, Starfish supports unmodified applications and unmodified vision libraries without their source code, and guarantees correctness to vision applications.

In doing so, Starfish improves the performance and energy efficiency of concurrent vision applications. Leveraging work in remote invocation [4, 21] and computation memoization [13, 19], Starfish contributes novel techniques that meet its unique constraints.

We also report an implementation of Starfish on Android KitKat. Our work intercepts calls to OpenCV by replacing its C++ linked library with our own, and forwards calls over an Android Binder with `MemoryHeapBase` shared memory regions. We support 22 OpenCV functions and 19 OpenCV objects and design interfaces to ease Starfish library development and reduce source code redun-

MobiSys'15, May 18–22, 2015, Florence, Italy.

http://dx.doi.org/10.1145/2742647.2742663.

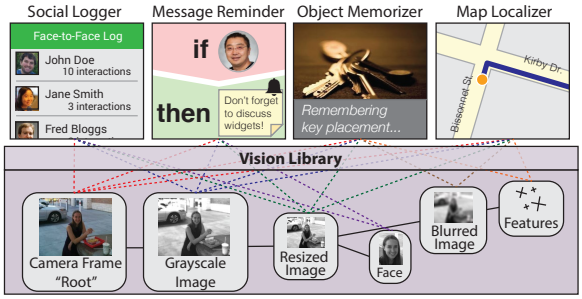


Figure 1: Vision applications use a vision library to request computations on the root data to create derivative vision objects. Multiple vision applications often require the same vision objects.

dancy. To fulfill the need to provide concurrent vision applications with camera frame access, we design a frame service that operates on top of the OpenCV camera framework mechanisms, allowing Starfish to send frames to foreground and background applications with no modification to application functionality.

We evaluate the Starfish design and implementation on a Google Glass. We use microbenchmarks to understand the per-call overhead of Starfish library call requests, validating the effectiveness of our split-process and function caching design choices. We measure that our optimizations reduce the per-call Starfish overhead from 20.9 ms to 5.9 ms. We then evaluate the effectiveness of Starfish on a combination of application benchmarks that share computations and do not share computations to explore the range of benefits that Starfish provides. We show that Starfish provides minimal negative impact to applications that do not share computations, while providing substantial benefits to applications that do share computations. While running multiple face recognition applications typically incurs significantly increasing CPU utilization, scaling up to 60% for ten concurrent vision applications running at 0.3 frames per second, Starfish maintains a CPU utilization, staying below 15%. Under Starfish, the system consumes 19% – 58% less power, as Starfish maintains a power consumption at approximately 1210 mW while running the concurrent application workloads. Starfish thus successfully provides system efficiency benefits to concurrent vision applications.

The rest of the paper is organized as follows. §2 provides a background of vision applications and libraries. §3 presents a high level overview of the Starfish design. §4 describes design strategies that reduce the overhead of the split-process execution. §5 details caching policies that allow reuse of previously computed library results. §6 offers details regarding our Android + OpenCV Starfish implementation, including a shared frame delivery service among multiple applications. §7 evaluates Starfish using a combination of application benchmarks, and §8 discusses related work.

2. COMPUTER VISION BACKGROUND

We derive our Starfish design through examining contemporary implementations of vision applications. Typical computer vision applications read frames from the camera service and use libraries of computer vision algorithms to extract important information about the content or geometry of a camera frame. In this section, we discuss various systems implications and opportunities of computer vision applications.

2.1 Resource usage

Vision applications are resource-intensive, consuming memory, memory bandwidth and CPU cycles. Sampling a full color image uses 0.5 MB for a 640×360 color frame. At 30 frames per second, this introduces 15 MB/s of data creation and usage. To maximize memory reuse, Android applications typically issue a pre-allocated multi-frame buffer to allow the Android camera service to write frames to pre-allocated memory with every new frame capture. In addition to removing the latency of allocating memory, this allows the camera service to record new frames while others are being processed or retained by the application.

Vision operations on received frames consume further CPU cycles; computing corners on the frame can take 2.2 seconds at 40% CPU utilization on a Google Glass. As such, applications typically operate on subsampled frames to reduce memory-handling and computation bottlenecks. Still, real-time vision operations need to maintain an acceptable frame rate while performing tasks that may be highly computationally expensive and memory intensive. This implies a necessity to optimize frame and feature processing.

2.2 Frame and feature redundancy

For continuous vision applications, the camera service streams frames from the camera to the application as they become available. This frame becomes the root data for repetitively computed vision processing operations within an application. First, frame-level image processing operations, such as resize and blur convolutions, create derivative frames from the root data. Other processing operations then generate successive derivatives when computing corners, edges, or other feature locations and descriptors, as illustrated in the bottom half of Figure 1.

These streams of root data and derivative objects form common first steps of many algorithms and applications. Many applications utilize the same camera resource, and therefore can share the same root data, as shown in the top half of Figure 1. Executing on the root data, many applications follow the same basic image processing steps. For example, for efficiency, many vision algorithms begin by downsampling a frame and converting it to grayscale. Our benchmarks for face recognition, scene geometry, and object recognition all perform both of those tasks as common first steps. Moreover, low-level vision features are commonly used across vision algorithms; an integral image serves as the basis for Viola-Jones object recognition for face detection and character recognition [2]. Our scene geometry and object recognition benchmarks both employ SURF features. Even the same high-level features, such as detected and recognized faces, can be utilized by diverse applications, including life-loggers [26], photography taggers [28], and alert notification services [14].

2.3 Computer vision libraries

To incorporate packaged vision algorithms, developers of vision applications harness libraries, such as the OpenCV Library for iOS, Android, and Windows Phone, or Qualcomm's FastCV Library for the latter two. While the libraries are useful for rapidly prototyping applications, the recent trend of implementations have become attractive for product-grade application performance and efficiency. OpenCV versions increasingly leverage hardware to optimize the processing, e.g., SIMD processing with Intel SSE and ARM NEON, GPGPU computing through CUDA and OpenCL, and vision hardware support through OpenVX. As the library implementations handle the hardware acceleration, vision libraries provide an easy pathway to efficient vision performance. Optimizing library use will multiply the benefits of the library processing acceleration.

An Android application incorporates OpenCV by linking to the OpenCV libraries and calling the appropriate functions. OpenCV Java functions call their associated native C++ functions to execute.

For further speed and control, many developers use the Java Native Interface to directly call the OpenCV C/C++ functions. At that level, an application invokes shared library functionality by using classes and functions prototyped in header files. As of March 2015, opencv.org reports a user base of 47,000 developers and 9 million downloads [15].

2.4 Motivational observations

To summarize, we have the following insights about vision applications:

- *Vision is memory-expensive and compute-intensive.* Vision uses heavy resource utilization to process frames and features. There is much need to alleviate memory bandwidth and CPU utilization.
- *Vision libraries are common-case utilities.* The code reuse of the shared library is valuable for a variety of applications. Optimizing vision library performance thus improves a widely-applicable developer tool.
- *Library calls are redundant across multiple applications.* Multiple applications call the same streams of library functions on the same data for both low-level and high-level computations. This redundancy provides an opportunity to collectively share computation results.
- *Library calls are repetitive within an application.* Vision applications compute frames and features on incoming streams of frame data; a single application calls the same set of library functions on incoming frames. Thus, the opportunity for sharing library execution occurs not just once, but repeatedly throughout an application's execution.

These observations motivate our Starfish design, which services concurrently running applications through memory and computation reuse through a split-process execution.

3. DESIGN OF STARFISH

We present a software system design called ***Starfish***. Starfish enables a system to concurrently run multiple vision applications with improved overall system efficiency, reducing computational overhead, memory pressure, and energy usage. Starfish employs two key techniques. First, it features split-process execution that uses a separate process to service vision library calls. Second, Starfish allows that process to serve all applications with function caching and shared memory distribution to reduce compute and memory redundancy.

We constrain our design to maintain *application transparency* for development: application developers use Starfish as though it is the original library, with no need to understand the Starfish system's underlying mechanisms. We also preserve *library transparency*: our design does not require modification of the library binaries, allowing Starfish to work with closed-source libraries and open-source libraries that can be rapidly-changing. Finally, we target application *performance preservation*: an application should perform as though it is a single application running without Starfish.

3.1 Split-process execution

Starfish splits a vision application's execution into a pair of processes, using the library call as the boundary. This allows Starfish to: (*i*) separate vision processing from the calling application; and (*ii*) use a single process to centrally service vision library calls across all applications. The library boundary provides transparent function indirection and allows Starfish to create a centralization point for shared computations.

