mTags: Augmenting Microkernel Messages with Lightweight Metadata

Augusto Born de Oliveira
Department of Electrical and Computer Engineering
University of Waterloo, Canada
a3olivei@uwaterloo.ca

Ahmad Saif Ur Rehman
Department of Electrical and Computer Engineering
University of Waterloo, Canada
asaifurr@uwaterloo.ca

Sebastian Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo, Canada
sfischme@uwaterloo.ca

ABSTRACT

In this work we propose mTags, an efficient mechanism that augments microkernel interprocess messages with lightweight metadata to enable the development of new, system-wide functionality without requiring modification of the application source code. As such it is well suited for systems with a large legacy code base or third-party applications like phone and tablet applications.

We explored mTags in a variety of different contexts in local and distributed system scenarios. For example, we detail use cases in areas including messaging-induced deadlocks and mode propagation. To demonstrate that mTags is technically feasible and practical, we implemented it in a commercial microkernel and executed multiple sets of standard benchmarks on two different computing architectures. The results clearly demonstrate that mTags has only negligible overhead and strong potential for many applications.

1. INTRODUCTION

Software integration poses a challenge for system developers. Closed-source software makes integration and the development of system-wide functionality difficult. Access to all source code can simplify this effort, but retrofitting a system-wide feature into a large code base will still be very time consuming.

In the absence of source code, software integration and system-wide functionality can still be retrofitted from the operating system side. For example, tracing, profiling and deadlock detection are all dynamic functionalities that require runtime access to a program's execution flow, but are orthogonal to program functionality. Implementing them without changing their target program's source code has the benefit of making them instantly reusable with other programs.

One way to provide dynamic access to a program's execution flow is to attach information with programs and propagate this information as the program communicates with other system components. Other programs or the operating system can then act upon this metadata to integrate the applications and implement the required functionality. Tracing, for example, could be performed by attaching a marker to a program and logging its trajectory as that program communicates with other system components. This has been done with great overhead on the system by past work [25].

In this work we propose mTags, an efficient mechanism for attaching lightweight metadata with threads and propagating it along with messages in a microkernel OS. mTags enables the development of new, system-wide functionality without requiring modification of application source code. The well-defined messaging interface present in microkernels allows precise tracking of information exchanges between threads, and serves as a foundation for mTags' metadata propagation mechanism.

To evaluate the concept, we implemented a full prototype of mTags on a commercial microkernel and tested several use cases ranging from simple tracing to deadlock detection to mode change propagations. We show that mTags can be used to detect message-passing–related deadlocks between threads and to implement new system-wide features through mode changes in the presence of closed-source applications. By using microkernels as a base, mTags support transparent distribution. Our evaluation shows that mTags has only negligible overhead on system performance, while enabling the implementation of functionality with a fraction of the effort necessary otherwise.

The remainder of the paper follows this structure: Section 2 introduces the system model and terms for mTags. Section 3 defines measures that can be used to limit tag propagation. Section 4 outlines our implementation using the QNX Neutrino microkernel system. Section 5 motivates why our approach is useful by listing different use cases. Section 6 describes our experimental method to evaluate the system. Section 7 discusses the results of the measurements and evaluation. Section 8 goes over some of the related work. Section 9 discusses lessons learnt and specific corner cases. Section 10 describes our future work plans for mTags. Finally, Section 11 sums up and closes the work.

2. SYSTEM MODEL & TERMINOLOGY

Our approach assumes a *microkernel architecture* for the operating system similar to QNX [14], Chorus [30], Mach [1], Singularity [15] and L4 [23]. The microkernel strictly divides essential and optional services. A *service* can range from system services such as virtual memory, file servers, and device drivers, to user services like applications, web servers, and database server. The microkernel itself implements services such as a messaging layer, low-level hardware access while other services, for example drivers and managers execute in user-space. System threads implement microkernel ser-

vices whereas user threads represent all the services in user space. Services communicate via *message passing* which the microkernel provides. A *message* has a sender, a receiver, and a payload. The sender creates the message, the microkernel delivers the message to the receiver, and the receiver processes the message and the payload. The system contains two types of messages: requests and replies. *Request messages* initiate communication between two services. *Reply messages* answer request messages. Modern microkernels implement transparent distributed messaging and further message types such as pulses and signals for special purposes. Pulses and signals implement asynchronous messages.

Our approach assumes multiprogramming and multithreading. The system hosts multiple *processes* which represent resource ownership and implement services. Each process contains multiple *threads* which are the unit of schedulable processing. Furthermore, in networks of systems that run compatible microkernels, messages can transparently pass through the network from one node to another node.

The key abstraction for our work is the notion of a tag. A *tag* is an abstract entity like a label, that users or programs can attach to threads. We extend the concept of microkernel messages to include the propagation of tags. A *tagged message* is a message sent from a sender to a receiver with a payload and a tag. The payload is part of any messaging framework, the tag is our extension. The receiving thread will acquire the tags the sender had at the time of transmitting the message, while keeping the tags it had previous to this message pass. Note that we only tag request messages. A reply message will not contain or propagate any tags. We found this propagation mechanic to be intuitive and sufficient for a large variety of use cases. All future messages transmitted by the receiver will carry its current tags unless the developer deliberately chooses to change this behaviour. For example, the developer can manually add or remove tags to a thread or choose to explicitly send a plain message.

In addition to the request messages, the tagging mechanism also propagates tags through other IPC mechanisms such as pulses, signals and shared memory.

Tags can be created at any time throughout a system's lifetime. Tags that are known at start time may be assigned by the developer in the boot image for an embedded application or at the start time of a process. The main thread of the child process inherits the tag of the calling thread of the parent process in the case of a fork() call. Tags can be dynamically created and deleted, so testers can use a system call to set and remove them from processes and threads while they are running. By using the mTags API, programs can also create and set tags themselves.

mTags is a general, abstract metadata propagation mechanism. In the following, we explain some of the possible mechanics and semantics mTags can enable through the example use case of identifying a message flow in the microkernel. First, we present an abstract notion of how tags are propagated, and second, we present a more concrete use case that makes use of mTags in a real scenario.

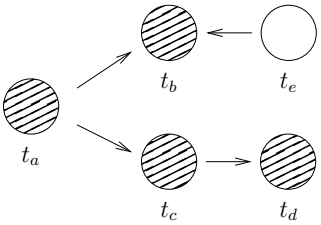

Figure 1: Abstract example of tag propagation

EXAMPLE 1. Tag Propagation: *Assume a system with threads t_a to t_e as shown in Figure 1. A circle represents a thread, and a line between two threads indicates message passing with the arrow specifying the direction of the message flows. Shaded circles represent the threads with tag τ_1. Initially, thread t_a creates the tag τ_1.*

As soon as t_a sends a message to another thread, it will pass τ_1. After receiving a message from t_a, the receiving thread's tag field will also contain τ_1 in addition to its previous tags. The further propagation of τ_1 only occurs by interaction of threads through message passing. For example in the figure above τ_1 will propagate to t_b as a result of a message pass from t_a to t_b. Since t_b never initiates any communication, it will not pass τ_1 to another thread, even when it receives messages from t_e. t_c, on the other hand, passes the tag forward to t_d with its messages.

