Active Segments: A Virtual Memory Abstraction

ACM Reference Format:
. 2025. Active Segments: A Virtual Memory Abstraction. In *Proceedings of Workshop on Hot Topics in Operating Systems (HotOS).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

The virtual memory of a process, originally intended to abstract physical memory on the machine where the process is running, has been used to abstract a wide variety of resources, including files, distributed shared memory [3], remote memory resources [2], persistent memory resources [3], and more. It is often a convenient interface, as processes can access the resources with CPU-supported load and store machine instructions, allowing applications to run unchanged on new resource types. Moreover, virtual memory can be leveraged to implement transparent checkpointing and recovery [3], process replication [18], speculative execution [14], and debugging tools. Unfortunately, as an abstraction, it is hard for system developers to offer it, as the abstraction is implemented in the kernel of an operating system and therefore hard to customize. The Linux virtual memory interfaces provide only limited customization.

This paper proposes *active segments*, a virtual memory abstraction that can be efficiently offered by non-privileged services running in user space. Each active segment is managed by a *segment process*. As a result, each application runs as a collection of processes, one running the main logic of the application, and the others managing the various virtual memory segments of the application, each of which can be an important part of the application logic as well.

1 Background

In modern operating systems, applications have essentially two abstractions available to access data: They can use *stream* system calls such as `open`, `read`, `write`, and `close`, or they can map data into their address space. Stream system calls are primarily designed for resources such as keyboard input and network connections, and for applications where only part of the required or generated data is available at a time. When virtual address spaces were small, stream interfaces also made sense for so-called *out-of-core* processing, where only parts of a file were read at a time. SpaceJMP [8] addresses this issue by making multiple virtual address spaces available to processes, but virtual address spaces are no longer small.

Increasingly applications operate on very large data sets, and mapping the entire data set in virtual memory is more convenient and efficient then streaming the data set. Stream interfaces inherently require copying, which is expensive, and they do not easily support (real-time) sharing, pointers, and locking that are inherently available in the virtual memory abstraction. 5-level page tables are increasingly common, allowing extremely large data sets (up to 128 Petabytes) to be mapped. Moreover, memory mapped data sets are amenable to checkpointing, which can be important for fault tolerance and effective debugging of applications.

Modern operating systems provide various interfaces to manage the virtual memory of a process, but they were initially designed to support dynamic libraries and are limited in their capabilities. Importantly, the kernel is in full control of when virtual pages are mapped, when and where they are saved to backing store, when they are restored, and so on. An application may be able to advise the kernel on how it uses the virtual memory (e.g., using the Linux `madvise` system call), but the kernel is still in charge.

As of recently, Linux also provides `userfaultfd`, supporting delegation of page-fault handling to user space. A process can register certain regions of its address space. When a page fault occurs in such a region, the faulting thread is put to sleep and another thread is notified. The fault handling thread can initialize the page and restart the faulting thread. Dirty page tracking can be accomplished by write-protecting clean pages.

The `userfaultfd` interface provides applications with control over prefetching and cache eviction policies. UMap [16, 17] is a convenient interface over `userfaultfd` to customize memory mapping large data sets for various applications and backing stores. However, `userfaultfd` has various shortcomings. The kernel still decides when to evict a page and which page to evict. Also, because the mechanism does not give access to the page tables, it does not support Copy-on-Write, which is essential for efficient checkpointing, replication, and speculative computing.

HotOS, Banff, Alberta, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Microkernels [1, 4, 13] provide user-space management of paging and simplify customization. However, the management is done by privileged processes, so ordinary applications cannot easily customize paging to their needs.

Exokernels [10, 11], which leave the implementation of abstractions to user space, do enable a significant amount of customization per application as paging is managed by a *library operating system*. We instead propose a kernel-supported abstraction to enable efficient virtual memory customization in conventional operating systems.

Dune [5] is a hybrid between a process and virtual machine model that gives processes access to hardware VM features, including (nested) page tables. While this gives unlimited flexibility, it does not offer sharing of segments to processes and requires programming of non-portable, low-level abstractions.

This paper proposes *active segments*—segments that are managed in user space by unprivileged processes—a new virtual memory abstraction that is intended to be both powerful and easy to use and that can be incorporated in conventional operating systems.

2 Use Cases

In this section we highlight some use cases we envision.

