UFO: A General-Purpose User-Mode Memory Protection Technique for Application Use

Naveen Neelakantam and Craig Zilles
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {neelakan, zilles}@uiuc.edu

Abstract

In this paper, we advocate for a general-purpose, fine-grain memory protection mechanism for use by applications and software development tools. The hardware component of this mechanism consists of two bits (fault-on-read and fault-on-write) that are associated with each cache block and transferred with the block by the cache coherence system. When an access is performed that violates the protection on the block, a user handler is called, which can perform an arbitrary software-defined action. We describe a complete implementation that handles real world issues like protecting arbitrarily large regions, context switches, swapping, and multiprocessing.

We demonstrate three examples of this mechanism in applications: 1) to detect self-modifying code (SMC) in a dynamic translator, 2) to detect common memory-related software bugs, and 3) to guard mostly-invariant heap variables in order to enable speculative optimizations. Because checking for conflicts does not directly impact performance, we find our mechanism has run-time overhead that is either negligible or at least comparable to special-purpose hardware for the same application. Furthermore, by virtue of providing a software interface to use the fine-grain protection mechanism, we can support multiple uses of the protection mechanism within the same application, enabling arbitrary composition of tools and libraries which use it.

1. INTRODUCTION

In this paper, we advocate for a general-purpose, fine-grain memory protection mechanism for use by applications and software development tools. We hope to demonstrate that doing so is: 1) useful, in that it provides functionality that is difficult (if not impossible) to efficiently achieve through conventional instruction sets, and 2) relatively inexpensive and straight-forward to implement.

We call the proposed architecture User-mode Fault-On (UFO) bits, because we have extended the memory system to include two additional bits (fault-on-read and fault-on-write) that can be set/cleared in user mode and that raise a “fault” when the block is later accessed by the application. By implementing a simple software layer on top of this hardware support, we can provide application and tool programmers a simple API that allows callback functions to be registered for single addresses or whole ranges of addresses. By invoking the callback function before the faulting data access completes, the callback function can perform fix-up activities required before the access, or to emulate the data access. In Section 3, we describe the salient features of the UFO API.

We demonstrate the flexibility and utility of the UFO API using three types of application:

- lightweight detection of self-modifying code in a dynamic translator (Section 4.1)
- detection of common memory-related software bugs (Section 4.2)
- guarding mostly-invariant memory variables in order to support speculative optimizations (Section 4.3)

For each application, we describe how the fine-grain memory protection afforded by UFO is exploited.

The UFO API is implemented by a combination of hardware and software: software controls setting the UFO hardware state and delivering UFO faults, and the hardware is responsible for detecting access violations with a minimum overhead. In fact, the primary goal of UFO’s implementation is to not slow down execution unless the the program is making a UFO API call or handling a UFO callback. While beneficial for all applications, this characteristic is absolutely essential for using fine-grain memory protection to support software speculation, as any overhead reduces the achievable speedup of those techniques.

Zero execution overhead is achieved by virtue of UFO’s memory-centric approach. By bundling UFO permissions with data wherever it is stored in (and as it moves through) the memory hierarchy, there is no need for extra memory operations to check permissions. The UFO permission check occurs as part of a cache tag look-up and is not on a processor critical path. Furthermore, since the state is stored in memory, there is no work to perform on a context switch.
UFO’s hardware support is described in detail in Section 3.2. The primary components include a 0.4% storage overhead for processor caches, modifications to the memory controller to permit storage of the UFO bits in main memory’s ECC, and two new instructions for reading and writing UFO bits. We discuss how UFO bits, which aren’t renamed, can be written without serializing the machine and how UFO supports multiprocessors without any changes to the coherence protocol.

Section 3.3 describes UFO’s software layer, which tracks the association of UFO callbacks to regions of memory, controls setting and clearing the UFO bits, and the delivery of UFO faults to the application callbacks, and, in Section 3.4, we describe the operating system support for preserving UFO state while pages are swapped to disk. Because the UFO software layer, in effect, virtualizes the UFO hardware (by supporting independent overlapping UFO regions), a single application can concurrently use UFO for multiple purposes, enabling arbitrary composition of libraries, tools, and compilation techniques that use UFO.

In Section 5, we discuss two implementations of UFO. The first is a functional simulation implemented using the Simics full-system simulator; with this implementation we sought to validate the correctness and completeness of our UFO hardware, software layer, and O/S support implementations. With the second implementation, we sought to measure the performance impact of UFO. As the overhead of UFO derives entirely from its software components, we used real machine executions for these experiments, modelling the reads and writes to UFO state with normal loads and stores. For two of our applications we find the UFO overhead to be negligible and for the third it is comparable to special-purpose hardware for that application.

Specifically, our paper makes the following contributions:

- We provide a detailed description of a user-mode fine-grain protection mechanism, which handles real world issues like protecting arbitrarily large regions, context switches, swapping, and multiprocessing,
- We describe an implementation and measure the performance impact of O/S support for maintaining per-page data in the presence of swapping pages to disk,
- We describe two novel applications of a fine-grain protection mechanism (detection of self-modifying code for binary translators and support for speculative optimizations), and
- We perform real machine experiments to measure the overhead of UFO in a variety of programs.

2. Related Work

In this section, we discuss the closely related work—including virtual memory page protection, Mondrian memory protection, iWatcher, and ECC-based schemes for fine-grain memory protection—and describe how UFO differs from these schemes.

