Tasks and Revocation for Java
(or, Hey! You got your Operating System in my Language!)

November 13, 1999 Draft

Chris Hawblitzel and Thorsten von Eicken
Cornell University

Abstract: Current Java systems do not adequately support revocation, domain termination, thread control, global optimization in the presence of dynamic loading, and resource control. To solve these problems, this paper introduces a *task* model for Java based on a type system which distinguishes between task-local pointers and *remote pointers* shared between tasks. These extensions to Java are implemented by a system called Luna, based on the Gauvac and Marmot compilers, extended with special optimizations to support both fast inter-task communication and dynamic access control.

Introduction

Traditional operating systems such as Unix, Windows NT, and more recent microkernels use virtual memory to protect different programs from one another and to divide a computer's finite resources among the programs. In these systems, the *process* or *task* is the central unit of protection and resource control, and cleanly encapsulates each program's memory usage and processor usage. In recent years, software based protection has become a viable alternative to virtual memory based protection for many types of extensible applications. For instance, the Java programming language is now used to extend browsers with applets, servers with servlets [Java], active network routers with new protocols [WGT98], databases with customizations [GMS+98], and agent systems with mobile agents [Gen]. In these systems, the language subsumes the role of a traditional operating system, and is responsible for protecting programs from one another. As safe language protection becomes more widespread, the demands on the language's protection features increase: a denial of service attack on a browser may be merely annoying, but such an attack on an active network or server could cause serious damage. Even in the absence of malicious attacks, these systems must be robust enough to deal with buggy code or code that overwhelms the system resources.

Unfortunately, current safe languages do not have the support for resource control and clean program termination that is present in traditional operating systems. For example, Sun recently deprecated Java's `Thread.stop` and `ThreadGroup.stop` API, leaving no officially sanctioned way to stop an applet [Javb]. These `stop` methods were based on the idea that terminating an applet only requires stopping the applet's threads. While this approach to termination seems reasonable on first glance, closer inspection reveals several difficult problems:

- The wrong code gets interrupted: in Java, a call from an applet to the system is just a function call, so that the system code runs in the applet's thread. The applet can kill or suspend the thread while the system code is running, possibly leaving the system in an inconsistent or deadlocked state.
- Malicious code eludes termination: terminating a malicious program's threads does not terminate the program's code, which may still exist in overridden methods of objects that the malicious program created and passed to other protection domains. If one of these other domains invokes the rogue object methods, the malicious code is revived. Java mitigates this problem by making key datatypes *final* (e.g. `String`, array types) or *primitive* (e.g. `int`, `float`) so that they cannot contain overridden methods, but a purer object-oriented language would have more difficulties.
- Damaged objects violate abstract datatype integrity: under Java's synchronization mechanisms, a thread may enter an object's method, acquire a lock, start an atomic operation, and then get terminated in the middle of the operation, releasing the lock and leaving the object in a "damaged" state that violates its intended abstraction.
- Resources aren't reclaimed: when a traditional operating system shuts down a process, both the process's threads and memory are shut down. Java's termination is weaker, stopping threads but not necessarily reclaiming other resources.

Solving these problems often means compromising some of the language-based protection's promise. For example, DrScheme/MrEd [FFK+99] restricts communication to a hierarchical parent-to-child pattern, providing no support for peer-to-peer sharing (e.g. applet-to-applet or agent-to-agent sharing), in order to allow a parent to cleanly terminate a

child. DrScheme's approach also leaves much of the responsibility for clean termination to the programmer. For example, the parent is expected to explicitly relinquish pointers to objects in a child domain in order to avoid damaged objects and ensure memory reclamation; MrEd's "custodians" do not do this automatically.

At the opposite extreme, the J-Kernel [HCC+98] allows communication between arbitrary protection domains (even mutually suspicious domains), but only allows the domains to share special "capability" objects through which they can perform remote method invocations. The system ensures that these capability objects are revocable, so that when a protection domain is shut down, the capabilities that it exported are revoked and the domain's memory is reclaimed. Unfortunately, the J-Kernel's heavyweight approach abandons many of the advantages of lightweight protected sharing.

