Using Interpreted *CompositeCall*s to Improve Operating System Services

Francisco J. Ballesteros[1]* Ricardo Jimenez[2†] Marta Patiño[2†] Fabio Kon[3‡]
Sergio Arevalo[2†] Roy H. Campbell[3§]

[1]Systems and Communications Group. Carlos III University of Madrid
[2]Distributed Operating Systems Group. Technical University of Madrid
[3]Systems Research Group. University of Illinois at Urbana-Champaign

Abstract

A high number of protection domain crossings and context switches is often the cause of bad performance in complex object-oriented systems.

We identified the *CompositeCall* pattern which has been used to address this problem for decades. The pattern modifies the traditional client/server interaction model so that clients are able to build compound requests which are evaluated in the server domain.

We implemented *CompositeCall*s for both a traditional OS, Linux, and an experimental object-oriented research μkernel, *Off*++. In the first case, we learned about implications of applying *CompositeCall* to a non-object-oriented "legacy" system. In both experiments, we learned when *CompositeCall*s help improving system performance (and when they do not help). In addition, our experiments gave us important insights about some *pernicious design traditions* extensively used in OS construction.

1 Introduction

In operating systems, invoking a system service is usually a heavy-weight operation due to protection domain crossing. In distributed systems, invoking remote services is more expensive than invoking local services due to network latency and processing overhead. However, many applications spend most of their time within a tight loop, issuing repeated calls to objects in a different protection domain or in a different node. A non-negligible portion of the processor time consumed by these applications is entirely spent in domain-crossing.

Service designers have to decide whether to provide non-primitive operations (i.e. those which could be built using already implemented operations) or not. If they are included, the interface gets more complex and changes in the primitive operations may affect the non-primitive ones [1]. If they are not included, a larger number of domain crossings (or messages) might be needed at run time.

To state it more clearly, consider for instance a system service such as a name service, a connection service, or even a complete operating system. It is typical for a single application to issue several calls to the domain where the service resides. A program like `pc_copy`, which uses a file server, can be an example of such system usage pattern:

*Partially supported by Spanish CICYT grant # TIC-98-1032-C03-03.

†Partially supported by the Spanish Research Council CICYT grant # TIC-98-1032-C03-01 and by the Madrid Regional Research Council grant number CAM-07T/0012/1998

‡Fabio Kon is supported in part by a grant from CAPES, the Brazilian Research Agency, proc.# 1405/95-2.

§The Systems Research Group is supported in part by a grant from the National Science Foundation, NSF 98-70736.

[1]As the implementor may fall into the temptation of using some internal feature of the service.

```
// Using primitive calls
pc_copy() {
 while (FileSys::aFile.read(buf))
   write(FileSys::otherFile.write(buf));
}
```

Calls to either operating system or remote services (e.g to `FileSys`) are much more expensive than calls within the client domain. Therefore, it would be not just convenient but also much more efficient to use a non-primitive operation like `copy`.

```
// Using composite calls
cc_copy() {
  FileSys::otherFile.copy(FileSys::aFile)
}
```

The difference between the original `pc_copy` and `cc_copy` is that the former uses four domain crossings per loop (two per call). The latter uses just two domain crossings, no matter what the number of iterations is.

Sadly, well-designed servers provide just primitive operations (i.e. operations which cannot be built using other operations already provided by the server). Therefore, a operation like `copy` is seldom provided. What the client could do instead, is to send the whole `while` loop to the file server. A single cross-domain call (i.e. two domain crossings) would be enough to perform the file copy.

The *CompositeCall* pattern enables the extension of server interfaces for safe execution of repeated sequences of service calls and *simple* control structures. It provides the means to compose separate calls to a server into a single one. A *CompositeCall* is indeed a program a client sends for execution in the server domain.

It is well known that some existing systems, like SPIN [3], support code-downloading as a means for extensibility. Our main contributions are that we have identified the pattern being discussed, and we are applying it to systems not designed to support such feature. Also, we have applied it to a distributed adaptable μkernel, *Off*++. Besides, we have employed very light-weight interpreters, which performed surprisingly well (at least when compared with those heavy-weight interpreters used in previous systems).

In the case of bulk data transfer operations, a very large amount of data copying can be avoided by using *CompositeCall*s. Compare, for instance, `pc_copy` and `cc_copy` considering that the file service is provided by a remote NFS server. In the first case, the whole file must be sent to the client and back to the server. In the second case, by means of *CompositeCall*s, the file content does not need to leave the server just to be copied back to the place where it came from.

We found that other systems concepts such as gather/scatter IO, message batching, deferred calls, and heterogeneous resource allocation could be seen as instances of this pattern. By allowing clients to compose calls, all these abstractions can be provided by a single piece of code as described below.

Using *CompositeCall*s helps to keep the system (server) small, as only *primitive* operations must be included. Non-primitive operations can be provided by programs built by clients.

