Language Support for Fast and Reliable Message-based Communication in Singularity OS

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel,
Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi

Microsoft Research

ABSTRACT

Message-based communication offers the potential benefits of providing stronger specification and cleaner separation between components. Compared with shared-memory interactions, message passing has the potential disadvantages of more expensive data exchange (no direct sharing) and more complicated programming.

In this paper we report on the language, verification, and run-time system features that make messages practical as the sole means of communication between processes in the Singularity operating system. We show that using advanced programming language and verification techniques, it is possible to provide and enforce strong system-wide invariants that enable efficient communication and low-overhead software-based process isolation. Furthermore, specifications on communication channels help in detecting programmer mistakes early—namely at compile-time—thereby reducing the difficulty of the message-based programming model.

The paper describes our communication invariants, the language and verification features that support them, as well as implementation details of the infrastructure. A number of benchmarks show the competitiveness of this approach.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Performance, Reliability

Keywords

channels, asynchronous communication, static checking, protocols, data ownership

EuroSys'06, April 18–21, 2006, Leuven, Belgium.

1. INTRODUCTION

Process isolation and inter-process communication are among the central services that operating systems provide. Isolation guarantees that a process cannot access or corrupt data or code of another process. In addition, isolation provides clear boundaries for shutting down a process and reclaiming its recources without cooperation from other processes. Inter-process communication allows processes to exchange data and signal events.

There is a tension between isolation and communication, in that the more isolated processes are, the more complicated and potentially expensive it may be for them to communicate. For example, if processes are allowed to share memory (low isolation), they can communicate in apparently simple ways just by writing and reading memory. If, on the other hand, processes cannot share memory, the operating system must provide some form of communication channels, which typically allow transmission of streams of scalar values.

In deference to performance considerations, these tradeoffs are often resolved in the direction of shared memory, even going as far as to co-locate components within the same process. Examples of such co-location are device drivers, browser extensions, and web service plug-ins. Eschewing process isolation for such components makes it difficult to provide failure isolation and clear resource management. When one component fails, it leaves shared memory in inconsistent or corrupted states that may render the remaining components inoperable.

At the other end of the spectrum, truly isolated processes communicating solely via messages have the advantage of independent failure, but at the costs of 1) a more complicated programming model (message passing or RPC) and 2) the overhead of copying data.

This paper describes how we overcome these costs in the Singularity OS [22, 23] through the use of strong system-wide invariants that are enforced by the compiler and runtime system. The main features of the communication infrastructure are:

- Data is exchanged over bidirectional channels, where each channel consists of exactly two endpoints (called Imp and Exp). At any point in time, each channel endpoint is owned by a single thread.

- Buffers and other memory data structures can be transferred by pointer, rather than by copying. These transfers pass ownership of blocks of memory. Such trans-

fers do not permit sharing between the sender and receiver since static verification prevents a sender from accessing memory it no longer owns.

- Channel communication is governed by statically verified channel contracts that describe messages, message argument types, and valid message interaction sequences as finite state machines similar to session types [17, 21].
- Channel endpoints can be sent in messages over channels. Thus, the communication network can evolve dynamically.
- Sending and receiving on a channel requires no memory allocation.
- Sends are non-blocking and non-failing.

The next section provides context for the present work by describing Singularity. Section 2 presents channels and how programs use them. Section 3 describes the programming model allowing static verification of resource management. The implementation of channels is described in Section 4 along with some extensions in Section 5. Section 6 discusses benchmarks and experience with the system. Finally, Sections 7 and 8 discuss related and future work.

1.1 Singularity

The Singularity project combines the expertise of researchers in operating systems, programming language and verification, and advanced compiler and optimization technology to explore novel approaches in architecting operating systems, services, and applications so as to guarantee a higher level of dependability without undue cost.

The increased reliability we are seeking stems in good part from the following design and architectural decisions:

1. All code, with the exception of the hardware abstraction layer and parts of the trusted runtime, is written in an extension of C# called Sing#, a type-safe, object-oriented, and garbage collected language. Sing# provides support for message-based communication. Using a type and memory safe language gurantees that memory cannot be corrupted and that all failures of the code are explicit and manifest as high-level exceptions (possibly uncaught), not random crashes or failures.

2. The operating system itself is structured as a microkernel in which most services and drivers are separate processes communicating with other processes and the kernel solely via channels. Processes and the kernel do not share memory. This promotion of smaller independent components allows for independent failure of smaller parts of the system. Failure can be detected reliably and compensating actions can be taken, for example restarting of services.

Implemented naively, these design decisions lead to an inefficient system due to the high frequency of process boundary crossings implied by the large number of small isolated components, the cost of copying message data from one process to another, and the overhead imposed by a high-level language and garbage collection. Singularity avoids these costs using the following techniques:

1. Isolation among processes and the kernel is achieved via the statically verified type safety of the programming language rather than hardware memory protection. Software based isolation allows all code to run in the highest privileged processor mode and in a single virtual address space, thereby removing the cost of changing VM protections and processor mode on process transitions.

2. The efficient communication technique described in this paper enables the exchange of data over channels without copying. Such an approach is hard to make safe in traditional systems not based on type safe languages. In our setting, we obtain safety via compile-time verification guaranteeing that threads only access memory they own. The static verification of this property is vital to the integrity of process isolation.

3. All code of a processes is known on startup (no dynamic code loading), enabling the compiler to perform a whole program analysis on each process during compilation to machine code. Global program optimizations can eliminate many of the costs incurred with high-level object-oriented languages, such as for instance crossing many levels of abstraction and object-oriented dispatch. Additionally, since each process has its own runtime system and garbage collector, processes do not have to agree on common object layouts and GC algorithms. Each process can be compiled with the object layout (including the removal of unsed fields) and GC algorithm best suited to its needs.

The Singularity operating system prototype consists of roughly 300K lines of commented Sing# code. It runs on x86 hardware and contains a number of drivers for network cards, IDE disks, sound and keyboard devices, a TCP/IP network stack, and a file system. All drivers and services are separate processes communicating via channels. Thus, even network packets and disk blocks are transmitted between drivers, the network stack, and the file systems as messages.

2. CHANNELS

A channel is a bi-directional message conduit consisting of exactly two endpoints, called the channel peers. A channel is loss-less, messages are delivered in order, and they can only be retrieved in the order they were sent. Semantically, each endpoint has a receive queue, and sending on an endpoint enqueues a message on the peer's queue.

Channels are described by channel contracts (Section 2.3). The two ends of a channel are not symmetric. We call one endpoint the importing end (Imp) and the other the exporting end (Exp). They are distinguished at the type level with types C.Imp and C.Exp respectively, where C is the channel contract governing the interaction. The next sections describe in more detail what data is exchanged through channels, how channel contracts govern the conversation on a channel, and what static properties are enforced by verification.

2.1 The Exchange Heap

Processes in Singularity maintain independent heaps and share no memory with each other. If we are to pass data from one process to another, that data cannot come from a process' private heap. Instead, we use a separate heap,

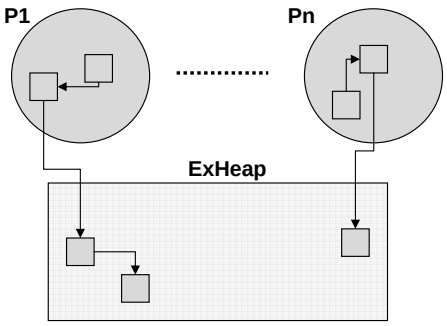

Figure 1: Process heaps and the exchange heap

called the *exchange heap*, to hold data that can move between processes. Figure 1 shows how process heaps and the exchange heap relate. Processes can contain pointers into their own heap and into the exchange heap. The exchange heap only contains pointers into the exchange heap itself. Although all processes can hold pointers into the exchange heap, every block of memory in the exchange heap is owned (accessible) by at most one process at any time during the execution of the system. Note that it is possible for processes to have dangling pointers into the exchange heap (pointers to blocks that the process does not own), but the static verification will prevent the process from accessing memory through dangling references.