3.1.1 Library Call Indirection

Because Starfish splits the application execution at the library boundary, it supports legacy applications and legacy libraries without requiring their source code. The API documentation and headers of library structures and functions provide important information that guide a Starfish implementation. Specifically, we can glean the following understanding from the API:

- *Memoizable Library Calls:* The API documentation can reveal which library calls are deterministic, computationally intensive, and widely useful among many applications. Such functions will be targets for Starfish to optimize. Other functions will be passed through to the original library for computation.
- *Object Structure:* The library's header files reveal the structure of library objects, allowing the Starfish implementation to specify efficient methods for communicating marshalled structures between applications.

The library boundary allows Starfish to use the above insights to ensure correctness and improve efficiency. Using the library as the boundary has a caveat: Starfish functions must strictly follow the API specification, taking expected library input and returning expected library output. This requires that Starfish does not alter object structures defined by the original library, e.g., by adding a member.

3.1.2 Centralization point for shared computation

By transferring library call computation out of the application process and into the central service process, Starfish creates opportunities to share common computations among multiple applications.

Starfish matches library calls with identical arguments, reusing computation results to reduce system-wide computational redundancy, as described in §5. Centralizing library calls also allows efficient library designs — such as hardware acceleration or offloading optimization described in §2.3 — to extend to all applications.

Furthermore, centralized service of library calls avoids possible security and privacy flaws introduced by sharing library call results among multiple untrusted applications. For example, a distributed computation scheme in which each application computes library calls and shares the output would require trust of all participating applications; an untrusted application could create accidentally or maliciously erroneous objects, which would then be propagated to other applications as putatively valid results. Instead, by using library calls themselves as the boundary and handling vision processing in the central process, Starfish provides tight security and control over library computation results.

Unfortunately, computing library calls on the central service introduces communication overhead, as input and output arguments need to be transferred between processes. We reduce this overhead by efficiently reusing arguments over shared memory regions, as discussed in §4.

3.2 Starfish overview

Using the vision library API as a boundary, we design Starfish, illustrated in Figure 2, as a split-process execution system to efficiently perform library computations.

The Starfish *Core* process centrally services a library call and caches its arguments (both input and output). The computer vision applications that use the library constitute the *Apps* of Starfish. Our execution model assumes that the Core and App processes run on an operating system that supports shared memory for inter-process communication.

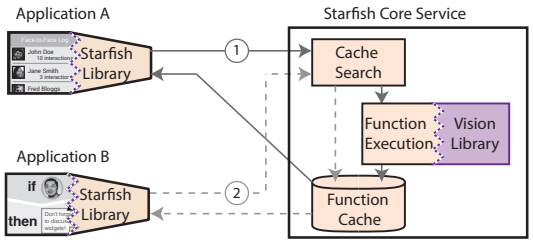


Figure 2: Apps compile to original vision library headers, but link to Starfish Library, which communicates with the Core to execute library calls. For a first library call ①, the Core uses the original library to compute the result, which is stored in the function cache and returned to the caller. For subsequent identical calls ②, the Core retrieves the cached result, reducing the overhead of redundant operations.

3.2.1 Starfish App

At runtime, an application's binary dynamically links with our *Starfish Library* in place of the original computer vision library. This linked pair of application binary and Starfish Library becomes the Starfish App. Dynamic linking is supported by all major languages (C/C++/Java/Obj-C/C#) to efficiently distribute and reuse third-party codebases.

To redirect library calls, the Starfish Library replaces the original library in the filesystem, intercepting the dynamic linking. We move the original library elsewhere for use by the Starfish Library and Core. This theoretically allows Starfish to support unmodified applications. However, to enable our Android + OpenCV implementation to distribute camera frames, Starfish requires the developer to change a single SurfaceView object in the layout file, as described in §6.2.

To implement the function behavior, the Starfish Library makes a request to the Core process to service library calls. Starfish blocks the calling thread on the App while waiting for the library call request to return. This preserves the behavior of original library call execution, which also blocks execution on the thread. This also allows the operating system to schedule other threads while the Core is handling the library call request.

3.2.2 Starfish Core

The Starfish Core is a central process that executes and tracks shared computations and objects among multiple Apps; it maintains a function cache of recently performed library calls, along with their input and output arguments. The Core links to the original vision library to execute the computations.

The Core is responsible for executing vision library calls. When an App makes a library call, the Starfish Library makes a library call request by passing input arguments to the Core. On the Core, library calls enter a queue of call requests. Multiple threads on the Core operate on requests, searching the function cache for an input argument match. If there is no match, the Core executes the function using the original vision library and marshals output arguments into the function cache, as ① in Figure 2. Otherwise, the Core retrieves output arguments from the cache, providing them to the calling App's Starfish Library to complete the library call, as ② in Figure 2.

3.2.3 Unsupported library calls

Because Starfish relies on library call arguments to pass states between Core and App, it does not support *functions that modify program/system state beyond the arguments*, e.g., functions that modify global variables or write to the filesystem; Starfish does not maintain a unified namespace across Apps.

Moreover, because the Core reduces computation redundancy by reusing library calls among multiple Apps, Starfish cannot support *non-deterministic functions*, such as those that use random variables or data from a changing network resource; providing cached results would violate correctness for non-deterministic functions.

Finally, Starfish does not efficiently support all library calls. For *lightweight functions* that are quick to compute, the overhead of retrieving values from the Core outweighs the benefit of computation reuse in some cases, e.g., for single pixel value changes. A Starfish implementation should choose not to support lightweight functions.

The implementation of the Starfish Library links to the original vision library to execute all unsupported functions in the App, returning the resulting values (not shown in Figure 2 for clarity). This fully preserves original library functionality and correctness.

Nevertheless, we observe that Starfish supports a significant portion of OpenCV vision functions; none of the functions in our evaluation benchmarks exhibit system-modifying or non-deterministic behavior. Blur, SURF, and face detection operations are some of the many deterministic compute-heavy operations that Starfish supports.

4. SPLIT-PROCESS EXECUTION

The Starfish Core services library calls as a separate process. This design naturally invites the use of remote procedure calls (RPCs) over shared memory. On top of standard RPC mechanisms, we design zero-copy strategies for argument passing and shared memory management that reduce the overhead of passing calls.

4.1 Argument passing via shared memory

While RPC argument passing has been researched and optimized since its invention [29], Starfish is subject to a unique set of constraints that are not met by existing solutions such as those used in [4, 21] or in the recent mobile system literature [10, 8, 6]. In particular, supporting unmodified applications and libraries prohibits the use of shared pointer structures. Furthermore, as the Starfish Library must be transparent to the application, Starfish must work with virtual addresses prescribed by each application; we cannot forcibly maintain a common virtual address space across multiple applications.

Instead, we design Starfish argument passing strategies over a reusable pool of shared memory regions for all Starfish Apps, guaranteeing both efficient and correct object marshalling. Specifically, Starfish minimizes the number of *deep copies* during argument passing. A deep copy involves a full traversal of an object's structure and is thus particularly expensive: a 640×360 color image takes approximately 3.5 ms to copy using `memcpy` on a Google Glass. Typical RPC solutions on library calls incur four deep copies: two per input argument and two per output argument. Starfish avoids the inefficiency of these deep copies by: (1) reusing input and output arguments across multiple applications, and (2) guiding function execution output directly into memory shared by the Core and the Apps.

4.1.1 Starfish marshalling sequence

Here we use a simple yet common code pattern in computer vision algorithms, `{b=foo(a); c=bar(b,a);}`, to explain how Starfish passes arguments efficiently. Both `foo()` and `bar()` are vision library functions. Figure 3 shows how the code works in six steps. When a Starfish App makes a library call request `b=foo(a)`, the Library and the Core follow a standard RPC sequence. First, the Library marshals a serialized representation of input argument `a`'s member fields into a shared memory region **(1)**. The Core un-

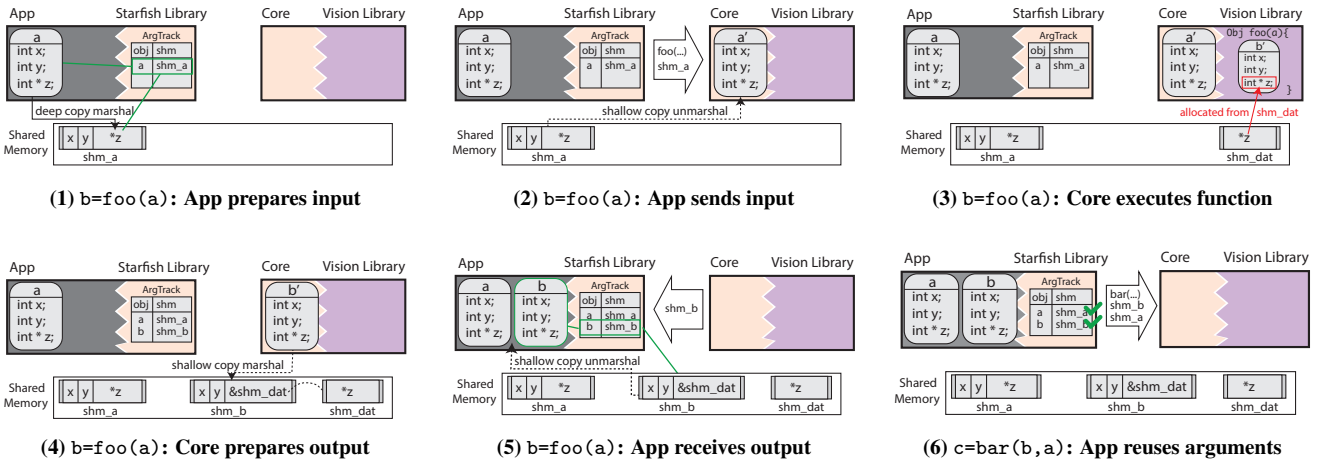


Figure 3: Example code: `{b=foo(a); c=bar(b,a);}`, where `foo()` and `bar()` are vision library functions. Starfish marshalling procedures minimize deep copies by tracking argument marshals and directly allocating library call output data into shared memory between the App and the Core.