Most virtual memory architectures provide mechanisms to independently control permission to read, write and execute memory at a page granularity. mprotect() is a POSIX system call that allows applications to specify which pages of their memory have which permissions. If a program attempts to perform an operation on a memory location that is not allowed by these permissions, it is sent a SIGSEGV (segmentation violation) signal, which can be handled by the application. A number of applications and tools have used mprotect to add functionality (e.g., TreadMarks [1], which uses it to implement distributed shared memory on a network of workstations) or prevent errors/increase security (e.g., StackGuard [6], Oracle [14], and Dynamo [2]). In principal, UFO provides similar functionality as mprotect, but is a lighter-weight mechanism, in two aspects: 1) it enables monitoring smaller granularities than full pages, and 2) it does not require a system call to add or delete a UFO region. While the absence of the system call and the checks it performs to validate an mprotect request reduces overhead, it also makes UFO unsuitable for security/protection applications, as it relies on the cooperation of all code within the application that use UFO.

Mondrian memory protection (MMP) [22] eliminates one of the constraints of tradition virtual memory protection by enabling regions of arbitrary size to be protected. MMP is comprised of two main components: an in-memory data structure (much like a page table) that stores the regions and their protections and a permissions lookaside buffer (PLB) that caches the permissions (much like a translation lookaside buffer). MMP’s functionality is a superset of UFO’s, as it can also be used for true memory protection and can support granularities down to a single byte, but this additional functionality comes at a cost of more significant hardware support and performance and memory overheads. MMP’s hardware support includes additional processor storage for register sidecars and the PLB, as well as a hardware walker for the in-memory protection data structure. While both UFO and MMP have overheads to change protections, MMP also has overhead in normal execution. While the memory
and performance overheads of MMP are insignificant (less than 1%) when the features of MMP are not actively used, when fine grain protection is significantly used, memory overheads of 4-8% and memory bandwidth of 1-6% are observed; these overheads are not present in UFO. MMP also requires more significant modification to the operating system, including the implementation of a call gate interface to the operating system in order to avoid the overhead of a system call to change permissions on a region of memory.

Perhaps the closest related work to UFO is iWatcher [23], which was proposed as a flexible architecture for location-based monitoring for software debugging. The key idea is that since accesses to the bulk of memory locations are not problematic, monitoring should be memory location centric rather than instruction centric. In particular, instrumentation can be inserted into a program to identify potentially problematic memory regions to the hardware, which notifies the software when these regions are accessed. While many architectures, including x86, already support location-based monitoring in the form of hardware watchpoints, their limitations—IA-32 can only track four addresses—prevent them from being exploited by dynamic checking tools. If an unbounded number of watchpoints were supported, they could be effectively exploited by these tools.

While iWatcher and UFO provide similar APIs to the programmer, the hardware support is organized differently. iWatcher takes a processor-centric approach by allocating three hardware structures associated with a processor and its caches: 1) the Range Watch Table (RWT) specifies watched virtual address ranges and is accessed in parallel with the TLB, 2) per-word WatchFlags for every line in the caches, and 3) a Victim WatchFlag Table (VWT) to hold the WatchFlag state for lines that have been kicked out of the caches. The two main drawbacks of this approach are: 1) as proposed, watched regions are only guaranteed to be watched by the thread that requested the watching, and 2) context switching is significantly impacted.

Because the RWT’s are, in effect, processor registers, writes to them performed by one thread are not visible by other threads within the same application. In order for all threads to watch a range, the operating system would need to be invoked to update the contexts of other threads within the process, performing an interprocessor interrupt on any of the threads that are currently executing. Likewise, because the WatchFlags are maintained only within the on-chip cache hierarchy, they are not visible to a thread executing on another chip. In contrast, UFO’s memory-centric approach always stores its permissions with the block wherever it is in the memory hierarchy, making them accessible to any thread on any processor.

The second drawback of a processor-centric approach is that, when a thread is context switched, its iWatcher state must be also. Swapping the RWT is straight-forward and only add overhead to the context switch time. Swapping WatchFlags and VWT state is performed on demand; when a block with WatchFlags is evicted from the on-chip hierarchy its address and WatchFlags are recorded in the VWT. If this happens when the VWT is full, an asynchronous exception is raised and the O/S mprotects the whole page and records the block and its WatchFlags. If the page is later accessed, a page fault will occur and the O/S can restore the VWT entries for the page (possibly victimizing other VWT entries) and restoring the page’s protection. Again, UFO’s memory-centric approach obviates the need for any additional activity at context switch time.

In addition, UFO requires less hardware than iWatcher, not having structures that correspond to the RWT or VWT nor incorporation of thread-level speculation. Also, iWatcher also assumes that pages with WatchFlags are pinned in physical memory, which could be implemented by performing a system call when watchers are installed (resulting in additional overhead), something not required in UFO since UFO bits can be swapped by the operating system.

UFO harvests space for storing the UFO bits in memory by re-encoding ECC at a larger granularity. Previous work has used the existing ECC hardware for performing fine-grain memory protection by purposefully mangling ECC codes (i.e., introducing uncorrectable errors). Blizzard-E [18] used precise exceptions from this technique in order to implement distributed shared memory. SafeMem [17] used this technique to build software tools for detecting memory leaks and memory corruption bugs.