This paper attempts to bridge the gap between these two approaches. It describes an extension to the Java language called Luna, which implements multiple *tasks*, each with its own objects, threads, and code. When a task is terminated, its objects, threads, and code are deallocated together, ensuring that damaged objects and unwanted code are cleanly and completely disposed of. Inter-task communication is organized around the concept of a revocable *remote pointer*, which is built into Luna's type system. Pointers from one task to an object in another task have a special type indicated by a tilde (e.g. "`String~`", "`Hashtable~`"). These remote pointers are dynamically revocable, so that when a task is shut down, pointers into the task are revoked, and the task's objects, code and threads are safely deallocated. This allows fine-grained sharing between arbitrary tasks to coexist with clean termination and resource management.

Luna is implemented as a source-level extension to the Gauvac source-to-bytecode compiler, which produces special Luna bytecode that runs on an extension to the Marmot, a sophisticated globally optimizing Java virtual machine [FKR+99]. The extensions to the source language are small, while the extensions to Marmot's run-time system are more involved. Note that tasks and remote pointers would apply to languages besides Java; an earlier paper [HvE99] describes the application of these ideas both in a C-like procedural language with modules and as a formal extension of the typed lambda calculus for which we proved soundness.

The structure of the paper is as follows. First, the paper discusses drawbacks of language-based protection in its current state, and how the task model can help overcome these drawbacks. Afterwards, the paper describes remote pointer types, and then describes how tasks and remote pointers are implemented and optimized, focusing on what we found to be the hardest problem in Luna: optimizing field and array accesses to revocable remote pointers. Finally, the paper briefly describes an extensible web server written in Luna, and compares Luna to related work.

Limitations of language-based protection; how the task model can help

Safe languages hold many potential advantages over virtual memory based systems: fine-grained sharing, no expensive address space switches, capability-based access control built with unforgeable pointers, abstract datatype enforcement, and code portability. Unfortunately, this list of language protection advantages is matched by an equally long list of drawbacks. The performance of safe languages tends to lag behind lower level languages like C, typical language-based systems support only one language, and relying on a language for protection requires trusting that the language's compiler (the just-in-time compiler in Java's case) and run-time system are written correctly. Other research has made great strides in attacking these problems ([BSP+95], [HLP98], [MWC+98], [NL98], [Sha97]). This paper focuses on a different set of problems, arising from the lack of OS features and structures in current language-based systems. In addition to the termination problem discussed above, current language-based systems do not adequately support:

Resource control

When objects are shared at a fine granularity, which programs should be charged for the objects and how can memory usage be tracked? One solution is to charge for all the objects reachable from a program's roots. While this may work in traditional operating systems with fixed sized shared memory buffers of "raw data", it is dangerous for abstract data types and pointer based data structures. Program A can give program B a single object with some private field pointing to all of A's private data, and program B gets charged for all of A's data. Fine-grained sharing looks less pleasant when any shared object can act as a resource trojan horse. Luna offers a simple alternative: a task pays only for those objects that it explicitly allocates with the Java `new` operator; it is not charged for objects allocated by other tasks.

Revocation

In ordinary Java, pointers can serve as capabilities, but once a program is given a pointer to an object, that pointer cannot be revoked. Why would anyone want to revoke a capability? Revocation assists termination: Luna revokes all remote pointers into a task when the task is terminated. It also helps to implement the principle of least privilege, since it is better to give someone access to something for only the necessary duration and then revoke the access than to give them access forever. Revocation accommodates changing preferences over time. An agent once considered trustworthy may abuse the trust and necessitate revoking its privileges. At a lower level, revocation is a good way to give someone temporary, fast access to a resource, such as idle network buffer space or a rectangle in video memory. In addition to these device-specific examples, revocation is also used in general purpose operating system mechanisms: for instance, FBufs [DP93] dynamically revoke write access to buffers when data is transferred between protection domains.