To make the static verification of the single owner property of blocks in the exhange heap tractable, we actually enforce a stronger property, namely that each block is owned by at most one thread at any time. The fact that each block in the exchange heap is accessible by a single thread at any time also provides a useful mutual exclusion guarantee.

2.2 Exchangeable Types

Exchangeable types encompass the type of all values that can be sent from one process to another. They consist of scalars, *rep structs* (structs of exchangeable types), and pointers to exchangeable types. Pointers can either point to a single exchangeable value or to a vector of values. Below, we explain in more detail how these types are declared.

```
rep struct NetworkPacket {
  byte[] in ExHeap data;
  int bytesUsed;
}

NetworkPacket* in ExHeap packet;
```

The code above declares a rep struct consisting of two fields: data holds a pointer to a vector of bytes in the exchange heap and bytesUsed is an integer. The type of variable packet specifies that it holds a pointer into the exchange heap pointing to a NetworkPacket struct.

Allocation in the exchange heap takes the following forms:

```
byte[] in ExHeap vec = new[ExHeap] byte[512];

NetworkPacket* in ExHeap pkt = new[ExHeap] NetworkPacket(...);
```

The syntax for **new** is retained from C#, but the ExHeap modifier makes it clear that the allocation is to be perfomed in the exchange heap. Blocks in the exchangeable heap are deleted explicitly via the statement **delete** ptr, modeled after C++. Section 3 shows why this cannot lead to dangling pointer accesses or leaks.

Endpoints themselves are represented as pointers to rep structs in the exchangeable heap so that they can be exchanged via messages as well. Section 4 describes in more detail how endpoints are implemented.

2.3 Channel Contracts

Channel contracts in Sing# consist of message declarations and a set of named protocol states. Message declarations state the number and types of arguments for each message and an optional message direction. Each state specifies the possible message sequences leading to other states in the state machine.

We explain channel contracts via a simplified contract for network device drivers.

```
contract NicDevice {
  out message DeviceInfo (...);
  in  message RegisterForEvents(NicEvents.Exp:READY evchan);
  in  message SetParameters(...);
  out message InvalidParameters (...);
  out message Success();
  in  message StartIO();
  in  message ConfigureIO();
  in  message PacketForReceive(byte[] in ExHeap pkt);
  out message BadPacketSize(byte[] in ExHeap pkt, int mtu);
  in  message GetReceivedPacket();
  out message ReceivedPacket(NetworkPacket* in ExHeap pkt);
  out message NoPacket();

  state START: one {
    DeviceInfo! → IO_CONFIGURE_BEGIN;
  }

  state IO_CONFIGURE_BEGIN: one {
    RegisterForEvents? →
      SetParameters? → IO_CONFIGURE_ACK;
  }

  state IO_CONFIGURE_ACK: one {
    InvalidParameters! → IO_CONFIGURE_BEGIN;
    Success! → IO_CONFIGURED;
  }

  state IO_CONFIGURED: one {
    StartIO? → IO_RUNNING;
    ConfigureIO? → IO_CONFIGURE_BEGIN;
  }

  state IO_RUNNING: one {
    PacketForReceive? → (Success! or BadPacketSize!)
      → IO_RUNNING;
    GetReceivedPacket? → (ReceivedPacket! or NoPacket!)
      → IO_RUNNING;
    ...
  }
}
```

A channel contract is written from the exporting view point and starts in the first listed state. Message sequences consist of a message tag and a message direction sign (! for Exp to Imp), and (? for Imp to Exp). The message direction signs are not necessary if message declarations already contain a direction (**in**, **out**), but the signs make the state machine more human-readable.

In our example, the first state is START and the network device driver starts the conversation by sending the client (probably the netstack) information about the device via message DeviceInfo. After that the conversation is in

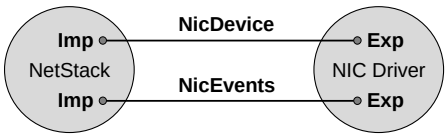

Figure 2: Channels between network driver and netstack

the IO_CONFIGURE_BEGIN state, where the client must send message RegisterForEvents to register another channel for receiving events and set various parameters in order to get the conversation into the IO_CONFIGURED state. If something goes wrong during the parameter setting, the driver can force the client to start the configuration again by sending message InvalidParameters. Once the conversation is in the IO_CONFIGURED state, the client can either start IO (by sending StartIO), or reconfigure the driver (ConfigureIO). If IO is started, the conversation is in state IO_RUNNING. In state IO_RUNNING, the client provides the driver with packet buffers to be used for incoming packets (message PacketForReceive). The driver may respond with BadPacketSize, returning the buffer and indicating the size expected. This way, the client can provide the driver with a number of buffers for incoming packets. The client can ask for packets with received data through message GetReceivedPacket and the driver either returns such a packet via ReceivedPacket or states that there are no more packets with data (NoPacket). Similar message sequences are present for transmitting packets, but we elide them in the example.

From the channel contract it appears that the client polls the driver to retrieve packets. However, we haven't explained the argument of message RegisterForEvents yet. The argument of type NicEvents.Exp:READY describes an Exp channel endpoint of contract NicEvents in state READY. This endpoint argument establishes a second channel between the client and the network driver over which the driver notifies the client of important events (such as the beginning of a burst of packet arrivals). The client retrieves packets when it is ready through the NicDevice channel. Figure 2 shows the configuration of channels between the client and the network driver. The NicEvents contract is shown below.

```
contract NicEvents {
  enum NicEventType {
    NoEvent, ReceiveEvent, TransmitEvent, LinkEvent
  }

  out message NicEvent(NicEventType eventType);
  in  message AckEvent();

  state READY: one {
    NicEvent! → AckEvent? →READY;
  }
}
```

So far we have seen how channel contracts specify messages and a finite state machine describing the protocol between the Imp and Exp endpoints of the channel. The next section describes how programs use channels.

2.4 Channel Operations

To create a new channel supporting contract C, the following code is used:

```
C.Imp imp;
C.Exp exp;
C.NewChannel(out imp, out exp);
```

Two variables imp and exp of the corresponding endpoint types are declared. These variables are then initialized via a call to C.NewChannel which creates the new channel and returns the endpoints by initializing the out parameters.[1]

Endpoint types contain method definitions for sending and receiving messages described in the contract. For example, the endpoints of the NicEvents contract contain the method declarations shown in Listing 1. The semantics of these methods is as follows. Send methods never block and only fail if the endpoint is in a state in the conversation where the message cannot be sent. Receive operations check that the expected message is at the head of the queue and if so, return the associated data. If the queue is empty, receives block until a message has arrived. If the message at the head of the queue is not the expected message or the channel is closed by the peer, the receive fails.

As is apparent from these declarations, there is no need to allocate a message object and fill it with the message data. Only the message arguments are actually transmitted along with a tag identifying the message. The sender and receiver only manipulate the message arguments, never an entire message. This property is desirable, for it avoids the possibility of failure on sends. Furthermore, as we discuss in Section 2.6, it simplifies the implementation.