EXAMPLE 2. Tracing Distributed File Writes: *Assume a network with two nodes: a filesystem server and a workstation. Remember that since we are discussing microkernels, distributed messaging is transparent to applications, and every non-essential service is modelled as its own process containing its own threads, including the filesystems. By tagging the application threads, we are able to track all interaction between them and the filesystem on the server.*

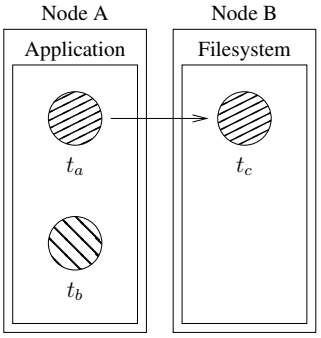

Figure 2: Tracing file writes with tags

Assume, as shown in Figure 2, that the application process is composed of two threads, t_a and t_b. Tagging each of those threads with its own individual tag (τ_a and τ_b, represented by the different striped patterns) will cause those tags to be passed to the filesystem thread t_c whenever one of them communicates with it. For example, tag τ_a or tag τ_b will propagate to the receiver thread as soon as the thread t_a or the thread t_b sends a message to another thread, respectively. In

this example, only thread t_a communicates with the filesystem thread t_c and therefore only τ_a is propagated to it. Figure 2 indicates the propagation by shading filesystem thread t_c with the striped pattern of τ_a. If both t_a and t_b were to communicate with t_c, t_c would receive both tags, τ_a and τ_b, keeping both simultaneously.

By listing the thread identifiers which contain tags τ_a and τ_b we can identify which threads access files. It is important to note that, through mTags, this can be done at run time and without modifying any application or filesystem source code.

3. LIMITING TAG PROPAGATION

During our implementation and testing of mTags, it became clear that sometimes limiting tag propagation would be of interest to the developer. Oftentimes tagging a single component would result in the eventual tagging of most system threads which would then tag the remaining userspace threads. To resolve this, we provide three concepts to control the dissemination of tags: time-to-live, tag terminators and the special treatment of system threads. The time-to-live concept is similar to network packets in IPv4 [16]. A tag has a *time-to-live* and each time the tag gets propagated, its time-to-live decreases. Once the time-to-live value reaches zero, the tag will no longer propagate to other threads. While the time-to-live provides a dynamic containment for tags, terminators create static boundaries. *Tag terminators* on threads filter out tags which will no longer propagate to and past the current thread. Developers need to set tag terminators explicitly. Finally, non-system tags can be configured to not spread to system threads and vice-versa.

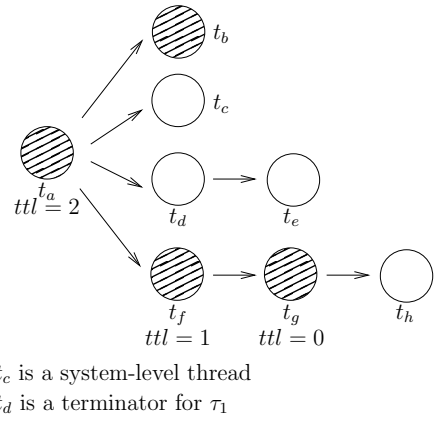

Figure 3: Abstract example of tag termination

To exemplify these mechanics, assume a system with user threads t_a to t_h as shown in Figure 3. t_c is a system-level thread. Thread t_a has the tag τ_1 as its tag with the time-to-live of $ttl = 2$. According to the propagation limiting mechanics, the following occurs:

- System-level threads do not receive user tags. t_c, the only system-level thread in the example, will reject τ_1 from t_a, because τ_1 is a tag but not a system-level tag.
- Terminators contain the propagation of tags. t_d is a terminator for τ_1, therefore t_d will reject τ_1 as it receives a message from t_a. Since t_d rejects the tag, it will not pass the tag to other threads such as t_e.
- The TTL dynamically bounds tag propagation. The TTL value for τ_1 at t_a is two. As soon as t_a passes τ_1 to t_f, the TTL value of τ_1 at t_f will be decremented to one. As shown in the figure, t_f further sends a message to t_g, hence propagating τ_1 to t_g with the decreased TTL value of zero. The zero TTL value restricts t_g from further propagating τ_1, therefore t_h will not receive τ_1 with the message from t_g.

While it has not come up in our current use-cases of mTags, it may be of interest to the developer to limit the lifetime of a tag by real time, as opposed to a number of propagations or by thread terminator boundaries. We believe that such a feature would have a straightforward implementation, by having a tagging module (described further in Section 4) associate a timer to a tag upon its creation, and handling the timer event by deleting that tag.

4. IMPLEMENTATION

To be able to evaluate the mTags concept presented in Section 2, we implemented the tag passing mechanism on a modified version of QNX Neutrino, one of the leading commercial real-time operating systems. QNX Neutrino follows a microkernel design and therefore already implements message passing through kernel primitives such as MsgSend(), MsgReceive(), MsgReply(). Because of this, most elements from the system model presented in Section 2 directly translate into our implementation.

We implemented tags by adding a bit vector to the thread control block. Each bit encodes a different tag. Using this design, if the tag vector is configured to be 32 bits wide, the developers can assign 32 different tags to a single thread. The vector size is a tunable variable in our implementation.

In addition to the tag bit vector, the thread control block also contains a terminator mask for tags. The terminator mask of a thread controls the further propagation of the tag from the thread. The thread checks the terminator mask against the tag bit vector by logical AND operation. If the respective bit for a particular tag is set in the terminator mask of a thread than the thread will not propagate the tag.

We use Neutrino's message passing routines to propagate the tag from the source thread to the receiver thread. The source thread will copy its tag vector to the message immediately in the MsgSend() routine, and, since QNX Neutrino message passing is synchronous, become *send-blocked*. The actual transfer of the active tag does not occur until a receiving thread receives the message by executing MsgReceive() routine.

Passing a tag consists of a bitwise OR and updating the tag vector of the receiver. The overhead added to the message pass system call is of a few instructions: the OR operation (between the source's and the receiver's tag vector) and the assignment operation (of the resulting tag vector on the receiver).

In case of shared memory, the kernel associates the tags of the creating thread with the shared memory object. The

kernel will pass on the associated tags to threads that use that memory region when they map it to their address space.

The TTL is an upper bound on how many steps from its originator thread a tag may be propagated. The tag structure contains the TTL as a decrementing counter. After the TTL for a tag is set, each further transfer of that tag decrements the TTL value by one. The transmission of that tag stops as soon as the TTL value reaches zero. A thread can deactivate the transfer of a tag by setting its TTL value to zero.