marshals `a`, creating an object from the member fields **(2)**, and then executes `foo` using the vision library **(3)**. The Core marshals the output argument `b` **(4)**. Finally, the Library unmarshals `b` and supplies it as the return of the function **(5)**. The App reuses `a` and `b` for a subsequent call `bar(b,a)` **(6)**.

During unmarshalling, Starfish will direct member pointers to the data in the shared memory region, i.e., Starfish unmarshals using shallow copies. The Core and the Starfish Library mark all unmarshalled object data with Copy-on-Write, such that object modifications will not affect the physical memory, which may be shared with other Apps.

However, marshalling object data typically requires deep copies to write into the shared memory regions. We design Starfish to guide data creation directly into the shared memory regions and reuse previously-marshalled arguments, limiting this deep copy overhead of the marshalling procedure.

4.1.2 Direct data marshalling of library output

To reduce deep copying of library output. Starfish writes objects directly to shared memory regions during library execution, as shown in Figure 3:**(3)**. While the original library performs the library call, the Starfish Core intercepts calls to `malloc`. Starfish defines any memory allocation requested from the heap during a function execution to allocate memory from a shared memory region instead. While this may generate temporary intermediate shared memory allocations during execution, Starfish frees any non-output shared memory regions when the library call returns to minimize the shared memory footprint.

After the library function executes, the Core can marshal the specially-allocated output arguments by directing member pointers to addresses in the shared memory region, so no deep copying is required. This is illustrated in Figure 3:**(4)**.

4.1.3 Tracking arguments for shared memory reuse

The Starfish Library tracks opportunities to reuse arguments already marshalled into shared memory. If a previously-marshalled object has not been modified, then it is consistent with its marshalled representation in shared memory and does not need to be marshalled again. The Starfish App performs this tracking by maintaining *ArgTrack*, a map between objects in the virtual address space of the App and their shared memory region ids. The Starfish Library places a library call's input arguments into ArgTrack after marshalling the objects into their shared memory region, as shown in Figure 3:**(1)**. The Library also places output arguments into ArgTrack after unmarshalling the objects from their shared memory regions, as shown in Figure 3:**(5)**.

Starfish tracks an object's consistency with the shared memory region by applying `mprotect` on the argument objects on the App, trapping write operations to the object. If the application attempts a write access to a protected region, the Starfish Library removes the arguments from ArgTrack, lifts the memory protection, and allows the write.

Starfish references ArgTrack when marshalling input arguments; if an object has already been marshalled, the Library simply sends the shared memory region ID, removing the need for a deep copy for reused arguments, as shown in Figure 3: **(6)**. Searching an ArgTrack of 100 elements induces minimal overhead, consuming less than 100 µs. Similarly, trapping a write operation incurs a negligible overhead of only 1 ms. By removing the deep copies, ArgTrack greatly minimizes the Starfish marshaling overhead.

Thus, Starfish completely eliminates all deep copies from each library call except: (*i*) to transmit previously unused input arguments, (*ii*) to record an output argument if it writes data to a specified pointer address, and (*iii*) to maintain shared memory immutability for application correctness.

At worst, each library call uses two deep copies per input argument and 1 deep copy per output argument. However, as an application derives most input arguments from previous Starfish calls, Starfish can reuse the arguments, eliminating all deep copies in many cases. Minimizing deep copies greatly reduces the shared memory pressure and communication overhead for performing remote procedural calls.

4.2 Shared memory region management

Operating on a pool of shared memory regions allows the Starfish Core to reuse input and output arguments among many Apps. However, allocating a new shared memory region for each passed argument would be expensive; a Google Glass takes approximately 0.6 ms to perform a `MemoryHeapBase` allocation of a 128 KB region. Starfish must also judiciously use the limited heap memory provisioned by the Dalvik VM; Google Glass processes may have at most 192 MB of heap memory. Hence, we conservatively limit

the total size of all of the Starfish shared memory regions, e.g., to 128 MB. While the operating system handles heap management issues of memory allocation, e.g., physical memory fragmentation, the Starfish Core centrally manages the distribution of the shared memory regions for argument passing. Exploiting this central control, the Starfish Core adopts memory reuse techniques to reduce the cost of shared memory allocation, while maintaining a heap memory size limit.

To foster reuse of shared memory regions, the Core handles shared memory requests using free lists of reusable shared memory regions. Starfish designates a separate free list for a particular fixed memory size, e.g., 2 KB, 16 KB, 128 KB, 1 MB, 2 MB. Memory allocation requests pull empty regions from the free list of the smallest fixed size that would satisfy the request. Upon releasing the memory region, the Core returns the region to the free list for reuse. If the stack is depleted upon allocation request, the Core allocates a new shared memory region from the OS. Such fixed-size free lists, as used by the GCC standard library's multi-thread memory allocator [27] and by the classical buddy system memory management [16], allow for rapid reuse of memory regions, as applications recycle empty memory regions without redundantly allocating new physical memory or virtual memory regions. Fixed-size free lists come with the drawback of wasting space when rounding up to the next-largest free list region size. However, using a common fixed-size promotes the reusability of memory regions, which allows Starfish to reuse allocated shared memory.

To provision for dynamic region-size partitioning of the shared memory, Starfish does not fix the number of shared memory regions in each free list. Instead, Starfish handles each free list as a dynamically allocated stack, growing and shrinking as regions are added or removed.

If a shared memory region allocation would cause the Core to allocate more than the allotted total shared memory size (e.g., 128 MB), Starfish must free allocated memory regions. To maintain function reusability, the primary candidates for clearing are the empty regions in the free lists, regardless of size. However, if the free lists are vacant, Starfish then clears regions referenced only by the Core, ordered by least recent use. Starfish releases these least recently used regions until there is enough free memory to allocate a new region. These free list structures and policies allow Starfish to efficiently allocate and reuse shared memory regions within a fixed total memory size.

5. OPTIMIZING CENTRALIZED LIBRARY CALL EXECUTION

As library calls into the Starfish Core arrive from multiple applications, the Core tracks identical library calls and reuses computed results across multiple applications.

While exploiting this opportunity reduces computational redundancy and promotes memory reuse, it also introduces caching and concurrency challenges, including:

- Efficiently memoizing library calls across Starfish Apps;
- Enforcing consistent code behavior;
- Handling concurrent library call requests; and
- Promoting memoization through camera frame reuse.

We address these problems through thorough bookkeeping structures and policies, elaborated below.

5.1 Cached execution as function lookup table

The Core caches computed results to reuse library call outputs for identical library calls with identical arguments. We design *MemoCache*, a two-layer structure on the Core that tracks library call input and output arguments.

Because the search for a match must be efficient and accurate, we organize MemoCache as a function lookup table. At its top-level, MemoCache consists of a list of *function vectors*, each representing a library function symbol, e.g., `resize`. Each function vector contains a list of MemoCache entries, representing previous library calls to the library function. A MemoCache entry includes a list of pointers to its input argument shared memory regions and a list of pointers to its output argument shared memory regions.

Starfish consults the MemoCache to handle library call requests, as illustrated by Figure 4. The Core first retrieves the MemoCache function vector corresponding to the function symbol, creating the function vector if it does not exist ①. The Core then matches the request's input arguments against the contents of each entry in the function vector ②. Because each function vector maintains few entries (e.g., 100s), running a memory compare (`memcmp`) against entry arguments is significantly more efficient than performing a non-cryptographic hash on a library call's arguments. For example, on a Google Glass, we measure that computing a `fnv1` hash requires approximately 19 ms of processing on a 640×360 frame, whereas a `memcmp` requires only 6 µs.