Direct calls to the receive methods are not useful in general, since a program has to be ready to receive one of a number of possible messages. Sing# provides the **switch receive** statement for this purpose. Here's an example of using the NicDevice channel endpoint in the server:

```
NicDevice.Exp:IO_RUNNING nicClient ...

switch receive {
  case nicClient.PacketForReceive(buf):
    // add buf to the available buffers, reply
    ...

  case nicClient.GetReceivedPacket():
    // send back a buffer with packet data if available
    ...

  case nicClient.ChannelClosed():
    // client closed channel
    ...
}
```

```
rep struct Imp {
  void SendAckEvent();
  void RecvNicEvent(out NicEventType eventType);
}

rep struct Exp {
  void SendNicEvent(NicEventType eventType);
  void RecvAckEvent();
}
```

Listing 1: Methods on endpoints

[1] In C# an out parameter is like a C++ by-ref parameter, but with the guarantee that it will be initialized on all normal code paths.

In general, each case can consist of a conjunction of patterns of the form

```
pattern−conjunction :− pattern [ && pattern−conjunction ]
                     |  unsatisfiable
           pattern :− var.M(id ,...)
```

A pattern describes a message M to be received on an endpoint in variable *var* and a sequence of identifiers *id* ,... that will be bound to the message arguments in the case body.

A pattern is satisifed if the expected message is at the head of the endpoint's receive queue. The execution of a **switch receive** statement proceeds as follows. Each case is examined in order and the first case for which all pattern conjuncts are satisfied executes. The messages of the matching case are removed from the corresponding endpoints and the message arguments are bound to the identifiers before execution continues with the case block. Blocks must end in a control transfer, typically a **break**.

If no case is satisfied, but some cases could be satisfied if more messages arrive, the **switch receive** will block until the arrival of new messages. If on the other hand, none of the cases could be satisfied by more message arrivals, the **switch receive** continues with the **unsatisfiable** block.

2.5 Channel Closure

Channels are the only ties between processes and thus act as the unique failure points between them. We adopted the design that channel endpoints can be closed at any time, either because a process explicitly closes an endpoint via **delete** ep, or because the process terminates (normally or abruptly) and the kernel reclaims the endpoint. Each endpoint is closed independently but a channel's memory is reclaimed only when both ends have been closed.

This independent closure of endpoints makes it easier to provide a clean failure semantics and a single point where programs determine if a channel peer has closed its endpoint. As we mentioned above, sends to endpoints never fail if the endpoint is in the correct state in the conversation, even if the peer endpoint is already closed. However, on receives a special message ChannelClosed appears in the receive queue once all preceeding messages have been received and the peer has closed its end. Once an endpoint has been closed, the compiler verifies that no more sends or receives can be performed on that endpoint. The ChannelClosed messages are implicit in the channel contracts.

2.6 Channel Properties

A main requirement of the channel implementation for Singularity is that sends and receives perform no memory allocation. The requirement has three motivations: 1) since channels are ubiquitous and even low-level parts of the kernels use channels, we must be able to send and receive in out-of-memory situations, and 2) if memory allocation occurred on sends, programs would have to handle out of memory situations at each send operation, which is onerous, and 3) make message transfers as efficient as possible.

In order to achieve this no-allocation semantics of channel operations, we enforce a *finiteness* property on the queue size of each channel. The rule we have adopted, and which is enforced by Sing#, is that each cycle in the state transitions of a contract C contains at least one receive and one send action. This simple rule guarantees that no end can send an unbounded amount of data without having to wait for a message. As we will describe in Section 4, this rule allows us to layout all necessary buffers in the endpoints themselves and pre-allocate them as the endpoints are allocated. Although the rule seems restrictive, we have not yet seen a need to relax this rule in practice.

The second property enforced on channels is that they transfer only values of exchangeable types. Allowing a reference to an object in the GC'ed heap to be transferred would violate the property that no processes contain pointers into any other processes heap. Thus, enforcing this property statically is vital to the integrity of processes.

The third and final *send-state* property concerns endpoints transferred in messages. Such endpoints must be in a state of the conversation where the next operation is a send on the transferred endpoint, rather than a receive. This property is motivated by implementation considerations. As we will discuss in Section 4, each block in the exchange heap (thus each endpoint) contains a header indicating which process owns it at the moment. In order for send operations to update this information correctly, one has to avoid the following scenario: process A sends endpoint e to process B. Before it has transferred e's ownership to B, process C, holding the peer f of e tries to send a block m on f. C finds that the peer is owned by A. After that, A transfers e to B, but C still thinks it needs to make A the owner of m, whereas B should be the owner.

This scenario is essentially a race condition that could be attacked using various locking schemes. But such locking would involve multiple channels, is likely to cause contention and is difficult to implement without deadlocks. The send-state property rules out this race statically and allows for a simple lock-free implementation of transfers.

3. RESOURCE VERIFICATION

One of the goals for the Singularity project is to write code in a high-level garbage-collected language, thereby ruling out errors such as referencing memory through dangling pointers. However, garbage collection is local to a process and ownership of memory blocks transferred through channels requires the reintroduction of explicit memory management into the language.

Consider the operation ep.SendPacketForReceive(ptr) from the NicDevice contract. The ptr argument points to a vector in the exchange heap (type **byte** [] **in** ExHeap). After the send operation, the sending process can no longer access this byte vector. From the sender's perspective, sending the vector is no different than calling **delete** on the vector. In both cases, the value of ptr constitutes a dangling reference after these operations.

Avoiding errors caused by manual memory management in C/C++ programs is a long standing problem. Thus, it would appear that adopting an ownership transfer model for message data would set us back in our quest for more reliable systems.

Fortunately, the programming language community has made important progress in statically verifying explicit resource management in restricted contexts [10, 12, 32]. The rules described in this section for handling blocks in the exchange heap allow a compiler to statically verify that 1) processes only access memory they own, 2) processes never leak a block of memory, and 3) send and receive operations on channels are never applied in the wrong protocol state.

3.1 Tracked Data

In order to make the static verification of block ownership tractable, we segregate data on the GC heap from data on the exchange heap. This segregation is explicit at the type level, where pointers into the exchange heap always have the form R∗ **in** ExHeap or R[] **in** ExHeap. Any other type is either a scalar or must live in the GC heap. We use the term *tracked pointer* to refer to pointers (including vectors) to the exchange heap because the verification tracks ownership of such pointers precisely at compile time. Pointers into the GC'ed heap need not be tracked generally, since the lack of explicit freeing and presence of garbage collection guarantee memory safety for those objects.

In the following sections, we first present a very restricted, but simple method of tracking ownership, and then gradually relax it to allow more programming idioms to be expressed and verified.

3.1.1 Basic tracking

The simplest form of ownership tracking restricts tracked pointers to appear only on the stack (i.e., as method parameters, return values, and local variables). The compiler simply rejects tracked pointer types in any other position. With these restrictions, GC'ed objects never point to tracked blocks, and tracked blocks themselves only contain scalars. Now it is fairly easy to classify which pointers are owned within the context of a method by considering how ownership is acquired and is lost. There are three ways ownership of a tracked block is acquired by a method:

1. A pointer to the block is passed in as a parameter
2. A pointer to the block is the result of a method call
3. A pointer to the block is the result of **new** operation

Similarly, there are three ways a method can lose ownership of a tracked block:

1. A pointer to the block is passed as an actual parameter in a call
2. A pointer to the block is a result of the method
3. A pointer to the block is the argument to **delete**

Let's look more closely at how ownership of blocks is transferred from a caller to a callee. There are two common cases when passing a tracked pointer to a method:

- The ownership of the block pointed to by the parameter is passed to the callee temporarily, i.e., upon return, ownership reverts back to the caller. In the classification above, we consider that as two transfers: from caller to callee as a parameter, and then as an implicit result from callee to caller upon return. We consider this the default case for method parameters (including **this**).
- The ownership of the block pointed to by the parameter is passed to the callee permanently (e.g., arguments to Send methods). We say that ownership of such parameters is *claimed* and use the annotation [Claims] on such parameters.