After identifying the *CompositeCall* pattern, we have applied it to improve the performance of user programs in two different environments, Unix and *Off*++ [1, 15]. In *Off*++, the pattern is also applied to provide support for disconnected operations, gather/scatter I/O, and heterogeneous resource allocation—these services being not provided as primitive operations.

The remainder of this paper is organized as follows. We introduce the *CompositeCall* pattern in section 2. Section 3 discusses some issues related to the application of *CompositeCall* to OS services. This includes the design and implementation of the pattern on Linux (section 3.1) and on *Off*++ (section 3.2). Section 4 presents experimental results and lessons we learned. Finally, section 5 is devoted to related work and section 6 presents our conclusions and future work.

2 The *CompositeCall* pattern

The *CompositeCall* pattern combines a simple control command language with an existing server as shown in figure 1 (this figure and the following ones follow the OMT notation [16] variant used in [6]).

The goal of *CompositeCall*s is to enable users to send simple groups of calls (or programs) to the server. Issuing separate single calls can thus be avoided under many circumstances. In fact, the programming model for using system services is being shifted from a "protected library" providing several entry points to an "interpreter" executing client programs to service requests.

Clients compose primitive calls to build a *CompositeCall*, also known as `Program`. The program is then sent to an extended server, or `InterpServer`. A single instance of `InterpServer` resides in the server protection domain. The `InterpServer` implements `execute` as an alternate entry point into the server.

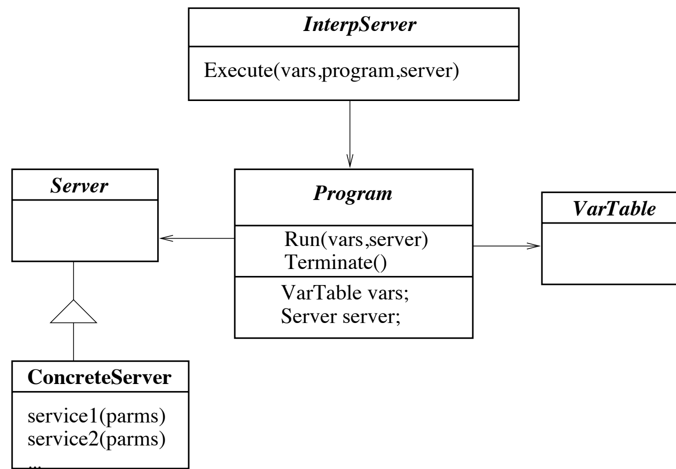

Figure 1: Main participants in the *CompositeCall* pattern.

To enable the use of a single `InterpServer` with different servers, a reference to an abstract `Server` is supplied to `execute` whenever a program needs to be executed. Concrete `Server` subclasses wrap existing servers, providing a way for the program to issue calls to legacy services.

A complete view of the entities involved in the *CompositeCall* pattern is depicted in figure 2.

All `Programs` are made of `Commands`. The set of `Commands` accepted by a `Program` can be divided into:

- Control commands: which allow the construction of simple control structures (e.g. `While` in figure 3).
- Call commands: which issue calls to primitive server entry points. (e.g. `Read` and `Write` in figure 3).

Depending upon the chosen control command language, different interpreters can be used. In particular, we have designed and implemented both a high-level command language (`HighLevelProgram`) and a low-level byte-code based language (`LowLevelProgram`).

Our goal is to let users write programs – like the one shown in figure 3 – and compile them to generate low-level programs which can be interpreted more efficiently. In fact, depending on the latency of domain-crossing operations, low-level programs might not be needed at all. If the extra latency introduced by domain-crossing is very large, like on WAN distributed applications, high level programs will already produce significant performance improvements. Note that, as discussed in section 3, there might be more reasons than just latency to use *CompositeCall*s.

A detailed description of how high-level programs are built and compiled is out of the scope of this paper (see the extended version in [15]). However, an example of a high-level program is presented in figure 3.

Constructors for concrete classes representing "control structures" and "server call" commands allow for a convenient syntax. Instances representing the structure of the program are built while constructors are invoked. They end-up building a (syntax) tree which represents the program structure.

Such "high-level" programs are built by the user (or the client) and can be serialized and sent to the server, where it is deserialized for interpretation. Alternatively, it can be compiled by the client into a "low-level" program before being sent, as suggested in line 11 of figure 3.

The compilation triggered in line 11 of figure 3 is what could be called on-line compilation. Of course, it is always feasible to compile the program off-line and then include just the low-level program into the user application.

`run`, the main method of `Program`, triggers program execution by calling the `do` method of the proper `Command` (e.g.: In figure 3, the cross-domain call to `execute` will call to `program.run`, in the server domain; afterwards, `program.run` calls to `Sequence::do` in the `Sequence` instance).