Revocation has been a historical problem for capability systems [WLH81], but it is particularly difficult at the language level. While adding a level of indirection to every capability may be a way to implement revocation in an operating system with coarse-grained capabilities [Red74], adding a level of indirection to every Java object is undesirable. Luna's static distinction between local and remote pointers ensures that only pointers shared between tasks incur the overhead of supporting revocation (as described below, the high cost of locks on modern processors makes this overhead a serious concern).

Controlling threads

If a call from one program to another is just a function call, as it is in standard Java, the caller and callee programs share the same thread. Who is allowed to terminate or suspend the shared thread? In current Java implementations, an applet can call the browser and then kill the thread while the browser is in the middle of a sensitive operation. Luna, on the other hand, switches threads during a method invocation on a remote pointer, so that the caller and callee each have their own threads, which is similar to IPC and RPC mechanisms in traditional operating systems. Luna's remote pointer types statically signal which method invocations are on local objects (needing no thread switch) and which are *cross-task* method invocations on remote objects.

Reasoning about the system structure

Analyzing the security of a system requires analyzing the communication patterns between programs. Limited, well-defined communication channels make this easy; unconstrained fine-grained sharing makes a mess. Luna's remote pointer types statically mark the boundaries between tasks, and thus form a static blueprint of the system's cross-task communication patterns.

Global optimization

While dynamic loading is a useful tool for system extensibility, many optimization techniques rely on global information (such as "this method is never overridden and may therefore be inlined") that may be invalidated as code is dynamically loaded. Luna solves this problem by making the task the fundamental unit of loading, instead of loading classes individually. This enables task-wide global optimizations of operations on local pointers, so that Luna is able to exploit Marmot's inlining and static method binding. The only global optimization that caused difficulties was Marmot's object stack allocation, which adds extra method table entries to some classes (Luna must ensure that method tables and field layouts are consistent across tasks, to support operations on remote pointers).

Remote pointers

In order to preserve the advantages of safe language protection (fine-grained sharing, low cross-task call overheads, simple capability-based access control, and enforcement of abstract data types), remote pointers support the same operations as local pointers: field/array element access, method invocation, synchronization, equality testing, casting, and instanceof testing, although most of these operations have different semantics and performance for remote pointers. Figure 1 summarizes Luna's type system; standard Java types are written in plain text and the new types are written in bold. For convenience, we will refer to the objects pointed to by local and remote pointers as "local objects" and "remote objects" respectively.

```
Type = PrimitiveType | ReferenceType | ReferenceType~
PrimitiveType = boolean | byte | short | int | long | char | float | double
ReferenceType = ClassType | InterfaceType | Type[]
```

Figure 1: Luna's type system

The key difference between remote pointers and local pointers is revocation. Luna's task model requires that remote pointers into a task be revoked when the task is killed, but the previous section argues that revocation is also useful at a finer granularity. To realize these uses, Luna gives the programmer a special handle with which to control access to remote pointers. This handle is a Java object called a *permit*. A permit is allocated in an unrevoked state, and can later be revoked:

```
public class Permit {
    public Permit(); // constructor to allocate an unrevoked permit
    public void revoke(); // method to revoke the permit
    ...
}
```

A remote pointer is implemented as a two word value that consists of a local pointer paired with a permit. The @ operator converts a local pointer into a remote pointer:

```
Permit p = new Permit();
String s = "hello";
String~ sr = s @ p;
```

Once a task has used the @ operator to create a remote pointer, it can pass the remote pointer to other tasks, which can use them until the remote pointer's permit is revoked. Operations on remote pointers perform a run-time access check: the expression "`sr.length()`" will evaluate `sr`'s length if `p` is unrevoked, and raise an exception if `p` is revoked. Permits can selectively revoke access to data: if permit `p1` is revoked while `p2` is not, then `s` is accessible through the remote pointer `(s @ p2)` but inaccessible through the remote pointer `(s @ p1)`. Note there is no way to decompose a remote pointer into its two parts: the local pointers to `s` and `p` can't be extracted from `sr`; this prevents other tasks from gaining direct access to them. In other words, a task's access to another task's data is always mediated by a permit ("complete mediation" is one cornerstone of a secure system [SS75][WBD+97]).