A global tag list contains all the tags in the system. The number of entries in the global tag list depends on the maximum number of tags supported by the system. Each entry in the tag list contains the name of the tag, the bit identifier, and a circular buffer for a thread list, which we will discuss later. The name of the tag is a user-assigned string. The bit identifier points to the particular bit in the tag vector that represents this tag. The global tag list resides in the kernel space; this is justified since it is the kernel who handles all tag propagation, the tag list needs to be safeguarded from user space tampering, and tags may outlive their creating processes. This does mean that user level applications will need to retrieve tag list information through a system call.

For every tag, we also maintain a thread list. This list contains all threads that have received the tag since its creation. Each entry in a thread list holds the thread identifier, the process identifier, and a timestamp. Whenever a thread receives a particular tag, it will add itself to that tag's thread list. To limit the overhead incurred on the system, the thread list is a circular buffer with a configurable size. In case of list overflow, the system starts overwriting the old entries in the list.

Our implementation of the mTags mechanism allows the user to log and timestamp the flow of tags through different threads in the system. We call this mechanism the *lifeline* of the tag [11]. The lifeline mechanism is built on the top of a tag's thread list. The presence of a thread in a tag's thread list, in addition to the associated timestamps, indicate the reception of that tag by that thread at that particular time. The user can access this information through a system call. Our current implementation registers the local wall time at the time of propagation, but for the distributed case a logical clock, vector clock, or matrix clock [7] would be required.

The user can configure the tagging module to enable or disable the lifeline capability. The lifeline mechanism introduces extra overhead such as the memory for the thread list and execution time for managing the list. The benchmarks presented in Section 7 show that this overhead is negligible.

The user can use a system call to remove tags from the system. The call not only removes the tag from the system but it also scans the tag thread list of the removed tag to clear the particular bit (pointing to the removed tag) from the tag vector of all the threads in the list. Furthermore, a thread can also remove a particular tag from its own tag vector.

The user can use a system call to remove tags from the system. The tag removal system call scans the tag thread list of the removed tag to clear the particular bit (pointing to the removed tag) from the tag vector of all the threads in the list. Furthermore, a thread can also remove a particular tag from its own tag vector.

To use mTags, the developer has two options: a) using command line tools, which allow the use of tags without the modification of any source code or b) include the mTags library in the application, which allows him to use a tagging-specific API from within the application. Choosing a method depends on factors such as access to and familiarity with the target source code, and the granularity with which tag creation and passing needs to be performed.

The mTags library implements functions such as SetTagsField(char *tag), which sets the tag of the calling thread, GetTagsField(), which returns the tag vector for the calling thread and LookupTag(char *tag) to do name lookups for tags. Other functions allow the creation and destruction of tags, and the listing of threads with a particular tag. These functions are also available via the command line tools.

All these functions use Neutrino's ThreadCtl() kernel call. It allows the user to access Neutrino-specific thread settings. The commands for each of the functions above are defined and passed to ThreadCtl() as its parameter. ThreadCtl() resolves the command and calls the kernel-level functions to manipulate tags at the kernel level.

To implement distributed tags, we modified Neutrino's QNET library, which is used by Neutrino's network stack (the io-pkt process). The kernel is responsible for forwarding outgoing distributed messages to this library, which then performs all marshalling and network transmission operations. By adding a tag vector to the header of all network messages, we were able to pass tags through the network exactly as we do locally. In the receiving node, the kernel reads the incoming tag vector from that header and applies it to the virtual thread that represents the remote sender thread in the local node. That virtual thread then propagates its tags exactly as a regular thread would.

Distribution brings with it two problems: 1) dealing with the limited number of possible tags and 2) maintaining global tag coherence. An active tag manager component can solve both these problems, at the expense of performance; tag creation and deletion would be synchronized across machines, and the size of tag vectors would be made variable to accommodate the tag creation needs of several applications. Another solution to this is to allocate a range within the tag vector to have globally static names and semantics, and use the rest of the vector for dynamically allocatable, exclusively local tags. For the sake of simplicity and speed, our current implementation only supports the latter solution. The expected network overhead added by mTags is minimal, as one tag vector per message (not packets, which may be more numerous) is added to transmissions. This is confirmed by the experiments shown in Section 7.

5. USE CASES

mTags is a versatile mechanism. The developer can use it in different contexts for different use cases. The following

outlines some of use cases that we have successfully tested with our prototype implementation. They should convey the value mTags adds to a system through its applicability and adaptability.

5.1 Use Case 1: Profiling/Debugging

To aid the system designer in understanding the interaction between system components, mTags enables comprehensive tracing of those interactions. If a thread creates a tag, it will be passed on with each message it sends and, eventually, all components in the system that it interacts with (directly or indirectly) will also have received that tag. Additionally, lifeline implementation offers additional information for profiling. The lifeline for a particular tag shows the complete flow of the tag through different threads in the system, helping the developer to identify how much time is spent in each component, the number of involved threads, order of execution, and the termination of the flow (either expected or unexpected).

EXAMPLE 3. Profiling a Media Decoder: *To evaluate the effectiveness of tagging as a profiling tool, we performed the following experiment: we applied tags to a media decoder (madplay [34]) to identify the frequency in which it would read and write its input and output. In our particular experiment, the input file (a large MP3 file) was stored on a disk local to the machine, and the output (the PCM WAV file that results from decoding the MP3) was stored in a CIFS share over the network. While the decoding was in progress, a tag was applied to madplay; this tag was then propagated to both devb-eide, Neutrino's IDE disk driver, whenever it read its input. The same tag was propagated to* fs-cifs*, Neutrino's implementation of the CIFS filesystem and io-pkt-v4-hc, Neutrino's network stack, when it wrote to the network share. This tagging of io-pkt-v4-hc happened indirectly, via fs-cifs, which helps the user understand the hierarchy of the system.*

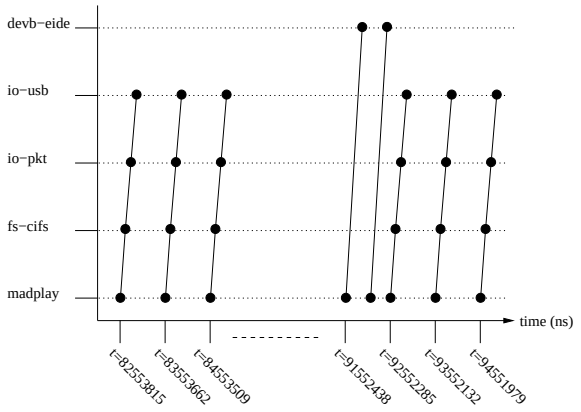

Figure 4: Profiling a Media Decoder

Figure 4 shows a plot of these tagging events over time. It contains the lifeline data for the tag, which was extracted via the command line utilities. On it the user can see the periodicity of communication between madplay and the different resource managers. Madplay sends a request to the fs-cifs resource manager approximately every millisecond. There is a delay after three of these 1ms periodic cycles where madplay reads the file from the disk to fill the input buffer. This shows how tagging can be useful to profile disk and network access in this particular case, but the same usage can be applied in a variety of other interactions.