A single storage area, named `VarTable`, is required to run a program. It is supplied as a parameter to `execute`. Some entries in `VarTable` act as input or output arguments for the program, others behave as local temporary variables. The storage area is returned to the user upon program com-

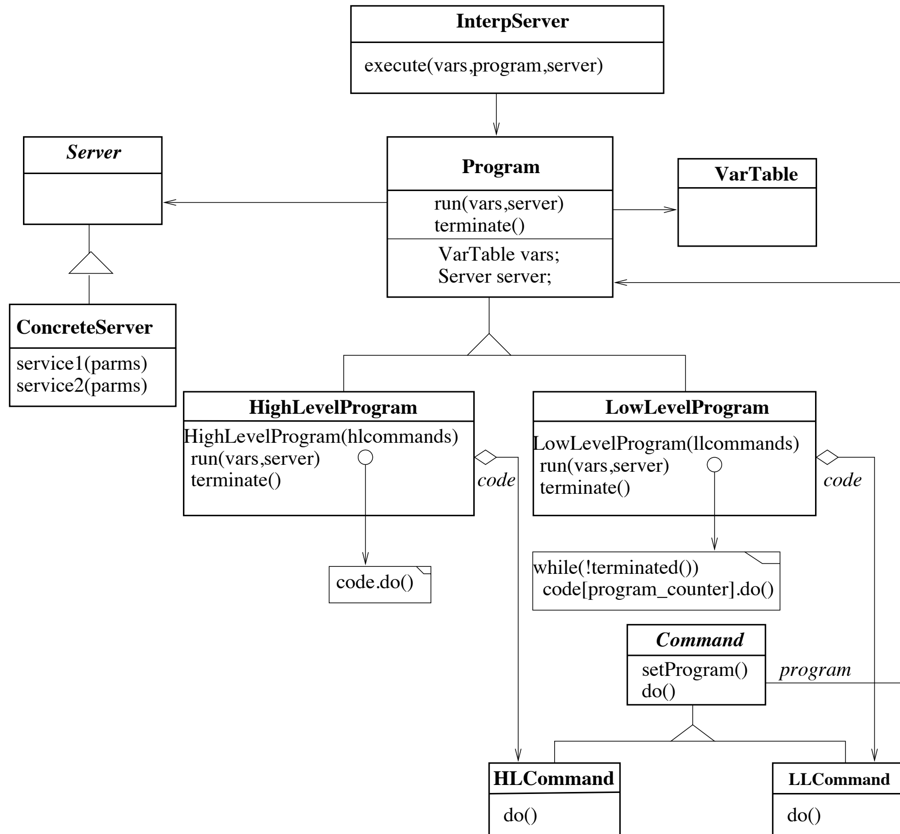

Figure 2: *CompositeCall*s: The whole picture.

```
1: VarTable vars;                  //  Declares a variable pool.
2: StringVar buf(vars,100);        //  Allocates a string of up to
3:                                 //    100 characters in vars.
4: IntVar    len(vars);            //  Allocates an integer in vars.
5: HighLevelProgram hprogram =
6:         Sequence(                     // These Constructors can
7:              Read(buf,len),           // initialize hprogram by
8:              While( Greater(len,0),   // building a tree
9:                Write(buf,len)
10:           ));
11: LowLevelProgram program = hprogram.compile(); // which can be
12:                                               // translated
13:                                               // to byte-code
14: InterpServer::execute(vars,program,server);   // and executed.
```

Figure 3: A high-level user program for copy: this code, which executes within the client, builds a program for copy (lines 5 to 10), and sends the program for execution into the server domain (line 14).

pletion.

The pattern is completely independent of the transport mechanism used to deliver calls to the server. It can be used in systems using trap-based system calls, remote method invocations, Rendez-Vous, or any other IPC mechanism.

2.1 Related patterns

A `Program` is actually an interpreter for programs made of `Commands`. The *Interpreter* pattern [6] is thus used to implement the desired command language. In turn, `Commands` are usually *Composite*s [6], so high-level constructs (e.g. loops) can be expressed cleanly.

The *Visitor* [6] pattern can be used to *compile* high-level programs into byte code to be sent to the `InterpServer`. Different mechanisms can be used to issue the call from the client to the server, like for instance the *RendezVous* pattern [7].

The *Active Objects* [11] pattern can also be used to decouple the client from the server, by decoupling method invocation from method execution. It can be combined with *CompositeCall* pattern, so that the *CompositeCall* is isolated from server concurrency issues.

3 Using *CompositeCall*s

Now, we discuss some issues regarding the use of *CompositeCall* in Operating Systems.

Should *CompositeCall* be used? Using *CompositeCall* is worthwhile when enough number of calls are issued from within the program. Otherwise, the overhead introduced by having to generate, send and interpret the program will be larger than the gain from using *CompositeCall*. In some cases, the relative overhead is so small that it is worthwhile to use *CompositeCall* to provide simple non-primitive operations.