3. UFO API and Implementation

In this section, we describe hardware/software implementation for the UFO system. The goal of this system is to provide a simple application program interface (API) to the programmer that allows callback functions to be registered when specific data blocks are accessed. In the next subsection, we describe a representative API, followed by descriptions of the hardware support (Section 3.2), the software support to implement the API, including its utilization of the existing signal handler interface (Section 3.3), and operating system support to preserve the UFO bits during paging (Section 3.4).
3.1. UFO Application Program Interface

The two components of the API are the callback handlers and the registration and de-registration of those handlers. The handlers have the following form:

```c
typedef enum {NONE, READ, WRITE, EITHER} access_type_t;
typedef void (*UFO_handler)(addr_t fault_addr, access_type_t access_type,
                             void *registered_data, void *ucontext);
```

Callback functions are invoked with four arguments: 1) the virtual data address that was being accessed that caused the fault, 2) the type of access (READ or WRITE) that caused the fault, 3) a pointer to an arbitrary data structure (interpreted by the callback function) that was supplied when the callback was registered, and 4) a pointer to the register context of the thread that raised the fault. The first two arguments enable the callback to understand why it was being called. By allowing arbitrary data to be associated with a UFO region, the third argument enables a single callback function to be used for multiple independent UFO regions. The fourth argument enables the callback function to inspect the state of the faulting thread.

The API enables applications to register callback functions to be associated with an arbitrarily-sized range of addresses and invoked only on read, only on write, or on either type of access:

```c
void UFO_add_region(addr_t start_addr, size_t size, access_type_t atype,
                    UFO_handler callback, void *registered_data);
```

Callbacks are invoked prior to the completion of the faulting memory operation; if a thread attempts to access one of the registered locations, our mechanism suspends the thread immediately prior to the access and invokes the callback function. A common important usage of UFO handlers is to perform some computation, clean-up, or logging just before an access to a region, after which the monitoring of the UFO regions is removed by the callback routine. The UFO API provides an interface for removing handlers:

```c
void UFO_del_region(addr_t start_addr, size_t size, access_type_t atype,
                    UFO_handler callback, void *registered_data);
```

Composition: An important feature of the UFO API is that it is general-purpose and, as such, it is important to permit the UFO hardware to be used concurrently for multiple purposes within an application. If this were not the case, then libraries and modules that used UFO could not be composed together to make larger pieces of software, a fundamental requirement of modern software engineering practice. UFO enables composition by discouraging direct access to the UFO hardware, instead presenting a clean API to the programmer that handles the composition internally.

A key requirement to enabling composition is supporting the registration of multiple overlapping regions; when an access occurs to a memory location where multiple callback functions have been registered, each is called before retrying the operation. In our current implementation, no guarantees are provided on the order in which they are called, but providing FIFO or LIFO order would be a relatively simple modification to the software.

In Section 4, we demonstrate the generality of this API, through its ability to implement three distinct applications: 1) efficient dynamic translation in the presence of self-modifying code, 2) debugging of common memory-management errors,
and 3) write barriers for speculative code specialization. In the remaining sections, we describe the hardware, software, and O/S components used to implement this API.

### 3.2. UFO hardware support

The primary components of the UFO hardware are three fold: a set of bits associated with each cache line in memory, a mechanism to raise an exception if a memory access is performed that conflicts with the bits as they are set on the cache line, and user-mode instructions to read and write these bits.

**Allocating Space:** We propose that each block of memory has two associated bits (fault-on-read and fault-on-write) that are transferred with the block as it moves around the system. This design is motivated by two goals: first, checking for UFO faults should not impact performance (which effectively necessitates that UFO bits are transported with the memory block so that a secondary memory request is not required), and, second, that an arbitrary number of arbitrarily large UFO regions can be supported. By associating a pair of bits with every block in physical memory, both goals can be achieved.

Because of the commodity nature of the DRAM industry, it would be prohibitively expensive to necessitate a new DRAM architecture. Bits can be made available for use by UFO by using ECC memory but encoding ECC at a larger granularity (e.g., 128b vs. 64b), as was done to provide storage for the Alpha 21364’s directory [10]. Given the increasing susceptibility to single-event upsets with decreasing feature size, it is likely that ECC will be pervasive throughout future systems.

The granularity at which UFO bits are associated to memory blocks is a trade-off between cost and the number of false positives. The relationship to cost is fairly clear; the smaller the granularity at which UFO bits are tracked, the more storage that is required in caches, DRAM, and on disk. The relationship to false positives is more subtle: While our API allows registration of UFO regions of arbitrary size and boundaries, the UFO hardware is limited to only track UFO regions at the granularity at which the UFO bits are allocated.

To guarantee all accesses will be detected, the UFO software layer has to set bits conservatively to cover application-specified region (as shown in Figure 2). The result is that accesses nearby a UFO region may result in a false positive, wasting time by causing an exception that is thrown away (i.e., not passed up to the application by the UFO software layer). Thus, the larger the granularity of the memory blocks associated with the UFO bits, the more potential there is for false positives.

![Fig. 2. UFO bits must be conservatively set to cover application-specified regions.](image)

In our implementation, we elected to allocate UFO bits at the granularity at which cache coherence is performed (64-byte blocks), which results in less than 0.4% storage overhead for the on-chip caches. In spite of this relatively large granularity, we found that false positives were not a problem, as they either naturally did not occur or could be mitigated by padding data structures (because the code that was using UFO also had control of memory layout for the monitored regions) with minimal impact on overall memory usage.

**Detecting Faults:** When a processor reads a block from memory, the entity that checks ECC (the memory controller in Figure 3) extracts the UFO bits from the ECC stored in main memory and stores them with the cache tags. The UFO bits are transported with the block’s data (and any ECC used to detect errors) on the interconnect network. When a processor accesses its cache and performs a tag check, the UFO bits are read to detect if they conflict with the type of access. If a conflict occurs, an exception is raised and the faulting address is captured in a processor control register. We discuss the exception handling sequence in the next subsection. While UFO may introduce small storage, wiring, and power overheads, because reading the
Fig. 3. **Hardware support for UFO.** All caches tags are extended by two bits to hold (the shaded) fault-on-read and fault-on-write bits. When a block is not stored in the cache, these bits are stored in main memory with the ECC state by encoding ECC at a larger granularity.