One unforeseen interaction that tagging brought to light was between io-pkt and io-usb, which happened as a consequence of every time madplay wrote to the network share. It is still unclear why this happened, but it might be because of the path managing system Neutrino uses. There could be cases where the developer might like to suppress this last tagging event; he may use one of the propagation limiting mechanisms described in Section 3 in those cases.

5.2 Use Case 2: Multi-mode Applications

Applications often have multiple modes of operation, which are activated depending on their input. In the case of embedded systems, this is strongly related to the state of the physical environment the device is operating in. A modern GSM cellphone, for example, is capable of switching from EDGE to 3G and back depending on service availability. Applications that are aware of mode changes can take advantage of this information. For instance, a media streamer might switch to a lower bitrate on EDGE and a higher one in 3G.

Since mode changes allow important run time optimization in the embedded and real-time system domain, several related publications can be found. Alonso and de la Puente [2] propose an implementation of mode changes through a manager, which serves as a central "mode server". Li et al. [21] present an algorithm for selecting power modes that accounts for component interdependencies. Especially in scheduling [29], mode changes and their implementation are well researched topics.

While mode change mechanisms can be implemented in user space, any mode change mechanism needs to provide an infrastructure to coordinate and propagate mode changes throughout the system. Our approach using tags inherently provides this mechanism and therefore eliminates the need for a separate infrastructure. Revisiting the media streaming example, if the developer uses two separate tags for EDGE and for a 3G connection, then the media streamer will only need to check the tag to choose the right bitrate. It will receive the tag from the network stack as it will propagate the appropriate tag depending on the current active connection.

Additionally, using mTags to implement mode changes has the benefit that mode tags can propagate even through legacy systems, closed source, and non-modal software.

```
1 #include <libtag.h>

3 // in main() function:
        int result;
5
        char *tag_hipow_name = "HIPOW";
7       result = CreateTagsField(tag_hipow_name);
        result = SetTagsField(tag_hipow_name);
```

Listing 1: Code for tag creation

```
#include <libtag.h>

// during init:
        int tag_hipow_number = 0;
        char *tag_hipow_name = "HIPOW";
        tag_hipow_number = LookupTag(
            tag_hipow_name);

// before each send:
        int curr_tag = 0;
        curr_tag = GetTagsField();
        tag_hipow_number = LookupTag(
            tag_hipow_name);

        if(curr_tag && tag_hipow_number) {
            do_hipow_send();
            DeleteTag(tag_hipow_name);
        } else
            do_regular_send();
```

Listing 2: Code for tag reception

Example 4. Multi-mode applications: *To verify the effectiveness of using mTags to propagate mode changes, we modified one of Neutrino's wireless network drivers to expect a tag signifying a high power mode change. To perform the creation and propagation of power mode tags, we modified Neutrino's version of the* ping *utility. By adding the code shown in Listing 1 to its* main() *function, all messages originating from ping will carry the high power tag. The objective of this is guaranteeing that, whenever a power tag aware network stack is present, ping's packets will be transmitted at the highest power possible.*

Before actually sending each packet, the driver reads the tags associated with its thread, in the expectation that whatever component communicates with it might have sent the high power tag. This was accomplished by adding the code shown in Listing 2 to the original source code. After sending high power packets, the driver deletes the high power tag to clear it from its own tag vector (and those of any intermediate network stack threads) so that following packets are not incorrectly sent as high power.

This complete scenario—including the intermediate components that sit between ping and the network driver—is illustrated in Figure 5, where the striped pattern represents power tag awareness.

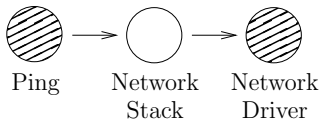

Figure 5: Power mode propagation with tags

Implementing this mechanism through mTags is advantageous in two main ways. First, the modified ping utility can be used with the original network stack with no problem whatsoever. The tags will still be propagated, but since the receiving code will not be there, they will just be ignored. Second, any components that are present between ping and the stack did not need to be modified, and, in fact, their source did not even need to be available to us. Tags are propagated through them transparently. To further illustrate this second point, Table 1 shows the number of lines of code needed to implement this through a) mTags, b) the modification of the interfaces from ping all the way down to the driver (more on this in Section 9) and c) the creation of a side channel, meaning messages are sent directly from ping to the network driver using Neutrino's interprocess communication facilities [33]. It also shows the number of components affected by the changes; in the case of the hierarchical interface column, the number varies by network stack but the number of components will be at least four: ping, libsocket, io-pkt and the particular network driver in use.

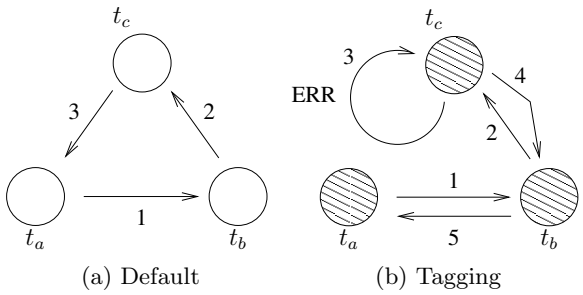

Figure 6: Deadlock detection and prevention with mTags

It is worth noting that, if the system designer decided to break the component hierarchy and create a side channel through which ping sent mode-changing messages directly to the network driver, he would be able to do so. However, this would not yield functionality equivalent to the one attained through our approach. This is because if, in the future, a power tag aware intermediate component was to be developed, neither our version of ping nor our network driver would need modification, since mode tags would already flow through all intermediate components. The new component would already receive the tag in all messages originating in ping.

	mTags	**Interfaces**	**Side channel**
Lines of code	17	40+	17
Components	2	4+	2
Full featured	Yes	Yes	No

Table 1: Evaluation of the different approaches for implementing the mode change

Under certain circumstances, we can retrofit an existing application with modes even without changing a single line of code. For example, if the modes of operation of the system are exclusively related to how threads are scheduled, then we will be able to use tags to attach scheduling-related information to threads. The OS scheduler can then use the tags in its short and long-term scheduling decisions.

5.3 Use Case 3: Detecting Message-Passing–induced Deadlocks

Most microkernel implement synchronous semantics for message passing, and this can cause deadlocks. While a process is waiting for a reply, it is in the *reply-blocked* state and cannot receive further incoming messages before its outgoing message has been replied. If a process in the reply-blocked stated receives a message as a consequence of its own outgoing message, then this will introduce a circular wait and the involved processes deadlock. For example, Figure 6(a) shows a circular wait with three threads. Thread t_a sends a message to t_b, which in turn sends a message to t_c, which finally sends a message to the originating thread t_a. This causes a circular wait and thus t_a, t_b and t_c will deadlock. Any such circular wait, regardless of the number of involved threads or the communication pattern, will result in a deadlock.

mTags provides a lightweight method to discover message-passing-induced circular waits at negligible cost (see Section 7), even in a distributed setting when using synchronous calls. Figure 6(b) illustrates the principle. The thread t_a initiates the call chain. The initiator creates a globally unique tag, in our case just DEADLOCK, and applies the tag to itself. This tag will identify all threads that are currently participating in a message flow. As t_a communicates with t_b, t_b will also acquire the tag DEADLOCK. Upon receiving the message, the kernel checks for the presence of the tag in the recipient. If the recipient current has that tag, the kernel will abort the message pass and return an error; otherwise the kernel will complete the message delivery. Figure 6(b) shows this behaviour as t_c tries to send a message to t_a. Upon delivery of an error-free reply message at the end of the interaction, the sending thread will clear its tag. Whereas, in case of a detected deadlock during message delivery, the kernel can list each participating thread's ID, the order in which they communicated if all interaction happened on the local machine only, and can start a recovery mechanism at the originating thread (in our case t_a).