*CompositeCall*s can be also used to decouple the service requester (the program builder) from the service provider and the calling mechanism. A *CompositeCall* program can be passed back and forth between different components of the client while calls, targeted to the server, are added to the program. Finally, the program is delivered to the server domain for execution.

The level of indirection provided by the program can be used as an *indirect call* [2], as one can transmit the program to the server by different means.

Our experience says that, in the cases in which the only motivation for using *CompositeCall* is efficiency, careful timing must be done. Depending on the interpreter used and the latency of domain crossing, it might, or might not, be worth the effort.

Existing services need no changes to support *CompositeCall*s. Since *CompositeCall* works by simply aggregating existing calls, legacy servers can be used off-the-shelf with this pattern.

The system call mechanism is used as-is, without changes, to transfer the program and the variable array down to the kernel. Once the program has reached the server-domain, it is verified and given to the interpreter (the implementation of the `execute` method).

Security is not compromised. No access is given to the user other than that granted by existing system services.

Verifying the program for safety is a very simple operation. Such process consists on ensuring that only valid commands are included in the program. The simpler the command language, the simpler the program verification. In the extreme case, when the command language is made just of call commands, the only check needed is ensuring that called entry points exist. We have found that the complexity of the interpreter heavily influences *CompositeCall* performance (i.e. the simpler, the better).

Note that every primitive system call still verifies its arguments before doing the actual work. The only difference is that these arguments now come from the `VarTable` instead of coming from the user space. Therefore, there is no difference regarding security between an interpreted program and the corresponding sequence of system calls. We discuss more about this issue in the following sections.

Error handling and recovery must also be addressed. When the user calls system services directly, (s)he is notified of any error condition. That happens usually immediately after the system call returns. However, what should be done if a command fails during the execution of a `Program` given to the `InterpServer`?

Our experience with *CompositeCall*s shows that users typically build programs assuming that either

1. every call will succeed, and no error condition is checked by any command in the program; or
2. calls are likely to fail and explicit commands are inserted in the program to deal with error conditions.

In the first case, it is convenient to let the interpreter abort the execution of the program as soon as a command fails. Error checking is performed implicitly by the call `Commands`, and the user does not need to insert more commands for that. In the second case, the interpreter must ignore error conditions, as the user `Program` will test such conditions in the following `Commands`.

In any case, it is the responsibility of the concrete `Program` to provide either `Commands` or any other means for the user to express the desired behavior (e.g. our interpreters include `AbortOnError` and `DoNotAbortOnError` commands).

Side effects may behave differently with clients issuing cross-domain calls and clients using *CompositeCall*s. With *CompositeCall*s, server calls are issued *within* the server domain, not from the client domain (i.e. they are issued from within the kernel in the pattern instances we built for Linux and *Off*++). Besides, depending on the command language, (infinite) loops might be dowloaded into the server on behalf of a single client process. This should be taken into account when implementing an instance of *CompositeCall*s.

The problem is that certain servers do not do all their work on response to entry point calls. Sometimes, some work might be done by the skeleton code between the transport (i.e. network, or caller domain) and the server entry point. An example could be a server creating new threads, acquiring or releasing locks, or executing pending background tasks, within skeleton code.

In modern and cleanly designed operating systems, this should not be a problem. In other cases (which include some instances of UNIX and Windows variants) that is certainly an issue as shown in section 3.1.2.

In general, if the skeleton code produces side-effects, they must be taken into account by the *CompositeCall* implementation. As a *CompositeCall* issues several calls without traversing all the skeleton from the network to the server, such side-effects might not be triggered as they were when using simple calls. Server implementations assuming that such side-effects will be honored frequently or *between* any two successive server calls might behave badly with *CompositeCall*s.

Section 3.1.2 describes how we addressed this problem in our experimental implementations.

3.1 Applying *CompositeCall* on Linux

We have instantiated the *CompositeCall* pattern using the Linux kernel as the `Server`. Even though the interpreter has been written in C, its implementation matches the design pattern here described. Therefore, all elements found in the pattern, as shown before, can be found in this instance. The overall picture is shown in figure 4.

An instance of the *CompositeCall* `InterpServer` was added to the Linux kernel as a new system call (`interp`).

```
int
interp(prog_t prog[], void *vars,
       int lp, int lv, int flags);
```

The `interp` system call receives the program to be executed `prog` (of length `lp`), a variable array `vars` (of length `lv`), and some flags.

The low-level interpreter implements the following concrete `LowLevelCommands` inside the Linux kernel:

- Simple arithmetic commands like `ADD` and the like which operate on two entries of the `vars` variable table.
- Comparison and branch commands. They compare two entries in `vars` and adjust the *CompositeCall* program counter if the test succeeds.
- An unconditional branch command.
- A `MOVE` command, used to perform copies within the argument array.
- A family of `LinuxCall` commands, used to issue system calls within the kernel.