UFO bits is performed as part of the existing cache tag look-up, there is no execution overhead introduced by UFO when faulting data is not accessed.

**Reading and Writing UFO bits:** Instructions must be provided for reading and writing the UFO bits. The simplest instructions are of the following form:

\[
\begin{align*}
\text{ufo\_write} & \quad \text{reg1/imm, offset(reg2);} & \text{## store format} \\
\text{ufo\_read} & \quad \text{reg1, offset(reg2);} & \text{## load format}
\end{align*}
\]

Both instructions use memory formats\(^1\), generate addresses normally, and move down the pipeline as loads and stores. Like stores, writes require exclusive coherence permission to the cache line and are performed at commit time, because the UFO bits are not speculatively buffered or renamed. UFO reads are performed speculatively (like normal loads) and must be invalidated based on coherence events as per the memory consistency policy on the platform. Uni-processor dependences must be observed\(^2\), but given that true (write \(\rightarrow\) read) dependences should be uncommon—we observed none within 128 instructions in our applications—forwarding should not be required; squashing when such a dependence occurs should be sufficient. Finally, UFO writes are treated as “writes” with regard to having write permission and setting dirty bits in the TLB/page table to ensure that UFO bits are swapped properly and the semantics of operations like copy-on-write are maintained.

### 3.3. Software Support

Since the UFO hardware does not directly implement the UFO API, a software layer is required to act as an intermediary. The three main roles of the software layer are to: 1) track the association of UFO regions to callbacks and associated data, 2) to correctly set and clear the UFO bits based on the registered UFO regions, and 3) to dispatch the appropriate callbacks when UFO exceptions are raised. We discuss each of these in turn.

**Tracking UFO regions:** The UFO bits do not track which callback is associated with a given memory block; this must be done in software and stored in normal memory. We have two goals in the design of this software: 1) minimizing the space overhead of this storage, and 2) minimizing the time overhead of inserting and removing UFO regions. These objectives turn out to be somewhat at odds.

For UFO regions that do not span cache blocks—what we’ll refer to as “singleton” UFO regions—a hash table can be used to provide O(1) insertion and removal time\(^3\). For large UFO regions, however, a hash table requires an entry for each cache block in the region, resulting in space overhead and insertion/removal times linear with the size of a region. In contrast, large regions can be represented efficiently using a balanced tree (e.g., an AVL or red-black tree) that allow ranges to be represented as a single node, providing O(1) space overhead. The drawback of a balanced tree is that its insertion/deletion

---

\(^1\)The offset field is generally not necessary and can be omitted if doing so enables a smaller encoding.

\(^2\)This checking requires a simple extension to existing load-store queues to distinguish normal operations from UFO operations.

\(^3\)We are assuming that the hash table is sized such that chaining is relatively uncommon; the hash table can be grown dynamically as necessary to ensure this property.
time is logarithmic with the size of the tree, making it significantly slower for a large number of singleton regions. To achieve a good balance of performance for large and small regions, our current implementation is a hybrid solution that uses a hash table to store singleton regions and a balanced tree to store the rest.

As shown in Figure 4a, our balanced tree design consists of two data structures: tree nodes, which specify the range covered, and callback descriptors, which hold the callback information. In order to support overlapping regions, each tree node has a list of callback descriptors. When regions do not perfectly overlap, we sub-divide a collection of regions into segments of exact overlap. This sub-division is demonstrated in Figure 4a, where distinct tree nodes are present for the non-overlapping and overlapping regions of UFO regions A and B from Figure 2. As new regions are inserted and deleted we split and merge regions as necessary.

Our hash table structure is shown in Figure 4b. A chaining-style hash table is used where each entry in the hash-table is made at the cache block granularity. To support multiple (overlapping or not) UFO regions within a single cache block, each entry of the hash table points to a list of region descriptors: 5-tuples that store the complete information about the UFO region [start_addr, end_addr, atype, callback, registered_data]. Each slice of the hash table has an associated lock to control concurrency, as described below.

**Setting and Clearing UFO bits:** If we momentarily ignore the fact that the UFO API supports overlapping regions and concurrent invocations from independent threads, then setting and clearing the UFO bits is quite straightforward. When the UFO API is invoked to add a region, the software layer inserts an entry into either the hash table or the balanced tree and then sets the bits for every cache block covered (or partially covered) by the new region. Similarly for deleting a region, we first remove the region from the software data structure and then clear the corresponding UFO bits.

To handle overlapping UFO regions, we need to set the UFO bits to fault on the union of the conditions prescribed by any UFO region that covers a given block. When adding a UFO region, we can simply perform a read-modify-write sequence on the UFO bits, where we bit-wise OR in the new fault sources. When removing a UFO region, we need to search the tree and perform a hash table look-up\(^4\) to compute the remaining fault conditions (if any) that should be applied to the block.

To handle the race conditions resulting from the potential for concurrent insertions and deletions, our current implementation uses locks to serialize updates to any cache line’s UFO bits.\(^5\) Our implementation allows either a single update to the balanced tree or parallel updates to independent entries of the hash table. This concurrency control is evoked by a multiple-readers/single-writer lock on the balanced tree—hash table writers acquire read permission to the balanced tree—and locks on each chain of the hash table.

**Translating UFO Exceptions to UFO Callbacks:** When a UFO access violation occurs, the application is notified using exception handling mechanisms, either in kernel-mode or user-mode. Because our experimental platform, x86, does not currently support user-mode exceptions, we elected to implement UFO faults using kernel-mode exceptions passed through the standard POSIX signal handling interface. This approach requires minimal changes to both hardware and the operating system, is

---

\(^4\) One of these accesses is performed in conjunction with removing the region from that data structure.