In order to preserve the invariant that only remote pointers cross task boundaries, local pointers cannot be written to remote object fields or passed as arguments to remote method invocations, while primitive types and remote pointers can. Reads from remote objects are more flexible: a local pointer can be read from a remote object or returned from a remote method invocation, but the local pointer is automatically *promoted* to a remote pointer. This means that in the function below, the expression "`list.next`" has type `List~`, not type `List`, because from the perspective of the `second` function, a local field of a remote pointer is still remote. At run-time, "`list.next`" evaluates to a remote pointer containing the same permit as the remote pointer `list` contained. This allows a single permit to control access to the entire list, not just the first element.

```
class List {
    int i;
    List next;
}
List~ second(List~ list) {
    return list.next;
}
```

Remote pointers cannot be used interchangeably with local pointers. For instance, a hash table object expecting keys of type "`Object`" cannot be passed a remote pointer (which has type "`Object~`"). This forces the programmer to either design a new type of hash table which can accommodate remote keys (and can deal with their revocation robustly), or, more commonly, to make a local copy of a remote object's data and use that as the hash table key. Similarly, remote pointers, like Java's primitive types, must be boxed by the programmer to be placed in Java's standard container classes,

such as `Vector`, that store `Objects` (this also serves a pragmatic implementation purpose, because remote pointers are a different size than local pointers, and are treated differently by the garbage collector).

Implementation

This section describes the implementation of remote pointer operations in more detail. Operations on remote pointers require revocation checks, and the compiler and run-time system cooperate to implement these checks. For field and array accesses to remote pointers, the compiler emits code which enters a critical section, checks a flag in the remote pointer's permit, performs the access (or raises an exception, if the permit is revoked), and exits the critical section.

Method invocations are more intricate. A method invocation on a remote pointer calls another task's code, and therefore must execute in one of the other task's threads. This thread switch allows the caller thread to be killed without abruptly terminating the callee thread. However, Luna is implemented over Win32 kernel threads, and switching kernel threads is an expensive operation. Therefore, cross-task calls only switch kernel threads lazily, when a call must be interrupted. In the normal case, a cross-task call only switches stacks, which can be done without involving Win32. To see how this works, consider a method in task A which calls a method in task B, which in turn calls a method in task C. This sequence executes in a single kernel thread, but involves 3 separate stacks. If task B is now terminated, B's stack is deallocated, C's method continues to run in the original kernel thread, and a new kernel thread is allocated which resumes A's method (and raises an exception in A's method to indicate that the call to B aborted abnormally). This model is similar to RPC models described for the Mach[FL94] and Spring[HK93] microkernels.

In more detail, then, a cross-task method invocation enters a critical section, checks the remote pointer's permit, grabs a stack from the target task's free stack pool, exits the critical section, switches the stack pointer, pushes the arguments, and makes the call. A return from the invocation enters a critical section, returns the stack to the free pool, and exits the critical section.

At this point, the reader would be correct to worry about the cost of entering and exiting a critical section. On a 266MHz P6 processor, a lock followed by an unlock, both written in hand-optimized assembly using an atomic compare and exchange instruction, takes 80 cycles when measured in a tight loop. The bottleneck is access to the bus: on a multiprocessor, an atomic operation must "lock" the bus while it executes. On a uniprocessor, the bus lock may be safely omitted, which cuts the cost of a lock/unlock sequence to 20 cycles. Because the choice of uniprocessor vs. multiprocessor has such a large impact on the cost of remote pointer accesses, we report both numbers in our benchmarks. User-level thread scheduling allows further reductions of lock costs [SCM99], but we have not implemented a user-level threads package for Luna. One last optimization is possible on a uniprocessor: remote pointer field reads (but not writes) can be implemented by first performing the read, and then checking the permit, without ever entering a critical section. Unfortunately, relaxed cache consistency prohibits this optimization on multiprocessors.