Arithmetic, branch, and move instructions are extremely simple. The references they use are indeed indexes into the program and variable array. Thus, they are not able to access any kernel data outside of the variable array.

Input values for the system calls might be either preset in the variable array when the user calls `interp`, or might be set at program (`prog`) run time by move or arithmetic instructions. Of course, an input value for a system call might come from an output value of a previous call.

In the following section, we discuss some issues related to this implementation.

3.1.1 Implementation

The `interp` system call has been statically linked to a Linux kernel, although a loadable module could have been used instead.

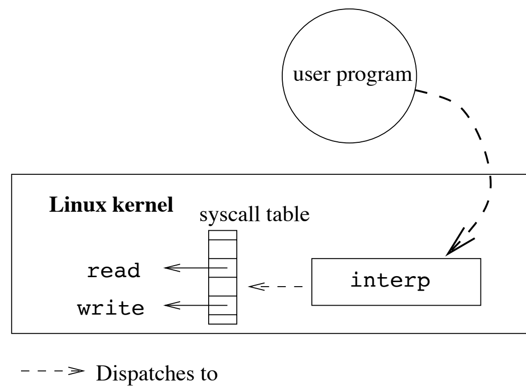

Figure 4: The *CompositeCall* instance for Linux system calls

As all arguments for existing Linux system calls fit into `long` integers, we have wrapped existing calls into just six different `Services` (these services are methods of the *CompositeCall* `ConcreteServer`, as shown in figure 1). For each `Service`, there is a low level command used to codify a system call in the downloaded `Program`. Concrete `LinuxCall` commands are named `call0` to `call5`, depending on the expected number of arguments[2].

Each concrete `LinuxCall` command contains:

- The system call ID number (also implicit in the command type).
- The number of arguments (also implicit in the command type).
- The index in the variable array where arguments start.
- The index in the variable array where the result should be placed.

Using the first two members we can dispatch to the actual system call. System call arguments and return values are handled by using the last two `LinuxCall` fields. Return values from system calls are stored in the `VarTable` (`vars`) at the specified slot. This slot can be verified and used in successive program instructions.

It could have been the first impression, after looking at the pattern, that additional argument data copying is needed. That is not the case. Note that, in calls accepting user supplied buffers, (e.g. `read` or `write`), such buffers do not need to be copied more times than when using traditional system calls. As an example, the buffer argument for `read` is a pointer to a user-space storage area which is still handled by `read` as if it were called by the user.

3.1.2 Side effects

Unfortunately, we faced some unwanted interactions between the interpreter and some Linux mechanisms. All of them did appear because some operations are triggered by checks performed *within* the system call return path. With *CompositeCall*, those checks were being honored at the end of the program.

Scheduling. Special care needs to be taken with the interaction between the `interp` mechanism and the Linux scheduler. As the kernel is non-preemptive, there is no opportunity to preempt the process during the `interp` system call. Of course, system calls issued by the user process using `interp` will still block and resume as usual, but the *end-of-quantum* event might not be honored until the interpreted program finishes.

Fortunately, the solution is simple: `interp` must check a flag set by the kernel whenever the processor quantum expires. This *needs-reschedule* flag must be checked after each system call. It must also be checked periodically by the interpreter, even when no system call has been issued. That is to prevent a program with an infinite loop from freezing the system.

If the flag is set, the interpreter calls the scheduler, as Linux would do, possibly preempting the current process. The interpreter remains in a “ready to run” state until placed again on a processor.

[2] That is indeed the way Linux (and most of other OSes) implements its system calls.

Signals. Yet another side-effect is the signal delivering mechanism. Signals are not actually *delivered* when they are sent. A flag is set in the process structure which is later checked. Such flag is precisely checked when system calls return. If a signal is sent to a process executing `interp`, it would not be delivered until the end of `interp`. Among other things, this has the undesirable effect of inhibiting the interrupt signal.

Again, the solution we found was to check the *pending-signals* flag within `interp`. It must be checked on a periodic basis and after every system call. Unfortunately, the routine delivering a signal assumes that the process is always returning from a system call, which is no longer the case. Such code operates on the process stack and behaves in different ways depending on the caller.

Although it could be expected that calling the signal delivering routine would suffice, it does not. We simply opted for aborting the whole interpreter program and returning an error code informing the user that a signal occurred.

Mechanisms to resume the program from the state where it stopped when the signal was delivered can be provided. For low-level programs it is just a matter of returning the program counter and the variable table to the user (perhaps, using a *Memento* [6]). The program can then be adjusted and redownloaded to complete its execution. Alternatively, it could be cached within the kernel to avoid repeated downloading.