\(^5\) A more hardware-intensive approach that enables optimistic concurrency would be to provide load-link/store-conditional equivalents for reading and writing UFO bits. These instructions, coupled with lock-free data structures (perhaps implemented using transactional memory [12]) enable a lock-free implementation.
flexible, and performs adequately provided that UFO access violations are uncommon, as they are in our applications. While user-mode exception handling represents a more significant change to the architecture, it is not complex [21] and could be used if exception handling performance was a concern.

In our experimental implementation for Linux on x86, we extended the POSIX signal handling interface by adding a SIG_UFO signal. Upon initialization, the UFO software layer registers a signal handler for the SIG_UFO signal using the POSIX sigaction system call. If a UFO fault later occurs, the operating system notifies the application, by sending it a SIG_UFO signal. As with other POSIX signals, the operating system builds a stack that contains the state of the faulting thread and the virtual address that caused the fault before invoking the UFO signal handler.

Once the UFO software layer has been called by the signal handler, it searches the balanced tree and the hash table for entries that correspond to the faulting address. For each matching entry, it invokes a callback. Once all callbacks are completed, execution is returned to the application.

### 3.4. Preserving UFO bits

To this point, we have only described how the UFO bits are stored in the memory hierarchy by extending processor cache tags and re-encoding ECC state in memory. The UFO bits are not (automatically) retained when pages are swapped to disk. To preserve these bits, we have implemented operating system support that harvests these bits when pages are swapped out and restores them when the page is brought back in.

We have implemented this support in a Linux 2.6.15.4 kernel by allocating an array with one element per swap location (much like the swap_map, which stores how many processes are sharing a given swapped page). At 2 bits per 64B block, each swapped 4KB page requires 16 bytes of storage. The bits are harvested in the add_to_swap function (when the page is removed from a process's page table), cleared in the arch_free_page function (when the physical page is freed), and restored in the read_swap_cache_async function (when the page is read back from the swap device). We verified that this code correctly preserves the UFO bits using the Simics full-system simulator modified to emulate the UFO hardware and a micro-benchmark that cyclically sets and reads back the UFO bits on a memory allocation larger than the simulated physical memory.

To observe the performance impact of these changes, we installed our modified kernel on a real machine. Since this machine does not include the UFO instructions, we emulated the reading of UFO bits with a byte load from the specified memory address; ufo_writes were emulated with a two instruction byte load-store sequence that does not change the data on the page, but does force exclusive coherence permission on the block to be attained, as would be done by the ufo_write instruction.

We find that the performance overhead of this modification is statistically insignificant in all of our application experiments. Only when we induce thrashing in the machine have we observed a significant overhead. Figure 5 shows an experiment involving a parallel build (e.g., make -j 6) of the Linux kernel on both an unmodified kernel and our modified kernel. When booted with 512MB of memory (where no thrashing occurs), we see no run-time variation. If we boot with only 64MB of memory, the machine thrashes taking (on average\(^6\)) 2.14 times as long to complete the parallel build (7 minutes 29 seconds vs. 3 minutes 29 seconds). Even during severe thrashing, our un-tuned implementation incurs only an 8% overhead.

![Fig. 5. Manually swapping UFO bits has negligible performance impact unless the machine is thrashing](image)

\(^6\)When thrashing, program execution time shows significant variability. We repeated each experiment 20 times and for the thrashing executions we had a standard deviation of 23 seconds, or 5% of the execution time. In the graphs, we have plotted a 95% confidence interval for each data point.
We tracked down the source of this overhead to be the additional swapping and memory pressure associated with storing the UFO bits in the `swap_info_struct`. We found we could mitigate the additional pressure on memory and disk bandwidth by only reading and writing the UFO bit storage array for swapped pages where at least one bit is set. If we avoid accessing it, the UFO bit storage array can itself be swapped out for long periods of time, not utilizing expensive resources. To achieve this goal, we allocate an additional array that stores a single bit per page indicating whether any of the UFO bits are set for the page. To set this bit we still have to traverse the page harvesting UFO bits, but if we set the bit to zero, we do not have to read or write the main UFO bit storage array. At one bit of storage required per 4k page, each page of this bitmap covers 128MB of swap space. With this optimization (labeled `mod2` in Figure 5) the slowdown when thrashing is reduced to 3% on average, which is not statistically significant at the 95% confidence level.

4. APPLICATIONS

In this section, we describe three applications that exploit the UFO API. In Section 4.1, we discuss how UFO provides an efficient and flexible means to detect self-modifying code in dynamic translators. In Section 4.2, we demonstrate how UFO, like iWatcher [23], can be used to facilitate software debugging. Finally, in Section 4.3, we demonstrate that UFO can be used to support speculative compiler optimizations, specifically run-time specialization.

4.1. Dynamic Translators and SMC

Dynamic translation is a technique that offers system architects and developers another design dimension by inserting a virtualization layer between a program and underlying execution hardware. Dynamic translators have been used to translate between ISAs, improve performance, and for program instrumentation [3]–[5], [7], [8], [15].

A dynamic translator operates by inspecting (at run time) an executable, from which it generates translated versions of code that it then executes. To make this process efficient, it translates groups of instructions at a time and caches these translations to avoid the cost of re-translating. This caching, however, is problematic if the application uses self-modifying code, as is the case for managed runtime workloads like the Java virtual machine. If the program modifies code that has already been translated and cached, then the dynamic translator must recognize this fact and invalidate all translations corresponding to the modified instructions.