Our method works transparently for messaging local or remote threads. Our extension of the network stack permits propagating tags through the network with QNET. For the example show in Figure 6(a), this means that if t_b would run on a different machine that t_a and t_c, mTags will detect the deadlock caused by the circular wait.

Note that this mechanism does not replace a full deadlock detection mechanism like [22, 10] or a distributed deadlock detection mechanism like [6, 20]. However, the method works well to detect deadlocks caused by message passing and the *reply-blocked* state while causing only negligible overhead in the system (see Section 7).

6. EXPERIMENTAL METHOD

Since the propagation of tags involves adding instructions to every message pass, it is imperative that the incurred overhead is minimal. To measure the overhead, we conducted three sets of benchmarks. The first set consists of OS benchmark programs that measure the performance of the Neutrino kernel and its closest components including the C Library, the process manager and the path manager. The second benchmark is an application-level benchmark based on the MiBench suite [13], which serves to illustrate the effects that tagging has on the performance of real world applications. MiBench has been widely used in academia to evaluate the performance of processors and other software systems [4, 17, 31, 32]. The third is a series of executions of the IOZone [26] benchmark to measure the overhead that propagating tags over the network adds to QNET.

The OS benchmark consists of 19 individual tests, each focused on stressing a particular part of the system. These tests might, for example, stress the creation of semaphores, writing to the null device or the passing of messages. Most of these test cases perform system calls which result in messages passed between the program and the service implementation. Thus, our extension affects the execution time of these system calls.

The MiBench suite is a collection of open-source benchmarks that was designed to mimic the set proposed by the EDN Embedded Microprocessor Benchmark Consortium [28] with open-source tools. Its "Consumer" category consists of five open-source applications: JPEG, MAD, LAME, TIFF and Lout, of which we used the first four for our experiments. We selected the consumer category, because it contains the most elaborate of the benchmarks found in MiBench, and it involves the most message passing between system components which is largely due to its file system operations.

To measure the overhead caused by mTags, we executed all benchmarks with and without tagging enabled in the kernel, and also with the extension for lifelines. In every test shown here, the tag vector width is set to 32-bits, the word size for the architectures used. We ran the MiBench benchmark on QNX Neutrino 6.4.2, running on a 1.8GHz Pentium 4 with 1GB of RAM. We executed the OS benchmarks on Neutrino 6.4.1 running on an Atmel AT91SAM9263-EK board, which contains a 200MHz ARM926EJ-S processor and 64MB of RAM. This should give us more deterministic results due to simpler cache and pipeline structures. In the case of the ARM platform, all code (benchmark and kernel) was compiled without GCC optimizations to eliminate compiler interference on the results; indeed, compiler optimizations made the tagged benchmarks execute faster than their untagged counterparts. The unoptimized numbers shown here are, therefore, the worst case overhead we observed. We ran the MiBench benchmark without lifeline support to avoid modification of its original source code, and also because the internal benchmarks conducted prior to them showed no evidence of significant slowdown.

To measure the execution time of each run of the OS benchmarks, we used the ClockCycles() libc function, which in the case of the ARM board, returns the value of an emulated clock with a 3.125MHz resolution. These emulated values allow performance comparisons despite being of a lower resolution than the actual CPU clock. For the MiBench benchmark, we used the clock_gettime() libc function, which allows measurements as precise as the system's free running counter. After each run of each benchmark, we recorded the execution time in a file for further processing. We analyzed the data using R 2.10.1.

To evaluate the overhead that distributed tagging incurs on QNET communication, we executed the IOZone benchmark between two Pentium 4 machines, one mounting a remote directory exported by the other through QNET. IO-

Zone is a filesystem benchmark with focus on file I/O operations. These operations include simple reads, random reads, strides, record rewrites, file rewrites, etc. Every file operation performed by IOZone in that setup lead to the propagation of tags between the two participating nodes, stressing message passing over the network. We collected statistics on a 10MB file with record sizes ranging from 4 to 8192 bytes resulting in more than 2.6 million individual measurements.

For MiBench benchmark, we collected 1000 measurements. For the OS benchmark, we collected between 250 (mmap) and 1750 (calls) values. We take 95% of the data to remove outliers due to excessive pipeline stalls, locking, cache and page misses.

7. RESULTS

MiBench. All programs have an execution time distribution that differs from the normal distribution. We established this using the Shapiro-Wilk test for normality and visual inspection. Figure 7 shows histograms of the execution time for the MiBench *lame* program on the Pentium platform. The x-axis show the execution time in seconds and the y-axis show the frequency of a particular execution time occurring in the sample. The distribution of execution times can be justified by several timing properties such as instruction scheduling anomalies [35]. Figure 8 shows the individual results for the MiBench *tiff2rgba* program on the Pentium platform. The x-axis shows the grouped results for the baseline and for tagging. The y-axis shows the execution time in seconds. Using visual inspection, both figures confirm the results of the Shapiro-Wilk test for normality. We therefore rely on robust statistics using for example the median and rank-based testing mechanism in the subsequent analysis.

Table 2 shows all results for the MiBench benchmark on the Pentium platform. None of the execution times significantly differ for any of the benchmark programs. The first column indicates the name of the benchmark program. The second column indicates whether the our basic tagging mechanics are enabled. The third column shows the median of the execution time. And finally, the last column shows the median absolute deviation for the runs. We used the Kruskal-Wallis rank sum test to check for significant slowdown when using tagging. The analysis showed no significant slowdown for any of the benchmark programs. The program *tiffmedian* has the largest difference in the median, however, it is still insignificant with a $p = 0.0458$ given a Bonferroni correction of seven tests on the data. Even if were significant, it would only be a negligible slowdown of a factor of 1.005 (0.5%).

System Calls. The distribution of the execution time for system calls also differs from a normal distribution. Similarly to the MiBench, we confirmed this using a statistical test and visual inspection. Figure 9 and Figure 10 also confirm this. We again use robust statistics instead of average and mean errors.

Table 3 shows all results of our comparison with the unmodified kernel on the ARM platform. All raw speed measurements are in CPU clock cycles. The first column lists the name of the system call tested in this row. The second column shows the median of the execution times for the baseline (i.e., the unmodified kernel). The third column shows the median absolute deviation of the baseline. The next two pairs of columns provide the same data for the modified version of the kernel with tagging and with lifelines. The last two columns show the ratio between the baseline and the tagging and the lifeline extension.