If the request's arguments do not match any function vector entries, Starfish creates a new entry with the request's input arguments and inserts it into the function vector ③. It then executes the function and supplies the list of output arguments to the MemoCache entry ④. Starfish creates the entry prior to function execution to support concurrent library calls, as elaborated in §5.3. After the function execution, the Core returns the output to the App ⑥.

If instead the request's input arguments match those of a previous MemoCache entry, the Core retrieves the corresponding entry ⑤ and delivers its output arguments to the App ⑥.

As detailed in §4.2, Starfish clears shared memory segments to create room for free shared memory regions. If a freed segment serves as an input or output argument of the MemoCache entry, the Starfish Core clears the entry from its parent MemoCache function vector. This effectively serves as a Least-Recently-Used (LRU) cache eviction policy and limits the size of MemoCache.

5.2 Consistent computation return timing

The rationale behind caching is that executing a library function takes much longer than retrieving cached results. This also implies that when a Starfish App make a library call, it could take a very different amount of time to return. This opens a source code vulnerability due to code privacy timing attacks, similar to web privacy timing attacks from browser caching [9]. That is, if a library call returns immediately, a developer could determine that another application had recently requested the same library call with the same input arguments. These side channel timing attacks pose a threat to the security of the application' proprietary algorithms. Fortunately, attacks can be easily deterred by mimicking execution time across multiple library calls.

Thus, Starfish emulates library execution time for memoized functions. When executing a library call, ④ in Figure 4, Starfish monitors the time it takes to run the function, the *execution delay*, and stores it in the MemoCache entry. When another App is to receive memoized results, we only return the function after blocking execution for the same amount of time ⑤. By emulating execution delay, Starfish makes it indistinguishable whether the Starfish Library is

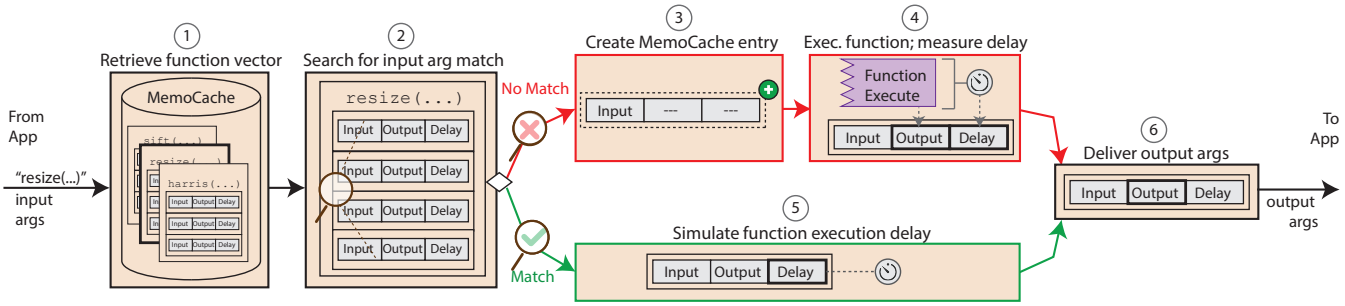


Figure 4: Starfish Core library call request. When receiving a library call request, the Core retrieves the corresponding MemoCache function vector ① and compares library call input arguments against the entries of the function vector ②. If there is no match, the Core creates a new MemoCache entry ③. The core then executes the vision function with the vision library, measuring the execution delay ④, and delivers the output ⑥. If there is a match, the Core simulates the execution delay ⑤, before delivering the matched entry's output ⑥.

returning a cached library call or performing the computation anew. As Starfish blocks an App's calling thread while waiting for the library call to return, the delay allows App threads to temporarily sleep, thus improving system efficiency.

5.3 Concurrent library call requests

The Starfish Core handles requests from multiple Starfish Apps, using multiple threads to operate on a queue of library calls in order of arrival. However, handling multiple asynchronous library call requests introduces two concurrency issues. First, multiple threads may wish to simultaneously read and write to a MemoCache function vector, introducing concurrency dependency conflicts. Second, a new library call request could arrive while another thread is computing the same function with identical arguments. Evaluating immediately would prohibit the use of memoized results, as they would not yet be ready.

Contention over a MemoCache function vector may be created by a thread accessing the function vector to search for an input argument match, while another thread alters the function vector by creating or deleting an entry. To solve this contention, the Starfish Core issues a readers-writer (RW) lock [7] on each function vector. RW locks allow simultaneous read operations, but require exclusive access for write operations. The Core issues a read-lock on a function vector while searching its input arguments, and obtains the write-lock when inserting new entries and deleting old entries. The RW lock thus allows concurrent searches over the function vector without contention, while only pausing to update the cache. To ensure timely updates, the Core issues write-preference on the RW lock; no new read locks are acquired while there are pending write locks. Using one RW lock per function vector is efficient; lock contention occurs only if there is a function symbol collision.

To handle concurrent library calls with identical arguments, the Core issues another lock on the delay element of the MemoCache entry – not the MemoCache entry itself – while the Core uses the original library to execute a function. This allows the Core to compare a library call request against any MemoCache entry's input arguments, but introduces a waiting period between concurrent requests; a library call request must wait for the completion of an earlier identical library call request. Starfish then can insert the execution delay from §5.2, subtracting the time spent waiting for the first library call to complete. After the delay, Starfish uses the output arguments as the results for the second library call.

Thus, we guarantee thread safety on the MemoCache structures, while allowing non-interfering concurrent execution to proceed. As our entry delay locks are independent, the system is deadlock-free.

5.4 Camera frame reuse

As camera frames are the root data of vision tasks, Starfish establishes *frame coalescing* to share frames between applications to maximize function and memory reuse. Because of the lenient timing allowance of computer vision applications, Starfish can relax the timing of frame requests, i.e., Starfish can supply a recently captured frame, or wait for the next frame to be captured.

Starfish does this by optionally allowing the application developer to annotate their frame capture with a *freshness* latency and a *patience* latency. When a App requests a frame, if Starfish has previously supplied a frame (to another App) within the freshness latency, we deliver it to the requesting App. Otherwise, the App waits for the patience latency. If Starfish captures a frame while the App is waiting, then it is delivered to the waiting App. If the patience latency expires and Starfish has not received a frame, Starfish initiates a capture to supply the frame immediately.

Depending on the App, different freshness and patience latency bounds may be justified. Developers of object recognition tasks may not be concerned with the real-time immediacy of the vision task, while motion estimation applications may have precise timing conditions. Without developer annotation, the Core measures the time between frame capture requests, and uses this as the freshness latency, and sets the patience latency to 0. This satisfies typical image request contracts, as the frame rate is usually processing bound.

Sharing the coalesced frame – and all its computed derivatives – significantly reduces the computation overhead, memory allocation, shared memory burden, and marshalling expense on the system.

6. IMPLEMENTATION

The Starfish system design described above is agnostic to vision library, device, operating system, and programming language, provided that there are inter-process communication and shared memory protocols, and that the vision library can be dynamically linked. We implement the Starfish System to retrofit OpenCV's C++ library on Android 4.4.2 "KitKat" to demonstrate its feasibility. As OpenCV Android Java calls operate using native JNI/C++ calls, our implementation naturally supports the OpenCV Android library. Although OpenCV is open-source, we treat its C++ library as closed-source, not altering any OpenCV code. However, we intercept Java support files to capture and distribute camera frames, explored in §6.2. Here, we discuss details specific to our Android implementation of Starfish. We evaluate this Starfish implementation performance on a Google Glass, explored in §7.

```
1 void resize(const Mat& src, Mat& dst, Size *
      dsize, double fx=0, double fy=0, int
      interp=INTER_LINEAR ){
2
3   //Input Marshallers and Inputs
4   vector<pair<IMarshaller*,void *> > mIn;
5   mIn.push_back(make_pair(&MatM,&src));
6   mIn.push_back(make_pair(&SizeM,(void*)dsize
        ));
7   mIn.push_back(make_pair(&DoubleM,&fx));
8   mIn.push_back(make_pair(&DoubleM,&fy));
9   mIn.push_back(make_pair(&intM,&interp ));
10
11  //Output Marshallers and Outputs
12  vector<pair<IMarshaller*,void *> > mOut;
13  mOut.push_back(make_pair(&MatM,&dst));
14
15  //Function Request
16  starfishCall(CV_RESIZE, mIns, mOut);
17 }
```

Listing 1: Library hook function wrapper for `resize` library call. Library hooks simply involve forwarding the correct arguments along with their object marshallers.