3.2 Applying *CompositeCall* on *Off++*

Off++ [1, 15] is a research distributed object oriented μkernel used by the 2k [10] operating system. In *Off*++, calls to system objects proceed through remote method invocation (RMI) into the kernel domain. Such RMI employs user and kernel wrappers, as seen in figure 5, and it might cross the network (*Off*++ is a distributed μkernel). The user wrapper is a proxy which delivers messages to the kernel domain; the kernel wrapper verifies user arguments and performs access checks.

Services provided by *Off*++ are mainly allocation and deallocation of (distributed) physical resources (page frames, address translations, processor slots, etc.). Therefore, it is common for users to issue several calls at a time (e.g. to allocate a page frame, allocate an address translation, and setup the translation so that it points to the allocated page frame.)

The *CompositeCall* pattern has been instantiated for *Off*++ and implemented using *C*++. In this case, we implemented two different command families. `off_ByteCode` is the one used in the Linux implementation (wrapped in *C*++). `off_CallArray` includes just the constructs needed for manipulating resource arrays. The latter permits allocation of multiple resources in a single composite call.

Depending on the control command family, we can build either `off_CallArray` programs or `off_ByteCode` programs. Both of them can be used as `off_Programs`.

Programs built using `off_CallArrays` can use the following high-level commands:

Repeat(Command, n) which performs the given Command `n` times.

Move(from, to, i, o, size, n) which copies `n` items, of the specified `size`. Items are taken starting at `from`, using a step of `i` bytes (e.g. the kth item will start at `from`$+k\cdot i$). Items are copied to the address `to`, using a step of `o` bytes.

These constructs can be used to allocate multiple resources which may be used on subsequent requests.

An `OffCall` command is required in both command families, to perform calls to kernel objects. The `OffCall` accepts as arguments the object and method the message is targeted to, an input message, and an output message. When the `OffCall do` method is called within the kernel, a call is made to in-kernel object wrappers. These wrappers were already present in *Off*++, as part of the system call mechanism, and they transform message delivering into object invocation (thus, there is not additional overhead). Arguments for the called object are taken from the input message. Result values are incrementally stored into the output message, which is returned to the caller.

As it happened with the Linux instance, access checks are performed (this time, by in-kernel wrappers) as usually within the kernel.

3.2.1 Implementation

Most of the *CompositeCall* implementation consists of including an `off_Interp` instance co-located with the *Off*++ kernel domain. The `off_Interp` instance is indeed our `InterpServer`. It provides a new `execute` entry point to the kernel.

As this implementation uses an object oriented language, the concrete type of the `off_Program` determines which implementation of the interpreter must be used.

In *Off*++, both kernel and user are preempted when needed; the kernel behaves like a *protected library* for user processes. There was no need to deal with side-effects.

There was no need to modify any kernel code to use *CompositeCall*, and the implementation follows the class

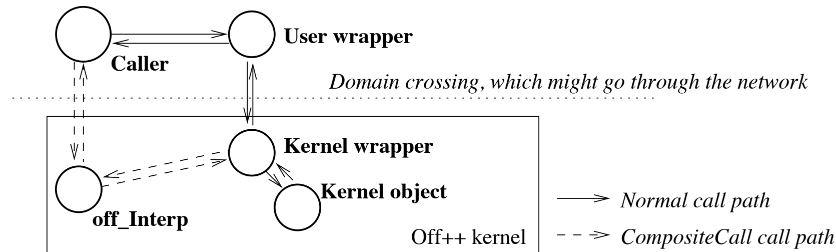

Figure 5: Normal system call path and *CompositeCall* call path in *Off*++.

diagram shown for the pattern in section 2. Thus, there are no further implementation issues to be discussed.

4 Experimental results

***CompositeCall* performance on Linux** was measured using a traditional `copy` program and a modified one, `icopy`, using the interpreted program shown in figure 6. Both of them copy what they read from their input into their output.

Because the `copy` program has to issue several system calls, the overhead imposed by the `interp` system call (building the program, copying it and the variable array, and decoding program instructions) may be outweighed by the time it saves on domain crossings.

The Linux system call path is well tuned. In our initial implementation, using *CompositeCall* was only worthwhile when more than 5,000 calls were issued by the same program. The program setup time was only amortized when `interp` could save, at least, 10,000 domain crossings.

After carefully tuning our `interp` implementation, we observed that the use of `interp` started to pay when the program issued more than 7 system calls within the interpreted program. A small difference in the performance of the interpreter inner loop can make the difference between achieving a speedup or a slowdown.

Frequently used programs can be kept within the kernel, so that users only need to supply the variable table. Programs may be installed in the kernel (they are small), and then used many times. As programs tend to match commonly used non-primitive operations, they can be aggressively reused by a process, by different processes and even by different users. Caching programs eliminates some overhead (due to program copying) and leads to the figures [3] shown in table 1 (see also figure 7). Cached *CompositeCall*-based programs can run faster than their traditional counterparts, even when *only two* system calls are issued within the *CompositeCall*.