Figure 2 shows the performance of local and remote field accesses, and local and remote method invocations to a function with an empty body. Both the local and remote method invocations perform a simple method table dispatch. All measurements indicate the number of cycles taken on a 266MHz P6 processor with 64MB of RAM and 16K/512K L1/L2 data cache, measured in a tight loop (note: an empty loop takes 2 cycles per iteration; this was not subtracted from the numbers below).

	local	remote (uniprocessor)	remote (multiprocessor)
field read	3	5	94
field write	3	34	94
method invocation	10	111	228

Figure 2: Remote pointer performance, without caching optimizations

The cross-task invocations, while slower than local invocations, are nevertheless faster than round-trip IPC on the fastest x86 uniprocessor microkernels [LES+97], and are orders of magnitude faster than Win32 LRPC calls. Unfortunately, the field accesses that require locks are slow to the point of being useless: if shared data were always so expensive to access, it would make more sense to copy data than to share it. Luckily, standard *caching* and *invalidation* techniques apply to remote pointer accesses, because accesses to the data are more frequent than revocation of the data. This is analogous to virtual memory, which stores access control bits for each page in a page table. The cost of virtual memory's access control is acceptable because a translation lookaside buffer (TLB) caches

the page table information, and is invalidated whenever the page tables are modified (which is expected to occur infrequently).

Caching permit information

This section describes how Luna imitates a hardware TLB to reduce the cost of repeated access checks of the same permit. However, Luna's approach differs from a hardware TLB in that a TLB operates entirely on dynamic information, while Luna takes advantage of static information to detect permit reuse. As a simple example, consider the following loop, which zeroes the elements of a remote list (using the list class defined in the previous section).

```
void zero(List~ list) {
    while(list != null) {
        list.i = 0;
        list = list.next;
    }
}
```

As Luna compiles this function from a typed high level representation to a typed low level representation, it adds type information to indicate repeated uses of the same permit. This information is computed using a straightforward intraprocedural dataflow analysis. In the `zero` loop, Luna's analysis sees that `list.next` has the same permit as `list`, and therefore every field access to `list` uses the same permit as the loop runs. The typed representation generated by the analysis is similar to the type system formalized in [HvE99]: the function is quantified over a *permit variable* ρ, and the two word remote pointer values are split into separate one word pointer and permit values, of type `List{ρ}` and `permit{ρ}`, respectively:

```
∀ρ.void zero(List{ρ} list0, permit{ρ} p0) {
    List{ρ} list1 = list0;
    while(list1 != null) {
        list1.i = 0;
        list1 = list1.next;
    }
}
```

This representation allows the one word pointer and permit values to be stored in unrelated stack slots and registers, and for multiple values of type `List{ρ}` to be controlled by a single permit value. Furthermore, except for the revocation check, the list traversal can treat the list traversal like a traversal of a local list, so the "`list1 = list1.next`" statement need not create a new two word remote pointer each time it executes.

From the intermediate representation, it is clear that ρ is checked repeatedly in the inner loop. Based on this, Luna inserts code outside the loop to cache this permit's revocation flag in a register or stack slot, so that an access check is done with a simple test of the cached value, with no locking. At run-time, this caching code adds the current stack frame to a doubly linked list held by the permit, and removes the stack frame from the permit's list after the loop is finished. The cost to add and then remove a frame from a permit's list is about 60 cycles on a uniprocessor and 175 cycles on a multiprocessor, which means that the optimization pays off after about 2 or 3 accesses to the shared data. If the permit is revoked, it invalidates the cached information in each stack frame in its list by suspending each affected frame's thread, advancing the thread to a safe point (which Luna places at every call site and backwards branch), and then using the GC information at the safe point to determine which registers and stack slots must be modified to invalidate the cached access control information.

Luna optimizes further under the following conditions: (i) the loop contains no call instructions, exception handlers, or other loops, (ii) the loop uses only one permit, (iii) all paths inside the loop from the loop header block back to itself check the permit for revocation. If these are satisfied (the `zero` loop, for example, satisfies them), Luna places safe points before the loop's revocation checks rather than at the loop's backwards branches, and then omits the code for the revocation checks entirely. If the permit is revoked, and the thread is advanced to a safe point inside the loop, then the thread is left at a point where a revocation check would have appeared, and a revocation exception is raised asynchronously in the thread at this point, so that it appears to the user that there was actually a check in the code there.