With these insights, it is simple to track the status of each pointer value at each program point of the method, via a data-flow analysis, to determine whether a pointer is definitely owned. A complication is that the analysis must keep track of local aliases. This issue can be dealt with using alias types [34, 15] and we won't comment on it further. Below is a well-typed example.

```
1  static void Main() {
2    int [] in ExHeap vec = GetVector();
3    for (int i=0; i<vec.Length; i++) { vec[i] = i ; }
4    Reverse(vec);
5    Consume(vec);
6  }
7
8  static int [] in ExHeap GetVector() {
9    int [] in ExHeap vec = new[ExHeap] int[512];
10   return vec;
11 }
12
13 static void Reverse(int [] in ExHeap vec) {
14   for (int i=0; i<(vec.Length+1)/2; i++) {
15     int peer = vec.Length−1−i;
16     int tmp = vec[i];
17     vec[i] = vec[peer];
18     vec[peer] = tmp;
19   }
20 }
21
22 static void Consume([Claims] int [] in ExHeap vec) {
23   delete vec;
24 }
```

Let's consider the ownership information inferred by the verification at each program point of the above program. At line 1 no blocks are owned since Main has no parameters. After line 2, the method owns the block pointed to by vec (case 2 of acquiring ownership). On line 3, the checker can correctly verify that the method owns the vector being indexed. At line 4, we check that the method owns the argument to the call, since that is what the method Reverse assumes of its parameter. This check passes. After the call to Reverse, the method still owns the block pointed to by vec, since Reverse only took temporary ownership of its parameter. Thus at line 5, we can verify that the method owns the argument to Consume. Since the parameter of Consume is claimed, the method does not own any more blocks at line 6. We have reached the return point of the method without any owned blocks and can thus conclude that Main does not leak any blocks.

Method GetVector starts out without any owned blocks (it has no tracked parameters). After line 9, the method owns the newly allocated block pointed to by vec (case 3 of acquiring ownership). At line 10, ownership of this block is consumed by virtue of returning it from the method (case 2 of consumption). After that, no more blocks are owned and thus no leaks are present.

The reasoning for Reverse is very simple. On entry, the method owns the block pointed to by vec. We can thus verify that all index operations act on an owned vector in lines 14–18. At the return of Reverse, the method must still own the block that was pointed to by vec on entry (since it is obliged to return ownership to the caller). It does, so ownership is reclaimed, leaving the method with no more owned blocks and thus proving that Reverse does not leak.

Finally, on entry, method Consume owns the block pointed to by parameter vec. The **delete** correctly operates on an

owned block at line 23 and consumes ownership (case 3 of consuming ownership). The method ends without any outstanding blocks and thus has no leaks.

To illustrate the kinds of errors that the verification will detect, suppose that the Main method calls Consume and Reverse in opposite order:

```
4  Consume(vec);
5  Reverse(vec);
```

The Sing# compiler emits the following error on that code.

```
(5,11): error CS6084: Accessing dangling reference
                      'Test.GetVector.returnValue'
```

The call to Consume consumes ownership of the block (since the formal parameter is annotated with [Claims]) and thus at the call to Reverse, the verification fails, since the argument pointer does not point to an owned block.

As another potential error, suppose we omit line 23 which deletes the vector in the Consume method. Sing# emits the following error in that case.

```
(24,1): error CS6095: Leaking owned reference 'vec'
```

To complete our discussion of the basic ownership checking, we need to address aliasing. If a method takes more than one tracked pointer, the assumption is that the pointers are distinct. The verification checks each call site to make sure no aliasing of parameters arises.

Also, **ref** and **out** parameters are handled in the obvious way. A **ref** parameter of tracked type requires ownership of the contents of the ref parameter on entry and returns ownership of the ptr in the ref parameter on exit of the method. For **out** parameters, no ownership transfer occurs on entry.

In practice, the verification needs to handle **null** specially, since no memory and thus no ownership is associated with such pointers. We use a "must-be-null" analysis before checking ownership in order to handle idioms such as:

```
1  void Consume([Claims] T* in ExHeap ptr) {
2    if (ptr != null) {
3      delete ptr;
4    }
5  }
```

The analysis needs to know that ptr is **null** on the control flow edge that skips the conditional block. This knowledge lets us determine at the merge point after the conditional (line 4) that no more resources are held.

3.1.2 Tracked pointers in the GC heap

The basic tracking is sufficient if all tracked pointers live on the stack. However, there are situations, in which one needs to escape the static checking of ownership or truly share some endpoint or other block in the exchange heap among multiple threads in the same process. To this end, Sing# provides a predefined class TCell<T> that is a GC wrapper for a tracked pointer. A TCell turns static verification of memory management into dynamic checking. Its interface is as follows:

```
class TCell<T> {
  TCell([Claims] T* in ExHeap ptr);
  T* in ExHeap Acquire();
  void Release([Claims] T* in ExHeap ptr);
}
```

The semantics of a TCell is it can be either full (containing a tracked pointer) or empty. On construction, a cell consumes the argument pointer and thus starts out as full. An Acquire operation blocks until the cell is full and then returns the tracked pointer, leaving the cell empty. A Release blocks until the cell is empty and then stores the argument pointer in the cell, leaving it full. TCells can thus be used to pass ownership of a tracked block from one thread to another within the same process.

As mentioned above, static verification of memory management is turned into dynamic checking by a TCell. For example, if a thread tries to acquire a cell twice and no other thread ever releases into the cell, then the thread blocks forever. Furthermore, TCells rely on the GC finalizer to **delete** the contained block if the cell becomes unreachable.

3.1.3 Tracked structs

So far we only have pointers from the stack and special cells into the exchange heap. But it is useful for pointers to link from the exchange heap to other blocks in the exchange heap as well. For example, the struct NetworkPacket used in the NicDevice contract contains a byte vector in a field. To verify code involving such fields we restrict how these fields are accessed: to access fields of tracked structs, the struct must be *exposed* using an expose statement as in the following example.

```
1 void DoubleBufferSize(NetworkPacket* in ExHeap pkt) {
2   expose(pkt) {
3     byte[] in ExHeap old = pkt→data;
4     pkt→data = new[ExHeap] byte[old.Length*2];
5     delete old;
6   }
7 }
```

In order to check this construct, the verification must track whether tracked structs are exposed or unexposed. An unexposed struct pointer satisfies the invariant that it owns all memory blocks it points to. When the struct is exposed, ownership of the pointed-to blocks is transferred to the method context doing the **expose**. At the end of the **expose** block, ownership of the current pointers in the fields of the exposed struct is transferred from the method to the struct.

Thus, in the example above, at line 2, the method owns the block pointed to by pkt and it is unexposed. After line 2, the same block is now exposed and the method also owns the block pointed to by pkt→data. After line 3, the method still owns the same blocks, but old points to the same block as pkt→data. After line 4, the method owns three blocks pointed to by old, pkt, and pkt→data. After line 5, ownership of old is consumed. At line 6, we unexpose the struct pointed to by pkt which consumes the contained pointer pkt→data. Thus, at line 7, the method owns only the block pointed to by pkt, and its status is unexposed. Since ownerhips of this block passes back to the caller at this point, we managed to verify this method. If the **delete** statement were omitted, the checker would complain that the old byte vector is leaking.