2.1 Memory Segments

These are perhaps the most obvious. A memory segment represents a region of zero-initialized virtual memory. The segment process is responsible for backing up pages if their frames are being reclaimed and for restoring their content after a subsequent page fault in these pages. It is useful for heap segments and stack segments.

A memory segment can be forked, creating a virtual copy. This can be useful for emulating the `fork` system call, where all writable segments must be forked. It can also be used for checkpointing segments, which is useful for roll-back in applications such as fault-tolerant computing, speculative computing [14], and reverse debuggers [9].

There may be different implementations of memory segments, including remote memory (e.g., CXL) and distributed shared memory with various forms of consistency guarantees [6, 7, 12].

2.2 File Segments

Similar to a memory segment, except that its initial content comes from a file. File segments come in various flavors. Read-only file segments are useful, for example, for code segments and shared libraries. Copy-on-write file segments can be written, but leave the initial file intact. This is useful for data segments. Finally, direct mapped file segments can be modified and these modifications are persisted. Persistence is eventual unless the application requests immediate persistence (in Linux, using the `msync` system call).

2.3 Disk Segments

A disk segment maps a contiguous range of blocks of a disk (local or remote) in memory. This is essentially a disk cache in the form of a segment. Such a segment may be used by a file server or a database server to access the disk. Similar to a file segment, updates can be persisted. Frame sharing between segments is desirable: a frame containing a part of a file can be shared between the disk segment that stores the content and the file segment that provides access to ordinary processes. Hence, this effectively leads to unified buffer and page caching [15].

2.4 Cache Segments

A cache segment is a read-only segment that maintains an in-memory copy of data that is derived from other data and that can be recomputed. For example, a cache segment may contain the uncompressed or decrypted form of some data set that is available through some other segment. It may contain a version of a data set, many versions of which are stored in a version control system. A cache segment may be used to present the materialized view of a database query, as long as it can be deterministically recomputed.

Note that when frames of a cache segment are being revoked, there is no need to save them to backup storage as their contents can be recomputed.

3 Design

An *active segment* (or just segment for brevity) is a first-class object, identified by a segment identifier or SID, much like a process is identified by a process identifier or PID. It consists of a region of virtual memory combined with a *segment process* that manages the required virtual memory resources. For the purposes of this paper, we assume that a virtual memory address consists of three parts: a segment number, a page number, and a page offset. The segment number indexes into the root page table. For example, for a 48-bit address, the segment number may occupy 9 bits, for a total of 512 segments in the virtual address space of a process. A segment then consists of 2^{27} 4K pages.

In our example configuration, a process can map up to 512 different active segments. We also allow multiple processes to map the same segment, or even the same process mapping the same segment multiple times in the process' address space. (There is no requirement that a segment be mapped at a particular address.) A segment provides RWX access rights to the processes that map it on a per-page basis.

3.1 Hardware Support

A segment is implemented using *frames*, which are physical memory chunks the same size as a page (4K in our configuration). Each segment has an associated Page Table Tree or PTT, the root of which is a child of the root page table of any process that maps the segment. Each node in the PTT is itself a frame containing an array of Page Table Entries (PTEs), which can optionally point to other frames. The PTT is (currently) of fixed depth, and the leaf frames point to data frames. A virtual address can be split into a sequence of indices that represent a path in the PTT to a data frame. The Memory Management Unit (MMU) of a processor automatically walks the entire page table tree (including the root table of a process), generating a page fault in case a page is not mapped or if access is disallowed.

3.2 Demand Paging

A key part of our design and the major innovation is to associate a user-space page fault handler with each segment. We accomplish this by assigning to each active segment a process, which we shall call the *segment process* or *segproc* for short. A segproc has limited access to the PTT that it manages. For obvious security reasons, a segproc cannot point PTEs to arbitrary frames. A segproc is only allowed to use the frames that it has access to, although, as we will see later, it is possible for multiple segments to share frames or even parts of the same PTT. To create an ordinary process,