In our experiment, for non-cached *CompositeCall*-based programs, 16 μseconds should be added to the execution times shown in table 1 (and figure 7). The reason is that it takes 16 μseconds to setup a new `copy` program for `interp`. Therefore, instead of just 2 system calls, non-cached programs must issue at least 7 system calls within the *CompositeCall* to run faster than their traditional counterparts.

We plan to implement the interpreter inner loop in assembler so that *CompositeCall*s could be even more useful in Linux environments. Nevertheless, even our simplistic interpreter achieves a speedup of more than 25%. These measurements correspond to a system with a relatively cheap, very well optimized user/kernel domain crossing.

On distributed systems, and object-oriented systems with expensive domain crossing, the performance improvements due to *CompositeCall* should be even higher. Note that our experiments do not capture the case in which *CompositeCall* avoids sending data through the network (as in the NFS `copy` example from section 1).

***CompositeCall* performance on *Off*++** was measured by implementing several services with both primitive system calls and with *CompositeCall*s. The chosen services were a user-level page fault handler, and a page frame allocator.

We describe, here, just the fault handler shown in figure 8. For the sake of simplicity, we have omitted some few additional parameters and declarations. The page fault handler allocates a page frame and installs a translation to

[3] Figures shown correspond to the mean of 10,000 experiments on a 100MHz Pentium-based Toshiba 110CS.

```
// CompositeCall-based program for copy. Slots in variable array are:
//     0: unused; 1-3: fd,buf,len for read;  4-6: fd,buf,len for write;
//     7: 0;  8: result;  9: PC for start (0); 10: PC for end (4)
START:
 call read/3, 1, 6  // call to read with 3 args. Take args from
                        //    slot #1 in vars. Store result at slot #6 in vars.
 jmple 6,7,10           // jump to PC in slot #10 if slot #6 <= slot #7.
                        //    i.e. jump to END if read result <= 0
 call write/3, 4, 8     // call to write with 3 args. Take args from
                        //    slot #4 in vars. Store result at slot #8 in vars.
 jmp  9                 // jump to PC in slot #9 (i.e. to START)
END:                    // terminate program execution.
 end
```

Figure 6: A *CompositeCall*-based program for copy.

number of calls:	1	2	3	4	5	6	7	8	9	10
Using system calls	36	46	57	68	78	89	98	108	119	131
Using *CompositeCall*	38	45	54	61	67	77	82	94	98	107

Table 1: Times (in μseconds) for copy programs on Linux issuing a fixed number of system calls

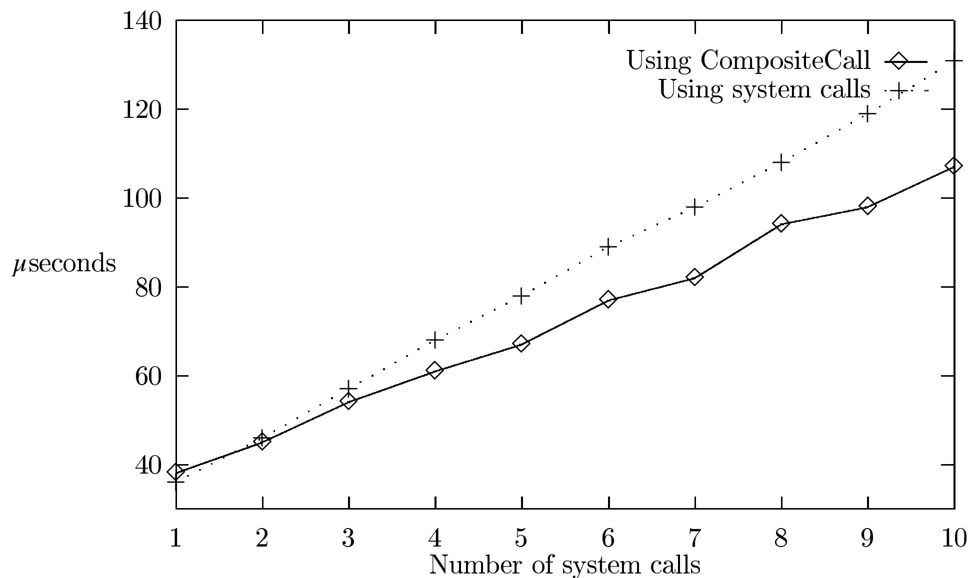

Figure 7: *CompositeCall*-based copy vs. Traditional copy on Linux.

it[4].

This routine can be transformed into another call to `off_Interp.execute`, passing to it the program shown in figure 9 (which performs the same task done by the one in figure 8). Figure 9 also shows one way to build the program with *CompositeCall*s. Only three instructions are needed: (1) a call to the page allocation method in the memory bank, (2) moving the identifier of the allocated page frame into the map request, and (3) issuing a map request (to install a new address translation).

The numbers shown in table 2 correspond to the execution of both handlers. It can be seen how a handler, using *CompositeCall*, executes 32% faster than a traditional one.