Two approaches exist for a dynamic translator to detect SMC: translation validation and memory protection. In translation validation [7], the value of each program instruction that contributed to a translation is captured, and a prologue is prepended to each translation. Before a translation is executed, the prologue must validate the translation by checking that the original program instructions used to generate the translation have not changed. If the program instructions have changed, the prologue jumps into the dynamic translator and the corresponding translation is invalidated. Clearly, this involves a significant overhead and, furthermore, correctly handling the case where a translation modifies itself requires hardware checkpointing support [7].

The alternative approach to supporting SMC is to use memory protection mechanisms provided by hardware. In essence, the dynamic translator marks as read-only any program region containing code that has been translated. If a protected region is later written, the dynamic translator will receive a signal, disable protection for the region and invalidate all corresponding translations. The drawback of this approach primarily results from the granularity at which the memory can be protected. If protection is only available at the page granularity, programs that frequently update code pages (e.g., JVMs) and applications that intermix code and data on the same page will incur significant unnecessary overhead when all translations on the page are invalidated.

Detecting SMC with UFO: For our experiments in this application area, we modified the Pin dynamic instrumentation tool [15] to detect self-modifying code. Our code works by allocating UFO regions as we produce the translation, write-protecting each cache block of instructions before we read them for translation, guaranteeing that a handler is invoked before the translation is invalidated. For the registered data passed to the `UFO_add_region` function, we pass a pointer to a data structure (`ufo_pin_data_t`) that stores: 1) where the translation resides in the code cache, and 2) the addresses of all of the blocks read by the translation. Because Pin frequently translates a single basic block as part of multiple translations, it takes advantage of the UFO API’s ability to support multiple overlapping UFO ranges.
In function “huft_free()”, the return address in the program stack is corrupted. The stack location holding the return address has its UFO write bit set in the function prologue and cleared in the epilogue. To avoid false conflicts, the stack frame is first padded below the return address.

A special memory allocator is used that does not reuse memory that has been freed and that pads to cache-block granularity. Freed memory has its UFO read/write bits set to detect references to freed regions.

In function “huft_build()”, an access past the boundary of a heap-allocated buffer. A special memory allocator is used which pads each allocation on each side with an aligned cache block. These padded regions have their UFO read/write bits set.

Table 1. iWatcher test cases and associated monitoring instrumentation.

If a subsequent SMC write occurs, a UFO handler will be invoked. By looking at the registered data, we know which translation is soon-to-be invalid. The handler unlink the translation from other translations and removes it from the map of original program PCs to translations, effectively invalidating it. Once the translation has been invalidated, it is safe to remove the UFO regions, which will allow the faulting thread to make further progress. This is accomplished using the `ufo_pin_data_t`'s list of addresses. Finally, the handler frees the memory for the `ufo_pin_data_t` structure, before returning to the application.

If the modified code region is executed in the future, Pin will have to re-translate it to generate a new cached translation and allocate a new set of UFO regions. In Section 5, we demonstrate that this use of UFO in Pin adds negligible overhead.

### 4.2. Debugging Memory-related Errors

As the UFO API is similar to the one that iWatcher [23] implements, we can implement the same sort of software debugging tools. Through the additional hardware support, UFO and iWatcher can perform the kinds of dynamic checking found in software reliability tools like Purify [11], StackGuard [6], and Valgrind [19], but with lower execution overhead. Tools like Valgrind use code-controlled monitoring where the software is instrumented at memory accesses to check for poor memory discipline. While typically only a small fraction of instructions may be problematic, most or all memory instructions are instrumented, generally resulting in order-of-magnitude slowdowns. With UFO, all of the program’s memory accesses do not have to be instrumented (eliminating most of the overhead); instrumentation is only required where the program allocates or deallocates memory and/or stack frames, which is largely done through using a special version of the memory allocator.

We implemented three of the bug detection scenarios from the iWatcher paper, shown in Table 1. For each of these experiments — involving bugs injected into the `gzip` benchmark — we used the instrumented applications provided by the iWatcher authors. In Section 5, we show that our real-machine experiments achieve performance results equivalent to the iWatcher simulations. In the rest of this section we discuss the minor differences between the two implementations.

The primary difference is the granularity at which memory locations can be monitored; while iWatcher allows monitoring down to individual 4-byte words (yielding a factor of 16 more space overhead in the cache), our UFO implementation uses a granularity of cache blocks. We find this not to be an issue in any of these three applications, as all false conflicts can be avoided by modifying the layout of data when the program is instrumented. For the memory corruption detector, we only had to modify the memory allocator to round up allocations to the nearest cache block. For the buffer overflow detector, the padding was required to be a full cache block and aligned. Finally, for the stack smashing detector, the function’s stack allocation needed to be padded below the return address and the function arguments, which reside above the return address, copied into locals before the return address is protected. In all of these cases the impact on performance of the padding was insignificant.

The other main difference is that when allocating a large UFO region, UFO requires setting the UFO bits for each cache block individually, whereas iWatcher can allocate a single entry in the RWT. If applications were frequently adding and deleting very large UFO regions this could become perceptible, but in all of our applications this time was negligible.

### 4.3. Run-time Specialization

Partial evaluation is an optimization technique where input data is received before code generation is completed. It is based on the observation that within a given program invocation, many program variables remain constant. As a trivial example, programs that support verbose output as a command-line option typically set a variable once at the beginning of the execution...
In general, no program breakpoints are set, and thus this method always returns 0.

The same sort of optimizations can be performed within a single program invocation within the context of a runtime-optimization system (e.g., a Java virtual machine), as demonstrated in recent work [20]. The runtime system can identify candidate variables that appear invariant and specialize the code based on these values. When the compiler cannot prove the invariance of the value, as is generally the case with heap variables, performing specialization in this manner is an inherently speculative technique. To avoid introducing errors, the “invariant” values must be monitored to ensure their values do not change. If an “invariant” value is modified, on-stack replacement (OSR) [9], [13] can be used to replace the speculatively optimized code with a new version.