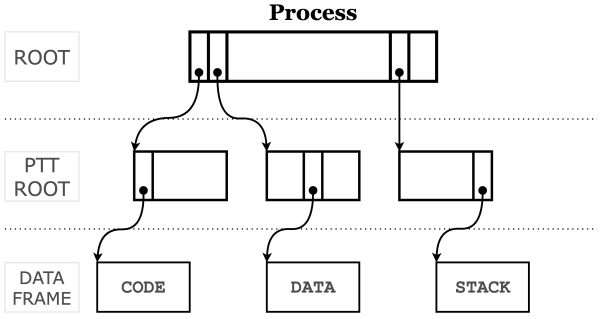

Figure 1: Demand paging of code, data, and stack PTTs (two level page table)

one must first create or obtain the segments that the process needs to run, which generally includes at least a code (aka text) segment, a data segment, and a stack segment. Each segment is mapped at a specific segment number, determining the address of the segment in that process. Figure 1 shows the mapping of the code and data segments at the lowest segment numbers in the process' address space, and the mapping of the stack at a higher segment number. The Process Control Block (PCB) of a process contains a *segment table* with this information. Each entry in the segment table contains the segment identifier (which in turns refers to the PTT and the segproc), the segment number, and the maximum RWX bits for its pages. Once the process' segment table is initialized, the process is then started.

When the process experiences a page fault, the kernel looks up the segment in the segment table. If there is no segment at the fault address, the process is terminated (segmentation fault). Otherwise, the kernel invokes the page fault handler of the segproc, much like a user-level interrupt. The kernel provides the segproc with the page number and the access (RWX). The process itself is put into a waiting mode so it is no longer scheduled.

The segproc ultimately is in charge of allocating and initializing frames and updating the PTT of the segment, for which it has a variety of system calls at its disposal (described later). When done, the segproc notifies the kernel that it has mapped the address. The kernel will then resume any process (or processes) that were waiting to access this page in the segment.

3.3 Frame Capabilities

A PTT is a tree of frames. It must be carefully managed to prevent processes from being able to access arbitrary memory or even devices. Moreover, to implement mechanisms such as copy-on-write, we allow different PTTs to share frames.

Inspired by exokernels [10], we use a capability-based security model for frames. Whereas exokernels commonly use cryptographically protected capabilities, we use kernel-protected capabilities. A capability for a frame in our system is simply the physical frame number of that frame. However, a segproc can only use a capability if the corresponding frame is reachable from its PTT. This then also provides the *delegation* mechanism: a segproc can delegate access to a frame by including it as part of a PTE inside a shared PTT frame.

A segproc can use the system call `frame_alloc`(*address*) to allocate a zero-initialized frame and map it at the given virtual address (within the address space of the segproc). In the current implementation, there is a single defined address that segprocs can use for allocating frames. After allocation, the segproc can initialize the frame in any way it sees fit.

A segproc can update a PTT by invoking the system call `ptt_map`(*address*, *level*, *frame*, *rwx*). This system call updates a PTT at the given level. Assuming 2^9 PTEs per frame, the most significant 9 bits of the address specify the segment, or, equivalently, which PTT must be updated. (A segproc can manage multiple PTTs in our design, thus multiple active segments can share the same segproc.) Level 1 replaces the root frame of the PTT and ignores the remaining bits of the address. (This update must be made in all processes that map

the segment.) Level 2 updates the entry in the root frame identified by the next 9 bits in the address. Level *n*, where *n* is the number of page table levels that the hardware supports, replaces a data frame.

Note that tuple (*address*, *level*) names the capability of a frame in the PTT. A segproc can only use those capabilities that it can name.

The *frame* argument is the address of the frame that must be mapped in the PTT. In our current implementation, it must be the same address as the one that was used for `frame_alloc`—the *frame* argument names the capability of the allocated frame. The *rwx* bits specify the required access to the frame. Specifying 0 causes the frame to be unmapped (and deallocated if there are no other references to it).

The kernel checks the security of the call. The kernel only allows the segproc to update segments that it manages, and the segproc can only map frames that are already mapped in the virtual address space of the segproc. When mapping a frame that is a PTT node, the kernel must make sure that it does not contain capabilities for frames that the segproc does not have access to.

3.4 Frame Reclamation

When physical memory runs low, it is advisable that some active segments release frames that they have allocated. Historically, it has been the domain of the operating system kernel to decide which frames to release and many page replacement techniques have been developed. Also, the kernel decides where to back up the page in case it is later accessed again.

In our design, we leave most of these decisions to the segprocs themselves. The kernel only determines an allocation of frames to active segments—that is, the kernel computes how many frames each segproc is allowed to use. If the segproc exceeds this limit for some time, then the kernel terminates the active segment and the processes that map it.

The kernel invokes the segproc when its allocation has changed, and in particular when the segproc uses more frames than its current allocation. The segproc has limited time to give up the required number of frames. The segproc first determines which frames it chooses to release. Some of these frames may be *dirty*. The segproc then has to write those frames to backing store first (unless there is a way to recompute their content some other way).