3.2 Vectors of tracked data

The final extension we present here is supporting vectors of tracked data. The problem in this setting is that the verifier cannot track the ownership status of an unbounded number of elements in a vector. Thus, we restrict access to the vector to one element at a time, requiring an expose of the vector

element in the same way as we exposed an entire struct. Below is an example showing the manipulation of a vector of network packets, using the previously defined method to double the buffer in each packet.

```
void DoubleAll(NetworkPacket*[] in ExHeap vec) {
  for (int i=0; i<vec.Length; i++) {
    expose(&vec[i]) {
      DoubleBufferSize(vec[i]);
    }
  }
}
```

Note that **expose** always acts on the slot of the vector, not the content of the slot. The invariant of a vector is that if the vector has no exposed slot, all the contents of all slots is owned. When exposing a slot, ownership of that slot's contents is transferred to the method, which can then act on it. On exit of the **expose** block, the ownership of the slot's contents is transferred back to the vector in order to satisfy its invariant.

3.2.1 Discussion

The **expose** blocks shown in the previous sections have no impact on the generated code or the instructions executed at runtime. The exposing and unexposing of vectors and structs are only hints for the static analysis on how to prove the code safe. All checking of ownership invariants is performed statically at compile time. The only runtime operations in conjunction with ownership are acquires and releases of TCells.

As mentioned earlier, channel endpoints are just pointers into the exchange heap and are thus tracked like other blocks in the exchage heap. Additionally, the verifier statically tracks the protocol state of each endpoint. This is again a relatively simple matter if we specify the protocol state of each endpoint whenever a message acquires ownership of them. Thus, we require endpoint arguments and results of methods to specify the protocol state of the endpoint. Similarly, message declarations must specify the state of endpoints they transport and TCells specify the state of the endpoints they encapsulate.

The verifier computes for each program point and each owned endpoint the possible protocol states of the endpoint. At Send calls, the verifier checks that the endpoint is in a state where the message can be sent. At Select calls, the verifier determines if the cases are exhaustive, i.e., whether there is a case for every message combination on the endpoints.

In summary, the static tracking ensures the following system-wide invariants:

- Each block in the exchange heap has at most one owning thread at any point in time.
- Blocks in the exchange heap are only accessed by their owner. (thus no access after free or transfer of ownership)
- A block not owned by a thread is owned by a TCell and thus effectively by the garbage collector's finalizer thread.
- The sequence of messages observed on channels correspond to the channel contract.

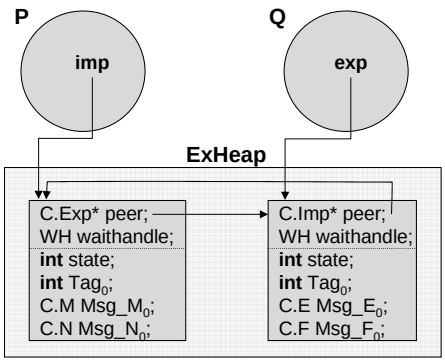

Figure 3: Channel representation in memory

4. IMPLEMENTATION

This section describes the runtime data and code representation for channels and switch receive statements.

4.1 Channel representation

Channel endpoints are represented as rep structs in the exchange heap. Each endpoint struct has a general part managed by the kernel and a contract-specific part. The kernel managed part contains a pointer to the channel peer, a special waithandle for signalling message arrival, and a boolean indicating whether this end of the channel has been closed.

A channel contract C is compiled into two rep structs C.Exp and C.Imp. Besides the kernel managed part, these structs contain a numeric field capturing the current protocol state of the endpoint, tag fields, and message argument buffers. Figure 3 shows a channel in memory.

The messages and protocol states of a channel (including implicit intermediate states in message sequences) are assigned numeric values. Then we compute maps C_QI_Exp and C_QI_Imp mapping protocol states to a queue index, thereby capturing how many messages are (maximally) waiting to be received at that protocol state. The queue index is computed by examining the protocol state machine and noting the maximal number of message receipts since the last send in the state machine.

For each queue index i, the endpoint contains a tag field Tag_i. The tag fields describe whether messages are present in the queue as follows: if a C.Exp endpoint is in protocol state X and $\mathsf{C\_QI\_Exp}(X) = i$, we can determine whether a message has arrived on it by examining field Tag_i. If the field is 0, no message is ready. Otherwise, the field value indicates the message tag that is present. Thus, if we are to send message m in state X on endpoint C.Imp, the sending code uses the queue index $\mathsf{C\_QI\_Exp}(X) = i$ for the peer endpoint, and then stores the message tag for m in the peer's Tag_i field. After storing the tag, the sender signals the waithandle on the peer. The reasoning in the other direction is symmetric.

The message data is handled similarly to the message tag. The compiler emits a struct type C.M for each message m whose fields are the message arguments. For each queue index i in an endpoint, if message m could be received in a state mapping to i, we add a message buffer field $\mathsf{Msg\_M}_i$ in the endpoint structure. Sending message m now involves also storing the message arguments in the peers $\mathsf{Msg\_M}_i$ field, prior to setting the message tag Tag_i. On multi-processors/cores, a memory barrier is needed af-

ter storing the message arguments and prior to setting the message tag.

4.2 Channel teardown

As is apparent from Figure 3, channel endpoints are partially shared, since the sender needs to access the peer endpoint to store message tags and message arguments, as well as to signal the waithandle. In fact, the cross-linked endpoint structure forming a channel is the sole mechanism for transfering information from one process to another.

In order to implement the desired failure semantics where sends can occur as long as the sending endpoint is not closed, we need to keep the channel data structure alive as long as there is at least one of the two parties still using the channel. We thus reference-count the endpoint structures. The ref-count starts at 2 for both ends, since there are exactly two references from the user processes, one to each endpoint, and one reference from each endpoint to its peer. The reference count never goes beyond two. When one side deletes an endpoint, we decrement its reference count and also that of its peer. When the reference count of an endpoint drops to 0, its memory and the associated waithandle are reclaimed. This scheme makes has the advantage that beyond the atomic decrement, no thread synchronization is needed during sends/receives or tear-down.

4.3 Block accounting

Normally terminating processes are guaranteed to free all exchange heap memory owned by the process by virtue of the static verification that accounts for every block manipulated and thus prevents such blocks from leaking. However, a system needs to be able to shutdown a process abruptly without letting the process clean up. Thus, in order to reclaim the blocks in the exchange heap that the process owned at the moment of its abrupt demise, the system must be able to find these blocks.

Our implementation uses a short header per block in the exchange heap containing its size and the owning process id. The size is used for vector operations in determining how many elements are in the vector. The owning process id is maintained by channel send operations as follows: When sending on an endpoint e, we determine the owning process pid of the peer f of e. Note that there is no race in determining the receivers pid, since the invariant discussed in Section 2.6 guarantees that an endpoint that is about to receive cannot be sent itself and thus its owning process cannot change. Once the receiver's identity is known, the sending thread changes the owning process id of all message arguments recursively. This traversal code is generated automatically by the compiler.

Additionally, the memory manager for exchange heap blocks keeps a list of all used blocks. The list is traversed by a background clean-up thread to reclaim blocks owned by non-existant processes.

4.4 Switch receive

For each **switch receive** statement, the compiler generates a static 2-dimensional integer array representing the message patterns that make up the cases. The matrix has k rows and n columns, where k is the number of **case**s in the switch, and n is the number of *distinct* endpoints used overall the cases. The matrix contains a non-zero entry in position (i, j), if the ith case contains a pattern for message m on the jth endpoint. The entry's value is the tag for message m. Each row thus indicates which messages must be received on which endpoints for that case to be enabled. Given this pattern matrix, the compiler compiles the **switch receive** statement into the following schematic code:

```
Endpoint*[] endpoints = new Endpoint*[]{ep1 ,..., epn};
int enabled_case = Endpoint.Select( pattern , endpoints );
switch (enabled_case) {
  case 0:
    // declare locals for bound message arguments
    // receive messages for case 0 on endpoints
     ...
    break;

  case 1:
    // declare locals for bound message arguments
    // receive messages for case 1 on endpoints
     ...
    break;

  ...
}
```

To make this more concrete, let's see how the following example is compiled:

```
switch receive {
  case a.M(int x) && b.Q(char[] in ExHeap vec):
    block0

  case b.R(int y):
    block1

  case a.ChannelClosed():
    block2

  case b.ChannelClosed():
    block3
}
```

The patterns mention 2 endpoints a and b. Assuming the compiler numbers them in that order, the pattern matrix for the switch is:

	a	b
case 0	M_tag	Q_tag
case 1	0	R_tag
case 2	-1	0
case 3	0	-1

The special ChannelClosed patterns are given tag -1. The code generated is shown in Listing 2. On line 1, we construct an array containing the endpoints involved in the pattern.[2] Then we call a static method Select, passing the pattern matrix and the endpoint array. This call will block until either a case is satisfied, returning the case number, or until it determines that none of the cases can be satisfied, returning -1. Note that the RecvX calls on lines 7,8, and 13 will not block, since the Select call will only return that case if the necessary messages are present at the head of the corresponding endpoints.