Luna advances Java code to a safe point by setting breakpoints rather than by polling, so this optimization reduces the per-iteration cost of the revocation checks to zero.

These optimizations preserve the original semantics of the Luna program: they only throw exceptions at points where the original Luna program could have thrown an exception, and precisely reflect the state of the program when the revocation exception is thrown.

Figure 3 shows the assembly language code generated for the body of the `zero` loop, as well as the code generated for an equivalent loop over a local list. Except for a difference in the instruction used to test for null, they are identical, and both execute in 3 cycles per iteration. Nevertheless, while the inner loops are identical, the remote traversal must manipulate the permit's linked list before and after the inner loop (in a critical section, of course), and the cost of this is significant: figure 4 shows the cost of repeatedly zeroing the elements of the same list, for lists of size 10 and 100. Luna's static analysis makes the speed of these loops reasonable, but not at good as the local case.

Even if Luna extended the static analysis to encompass more than inner loops, no amount of static analysis can eliminate all the run-time checking, and figures 2 and 4 show that penalty for each operation that acquires a lock is steep. This suggests, as future work, that dynamic reuse detection (such as keeping a hash table of recently used permits in each thread) may also be needed to avoid lock acquisitions.

local list traversal

```
loop:
  /* list.i = 0 */
  mov dword ptr [eax+8],0
  /* list = list.next */
  mov eax,dword ptr [eax+12]
  /* if(list == 0) goto done */
  test eax,eax
  je done
  jmp loop
...
done:
```

remote list traversal

```
loop:
  /* list.i = 0 */
  mov dword ptr [eax+8],0
  /* list = list.next */
  mov eax,dword ptr [eax+12]
  /* if(list == 0) goto done */
  cmp eax,0
  je done
  jmp loop
...
done:
```

Figure 3: inner loop assembly code for local and remote list traversals

	local	remote (uniprocessor)	remote (multiprocessor)
10 element list	46	105	221
100 element list	316	375	493

Figure 4: speed of local and remote list traversals

Garbage collection

GC information plays a part in caching/invalidation, and the garbage collector (an extension of Marmot's copying collector) treats remote pointers specially in other ways as well. When the collector traverses a remote pointer, it checks to see whether its permit was revoked. If the permit is unrevoked, then the remote pointer is treated as a strong pointer, while if the permit is null or revoked, the remote pointer is treated as a weak pointer, since the revocation makes the object semantically unreachable through the remote pointer. This allows a task's objects to be garbage collected even if there are outstanding revoked remote pointers to the objects. In particular, when a task is terminated, all of the task's permits are automatically revoked, causing all the task's objects to become collectable, so that a task's resources are reclaimed when the task is terminated.

Extensible web server

We ported a small web server, originally developed for the J-Kernel, to Luna. The server implements Sun's servlet API [Java]. To preserve this API, the server includes wrapper classes which lazily copy remote object data into local objects when the servlet requests the data, so that a servlet need not concern itself with remote pointers directly. Figure 5 shows the time taken to dispatch a request to a servlet (which lives in a different task from the server) and process the

response, for a servlet which returns a fixed sized message, not counting the time to transfer the data to or from the underlying sockets, and not counting the cost of spawning a thread to handle the request. The figure shows Luna's performance with and without the caching optimizations, and compares this to the performance of a modified server where both the server and servlet reside in the same task, so that no remote pointers are used. The measurements show that Luna's caching optimizations pay off for large messages, especially for a multiprocessor.

	Single task (local pointers only)	Multiple tasks, no caching (uni/multi-processor)	Multiple tasks, caching (uni/multi-processor)
10 byte response	730µs	730µs / 760µs	730µs / 730µs
100 byte response	770µs	800µs / 830µs	770µs / 770µs
1000 byte response	770µs	1100µs / 1400µs	970µs / 1000µs

Figure 5: servlet response speed

Comparison to related work

SPIN [BSP+95] used Modula-3 to safely extend an operating system kernel. It demonstrated that a high level language could eloquently express the interfaces between layers and enforce the layers' abstractions. Luna maintains both these properties: although the tilde throws in an additional constraint, Luna still expresses interfaces in the language itself, and uses the language features to enforce abstractions.