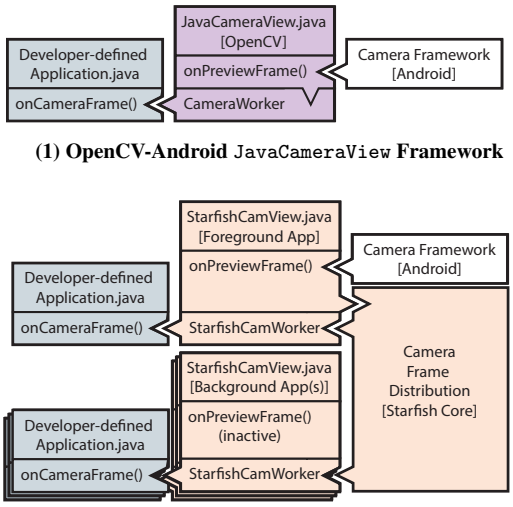


Figure 5: `StarfishCamView` replaces OpenCV's `JavaCameraView` to multiplex the single-application frame delivery to all Starfish Apps. Starfish defines the `onPreviewFrame()` callback to transfer frames to the Starfish Core. The Core sends frames to `StarfishCamWorker`s on all foreground and background Apps. Each `StarfishCamWorker` calls its App's developer-defined `onCameraFrame()` callback.

In our implementation, the base of the Core consists of 1099 lines of C++ code, while the Starfish Library 489. The StarfishCameraView.java camera distribution framework consists of 396 lines of Java code. These sets of source code can be reused to support additional libraries with little to no modification.

On top of the base code, Starfish marshalling supports 19 OpenCV objects with 853 lines of source code and 22 OpenCV functions using 1748 lines of source code in C++. This code is shared for the Starfish Core and Library. While seemingly large, the marshalling codebase is simple to write, and adding new marshallable objects and functions is straightforward, as shown in §6.1. Many lines of code are spent defining repetitive object oriented prototypes.

We compile the Starfish Library into a `.so` file to replace the OpenCV library. The developer's application links to this file to become the Starfish App. Meanwhile, we compile and run the Starfish Core as an Android background service.

Android specifies `Binder` as their inter-process communication interface. We implement the Starfish Core as an Android Binder service to receive calls from the Starfish Library. Starfish uses Android's `MemoryHeapBase` to generate shared memory segments for use across the Binder. These standard Android protocols serve as the substrate for our inter-process communication.

6.1 OpenCV function hooks

The Starfish Library intercepts OpenCV calls through library hooks, providing implementations for the OpenCV function headers. To complete the call, each library hook must marshal objects into shared memory and transmit them over the `Binder` interface. We simplify the library hook code itself to ease the development of additional library hooks into the Starfish Library.

To do this, we standardize an object marshaller interface `IMarshaller`. Each object marshaller has three functions: (1) prescribe the required size of a shared memory region, (2) serialize an object into a region, and (3) deserialize an object from a region. We provide an `IMarshaller` implementation for each argument data type.

To reduce source code redundancy in our implementation, we also create a reusable function `starfishCall()` to handle the duties of allocating shared memory regions for the inputs, marshalling input arguments, structuring the `Binder` transactions to the Starfish Core to execute the function, and unmarshalling the output arguments. `starfishCall` requires a function handler, a list of input objects and their object marshallers, and a list of output objects and their object marshallers. Then, as it simply needs to cycle through the lists of arguments and their marshallers, the `starfishCall` is able to generically serve any Starfish library call.

The `starfishCall` and `IMarshaller` thus reduce the effort of developing additional OpenCV library hooks. The implementation of a library hook simply needs to form input and output argument lists with their object marshallers, and call the `starfishCall` function, as shown in Listing 5.4.

6.2 Android camera frame distribution

At the root of all vision algorithms is the camera frame, containing raw pixel scene data. Starfish employs a frame distribution system that supports multiple applications under the existing developer interface while increasing the opportunity for frame memory reuse.

The camera device service on Android binds to a single application running in the foreground. Fortunately, because the OpenCV framework manages frame captures, we are able to augment the existing framework to use Starfish to capture and supply camera frames, allowing for centralized distribution of captured frames.

6.2.1 Legacy Android + OpenCV Frame Capture

OpenCV provides a camera framework for Android development that operates with minimal developer intervention, shown in Figure 5 **(1)**. The developer specifies code to run on incoming frames by providing an `onCameraFrame()` callback function with code to be run for each incoming frame. The developer also specifies an OpenCV `JavaCameraView` SurfaceView on which to draw results.

The OpenCV Java support files provide mechanisms to route camera frames to an application's callback. The `JavaCameraView` captures the frame by providing the `onPreviewFrame()` callback function and a foreground SurfaceView (itself) to the Android Cam-

era Framework. From the callback, it writes incoming frames into an OpenCV `InputFrame` buffer. A separate `CameraWorker` thread continuously runs the developer-specified `onCameraFrame` callback function on any incoming frames. Frames are captured continuously to limit latency, but the `CameraWorker` only triggers `onCameraFrame` after a previous `onCameraFrame` has completed. When an application leaves the foreground, the camera service pauses, preventing `onPreviewFrame` and `onCameraFrame` callbacks from triggering.

6.2.2 Starfish Camera Frame Distribution

We design a frame distribution system, illustrated in Figure 5 **(2)**, to provide camera frames to all running Starfish Apps while maintaining the same simple OpenCV callback interface. We do this by overriding the `JavaCameraView` object with our `StarfishCameraView`, which forces the foreground App to pipe frame captures through the Starfish Core. In the `StarfishCameraView` file, we use shared memory regions as our frame buffers and declare `onPreviewFrame` to deliver frames to the Core. Changing the JavaCameraView to the StarfishCameraView requires developer intervention in the Android XML layout file of their application.

As with the legacy system, we launch a `CameraWorker` thread from the StarfishCameraView. The CameraWorker requests and receives frames from the Core and calls the developer's `onCameraFrame` callback on received frames. This maintains the frame processing operation expected by the developer.

Backgrounding an application halts the Android camera service, pausing all vision activity. However, when another application's StarfishCameraView is brought to the foreground, its onPreviewFrame will resume providing frames to the Core, allowing all Starfish activity to resume. Thus, Starfish frame distribution can transition seamlessly between applications as long as at least one Starfish application is in the foreground with control of the camera.

The Starfish Core marshals frames through the same split-process shared memory system discussed in §4. Starfish calls can thus reuse frames in the Core without further frame marshalling or allocation.

6.3 Single-App mode

As characterized in our evaluation in §7, passing arguments to the Core presents a computational overhead – 7.2 ms per library call on our Google Glass implementation. While this overhead is negligible compared to the efficiency of sharing vision computations across multiple applications, the overhead substantially degrades the performance when running a single Starfish App. Thus, if there is only one active Starfish App, the Core tells the Starfish Library to enter Single App Mode, simply using the original vision library to execute library calls on the App. This eliminates the overhead of Starfish communication to preserve the application performance of the single vision application.

During our characterization of Starfish in §7, we disable Single App Mode to understand the overhead of Starfish on the vision applications.

7. EVALUATION

We evaluate our Starfish implementation on a Google Glass, powered by a TI OMAP4430 with a dual-core Cortex-A9. Because dynamic frequency scaling can introduce uncontrollable factors in our measurements, we pin the CPU clock frequency of our evaluated Glass to a constant 600 MHz. Our evaluation seeks to answer the following pair of questions:

How effective are major design decisions by Starfish in achieving their objectives? We evaluate our design decisions by performing a per-call analysis of the computation time of Starfish library calls. We analyze the impact of our optimizations discussed in §4 and §5 to validate the effectiveness of the marshalling reuse, shared memory re-allocation, and cache search strategies.

What is the computational benefit of Starfish with concurrent applications? We quantitatively analyze the impact of Starfish on multiple sets of concurrent applications running vision tasks, employing metrics of CPU utilization (using `mpstat`) and processing time for each frame. We run benchmarks of concurrent applications that employ the same vision tasks and concurrent applications that do not. This displays the range of benefits possible with Starfish. We also test the scalability of Starfish, evaluating its performance ability on many simultaneous applications and its effect on the power draw of concurrent vision applications.

7.1 Starfish per-call overhead

We microbenchmark a SURF feature `detect()` library call on a 160×90 frame, producing a vector of keypoints. The computationally intensive detect library call takes 214.5 ms, exhibiting the benefits of caching expensive library calls.

We also microbenchmark a `resize()` library call, reducing a 640×360 color image to 160×90. The `resize()` library call uses an entire frame as an input argument and produces another frame as an output argument, challenging the Starfish memory allocation and marshalling overhead. Furthermore, the relatively quick processing time of the native `resize()` function execution, taking 20.9 ms, highlights the need to optimize Starfish overhead.

We do a per-call analysis on our microbenchmarks by averaging 15 instances of library calls. We execute benchmarks for library calls before and after optimizations, for the first library call and subsequent library calls. We list the benchmark results in Table 1 and chart the results in Figure 6. While unoptimized Starfish introduces a substantial overhead of as much as 20.89 ms per call, our design optimizations reduce Starfish overhead by 55% – 75%.