The implementation of Select is relatively simple. For each row of the pattern matrix, it queries each endpoint for which there is a non-null entry in the row for the status of the endpoint. The endpoint returns either the tag of the first

[2]Our implementation avoids the allocation by reusing a thread-local array after the first switch receive of a given size.

```
1  Endpoint*[] endpoints = new Endpoint*[]{a,b};
2  int enabled_case = Endpoint.Select(pattern, endpoints);
3  switch (enabled_case) {
4    case 0:
5      int x;
6      char [] in ExHeap vec;
7      a.RecvM(out x);
8      b.RecvQ(out vec);
9      block0
10
11   case 1:
12     int y;
13     b.RecvR(out y);
14     block1
15
16   case 2:
17     block2
18
19   case 3:
20     block3
21
22   default:
23     throw new Exception("Unsatisfiable switch receive");
24 }
```

Listing 2: Generated code for switch receive

message, or -1 if there is no message and the peer endpoint is closed, or 0, if there is no message and the peer endpoint is not closed. Given this information, Select can determine if a case is satisfied and whether the case can still be satisfied by the arrival of new messages.

If no case is satisfied, but some cases are still satisfiable in the future, Select waits on the waithandles of all endpoints involved in the switch. When woken due to a new message arrival or channel closure, the patterns are scanned again.[3]

Note that the implementation uses no locking or other synchronization beyond the signalling of new messages via the waithandles. This lock-free implementation is again enabled by the global invariants provided by our ownership model: only a single thread can use a given endpoint in a **switch receive** at any time. Thus, we are guaranteed that when Select determines a case is satisfiable, the state of its endpoints won't change until the thread enters the corresponding case and removes the messages at the head of the endpoints involved. If multiple threads were to be able to receive on a single endpoint, the implementation would be much more involved, since it must atomically determine which case is satisfied and dequeue the corresponding messages.

5. EXTENSIONS

In practice, we use a couple of extensions to the channel mechanisms described so far. In this section, we briefly mention them as well as other ongoing work.

5.1 Endpoint sets and maps

Threads often need to handle an apriori unknown number of endpoints. This arises most frequently in services that have numerous connections to clients. In order to handle these situations, Sing# provides an *endpoint set* abstraction. An endpoint set contains endpoints that are all of the same type and in the same prototol state. Threads can create sets, add endpoints, and use these sets in **switch receive** statements to wait for messages on any of the endpoint in a set. When an endpoint has a matching message, the endpoint is removed from the set and bound in the **case** branch.

Endpoint maps are similar to sets, but associate arbitrary data with each endpoint in the set.

[3]Only the pattern column of the endpoint that caused the wakeup is scanned in practice.

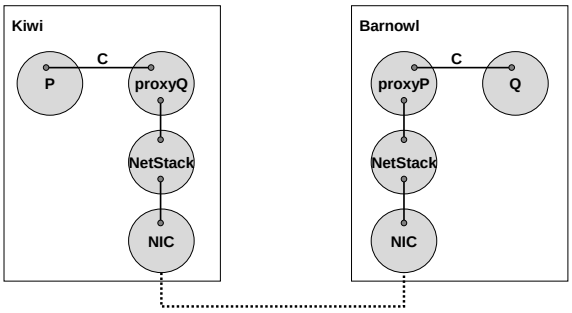

Figure 4: Cross machine channel via TCP proxy

5.2 Channel contract hierarchies

Channel contracts can form hierarchies such that a contract can extend another contract by adding more message sequences than the parent contract. In order for this to be sound, restrictions must be in place for when such extensions are allowed and how the corresponding types of endpoints can be cast to each other. A formal study of channel contract inheritance is the topic of an upcoming paper.

5.3 Process local channels

The channel implementation described does not require the two channel endpoints to be in different processes. The channels work just as well between threads of the same process. If such intra-process channels are declared as such, they can support a wider range of exchangeable types, since GC'ed objects could be safely transmitted over such channels.

5.4 Cross machine channels

The Singularity OS includes a TCP/IP stack and network interface drivers. Thus, communicating with other machines on the network is no different than on Unix or Windows. An obvious question though is whether the strongly typed Singularity channels can be extended to cross machine boundaries. Although we haven't implemented this feature, we believe it to be straight-forward as depicted in Figure 4. A process P on machine Kiwi wants to talk to a process Q on machine Barnowl using channel contract C. From the contract C we should be able to automatically produce proxy processes proxyQ and proxyP that marshall and unmarshall messages to and from the network and dynamically check the protocol of incoming network messages. The clients P and Q talk to the proxies over strongly typed channels, oblivious (up to latency) to the fact that they are communicating remotely. The semantics of our channels poses no immediate problem due to the fact that sends never block and that the channel failure is communicated only on receives. Thus, when proxies determine that the TCP connection has failed, they can just close the channel the their clients.

5.5 Running unverified code

To run on Singularity, applications must currently be expressible in verifiable MSIL. This requirement makes it im-

	Cost (CPU Cycles)		
	Singularity	Linux	Windows
Process-kernel call	78	324	445
Thread yield	401	900	763
2 thread wait-set ping pong	2,156	2,390	3,554
2 thread message ping pong	2,462	10,758	12,806

Table 1: Micro benchmark performance

practical to port existing software written in C++ or other unverifiable languages. We are currently investigating the combination of language-based safety with hardware protected address spaces in order to isolate unverified code from verified code using hardware sandboxes.

6. EXPERIENCE

The channel implementation described in this paper has been in use in Singularity for the last 8 months. The code base relies on 43 different channel contracts with roughly 500 distinct messages, and 135 different protocol states.

Passing ownership of memory blocks and endpoints over channels is used frequently. Among the 500 messages, 146 carry pointers into the exchange heap, and 42 messages carry channel endpoints themselves. Thus, the ability to configure the communication network dynamically by transferring endpoints is quite important.

The static verification of ownership has so far delivered on the promise of eliminating errors related to dangling pointer accesses and leaks of exchange heap blocks. We use two redundant verifiers checking the ownership: one running as part of compilation of the source language to MSIL [1], and a second verifier checking the MSIL. This redunancy helps shake out errors in the verification itself that would allow ownership errors to pass the compilation undetected.

One serious shortcoming of our current implementation is that we do not check exceptional program paths. This shortcoming stems from the fact that we don't have exception specifications in our code base. Without exception specifications on methods, the analysis of exceptional paths would lead to too many programs being rejected. We are actively working on specifying exceptional behaviors and integrating them with the ownership programming model and verification.

6.1 Performance

This section contains micro benchmarks comparing the performance of Singularity channel operations against other systems. All systems ran on AMD Athlon 64 3000+ (1.8 GHz) on an NVIDIA nForce4 Ultra chipset and 1GB RAM. We used Red Hat Fedora Core 4 (kernel version 2.6.11-1.1369_FC4), and Windows XP (SP2). Singularity ran with a concurrent mark-sweep collector in the kernel, a non-concurrent mark-sweep collector in processes (including drivers), and a minimal round-robin scheduler.