Although some results show a statistically significant difference, the overall differences are negligible and just a few clock cycles. The function most affected by tagging is msgpass and the results show no increase in the median. The reason is that (1) the best case, our extension adds eleven instructions and (2) in the worst case, our extension adds 58 instructions. Given the regular interference from the computer architecture resulting from pipeline stalls, cache misses, page alignments, and out-of-order execution, it is expected that the measurements show nearly identical values.

	Name	Tag	Median	MAD
1	jpeg	N	0.163	0.003
2		Y	0.165	0.001
3	lame	N	2.365	0.007
4		Y	2.368	0.009
5	mad	N	0.642	0.004
6		Y	0.645	0.006
7	tiff2bw	N	0.658	0.012
8		Y	0.656	0.013
9	tiff2rgba	N	0.913	0.048
10		Y	0.910	0.044
11	tiffdither	N	0.630	0.001
12		Y	0.630	0.001
13	tiffmedian	N	0.923	0.013
14		Y	0.918	0.015

Table 2: Performance summary for MiBench

Figure 7: Density plot of the execution time of the MiBench *lame* program.

	Name	Baseline		Tagging		Lifeline		Ratio	
		Median	MAD	Median	MAD	Median	MAD	Tagging	Lifeline
1	calls	2106	2972.613	2103	2969.648	2128	3005.230	0.999	1.010
2	channel	90	2.965	89	2.965	86	2.965	0.989	0.956
3	devnull	1072	65.234	1073	60.787	1080	250.559	1.001	1.007
4	devnullr	839	38.548	850	59.304	833	26.687	1.013	0.993
5	kill	80	2.965	80	2.965	76	1.483	1.000	0.950
6	malloc	109	2.965	108	2.965	106	2.965	0.991	0.972
7	msgpass	213	5.930	213	7.413	213	5.930	1.000	1.000
8	mutex	35	1.483	34	1.483	35	1.483	0.971	1.000
9	mutex_alloc	159	1.483	158	1.483	155	1.483	0.994	0.975
10	process	39934	169.016	39927	180.877	39991	140.847	1.000	1.001
11	sbrk	4123	54.856	4119	54.856	4124	35.582	0.999	1.000
12	signal	34829	1123.811	34953	1323.962	34735	853.978	1.004	0.997
13	syscall	729	382.511	736	357.307	736	367.685	1.010	1.010
14	thread	5841	40.030	5832	42.995	5867	34.100	0.998	1.004
15	timer	111	2.965	112	4.448	113	4.448	1.009	1.018
16	yield	534	10.378	537	8.896	540	8.896	1.006	1.011

Table 3: Slowdown for system calls in emulated clock ticks.

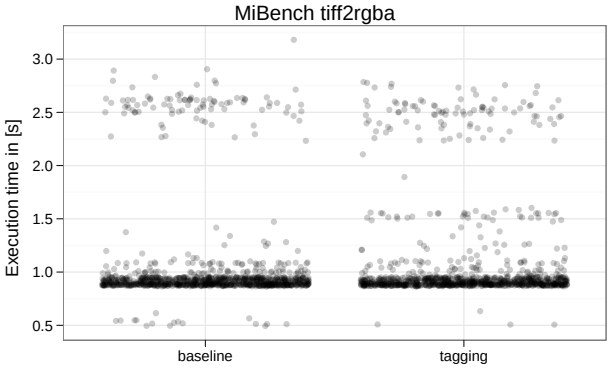

Figure 8: Individual results for MiBench *tiff2rgba* program.

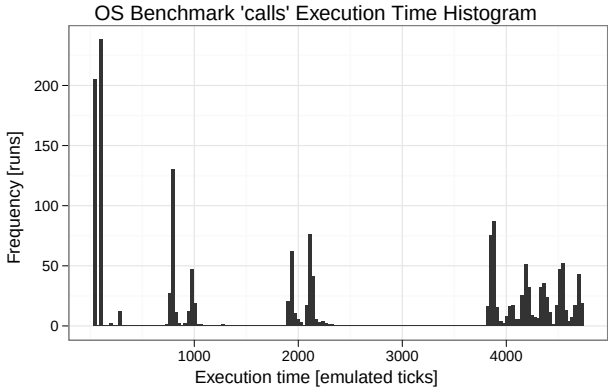

Figure 9: Histogram for the *calls* benchmark program.

Distributed Overhead. Since the addition of tags to QNET represents an extra 32-bits per message on the network, we expected the overhead to be very low. The results shown in Table 4 confirm this. The column titled 'Mean' shows the ratio between the results of IOZone over QNET with mTags enabled and mTags disabled. The column titled 'SEM' describes the standard error of the mean, and the column titled 'CI' shows the 95% confidence interval. Each row is the summary of a single record length and subsumes the results on the individual micro-benchmarks of IOZone like random read and block rewrite.

The table clearly shows that mTags, even in the distributed version with our modification of QNET at least for message sizes between 4 and 8192 bytes causes no significant overhead.

8. RELATED WORK

The use cases explored in Section 5 demonstrate the versatility of mTags; tags are more than a simple message logging or a profiling mechanism. Tag creation, deletion and propagation is completely dynamic and distributed. Furthermore, applications can act upon the presence of a tag at runtime, not only in after-the-fact trace analysis. It is, therefore, difficult to compare tagging with related approaches since we feel not many similar works are as versatile; we will, then, make a series comparisons by use case, highlighting why tagging generally has a versatility edge.

Asbestos [8] presents the idea of attaching labels to processes for controlling and tracking information flow. In Asbestos, each process contains two labels: a clearance label and a tracking label. The tracking label contains the level of all the information the process has seen whereas the clearance label represents the information level the process is allowed to see. A process can send a message to another process, if the tracking label of the sender process is less than or equal to the clearance label of the receiver process. If the receiver is cleared, its tracking label will be updated to represent the different level of information the process has viewed. In our proposed mechanism the propagation of tags could be used for similar means, by having the kernel stop messages that

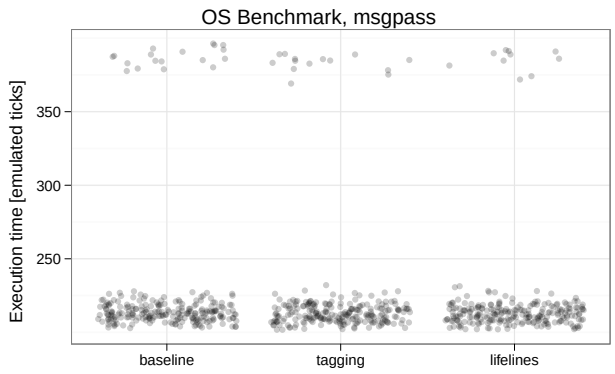

Figure 10: Individual results of the OS benchmark on the *msgpass* program.