3.5 PTT and Frame Sharing

Our design allows multiple segments to share the same frame, including frames that are part of a PTT. For example, we support segment *forking* to create a copy of a segment. Initially, the two segments can share their frames read-only. New (writable) frames must be allocated when a page fault occurs due to a write operation.

After forking a segment, there are two active segments and two segprocs but still only one PTT as the two segments share the same root frame. Both segprocs have full access to the PTT, as both can name all the frame capabilities inside the PTT. Figure 2 details this scenario, where segproc A forks itself into segproc B, subsequently sharing its PTT root frame. Both segprocs make a mapping in the shared PTT root to their corresponding `PATH` data frame, delegating access to the other segproc. It is possible that the segprocs attempt to modify the same PTT root entry at the same time. Thus updates to the shared PTT must be synchronized between the segprocs.

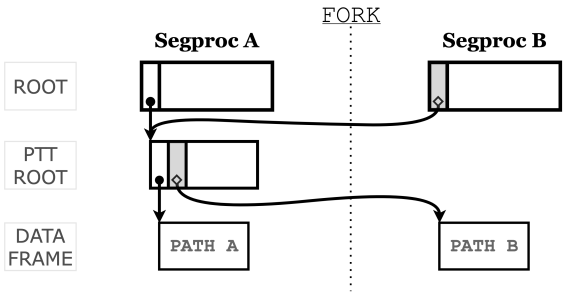

Figure 2: Depiction of segment forking, with both siblings updating the shared PTT (2 level page table)

When a process writes to one of its active segments, the corresponding segproc is notified. In the case of a Copy-on-Write segment, the first time this happens a new frame must be allocated, the original data frame must be copied to the new frame, and the PTT must be updated all the way up to the root. After this, the two segprocs have different root frames, but most of the rest of the PTT may still be shared. When the same page is updated in the other active segment, all that needs to happen is to enable the write bit in the PTE.

Frame reclamation may also require coordination between the segprocs. For example, it is possible that a frame was dirty even before the segment was forked, and hence the dirty frame is now mapped in both segments. When a segproc decides to unmap the frame, it needs to be saved in case it was the last reference to that frame (unless the data can be regenerated some other way). More specifically, if a PTT frame is shared and maps to the dirty frame, either segproc using the shared frame can write-back and unmap the dirty frame, subsequently revoking access from the other segproc. If the segprocs sharing the PTT frame use different page replacement techniques, there may be a conflict of interest on which frames should or should not remain, requiring coordination between the two segprocs to efficiently manage their shared memory.

To this end, the segprocs themselves need to maintain a shared active segment with metadata. This active segment in turn is managed by a third segproc. This shared segment is a simple memory segment that maintains locks on the shared PTT nodes as well as reference counts and dirty bits.

4 Implementation

We have implemented a prototype of active segments in the EGOS-2000 operating system [20] on a RISC-V platform. Our current implementation runs on a 32-bit RISC-V architecture, which only supports 2 level page tables, but we soon hope to move to a 64-bit system with deeper page tables. Part of the functionality related to sharing and eviction is still under development, so we discuss our current plans.

Segprocs are user space daemon processes that manage the virtual memory of an active segment. Our current implementation associates each segproc with exactly one active segment, where the active segment's PTT is mapped at the bottom of the segproc's virtual address space. Segprocs receive requests to operate on the corresponding active segment's data, including requests to handle page faults and requests to reduce the number of mapped frames.

When a segproc is requested to handle a page fault, it must have access to the active segment's PTT to know the state of PTEs along the faulting path. To simplify this, the kernel maps each PTT frame read-only inside the address space of the segproc. We call this the *flattened representation* (FR) of the PTT. The FR is maintained automatically as a side effect of mapping and sharing frames.

The FR does not include the data frames, but these are also accessible to the segproc because the PTT is used in its own address space. The FR requires a PTT of its own and must be synchronized when PTT frames are shared. A benefit is that a segproc will always have up-to-date access to capabilities held inside its PTT, possibly delegated by other segprocs that also share parts of the same PTT.

4.1 Managing Sharing

The kernel maintains an *ancestor set* for each frame, which contains the locations of valid PTEs (frame number, index) that map to the frame. For a given frame, its ancestor set is updated when another frame has an entry mapped to or unmapped from the given frame (except for the root PT of segprocs, which will always contain no ancestors).