7.1.1 Impact of argument and memory reuse

The transfer of input and output arguments dominates the unoptimized Starfish overhead, occupying 84% of the processing time. This overhead can be almost entirely eliminated through our marshalling optimizations from §4.

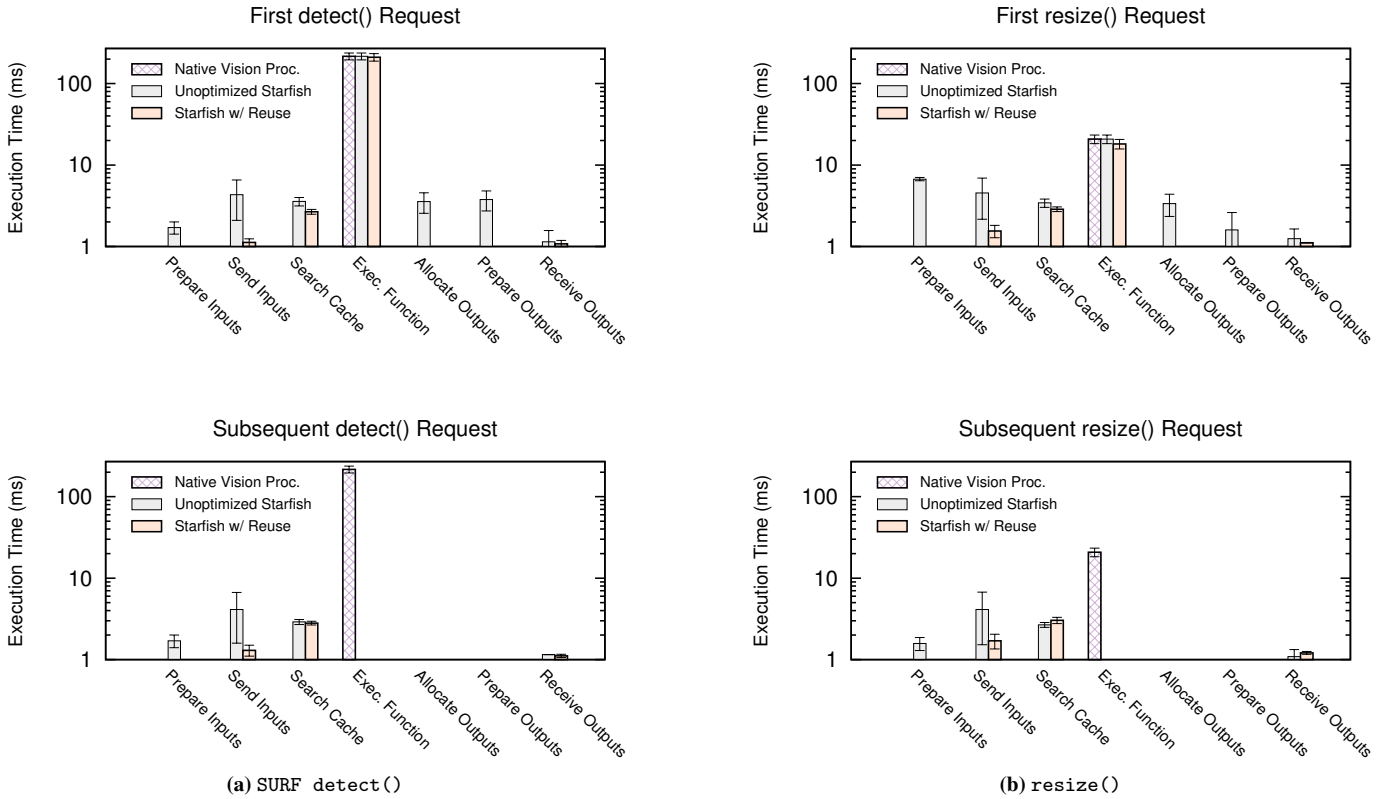
Using ArgTrack allows Starfish Apps to track and reuse already-marshalled arguments. This includes input frames, which are received from the Core and are thus already in shared memory. By sending the shared memory id from ArgTrack to send to the Core, the App completely evades the 4.5 ms overhead of marshalling the input arguments. Searching ArgTrack creates negligible overhead, on the order of hundreds of microseconds, in marshalling inputs and receiving outputs.

Unoptimized output arguments also pose a large overhead, requiring 3.36 ms to allocate and another 1.60 ms to deep copy. However, our Starfish Core directly allocates library outputs into shared memory during function execution, eliminating the overhead to allocate the shared memory region and deep copy the argument objects. Furthermore, because Starfish re-allocates shared memory regions, objects are allocated quicker, reducing the function execution time as well.

The combination of these remote execution optimization designs reduces the per-call 16.4 ms argument-passing overhead to only 3.0 ms for each library call.

7.1.2 Impact of cache search strategies

The MemoCache reduces the overhead of computing multiple identical library calls with identical arguments. Even with an unoptimized cache search, this poses great benefits, as Starfish Apps



(a) `SURF detect()` **(b)** `resize()`

Figure 6: Execution time of `resize` and `detect` library call requests, with and without marshalling reuse and shared memory reuse optimizations. Subsequent requests made by a separate App showcase the low cache retrieval overhead compared to the intensive library call.

avoid the large computational overhead of the function execution – 20 ms for `resize()`, and 215 ms for `detect()`. Starfish uses cache design described in §5 to further minimize the cache search overhead.

Our unoptimized Starfish MemoCache uses a lookup table structure with argument hash vectors to perform its searches. An `fnv1` hash operates on input arguments to organize the cache entries by hash value. This performs with a cache search time of 3.5 ms. By contrast, our decision to organize the lookup table by function vector only involves a direct lookup on the function symbol, without requiring any computation. We search the function vector by using `memcmp` on the entries. As the function vector size is fairly small, including only 10s to 100s of entries, this poses a smaller overhead than the `fnv1` hash computation. The result is a 16% reduction in cache search time to 2.9 ms.

In summary, using our design optimizations in argument reuse, shared memory reallocation, and cache structure, Starfish reduces the per-call overhead from 20.9 ms to only 4.2 ms. This lowers the computation time of repeated `resize()` calls by 71% and repeated `detect()` calls by 97%, reducing per-frame processing time by 15 ms and 210 ms respectively.

7.2 Multi-App performance

We next analyze the ability of Starfish to service concurrent Apps. We employ foundational vision tasks as benchmarks to examine the benefits and overhead of Starfish on Apps that do and do not use the same vision tasks.

7.2.1 Benchmark vision tasks

We evaluate Starfish App performance using benchmark vision tasks of Face Recognition, Object Recognition, and Scene Geometry. Apps employ these vision tasks to perform a diverse set of duties, as described below. While these tasks do not comprehensively cover the expansive OpenCV library, these selected computations serve as a basis for many vision applications that can potentially run concurrently. In our evaluation, each of our Apps receives the camera stream through the Starfish frame distribution. The App sends the frame through each benchmark vision task, which begins by converting the frame to grayscale, and resizing it to 160×90. We include these pre-processing steps in our evaluation results.

Face recognition algorithms detect and identify individuals in an image, useful for applications such as social network taggers and alert notification services. Face recognition consists of two components: *detection* and *classification*. The Viola-Jones face detection algorithm compares a computed integral image against a cascade of pre-trained Haar-like classifiers. If a region passes all cascade layers, the algorithm determines it to be the region containing a face. Face classification identifies detected faces by matching against a tagged dataset of faces. The Local Binary Pattern-based algorithm compares a matrix of spatial histograms from the image with a database of pre-trained spatial histogram matrices. The nearest-neighbor is determined to be the identity. Under controlled lighting and well-posed images, the face classification operates with 97% accuracy [1]. On the Glass, running natively outside of Starfish, face recognition performs at 4.36 frames per second (FPS) with a memory footprint of 61.28 MB.