Table 1 reports the cost of simple operations in Singularity and two other systems. On the Linux system, the process-kernel call was getpid(), on Windows, it was GetProcessId(), and on Singularity, it was ProcessService.GetCyclesPerSecond(). All these calls operate on a readily available data structure in the respective kernels. The Linux thread test ran on user-space scheduled pthreads. Kernel scheduled threads performed significantly worse. The "wait-set ping pong" test measured the cost of switching between two threads in the same process through a synchronization object. The "2 message ping pong" measured the cost of sending a 1-byte message from one process to another and then back to the original process. On Linux, we used sockets, on Windows, we used a named pipe, and on Singularity, we used a channel.

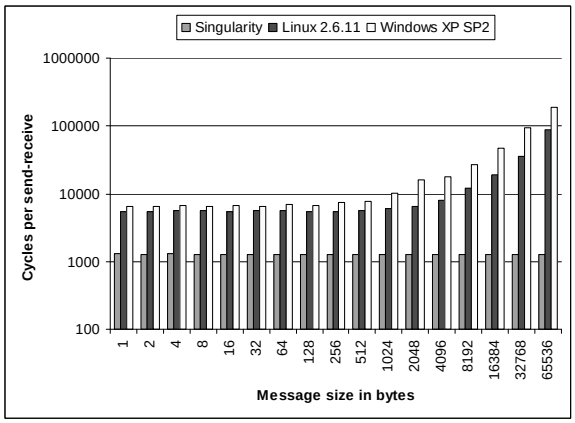

Figure 5: Send-receive cost vs. buffer size

As the numbers show, Singularity is quite competitive with systems that have been tuned for years. Particularly encouraging are the message ping pong numbers on Singularity as they show that the cost of using a channel between distinct processes is only about 15% slower than the cost of the wait-set ping pong on two threads of the same process. Channels on Singularity outperform the mechanisms available on other systems by factors of 4 to 5. These improvements don't even factor in the ability to pass pointers rather than copies of memory blocks.

Figure 5 shows the cost in cycles for sending and receiving a block of memory ranging from 1 byte to 64K. As expected, the cost on Singularity is constant around 1200 cycles. On the other systems, the cost starts to increase around buffer sizes of 1K. These measurements do not show the performance improvements possible if buffers of page size or larger (4K on x86) are remapped rather than copied. Such an optimization would of course require another IPC interface rather than sockets and named pipes.

7. RELATED WORK

There is a long history in the systems community of drawing on programming languages and safety guarantees to make systems more reliable, maintainable, and/or more performant. For example, SOLO was written in concurrent Pascal instead of assembly [19]. Early work around the Mesa language focused on memory safety through garbage collection, type safety, and modularity as well as features such as coroutines and recursion that would be burdensome to implement without runtime and language support [18, 30, 35]. The SPIN micro-kernel used Modula-3 to grant application specific extensions to run directly in the kernel to optimize performance [7]. These systems had very weak iso-

lation guarantees between processes, usually sharing a single heap managed by a single VM. In contrast, our processes are strongly isolated and can be killed and reclaimed individually.

Many systems and languages support channel-based interprocess communication. An early system using messages to communicate between tasks is DEMOS [6] on the CRAY-1. Channels (links in DEMOS) were unidirectional and could be passed over other channels. Hardware protection and copying was used to prevent inappropriate access and sharing rather than compile-time checking.

Channels in programming languages are typically unidirectional streams of uniform type, for example in the language OCCAM [26]. Other examples of languages with stream based communication types are CML [31], Newsqueak [29], and Limbo [13]. However, in contrast to the present work, none of the above cited work attempts to statically verify that exchanged data blocks are unshared without the need to copy memory.

Recent work on writing an operating system in OCCAM [4] has extended the language to support *mobile* data, which corresponds to what we call tracked data in this paper [5]. The main difference with the present work is that we have integrated mobile semantics into a main-stream OO-language without compromising safety or aliasing. Furthermore, the OCCAM channels are synchronous, uni-directional, and uniformly typed, whereas the channels described here are asynchronous, bi-directional and governed by channel protocols. Their reported message transfer times are extremely efficient ($\sim$ 80 cycles). The factor of 20 in increased performance over our approach is achieved by exploiting further invariants their system has over ours. The scheduling in their kernel is purely cooperative and assumes a single processor, whereas we assume a pre-emptive scheduler in a multicore/multi-cpu environment. Furthermore, an OCCAM thread waits on a single channel, whereas our threads can be blocked on any number of wait conditions, which causes more accounting on waking a thread. Our implementation also has to cooperate with two garbage collectors, the process' own, and the kernel's. On process-kernel boundaries, we push special stack markers that allow the respective GC's to avoid scanning stack frames they don't own. Finally, their implementation uses the channel as the synchronization event itself, whereas we currently use primitive events to implement the channel synchronization. We believe that we can further improve our channel performance by careful optimizations for the common case.

On the systems side, much research has gone into techniques to avoid copying of memory in the IO-system. For example, IO-Lite [28], uses read-only buffers accessed through mutable buffer aggregates to allow wide sharing of the buffers among different parts of the system. Buffer aggregates allow splicing together various regions of buffers. The IO-Lite approach uses VM pages and page protection to enforce read-only access. The main drawback of IO-Lite appears to be the need to know, during buffer allocation, which other processes will share the page in order to avoid remapping of pages and fragmentation. In comparison, the statically enforced mutual exclusion of access to buffers described in this paper does not suffer from page granularity restrictions. Similarly, the IPC mechanism in L4 micro kernel [20] is based on hardware protection and is only zero-copy if entire pages are transferred by remapping. Otherwise, the cost is proportional to the size of transferred buffers. Compared with measurements for L4 on the Alpha, our message round trip times are faster than L4's for all transfer sizes above 64 bytes (which L4 passes in registers).

The lack of read-only sharing in our system is one of its main drawbacks at the moment. We plan to address this issue in future work by incorporating ideas from IO-Lite while retaining a purely software based protection approach.

Ennals et al. use linear types to pass ownership of buffers in a language for packet processing compiled to network processors [14], thereby proving isolation between the sender and receiver statically as we do. Our resource tracking is rather similar to linear types and can be formalized using separation logic [32].

Supporting safe, explicit memory management in programming languages has been studied in Vault [12] and Cyclone [25]. The verification technique employed in Sing# builds on this earlier work. Type systems based on ownership hierarchies [9] or lexically-scoped regions [36] are not strong enough to prove the necessary non-aliasing properties at arbitrary ownership transfer points, e.g., when memory must be freed or transferred to other processes.

Real-time Java [8] is a version of Java for real-time constrained systems. It provides programers with memory areas that have a stack based lifetime. The explicit memory management in RTSJ is runtime verified, not statically checked as in our work, i.e., storing a pointer to an object in a memory area might cause a runtime error as opposed to a compile-time error if the target object lives beyond the memory area's lifetime. Scoped types [37] proposes a set of rules that statically guarantee the absence of such errors. Still, life-times of memory areas have lexical scopes whereas in our work, a block can get allocated by a process, get transferred to another process, and live beyond the lifetime of the allocating process. Furthermore, our memory model guarantees that every block is accessed by at most one thread at a time, thus providing statically enforced mutual exclusion. RTSJ to the best of our knowledge does not provide such features.