Several projects [FFK+99] [HCC+98] [VB99] [BTS+98] [BH99] have explored the idea of safe language tasks. Alta [BTS+98] restricts the types of objects available for inter-task sharing to some extent, but otherwise provides an all-or-nothing choice: either unrestricted sharing between two tasks or no sharing at all. Luna tries for the best of both worlds: allow sharing between arbitrary tasks, but use the type system to track the sharing.

KaffeOS [BH99] uses write barriers to restrict the propagation of inter-task pointers, so that sharing between tasks is confined to specially allocated buffers (it severely restricts the types of objects that may be placed in the buffers: no abstract datatypes with overridden methods, for example). KaffeOS doesn't have a static distinction between local and remote pointers, so the write barrier cost is incurred for all pointers, and errors are caught at run-time rather than statically.

Both KaffeOS and Alta concentrate on shared memory communication; while Alta supports inter-task RPC, it is more than an order of magnitude slower than Luna's cross-task calls. The J-Kernel [HCC+98], on the other side, supports RPC (with deep copies of arguments and return values) but not shared memory. The J-Kernel's copy routine is trusted and not customizable, whereas Luna lets the user write their own copy routines by using field accesses to remote objects (such as the lazy copies implemented in our web server).

Luna's remote pointer types and the lower level annotated typed representation were inspired by the region annotations of Tofte et al [TT94][CWM99]. Originally, we imagined having one "region" per task, but then decided to let a task allocate multiple "permits", so that it can selectively revoke some permits but not others.

Conclusions

When we first started working on language based protection, we had wild hopes of outperforming traditional operating systems and microkernels. The lack of separate address spaces would allow fine-grained sharing and make protection and communication dirt cheap. All that we needed to do was to solve the (apparently) minor problems of revocation, resource control, and terminating tasks so that the system would be robust enough to handle servers, agents systems, and active network systems. We found that these issues weren't nearly as easy to solve in a tightly coupled safe-language system as they were in a traditional system that clearly divides tasks into separate address spaces (and has a very fast TLB enforcing this division). Rather than admit defeat to traditional systems on these issues, we introduced a clear separation between tasks into Luna, but unlike the dynamic separation based on virtual memory address spaces, we made a static distinction in the type system. The contribution of this paper lies in:

- an argument for a *task* model in safe language setting, where each task has its own code, objects, and threads
- a type system extension (~) which allows complete mediation of inter-task communication

- a run-time mechanism that uses the ~ to implement revocation and termination. We found that the costs of revocation were reasonable for method invocations, but array and field accesses were challenging to optimize, requiring extensive cooperation between the compiler and run-time system.
- the application of these ideas to a real-world language, using the fastest optimizing Java virtual machine (Marmot) that we could get our hands on, to demonstrate that our mechanism did not preclude global optimizations or optimization of operations on local pointers, even in the presence of dynamic loading

The result is a system that combines the robustness, structure, and communication flexibility of a microkernel with the ease of expression, portability, and abstraction enforcement of a high-level programming language.

References

[BH99] G. Back and W. Hsieh. *Drawing the Red Line in Java*. Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems, Rio Rico, AZ, March 1999

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System*. 15th ACM Symposium on Operating Systems Principles, p. 267–284, Copper Mountain, CO, December 1995.

[BTS+98] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. *Java Operating Systems: Design and Implementation*. University of Utah Rechnical Teport UUCS-98-015, August 1998

[CWM99] K. Crary, D. Walker, and G. Morrisett. *Typed Memory Management in a Calculus of Capabilities*. To appear in 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages

[DP93] P. Druschel and L. L. Peterson. *Fbufs: A high-bandwidth cross-domain transfer facility*. Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, p. 189-202, Dec. 1993.

[FFK+99] M. Flatt, R. B. Findler, S. Krishnamurthi and M. Felleisen. *Programming Languages as Operating Systems (or, Revenge of the Son of the Lisp Machine)*. International Conference on Functional Programming (ICFP), Paris, France, Sep. 1999.