Two conventional approaches to monitoring “invariant” values exist: First, an “invariant” can be verified immediately prior to executing the speculative region. Not only does the overhead of this check reduce the benefit of optimization, but the compiler must ensure that the “invariant” is not modified within the optimized region. Alternatively, the address of each memory write in the program can be checked against the addresses of the “invariants” (via a write barrier) to detect whether any optimizations are invalidated. Write barriers can be removed from stores that the compiler proves to be unable to modify an “invariant” values. Thus, while the overhead of this second approach is prohibitive in contexts where little analysis can be done (e.g., a binary translator), Shankar et al. found that Java’s type safety enabled them to reduce the write-barrier overhead to 1-12%.

Guarding with UFO: The write barrier overhead, however, can be practically eliminated through the use of the UFO API, which provides an intuitive way for a runtime-optimization system to implement memory guards. When the runtime system speculatively optimizes a program region using specialization, it simply guards each assumed-invariant variable against future writes using UFO. If any invariant is later modified, the runtime will receive notification and can invalidate the appropriate speculative optimizations.

To explore the use of UFO for speculative optimization, we exploited a number of opportunities for speculative specialization within the benchmark m88ksim, a simulator for Motorola 88100 processor; we show one such opportunity in Figure 6. The method ckbrkpts determines whether a breakpoint has been set for either the current instruction or the target address of a memory operation. Typically, breakpoints are enabled, but not used and thus ckbrkpts always returns 0.

The method is called very frequently and therefore checking the breakpoints data structure accounts for 7.5% of all memory reads in the program (for the reference input set). In addition, the save and restore overhead of invoking ckbrkpts accounts for 3.5% of the stack reads and 4.1% of the stack writes in the program. This method and others like it indicate that significant potential exists for runtime specialization.

To exploit opportunities of this kind, we can wrap the invariant data structures in UFO regions, then partially evaluate the code with respect to the current values of the protected structures. To avoid false conflicts, we insert padding around the data structure so that only the data structures we intend to protect are covered by the UFO region. As previously noted, OSR is required to recover from a conflict, but a discussion on the resulting constraints to optimization is beyond the scope of this paper.
5. Evaluation

In this section, we evaluate UFO using implementations of each of the applications discussed in Sections 4.1, 4.2, and 4.3. The workload used for each application varies: we ran the SPEC2000 integer benchmarks on top of Pin\(^7\), we implemented three of the iWatcher bug detection strategies for gzip from SPEC2000, and we specialized the SPEC95 benchmark m88ksim.

For each of these applications, we verified their UFO functionality on a Simics-based [16] full-system x86 simulator that we extended to simulate the UFO hardware. These simulations give us confidence that our implementation of both the application and UFO itself (hardware, software layer, and O/S support) are complete and correct, which is important as this code is also used in our real machine-based performance evaluations.

In the subsections that follow, our goal is to reason about the performance of UFO in these applications. As noted in Section 1, UFO has three potential sources of overhead: 1) setting up and removing UFO regions, 2) checking for conflicts, and 3) handling conflicts. For the proposed system and the workloads that we investigated, we have found that the first term dominates. Previously, in Section 3.2, we described microarchitecturally how checking for access violations does not introduce execution overhead unless a fault is detected. In Section 5.1, we discuss why the overhead of conflicts need not be considered.

With minor contributions from the last two terms, the overhead from the first term dominates the overhead of the overall system. In Section 5.2, we describe real machine experiments to compute the overhead of the first term. We find the overhead to be negligible in some applications and non-negligible in others and to correlate strongly to the rate UFO regions are added and deleted by the application.

5.1. Handling Conflicts

When a conflict is detected, an application handler is invoked to perform a compensating action before the conflicting memory access is completed. In general, we are not concerned about the performance of this case for two reasons:

1) When a conflict occurs, the UFO bits have generally provided functionality to the application: for the Pin example it has detected self-modifying code, for the iWatcher examples it has detected bugs, and for the specialization example it has detected a modified “invariant.” It is not unreasonable to expect overhead to scale linearly with the conflict frequency.

2) Generally, the performance impact resulting from the conflict is going to be dominated by actions in the application’s callback function and not by the conflict handling mechanism. For our example applications, conflicts result in un-linking and invalidating translations (Pin), logging bugs for user review (iWatcher), and recompiling methods (speculation), all of which dominate the signal handler overhead.

The obvious exception to this line of reasoning is false conflicts, where the UFO covers data that the application did not intend to protect due to the granularity at which the UFO bits are allocated. As previously noted, false conflicts could be avoided in all of the applications we studied, but if there exist applications where this cannot be avoided, frequent access to a data item accidentally covered by UFO bits could result in a significant amount of meaningless overhead.

5.2. Measuring UFO overhead

In order to measure the UFO overhead, we need to run versions of the applications whose only difference is the use of the UFO API. That is, we need run workloads where no access violations will be detected; this means runs of Pin without SMC, runs of iWatcher where no bugs are encountered, and speculative optimizations whose invariants are not violated.

Because no faults will be detected, we are free to do the performance evaluation on a system without the UFO fault detection hardware. We exploit this fact to perform our performance evaluations on real machines (configuration shown in Table 2). In this way, we can accurately measure the performance impact of the UFO software, including secondary effects like additional pressure on the TLB. In addition, full speed execution enables us to use the SPEC reference input sets.