Table 1: Starfish execution time for first and subsequent calls to the same library functions. Reuse optimizations reduce Starfish overhead by 36%–71%

	`detect()`				`resize()`			
	Unoptimized Request	Starfish Reuse	Subsequent Request	Subsequent w/ Reuse	Unoptimized Request	Starfish Reuse	Subsequent request	Subsequent w/ Reuse
Prepare Inputs	1.71 ± 0.29	0.10 ± 0.01	1.70 ± 0.30	0.09 ± 0.02	6.72 ± 0.31	0.11 ± 0.02	1.58 ± 0.29	0.15 ± 0.03
Send Inputs	4.33 ± 2.24	1.12 ± 0.12	4.13 ± 2.54	1.30 ± 0.20	4.54 ± 2.38	1.55 ± 0.27	4.13 ± 2.61	1.70 ± 0.35
Search MemoCache	3.57 ± 0.42	2.67 ± 0.18	2.90 ± 0.20	2.81 ± 0.15	3.43 ± 0.40	2.87 ± 0.19	2.67 ± 0.18	3.04 ± 0.27
Execute Function	216.46 ± 20.84	210.94 ± 22.07	–	–	20.85 ± 2.56	18.17 ± 2.43	–	–
Allocate Out	3.56 ± 1.00	–	–	–	3.36 ± 1.02	–	–	–
Marshall Out	3.77 ± 1.04	0.25 ± 0.03	–	–	1.60 ± 1.01	0.21 ± 0.05	–	–
Transfer Out	1.14 ± 0.43	1.08 ± 0.11	1.15 ± 0.11	1.11 ± 0.05	1.25 ± 0.39	1.11 ± 0.21	1.09 ± 0.24	1.21 ± 0.05
Total Exec. Time	234.55 ± 26.26	216.15 ± 14.00	9.88 ± 1.07	5.31 ± 0.12	41.74 ± 3.60	24.02 ± 4.20	9.47 ± 0.97	6.10 ± 0.41
Starfish Overhead	18.08 ± 3.19	5.21 ± 0.38	9.88 ± 1.07	5.31 ± 0.12	20.89 ± 2.08	5.85 ± 0.39	9.47 ± 0.97	6.10 ± 0.41

Homography mapping estimates a camera's relative motion is through plane-to-plane mappings, which can be useful for applications such as indoor map localization and obstacle awareness. To compute a homography, we compute SURF corner points and match them between two frames in a camera stream. On the matched pairs, we use the RANSAC algorithm and a least squares regression to generate a homography matrix, revealing the geometric transform between two camera frames. Running natively, scene homographies performs at 3.54 FPS with a memory footprint of 59.97 MB.

Object detection recognizes food items, brand logos, and office keys, enhancing applications such as life-loggers and reminder notifications. A popular algorithm for object recognition uses Bag-of-Words (BoW), which forms a collection of salient visual features and compares them to a pre-trained dataset. We used SURF to form the input vector of corners and feature descriptors. These feature descriptors are then compared against the BoW dataset using a Support Vector Machine. Running natively, object recognition performs at 1.17 FPS with a memory footprint of 59.7 MB.

7.2.2 Overhead and benefits on App combinations

When running the Starfish Apps at a fixed 600 MHz clock frequency on the Google Glass, `mpstat` indicated that the system maintained a CPU utilization average of approximately 60%, regardless of the number and type of applications running. This is likely due to thermal throttling. However, the application workloads significantly affect the time it takes to process each frame through vision computations, impacting the effective frame rate. We use the processing time as our metric to evaluate Starfish performance. We list the processing time performance in Table 2.

Our benchmarks consist of seven combinations of Apps running vision tasks. In a potential scenario, a mobile device would run simultaneous background applications running multiple vision tasks, e.g., logging social interactions with face recognition, memorize object placement with object recognition, observing scene changes with homography fitting. This combination is represented by the 7th row of Table 2.

Different combinations of vision tasks hold different potentials for library call sharing. Object recognition and homography fitting both utilize SURF computations on the input frame but utilize different library calls later in their stream (RANSAC and SVM). Face recognition uses a different set of features entirely, computing Viola Jones and LBPH algorithms to detect and classify faces. All of our benchmarks resize the input frame and convert it to grayscale. We also simultaneously run two instances of the benchmarks (column x2). This represents running multiple Apps using the same vision task, such as a photo tagger and a lifelogger both using face recognition.

Our benchmarks reveal the overhead of running Starfish. In isolation, Starfish adds at most 30 ms of processing time per frame. At this low overhead, the performance impact of running Starfish on a single vision task only increases frame processing time by 10% at most. (Although, as mentioned in §6.3, one should not activate Starfish for use with a single App.)

Starfish's efficiency benefits become apparent when running multiple identical vision tasks. While native processing time is doubled when running two tasks, Starfish maintains the performance within 22% of the original frame processing time. This reduces the per-frame processing time of running two identical vision task instances by 35%–41%. Non-identical combinations combinations of applications also benefited from Starfish efficiency. Through sharing the `resize()` library call on the input frame; Starfish decreased the processing time of Homog/Face and Face/Obj by 5.5% and 4.7% respectively. Starfish performed best when sharing SURF features for Homog/Obj, reducing processing time by 22%. The combination of Homog/Face/Obj fell inbetween these, reducing processing time by 9.7%.

Computational benefits continue to scale with additional vision tasks. As shown in Figure 7, natively running additional face recognition Apps significantly limits the frame rate, dropping to 0.3 frames per second when running 10 Apps. However, using Starfish, the frame rate degrades at a much reduced pace, sustaining 1.8 frames per second when running 10 Apps. Thus, Starfish provides App scalability, as many Apps can run with minimally impacted frame rate performance.

7.2.3 CPU utilization and system efficiency

Stemming from constant activity, high CPU utilization contributes to the high power draw of systems running vision applications. For tasks with relaxed frame rates, the Starfish reduction of processing time reduces the CPU utilization, and in turn, the energy consumption and heat generation of the system.

We characterize this by running face recognition vision tasks at a fixed frame rate of 0.3 FPS. At this rate, a single application's average CPU utilization is 5.3%, processing at 60% CPU utilization for 230 ms, then idling near 1% for the remainder of a frame period. Without Starfish, the CPU utilization will grow linearly with the number of application instances, as each App will need to perform its own 230 ms of processing. Starfish allows multiple applications to share the same processing; when running 10 simultaneous face recognition applications at 0.3 FPS, Starfish keeps the average CPU utilization below 13%, limiting the power draw to an average of about 1210 mW, as measured with a Monsoon Power

Table 2: Average processing time per frame (seconds) on homography, face recognition, and object recognition benchmarks, running natively (NAT) and with Starfish (SF). x1 runs a single benchmark instance; x2 runs a pair of instances.

Benchmarks	NAT x1	NAT x2	SF x1	SF x2
Homog	0.282	0.513	0.310	0.332
Face	0.230	0.438	0.239	0.271
Obj	0.855	1.692	0.864	1.001
Homog/Face	0.577	1.135	0.547	0.611
Face/Obj	0.496	1.014	0.472	0.525
Homog/Obj	1.185	2.202	0.918	1.012
Homog/Obj/Face	1.288	2.616	1.164	1.355

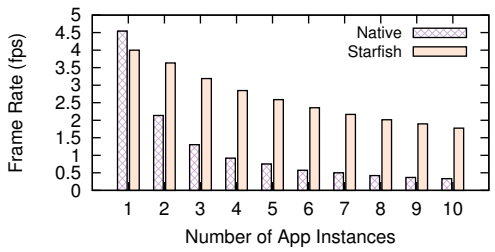


Figure 7: Maximum frame rate of running simultaneous face recognition apps, with and without Starfish.

Monitor. The same applications without Starfish natively draw an average of 2263 mW at 60% CPU utilization. The reduced power overhead also reduces the device heat generation. As reported in [18], each watt of average power consumption constitutes a 11 °C increase in surface temperature. Thus, Starfish provides more comfortable use and longer battery life to systems running concurrent vision applications with shared library calls.

In summary, we find that Starfish provides efficient, scalable support for multiple simultaneous vision Apps. The argument reuse, shared memory reallocation, and cache structure designs significantly reduce the per-call overhead of Starfish, which in turn allows for applications to transparently and efficiently share library computation results.

8. RELATED WORK

Three groups of research work are related to Starfish. First, *remote execution* or *offloading* partitions applications to run on disjoint resources, such as the cloud or cloudlet. Starfish has a different objective and faces different challenges. The goal of eliminating redundant computation by vision tasks across multiple applications dictates that Starfish use library calls as the boundary of splitting, instead of VMs, e.g., Gabriel [12], or threads, e.g., COMET [10] and CloneCloud [6]. Our need to support unmodified applications and libraries prevents the use of new programming models, e.g., Odessa [24]. MAUI [8] is related in its use of library call boundary and support of largely unmodified applications. Starfish, however, leverages that the Core and Apps run on the same system and optimizes RPC over shared memory, as shown in §4.

Second, *privilege separation* splits computations across processes to isolate data privileges to specific parts of an application for security purposes [5, 22, 30]. Some of these works actually split a program across the boundary of a library, like Starfish. For example, in [22], Murray and Hand use disaggregated libraries to run an application across multiple Xen hypervisor domains. This provides complete isolated protection between the host process and the disaggregated library. To share data, they use a shared memory area which only grants control to the library upon execution attempts. CodeJail [30] separates library calls into a separate process to isolate memory interactions. It can switch between trusted and untrusted library memory to ensure that library interactions only modify expected memory. Starfish, in contrast, does not place trust restrictions on the data provided to each App or the Core library processing. As vision algorithms only operate on the input and output arguments, all updates are provided through the library calls. Thus, as any state changes are encoded as output arguments of the function request, the Starfish Core does not alter existing App memory. Furthermore, each App has full privilege to read and write to its memory; Copy-on-Write preserves shared memory alterations from affecting other Apps. This allows Starfish to supply output arguments across many applications through our function cache, as described in §5.