Reclen	Mean	SEM	CI
4	0.992	0.004	0.008
8	1.001	0.009	0.018
16	1.003	0.008	0.015
32	0.997	0.000	0.001
64	0.980	0.018	0.036
128	0.999	0.000	0.001
256	0.999	0.000	0.000
512	0.990	0.021	0.041
1024	1.027	0.026	0.052
2048	1.018	0.036	0.071
4096	1.030	0.052	0.103
8192	0.979	0.042	0.082
Overall	1.001	0.018	0.035

Table 4: IOZone overhead summary results

broke the clearance relationship. Also, our approach can easily be used to implement the tracking of information flow with minimum overhead.

The concept of badges in sel4 [27] allows threads to provide multiple interfaces to incoming messages through the same endpoint. Badges propagates with the messages to let the recipient identify the source of incoming messages. mTags can be used on a much wider range of applications.

strace [12] is a tool used for profiling system calls made by a process in Linux. It logs all the system calls made by a process and the signals it receives. strace is useful for tracing the activities beyond the user space boundary into the kernel as both levels communicate through signals and system calls. Tracing of system calls is supported by mTags through the logging of interaction between user and system threads. Our implementation of mTags was done in the kernel, allowing it to trace different kinds of activities initiated by a process either from the user space or the kernel space. Furthermore, tagging is not limited to system calls, as it also profiles the interaction between threads at different system layers. Furthermore, applications can dynamically act upon the presence or absence of tags, a feature that is absent from systems that focus exclusively on tracing.

The Data Tomography [25] system proposes tracking data flow across multiple layers of abstraction by tagging the data in the system. The data tomography technique consists of inserting tags at the application, the network and the instruction level. It creates a tag map for each byte in the physical memory. The tag map of every byte stored in physical memory either the instruction or the data, points to some format of the tag. The format can vary from a simple collection of numbers to any other complex format. In contrast, our approach is to attach tags to threads rather than the physical memory in the system. Our mechanism incurs less overhead than the data tomography by avoiding the approach of tagging all the physical memory. Overhead reduction makes our tagging mechanism deployable in a production system rather than using it just for instrumenting purposes.

TaintDroid [9] is an extension to the Android [3] operating system that uses message-based taint tracking to detect the leak of sensitive information in mobile devices. While TaintDroid's approach is similar to ours (attaching metadata to IPC messages), mTags is a more general mechanic aimed at enabling a wide range of use-cases. Furthermore, because it is based on a real microkernel, mTags is capable of tracking interactions between system services and user applications; the same functionality would require modifications to the underlying native services in the case of Android. On the other, TaintDroid's is capable of tracking taints at the variable level while mTags only tracks them at the thread level.

Different kinds of related work in the past addressed the issue of dynamically tracing and debugging operating systems. This includes for example the Linux Trace tool [36], dynamic probes [24], kernel probes [19] and DTrace [5]. All of them provide mechanisms for inserting probes, sensors, and monitors into the system, with the objective of capturing data or the system state for tracing purposes. The tagging mechanism gives the user the provision of tracing the system at the granularity of the threads. The user can utilize the tagging without the deep understanding of the system and access to the source code. DTrace's D scripts are powerful, and may conceivably be used to achieve functionality similar to mTags, but they offer no direct support for distributed systems.

9. DISCUSSION

Types of Use-Cases. As our investigation of tagging progressed, and as different use cases were implemented, we were able to identify different "levels" of mTags integration in programs. Depending on the use case, the developer's use of mTags can range from what we call "application-agnostic" to "deeply integrated". We classify these levels of integration as follows, ranging from least to most integrated:

1. Application Agnostic: when the system designer uses tags without modifying any source code, and is looking to identify system-wide interactions with a low level of detail. An example of this would be tagging a thread to identify the message chains that go out from it, such as the example given in Figure 2;
2. Source Code Agnostic: when the system designer tar-

gets specific interactions, but still does so without modifying source code. The use case in Section 5.3 is an example of this;

3. Integrated: when tagging is used from within the source code of an application, but can be removed or turned off without breaking either the system or the application. The use case in Section 5.2 is an example of this;

4. Deeply Integrated: when mTags is integral to the functionality of the system, and its removal or disabling would require the reimplementation of at least part of the application.

Part of our future work is exploring more use cases at these different levels of integration, and what new interactions are enabled by different propagation mechanics and by tagging more elements of the operating system.

Tags vs. Raw Message Logging. Logging of message passing in microkernels is either standard functionality (as is the case on Neutrino) or easily implementable. The mTags mechanism differs from the simple logging of every message pass in three fundamental ways. First, it allows tags to be created in a way that they affect only a subset of all messages, effectively filtering and differentiating the particular message flow the developer is interested in. For example, two different tags can be created in either outcome of an if branch, and therefore differentiate between two types of messages that would look identical to a raw logger.

Second, tags can be very easily read and acted upon by applications at run time, which is not the case with an "eagle-eye" view such as the one provided by the Neutrino message logger. Without tagging, a thread can only know the sender of the messages it receives, while mTags enables it to construct a longer history of the message flow that lead to it and receive tag information from several hops away.

Finally, one can contain the propagation of tags—as discussed earlier in this section—without affecting the functionality of the system. The same cannot be said if one tries to limit message passing in any way.

Ease of Implementation. Our implementation of mTags on the Neutrino operating system is entirely modular and consists of relatively few lines of code. Its non-invasiveness and size are very valuable features, since they limit the likeliness of inserting new bugs into the kernel, and facilitate recertification of the mTags-compatible kernel if such a need arises. We feel these characteristics would carry to other microkernels as well.

Applicability to Non-Microkernel Systems. Message tagging as a concept is not necessarily tied to microkernel operating systems. Equivalent functionality could conceivably be implemented in monolithic systems through a couple of ways. One would be adding tag passing to every method call during the compilation process: the preprocessor can perform code insertion before each call, or tags could be handled by modified calling conventions in the compiler itself. Another way would be through aspect-oriented programming [18], where tagging itself would be an aspect. However, both of these approaches require access to the source code that is meant to use tagging.

Kernel Space vs User Space. We believe that mTags is best transparently implemented in the operating system. This is why we intercept the message passing functionality of the QNX Neutrino kernel to propagate tags across address space boundaries (or even network nodes). Normal POSIX applications that are ignorant of tags will normally receive and propagate tags.

To expand on the data presented in Section 5.2, we include here Listing 3. If a simple program that reads data from the network and writes it to disk is to implement tagging in user space, then it will have to be programmed as follows:

```
int main() {
    initialize_tags();

    send_tag(filesystem);
    file = open("filename");

    send_tag(network);
    socket = create();

    send_tag(network);
    listen(socket);

    send_tag(network);
    accept(socket);

    send_tag(network);
    while( !socket.empty() ) {
        send_tag(network);
        read(socket);

        send_tag(filesystem);
        write(file);

        send_tag(network);
    }

    send_tag(network);
    close(socket);

    send_tag(filesystem);
    close(file);
}
```

Listing 3: Tagging in user space

This implementation assumes the following: (1) the networking stack sends tag information to the application before each of its messages, (2) the filesystem supports tagging passes along tags, and (3) the semantics of the tag vector are uniform across across all system components. This implementation of mTags requires the modification of the source code of all participating components. This way of implementing tagging also requires careful tracking of every message pass to avoid bugs that may arise from omission. Performing all tagging in the kernel as we propose solves both these problems and is free of assumptions.