Guava [3] is a Java dialect providing statically guaranteed mutual exclusion through true monitors. Guava's stronger exclusion guarantees over Java enable compilers to optmize code more aggressively. Guava achieves mutual exclusion through hierarchical ownership domains [9] called regions. As discussed above, these techniques do not allow true ownership transfer as described in this paper. As a result, Guava relies on deep copying of data structures. It also does not address the cross process transmission of data, nor does it have any notion of message passing. As such, the ideas in Guava are very much orthogonal to those presented here and one could imagine combining these system, so that e.g., tracked data could be transferred between ownership domains without the need to copy it.

The Erlang language has been successfully used in the telecommunication industry [2]. It is based on a purely functional programming model in which all data is read-only and sharing cannot cause unwanted side effects. Depending on the runtime system employed [33], message data is either copied or processes share pointers into shared heaps. The latter scenario makes garbage collection a global affair. Erlang runtime systems can benefit from the tracked data approach of Sing# by keeping garbage collection process-local without the need to copy messages.

Compared to the channels described in this paper, Erlang uses port based communication and messages can be received out of order. The language does not statically guard against "message not understood errors", but a global static analysis ensuring that processes handle all messages sent to them is described in [11].

Specifying and statically checking the interaction among communicating threads via protocols is the subject of numerous studies [24, 21, 17, 27, 16]. From a theoretical view point, the work on session types for inter-process communication by Gay et al. [17] is closely related to ours. The session types have the same expressive power as our channel contracts and their type system also keeps track of aliases to determine proper ownership management of endpoints.

8. CONCLUSION

To our knowledge, this work is the first to integrate a full-fledged garbage-collected language with statically verified channel contracts and safe zero-copy passing of memory blocks. The main novelty of the work is the enabling of simple and efficient implementation techniques for channel communication due to the system-wide invariants guaranteed by the static verification. Our channel contracts are expressive and support passing channel endpoints and other memory blocks as message arguments with zero-copy overhead.

The message-based programming model is well supported by the language through a **switch receive** statement which allows blocking on complicated message patterns. The static checking prevents ownership errors and helps programmers handle all possible message combinations, thereby avoiding "message not-understood errors". The runtime semantics restricts channel failure to be observed on receives only, thus eliminating handling of error conditions at send operations where it is inconvenient.

Our programming experience with the language and channel implementation has been positive. Programmers on the team, initially skeptical of the value of contracts, now find them useful in preventing or detecting mistakes. The channel contracts provide a clean separation of concerns between interacting components and help in understanding the system architecture at a higher level.

8.1 Future Work

We already mentioned the need to deal with exceptional control flow in the static verification. We are also developing extensions to the channel contracts that enable static prevention of deadlock in the communication of processes. Finally, we are investigating how to build abstractions over channel objects and communication while retaining their asynchronous nature and the static checking of the protocol.

Acknowledgments

The present work would not have been possible without the effort of a large number of people. We would like to thank Paul Barham, Nick Murphy, and Ted Wobber for feedback and putting up with a research compiler and experimental static analysis. Thanks to Qunyan Mangus, Mark Plesko, Bjarne Steensgaard, David Tarditi, and Brian Zill for implementing support for the exchange heap in the Kernel and the Bartok native code compiler. Special thanks goes to Herman Venter for providing the inital compiler infrastructure and support.

REFERENCES

[1] Partition III: CIL Instruction Set. ECMA Standard 335 http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[2] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[3] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A Dialect of Java without Data Races. In *Proceedings of 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 382–400, October 2000.

[4] F.R.M. Barnes, C.L. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. In *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 269–288, September 2003. ISBN: 1-58603-381-6.

[5] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In *Communicating Process Architectures 2001*, number 59 in Concurrent Systems Engineering Series, pages 243–264. IOS Press, Amsterdam, The Netherlands, September 2001.

[6] Forest Baskett, John H. Howard, and John T. Montague. Taks communication in DEMOS. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 23–31, 1977.

[7] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gün Sirer. SPIN: An Extensible Microkernel for Application-specific Operating System Services. In *Proceedings of the 6th ACM SIGOPS European Workshop*, pages 74–77, 1994.

[8] Greg Bollella, James Gosling, Ben Brosgol, Peter Dribble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[9] David G. Clarke, James Noble, and John Potter. Simple Ownership Types for Object Containment. In *15th European Conference on Object-Oriented Programming (ECOOP 2001)*, volume 2072. Lecture Notes in Computer Science, 2001.

[10] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, January 1999.

[11] Fabien Dagnat and Marc Pantel. Static analysis of communications for Erlang. In *Proceedings of 8th International Erlang/OTP User Conference*, 2002.

[12] Robert DeLine and Manuel Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*, pages 59–69, 2001.

[13] Sean Dorward, Rob Pike, and Phil Winterbottom. Programming in Limbo. In *Proceedings of COMPCON*. IEEE, 1997.

[14] Robert Ennals, Richard Sharp, and Alan Mycroft. Linear Types for Packet Processing. In *European Symposium on Programming*, volume 2986 of *LNCS*, pages 204–218, Jan 2004.

[15] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, June 2002.

[16] Cédric Fournet, Tony Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-Free Conformance. In *LNCS*, volume 3114, pages 242–254, Jan 2004.

[17] Simon Gay, Vasco Vasconcelos, and António Ravara. Session Types for Inter-Process Communication. Technical Report TR-2003-133, Department of Computer Science, University of Glasgow, 2003.

[18] Charles M. Geschke, Jr. James H. Morris, and Edwin H. Satterthwaite. Early Experience with Mesa. *Commununications of the ACM*, 20(8):540–553, 1977.

[19] Per Brinch Hansen. The SOLO Operating System: A Concurrent Pascal Program. *Software–Practice & Experience*, 6(2):324–336, 1976.

[20] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The Performance of μ-Kernel-Based Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, October 1997.

[21] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *European Symposium on Programming, ESOP'98*, volume 1381 of *LNSC*, Jan 1998.

[22] Galen Hunt, James Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[23] Galen C. Hunt, James R. Larus, David Tarditi, and Ted Wobber. Broad New OS Research: Challenges and Opportunities. In *Proceedings of Tenth Workshop on Hot Topics in Operating Systems*. USENIX, June 2005.

[24] A. Igarashi and N. Kobayashi. A generic type system for the Pi-calculus. In *POPL 01: Principles of Programming Languages*, pages 128–141, 2001.

[25] Trevor Jim, Gregory Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX 2002 Annual Conference*, pages 275–288, 2002.

[26] Geraint Jones and Michael Goldsmith. *Programming in occam 2*. Web edition, 2001. http://web.comlab.ox.ac.uk/oucl/work/geraint.jones/-publications/book/Pio2/.

[27] Matthias Neubauer and Peter Thiemann. Session types for asynchronous communication. citeseer.ist.psu.edu/636671.html.

[28] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, February 2000.

[29] Robert Pike. The Implementation of Newsqueak. *Software–Practice & Experience*, 20(7):649–659, 1990.

[30] David D. Redell, Yogen K Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An Operating System for a Personal Computer (summary). *Communications of the ACM*, 23(2):81–92, 1980.

[31] John H. Reppy. CML: A Higher-Order Concurrent Language. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.

[32] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, 2002.

[33] Konstantinos Sagonas and Jesper Wilhelmsson. Message Analysis-Guided Allocation and Low-Pause Incremental Garbage Collection in a Concurrent Language. In *Proceedings of the 4th international symposium on Memory management (ISMM'04)*, pages 1–12, 2004.

[34] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias Types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, volume 1782 of *LNCS*, pages 366–381, 2000.

[35] Daniel C. Swinehart, Pollef T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, 1986.

[36] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the 21st Annual ACM SSymposium on Principles of Programming Languages*, pages 188–201, January 1994.

[37] Tian Zhao, James Noble, and Jan Vitek. Scoped Types for Real-time Java. In *25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 241–251, 2004.