Finally, reusing computation results, widely known as *memoization*, is an established computing methodology, as embodied by the dynamic programming paradigm. Indeed, advanced techniques exist for single applications, including automatic creation of memoization functions [13, 19] and dynamic analysis to determine the effectiveness of caching memoized results [11]. However, these techniques avoid complex objects and data structures in the input and output parameters of the functions. Additionally, the memoization takes place within a single application, whereas Starfish faces the challenge of maintaining consistent application performance and preventing developers from determining whether an object was cached or not, as shown in §5.2.

9. DISCUSSION

Library modification: We avoid modifying the vision library to preserve existing function behavior. To maintain backward compatibility, libraries rarely change APIs, even with iterative versioning. As of November 2014, OpenCV function headers have remained identical since OpenCV version 2.4.0, released April 2012 [23]. However, it is possible to fork OpenCV source code, and library integration would allow for further optimization. For example, data structures with shared pointers could be ordered for rapid marshalling.

Developer annotation: Starfish targets developer transparency, such that developers need not know the mechanisms underlying Starfish. However, developers may wish to publicize which vision library calls their applications use. By using annotation to relax developer privacy, this would allow for improved joint application performance as different developers share common library calls. Additionally, a slight difference in specific parameters of a library call will prevent an application from leveraging a previous computation. Developers could annotate a range of parameters that satisfies their function request. This would allow Starfish to select parameters that maximize computational sharing.

Optimizing Core execution: As mentioned in the related work, devices can offload execution to a network for efficiency. Devices can also use increasingly present vision hardware and heterogeneous units; the upcoming OpenVX API will increase the availability of vision hardware. The Starfish Core can act as the central hub for offloaded processing, aggregating decisions regarding tradeoffs in latency, efficiency, and workload across all library calls. The Core could also use centralized knowledge to improve system cache efficiency for the hardware resources, e.g., by ordering operations for optimal spatial reuse of the memory cache.

Broader use of Starfish: Beyond vision libraries, Starfish can enhance other libraries that are computationally intensive with frequent calls to deterministic functions, e.g., those processing sensor data such as speech and gesture recognition. We foresee that adoption of wearable devices will warrant more concurrent always-on tasks that are serviced by such sensor-processing libraries. In addition to efficiency benefits, Starfish also helps guard applications from untrusted libraries. With Starfish, an application can contain untrusted libraries in a separate process (the Core) and only needs to send the latter the data relevant to library computation.

10. CONCLUSION

Starfish employs split-process execution to reduce redundant vision computations across multiple applications. The Starfish design incorporates argument reuse, shared memory reallocation, and caching policies to minimize the overhead of operating on library call requests. We demonstrate Starfish's effectiveness through a Google Glass implementation, evaluating the system performance of running concurrent vision applications. We demonstrate that Starfish provides scalable benefits with additional applications; with a workload of two to ten concurrent face recognition applications that typically increases the computational burden from 10% to 60% CPU utilization, Starfish maintains a CPU utilization under 13%, which in turn reduces system power consumption by 19% – 58%, as Starfish maintains a power consumption at approximately 1210 mW. Starfish allows concurrent vision applications to run with efficient system performance, thus contributing an important step towards efficient wearable vision systems.

Acknowledgments

The authors thank Edward Hickman for his assistance in developing OpenCV marshalling code and benchmark tasks for Starfish. The authors are grateful for comments made by anonymous reviewers and the paper shepherd Dr. Rajesh Balan. This work was supported in part by NSF Awards CNS #1054693, CNS #1218041, and CNS #1422312. Robert LiKamWa was supported in part by a Texas Instruments Graduate Fellowship.

References

[1] Timo Ahonen, Abdenour Hadid, and Matti Pietikäinen. Face recognition with local binary patterns. In *Proc. European Conf. on Computer Vision (ECCV)*, 2004.

[2] Clemens Arth, Florian Limberger, and Horst Bischof. Real-time license plate recognition on an embedded DSP-platform. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2007.

[3] Paramvir Bahl, Matthai Philipose, and Lin Zhong. Vision: cloud-powered sight for all: showing the cloud what you see. In *Proc. ACM Workshop on Mobile Cloud Computing and Services (MCCS)*, 2012.

[4] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, 1990.

[5] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proc. USENIX Security Symp.*, 2004.

[6] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proc. European Conf. on Computer Systems (EuroSys)*, 2011.

[7] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[8] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making smartphones last longer with code offload. In *Proc. ACM Int'l. Conf. on Mobile Systems, Applications, & Services (MobiSys)*, 2010.

[9] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proc. ACM Conf. on Computer and Communications Security (CCS)*, 2000.

[10] Mark S Gordon, Davoud Anoushe Jamshidi, Scott A Mahlke, Zhuoqing Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2012.

[11] Philip J. Guo and Dawson Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *Proc. ACM Int'l. Symp. on Software Testing and Analysis*, 2011.

[12] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proc. ACM Int'l. Conf. on Mobile Systems, Applications, & Services (MobiSys)*, 2014.

[13] Marty Hall and J. Paul McNamee. Improving software performance with automatic memoization. *Johns Hopkins APL Technical Digest*, 18(2):255, 1997.

[14] Richard Hull, Philip Neaves, and James Bedford-Roberts. Towards situated computing. In *Proc. ACM Int'l. Symp. on Wearable Computers (ISWC)*, 1997.

[15] Itseez. OpenCV Computer Vision Library. `http://opencv.org`.

[16] Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[17] Robert LiKamWa, Bodhi Priyantha, Matthai Philipose, Lin Zhong, and Paramvir Bahl. Energy characterization and optimization of image sensing toward continuous mobile vision. In *Proc. ACM Int'l. Conf. on Mobile Systems, Applications, & Services (MobiSys)*, 2013.

[18] Robert LiKamWa, Zhen Wang, Aaron Carroll, Felix Xiaozhu Lin, and Lin Zhong. Draining our glass: An energy and heat characterization of Google Glass. In *Proc. Asia-Pacific Work. on Systems (APSys)*, 2014.

[19] Paul McNamee and Marty Hall. Developing a tool for memoizing functions in C++. *ACM SIGPLAN Notices*, 33(8):17–22, 1998.

[20] David Moloney, Brendan Barry, Fergal Connor, Martin O'Riordan, Cormac Brick, David Donohoe, David Nicholls, Richard Richmond, and Vasile Toma. Always-on vision processing unit (VPU) for mobile applications. *IEEE Micro*, 2014.

[21] Gilles Muller, Renaud Marlet, E-N Volanschi, Charles Consel, Calton Pu, and Ashvin Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proc. IEEE Int'l. Conf. on Distributed Computing Systems (ICDCS)*, 1998.

[22] Derek G. Murray and Steven Hand. Privilege separation made easy: trusting small libraries not big processes. In *Proc. European Work. on System Security*, 2008.

[23] Andrey Ponomarenko. API changes/compatibility report for the OpenCV library. `http://upstream.rosalinux.ru/versions/opencv.html`.

[24] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proc. ACM Int'l. Conf. on Mobile Systems, Applications, & Services (MobiSys)*, 2011.

[25] Swati Rallapalli, Aishwarya Ganesan, Krishna Chintalapudi, Venkat N. Padmanabhan, and Lili Qiu. Enabling physical analytics in retail stores using smart glasses. In *Proc. ACM Int'l. Conf. on Mobile Computing & Networking (MobiCom)*, 2014.

[26] Abigail J. Sellen, Andrew Fogg, Mike Aitken, Steve Hodges, Carsten Rother, and Ken Wood. Do life-logging technologies support memory for the past?: an experimental study using Sensecam. In *Proc. ACM SIGCHI Human Factors in Computing Systems*, 2007.

[27] Richard M. Stallman and GCC DeveloperCommunity. *Using The GNU Compiler Collection: A GNU Manual For GCC Version 4.3.3*. CreateSpace, Paramount, CA, 2009.

[28] Zak Stone, Todd Zickler, and Trevor Darrell. Autotagging Facebook: Social network context improves photo annotation. In *Proc. IEEE Computer Vision and Pattern Recognition Work. (CVPRW)*, 2008.

[29] James E White. A high-level framework for network-based resource sharing. In *Proc. National Computer Conf. and Exposition*, 1976.

[30] Yongzheng Wu, Sai Sathyanarayan, Roland HC Yap, and Zhenkai Liang. CodeJail: Application-transparent isolation of libraries with tight program interactions. In *Proc. European Symp. on Research in Computer Security (ESORICS)*, 2012.