As previously noted in Section 3.4, we model the `ufo_read` instruction as a byte load, and the `ufo_write` instruction as a byte load/byte store sequence to achieve the same caching and paging behavior of these instructions. As the UFO fault checking hardware does not introduce execution overhead, not modelling it does not introduce modelling error.

\(^7\)None of the SPEC benchmarks do self-modifying code but there are important workloads that do, including Adobe Photoshop and most Java virtual machines.
The relative performance of these runs are shown in Figure 7. As can be seen, the overhead for most of the test cases is in the single digits. This is not surprising because applications are spending most of their time doing things other than performing UFO operations, as indicated by the relatively small number of UFO allocations and deallocations shown in Table 3. There are two exceptions. First, m88ksim receives a speedup because the benefits of specialization are not at all discounted by the negligible overhead of enabling “invariant” protection. Second, iWatcher-STACK is constantly—at every function call and return—adding and removing UFO regions.

To get more insight on the source of UFO overhead, Table 3 provides data on a number of aspects of each application’s usage of UFO. First, we plot the number of distinct “handles” (callback/registered data pairs) which correspond to logical uses of UFO; for example, in the Pin workload there is a one-to-one correspondence between handles and generated translations. Second, there is the number of times the application invoked the API (e.g., once per extended basic block in Pin). Third is the number of times the software layer modified a cache block’s write permissions using the ufo write instruction, and, fourth, we include the amount of the UFO tagged data at its largest point. In addition, we provide the baseline run time for each application.

If we correlate the performance in Figure 7 with the UFO statistics in Table 3, we see that slowdown correlates most closely with the rate of API calls. iWatcher-STACK has by far the highest rate of API calls and also the highest amount of overhead, followed by iWatcher-B01 and iWatcher-MC. Note that while the Pin workloads have a lot of API calls these occur over much longer executions, so their overheads are negligible. Interestingly, despite iWatcher-STACK performing UFO API calls at a rate one thousand times greater than any other application, its overhead is only a factor of 10 worse than the other applications. This can be attributed to the fact that iWatcher-STACK generally has at most a few dozen UFO regions active at a given time and always allocates singleton (i.e., one cache block) UFO regions, so it can exploit the O(1) insertion/deletion time hash table.

Also noteworthy is that our results for the iWatcher experiments correlate well with those presented in the iWatcher paper [23]: 5% vs 10% (BO1), 3% vs. 9% (MC), and 72% vs 80% (STACK). While it is reassuring to see UFO achieve lower overheads than iWatcher, the experimental results are not directly comparable for at least two reasons. First, our experiments were performed on a real machine while theirs were performed in simulation; overhead from TLB misses, O/S activity, I/O, and

<table>
<thead>
<tr>
<th>Workload</th>
<th>distinct handles</th>
<th>UFO API calls</th>
<th>UFO bit writes</th>
<th>total UFO coverage</th>
<th>base time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin Spec2k</td>
<td>AVG 26,000</td>
<td>76,000</td>
<td>86,000</td>
<td>300 KB</td>
<td>347 sec</td>
</tr>
<tr>
<td></td>
<td>MAX 141,000</td>
<td>231,000</td>
<td>252,000</td>
<td>1 MB</td>
<td>467 sec</td>
</tr>
<tr>
<td>iWatcher</td>
<td>B01 68,000</td>
<td>135,000</td>
<td>135,000</td>
<td>9 MB</td>
<td>9 sec</td>
</tr>
<tr>
<td></td>
<td>MC 68,000</td>
<td>68,000</td>
<td>550,000</td>
<td>35 MB</td>
<td>9 sec</td>
</tr>
<tr>
<td></td>
<td>STACK 1</td>
<td>500,000,000</td>
<td>500,000,000</td>
<td>10 KB</td>
<td>9 sec</td>
</tr>
<tr>
<td></td>
<td>m88ksim 8</td>
<td>8</td>
<td>9</td>
<td>500 B</td>
<td>29 sec</td>
</tr>
</tbody>
</table>

Table 3. UFO Activity. Shows the number of distinct handles (callback/registered data pairs) used in UFO API calls, the number of times ufo write was called during insertions and removals of UFO regions, and the maximum amount of data with a UFO bits set at any time during the program.
context switching act as confounding variables. Second, we likely ran a larger input set, which could have resulted in a different balance of work being performed by the application.

6. Conclusion

In this paper, we presented the idea of User-mode Fault-On (UFO) bits, using state scavenged from recoding ECC at a larger granularity to tag all memory blocks with fault-on-read and/or fault-on-write protection modes. This hardware support is used to provide a simple general-purpose API that allows programmers to specify callback functions that should be invoked if particular memory regions are accessed. By invoking the callback function before the faulting data access completes, the callback function can perform fix-up activities required before the access.

The central goal of the design was to eliminate all overhead due to checking for faults. This goal is achieved through transporting the UFO bits throughout the memory hierarchy with their cache blocks (i.e., necessitating no additional memory requests) and storing them with the L1 cache tags so they can be read as part of a load or store tag check. With no overhead from checking, the observed overhead is proportional to the degree to which UFO regions are allocated/deallocated and the frequency at which faults occur; we believe this pay-for-use overhead structure is a nice property of the UFO system.

We demonstrated the use of the UFO API with three applications: 1) the detection of self-modifying code in Pin, 2) the detection of memory-related bugs, and 3) the guarding of mostly invariant memory values. In each of these applications, our experimental results indicate that the execution overhead is either minimal or at least comparable to hardware proposed specifically for the application.

7. Acknowledgments

This research was supported in part by NSF CAREER award CCR-03047260 and a gift from the Intel corporation. We thank Lee Baugh, Pierre Salverda and Nicholas Riley for feedback on this work.

References