[FKR+99] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgard, D. Tarditi. Marmot: An Optimizing Compiler for Java. Microsoft Research Technical Report MSR-TR-99-33, June 1999.

[FL94] B. Ford and J. Lepreau. *Evolving Mach 3.0 to a Migrating Thread Model*. Proceedings of the Winter Usenix Conference, January 1994.

[Gen] General Magic. Odyssey. http://www.genmagic.com/agents.

[GMS+98] M.Godfrey, T.Mayr, P.Seshadri, and T. von Eicken. *Secure and Portable Database Extensibility*. Proceedings of the 1998 ACM-SIGMOD Conference on the Management of Data, p. 390-401, Seattle, WA, June 1998.

[HCC+98] C. Hawblitzel, C. C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. *Implementing Multiple Protection Domains in Java*. 1998 USENIX Annual Technical Conference, p. 259-270, New Orleans, LA, June 1998.

[HK93] G. Hamilton and P. Kougiouris. *The Spring Nucleus: a Microkernel for objects*. Proceedings of the Summer 1993 USENIX Conference, p. 147-159, Cincinnati, OH, June 1993.

[HLP98] R. Harper, P. Lee, and F. Pfenning. *The Fox Project: Advanced Language Technology for Extensible Systems*. Technical Report CMU-CS-98-107, Carnegie Mellon University.

[HM98] J. Howell and M. Montague. *Hey, You Got Your Language In My Operating System!* Technical Report PCS-TR98-340, Dartmouth College, 1998

[HvE99] C. Hawblitzel and T. von Eicken. *Type System Support for Dynamic Revocation*. ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta, GA, May 1999.

[Java] JavaSoft. *Java Servlet API*. http://java.sun.com.

[Javb] JavaSoft. *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?* JDK 1.2 API documentation, http://java.sun.com.

[LES+97] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, T. Jaeger. *Achieved IPC Performance.* 6th Workshop on Hot Topics in Operating Systems, Chatham, MA, May 1997.

[MWC+98] G. Morrisett, D. Walker, K. Crary, and N. Glew. *From System F to Typed Assembly Language*. 25th ACM Symposium on Principles of Programming Languages. San Diego, CA, January 1998.

[NL98] G. Necula and P. Lee. *The Design and Implementation of a Certifying Compiler*. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation. Montreal, Canada, June 1998.

[Red74] D. D. Redell. *Naming and Protection in Extendible Operating Systems*. Technical Report 140, Project MAC, MIT 1974.

[SCM99] O. Shivers, J. W. Clark and R. McGrath. *Atomic heap transactions and fine-grain interrupts*. Proceedings of the 1999 ACM International Conference on Functional Programming (ICFP), Sep. 1999, Paris, France.

[Sha97] Z. Shao. *Typed Common Intermediate Format*. 1997 USENIX Conference on Domain-Specific Languages, Santa Barbara, California, Oct. 1997.

[SS75] J. H. Saltzer and M. Schroeder. *The Protection of Information in Computer System*. Proceedings of the IEEE, Volume 63, Number 9, p. 1278–1308, September 1975.

[TT94] M. Tofte and J.P. Talpin. *Implementation of the Typed Call-by-Value Lambda Calculus using a Stack of Regions*. 21st ACM Symposium on Principles of Programming Languages, p. 188–201, Portland, OR, January 1994.

[VB99] J. Vitek and C. Bryce. *The JavaSeal mobile agent kernel*. In First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA'99), October 1999.

[WBD+97] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. *Extensible Security Architectures for Java.* 16th ACM Symposium on Operating Systems Principles, p. 116–128, Saint-Malo, France, October 1997.

[WLH81] W. A. Wulf, R. Levin, and S. P. Harbsion. *Hydra/C.mmp:An Experimental Computer System*. McGraw-Hill, 1981.

[WGT98] D. Wetherall, J. Guttag, and D. L. Tennenhouse. *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols*. IEEE OPENARCH'98, San Francisco, CA, April 1998.