The implementation of tagging inside the kernel also allows us to enforce access control on tags, as it may be useful depending on the use case.

Security Model. The tagging mechanism provides different options to modify the behaviour of the tag propagation. The user can modify the tag features either through the command line options or APIs provided by tagging library. The API calls restricts unauthenticated modification by permitting only the respective threads to change the thread level features, such as tag terminator. The security model for the other features, such as TTL can be implemented by only allowing the tag owner to modify such features. In addition to these restrictions, we can restrict the command line access based on the current user privileges in the system.

10. FUTURE WORK

mTags has proved to be a promising and versatile mechanism, therefore our current plans are to continue to explore extensions to the current mechanism. We have three main lines of research we will investigate in the future: tagging more components, extending propagation mechanics and evaluating implementation on monolithic architectures.

Tagging files, devices, shared memory and possibly other operating system elements, should make mTags more expressive, and allow a whole new class of additional use cases.

Propagation mechanics have been deliberately kept simple, for performance and usability reasons; however, some use cases would benefit from different mechanics, such as "propagate on reply" in addition to or in place of propagate-on-send. We also plan to investigate the passing of data fields along with tags, to add to the expressiveness of tags. The main obstacle in this case would be the added overhead of copying data with every message pass.

Finally, the use of microkernels enable very straightforward tagging of messages, but we believe that through a mix of static analysis and dynamic tracking one could achieve similar if not equivalent functionality on monolithic kernels. The issues would be defining the edges between taggable entities and how to track all interaction between them without incurring excessive overhead.

11. CONCLUSION

In this paper we introduced the mTags, a mechanism to augment the messages of microkernel-based operating systems. We showed that mTags approach is useful in a number of contexts and situations ranging from simple tracing to multi-mode applications even in the presence of closed-source programs.

We presented the basic tag propagation mechanics and measured their impact on the system using MiBench, a system benchmark and IOZone for the distributed case. The measurements show that mTags has negligible impact on the system performance which demonstrates its adoptability for commercial applications.

12. ACKNOWLEDGEMENTS

This research was supported in part by NSERC DG 357121-2008, ORF RE03-045, ORE RE04-036, ORF-RE04-039, ISOP IS09-06-037, APCPJ 386797-09, and CFI 20314 with CMC.

13. REFERENCES

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX Summer Conference*, pages 93–112, 1986.

[2] A. Alonso and J. Antonio de la Puente. Implementation of Mode Changes with the Ravenscar Profile. In *Proc. of the 10th International Workshop on Real-time Ada Workshop*, pages 27–32, New York, NY, USA, 2001. ACM.

[3] Android. Android Operating System, 2011. `http://www.android.com`.

[4] I. Branovic, R. Giorgi, and E. Martinelli. A Workload Characterization of Elliptic Curve Cryptography Methods in Embedded Environments. In *Proc. of the 2003 Workshop on Memory Performance (MEDEA)*, pages 27–34, New York, NY, USA, 2003. ACM.

[5] B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX 2004 Annual Technical Conference*, pages 15–28, 2004.

[6] K. M. Chandy, J. Misra, and L. M. Haas. Distributed Deadlock Detection. *ACM Trans. Comput. Syst.*, 1:144–156, May 1983.

[7] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems (3rd ed.): Concepts and Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[8] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. In *Proc. of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*, pages 17–30, New York, NY, USA, 2005. ACM.

[9] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[10] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[11] A. C. for Science Department. NetLogger Toolkit - Example: Lifelines, 2010. `http://acs.lbl.gov/NetLoggerWiki/index.php/NetLogger_Toolkit`.

[12] J. Fusco. *The Linux Programmer's Toolbox.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of the IEEE International Workload Characterization (WWC-4)*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[14] D. Hildebrand. An Architectural Overview of QNX. In *Proc. of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

[15] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[16] I. S. Institute. Internet Protocol. RFC 791, Sept. 1981.

[17] A. Kejariwal, A. V. Veidenbaum, A. Nicolau, M. Girkar, X. Tian, and H. Saito. On the Exploitation of Loop-level Parallelism in Embedded Applications. *ACM Trans. Embed. Comput. Syst.*, 8(2):1–34, 2009.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[19] R. Krishnakumar. Kernel korner: kprobes-a Kernel Debugger. *Linux J.*, 2005(133):11, 2005.

[20] N. Krivokapić, A. Kemper, and E. Gudes. Deadlock Detection in Distributed Database Systems: a New Algorithm and a Comparative Performance Analysis. *The VLDB Journal*, 8:79–100, October 1999.

[21] D. Li, P. Chou, and N. Bagherzadeh. Mode Selection and Mode-Dependency Modeling for Power-Aware Embedded Systems. In *Proc. of the 2002 Asia and South Pacific Design Automation Conference (ASP-DAC)*, page 697, Washington, DC, USA, 2002.

[22] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin. Pulse: a Dynamic Deadlock Detection Mechanism Using Speculative Execution. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association.

[23] J. Liedtke. Toward Real Microkernels. *Commun. ACM*, 39:70–77, September 1996.

[24] R. J. Moore. Dynamic Probes and Generalised Kernel Hooks Interface for Linux. In *Proc. of the 4th Annual Linux Showcase & Conference (ALS)*, pages 35–35, Berkeley, CA, USA, 2000. USENIX Association.

[25] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and Visualizing Full Systems with Data Flow Tomography. *SIGARCH Comput. Archit. News*, 36(1):211–221, 2008.

[26] W. D. Norcott. Filesystem Benchmark iozone. `http://www.iozone.org/`.

[27] K. E. Philip Derrin, Dhammika Elkaduwe. sel4 Reference Manual. `http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf`.

[28] J. Poovey, T. Conte, M. Levy, and S. Gal-On. A Benchmark Characterization of the EEMBC Benchmark Suite. *IEEE Micro*, 29(5):18–29, 2009.

[29] J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[30] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Lonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. *Computing Systems*, 1:39–69, 1991.

[31] S. Schäfer and B. Scholz. Optimal Chain Rule Placement for Instruction Selection based on SSA Graphs. In *Proc. of the 10th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, pages 91–100, New York, NY, USA, 2007. ACM.

[32] B. Scholz, B. Burgstaller, and J. Xue. Minimal Placement of Bank Selection Instructions for Partitioned Memory Architectures. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–32, 2008.

[33] Q. S. Systems. System Architecture - Interprocess Communication (IPC), 2010. `http://www.qnx.com/developers/docs/6.4.1/neutrino/sys_arch/ipc.html`.

[34] I. Underbit Technologies. MAD: MPEG Audio Decoder, 2010. `http://www.underbit.com/products/mad/`.

[35] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.

[36] K. Yaghmour and M. R. Dagenais. Measuring and Characterizing System Behavior using Kernel-level Event Logging. In *Proc. of the Annual Conference on USENIX Annual Technical Conference (ATC)*, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association.