Ancestor sets enable the FR to be efficiently managed in shared regions. When a PTT frame is updated, the kernel performs an *ancestor-walk*, which finds the set of all segprocs that can access a given frame. The *ancestor-walk* recursively finds the set of all segprocs associated with each ancestor of a frame, walking up the PTTs, stopping at the set of frames with no ancestors. Frames where the walk stopped must be root PT frames, which are unique among segprocs. While walking, the kernel can also accumulate indices held in ancestor set entries and reconstruct the address each segproc has mapped to the updated PTT frame, determining how the FRs are modified.

As a result of the *ancestor-walk*, the kernel also maintains a *capability set* for each segproc, which holds the set of all capabilities accessible to a given segproc. On updates to a PTT frame, the *ancestor-walk* finds the set of all segprocs associated with the update, which enables the kernel to add or remove a capability from their capability sets based on the type of update. This lets the kernel know when to revoke frames from a segproc.

4.2 Security

A segproc should not be able to arbitrarily modify the PTT it manages. Therefore, the kernel enforces the following two constraints. Firstly, *if an invalid PTE is set to map to PTT frame F, then F must only contain invalid entries, or be shared by another segproc.* This prevents segprocs from using capabilities they cannot name. Secondly, *if a PTE currently maps to frame F, and F has descendants, the PTE cannot be made invalid, and can only have its mapping changed to F' if F' contains a copy of F*. This constraint prevents holes in the PTT, supports Copy-on-Write, and retains the original PTT structure.

5 Conclusion

Active segments are a new abstraction that allow application developers to customize the virtual memory of a process in user space. We hope that developers will make a wide variety of active segment implementations available for other developers to use. The active segments paradigm allows the in-kernel part of virtual memory management to be simplified, as page replacement algorithms are implemented in user space. This in turn may simplify work on formal verification of virtual memory management [19].

Going forward, we need to address several issues:

- In NUMA architectures, not all physical memory is the same. For example, we would like to provide support for NUMA architectures, non-volatile memory, etc. One possibility is to provide an extra argument to `frame_alloc` so the active segment developer can specify the type of memory that is desired or required.
- *Huge pages* (aka large pages or super pages) can reduce pressure on the TLB and the page table and generally improve performance for applications that use large data sets. For simplicity, we currently assume that all pages are the same size, but we will have to address this as well.

References

[1] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *Proceedings of the 1986 USENIX Summer Conference*, 1986.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 121–127, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):96–107, April 1991.

[4] Mohit Aron, Jochen Liedtke, Kevin Elphinstone, Yoonho Park, Trent Jaeger, and Luke Deller. The SawMill framework for virtual memory diversity. In *Proceedings of the 2001 Australian Computer Systems Architecture Conference*, pages 3–10, 2001.

[5] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association.

[6] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system . In *COMPCON Spring '93. Digest of Papers*, pages 528,529,530,531,532,533,534,535,536,537, Los Alamitos, CA, USA, February 1993. IEEE Computer Society.

[7] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Trans. Comput. Syst.*, 13(3):205–243, August 1995.

[8] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. SpaceJMP: Programming with multiple virtual address spaces. *SIGARCH Comput. Archit. News*, 44(2):353–368, March 2016.

[9] Jakob Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–6, 2012.

[10] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery.

[11] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 52–65, New York, NY, USA, 1997. Association for Computing Machinery.

[12] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989.

[13] Jochen. Liedtke. On micro-kernel construction. *SIGOPS Oper. Syst. Rev.*, 29(5):237–250, December 1995.

[14] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 191–205, New York, NY, USA, 2005. Association for Computing Machinery.

[15] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*, New Orleans, LA, February 1999. USENIX Association.

[16] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. UMap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 71–78, 2019.

[17] Ivy B. Peng, Maya B. Gokhale, Karim Youssef, Keita Iwabuchi, and Roger Pearce. Enabling scalable and extensible memory-mapped datastores in userspace. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):866–877, 2022.

[18] Thomas Ropars, Arnaud Lefray, Dohyun Kim, and André Schiper. Efficient process replication for mpi applications: Sharing work between replicas. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 645–654, 2015.

[19] M. Wang, Zhenhua Duan, and Cong Tian. Simulation and verification of the virtual memory management system with MSVL. *Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 360–365, 2014.

[20] Yunhao Zhang. EGOS-2000. https://github.com/yhzhang0128/egos-2000.