We have also used a program allocating a given number of page frames to get a picture of how *CompositeCall* behaves in *Off*++ as the number of issued system calls increases. Results can be seen in figure 10. The big amount of time spent on executing a system call is due to the expensive set of debug checks being done by the kernel version employed. We did not remove such checks to obtain performance measures for *CompositeCall* on a system with expensive system calls.

4.1 Lessons learned

The asymmetry between client and server code hurts. Although this issue is not strictly related to the *CompositeCall*, we learned that it was the asymmetry between the kernel and the user code the one causing most of the problems in the Linux implementation. All interactions with preemption and signal delivering showed because the kernel behavior is not symmetric with respect to user code, and kernel code can not be written in the same way user code is.

The non-preemption of the kernel (apart from degrading performance on multiprocessor systems) makes it infeasible to write system calls which can compute for an indeterminate amount of time. At least, without carefully calling the scheduler.

Also making the implementation of signal delivering not really asynchronous also makes infeasible to write system calls which can compute for an indeterminate amount of time.

This lesson can be extrapolated to a more general case: on servers using a single thread to serve all client requests, special care must be taken. If *CompositeCall*s are used with single-threaded servers, they may modify the server concurrency semantics by stealing the server thread for a long period. It is advisable to either forbid non-terminating programs or create additional threads to service requests from different clients. Thread processing may be encapsulated in concrete servers wrapping existing ones. In order to implement that, one could use the *Active Objects* pattern [11] to handle thread management in a clean way.

These problems were not encountered in the implementation of *CompositeCall* for *Off*++ because

- the kernel is structured as a set of servers which can be preempted in the same way that user code is preempted; and
- the system call mechanism does not present side-effects.

It is convenient to define non-primitive operations. Several *CompositeCall*s, corresponding to non-primitive operations on system services, began to appear soon. Some examples are `FileCopy` (which opens two files and copies the first one into the second), and `SendTCP` (which establishes a connection using TCP and then enters a loop sending the given buffer). One could have a whole family of composite operations for sending and receiving TCP and UDP data.

It would be very convenient to be able to use existing versions of these programs. Frequently used programs could be kept within the kernel, as mentioned before.

As an example, it is very common in *Off*++ to allocate a page frame and then install an address translation pointing to it. We could have provided an `allocate_and_install` entry point, but that would have mixed physical storage management with virtual memory facilities—which we prefer not to mix. Now, this operation could be implemented in a library using the *CompositeCall* mechanism.

Design patterns should be applied to legacy systems. There are some *apparently* disjoint pieces in almost every OS which indeed could be implemented by using *CompositeCall*s. Even though we have experience in the field of Operating Systems, we never imagined that a single piece of code could replace separate functions like gather/scatter IO and heterogeneous resource allocation.

By trying to identify common patterns in the design of different (already implemented) components, we can learn how to simplify both the design and implementation of our computing systems.

[4] In general, the user level page fault handler in *Off*++ performs additional tasks. It must, at least, analyze the reason for the page fault and decide what to do. Nevertheless, this handler is still useful for applications resident in pinned memory when they need to grow their stack or data segments.

```
// Handling a page fault on Off++ using primitive system services.
//
err_t pfhandler(off_PgFltReq *pf, off_MsgRep *r){
  off_uPFrame p;              // A page frame.
  extern off_uMBank mb;       // A memory bank.

  p = mb.alloc();             // Allocate a page frame.
  dtlb.map(pf->vaddr, p, mode);// Setup an address translation
                              // from the faulting address (vaddr) to
                              // the newly allocated page frame
                              // and install it at our protection domain.
  return (r->m_err=EOK);      // (Assuming that no allocation  fails).
}
```

Figure 8: Handling page faults in *Off*++

```
// The variable table contains:
//  PAGE_ALLOC_RQ: page allocation request message.
//  PAGE_ALLOC_REP:page allocation reply message.
//  DTLB_MAP_RQ: map request message.
//  DTLB_MAP_REP:map reply message.

cmd[0]= new OffCall(MBANK, PAGE_ALLOC_RQ, PAGE_ALLOC_REP);
cmd[1]= new Move(PAGE_ALLOC_REP + offset in that message for page frame id,
                 DTLB_MAP_REQ   + offset in that message for page frame id,
                 sizeof(page frame id),
                 1 // copy just one value
                 );
cmd[2]= new OffCall(DTLB, DTLB_MAP_RQ, DTLB_MAP_REP);

CallArray pfprogram(cmd, 3); // To be used for off_Interp::execute() calls.
```

Figure 9: Handling page faults with *CompositeCall* in *Off*++

Test	$\frac{time}{\text{timeofsinglecalltest}}$
Using single calls	1
Using *CompositeCall*	0.68

Table 2: Scaled times for page fault handlers in *Off*++.

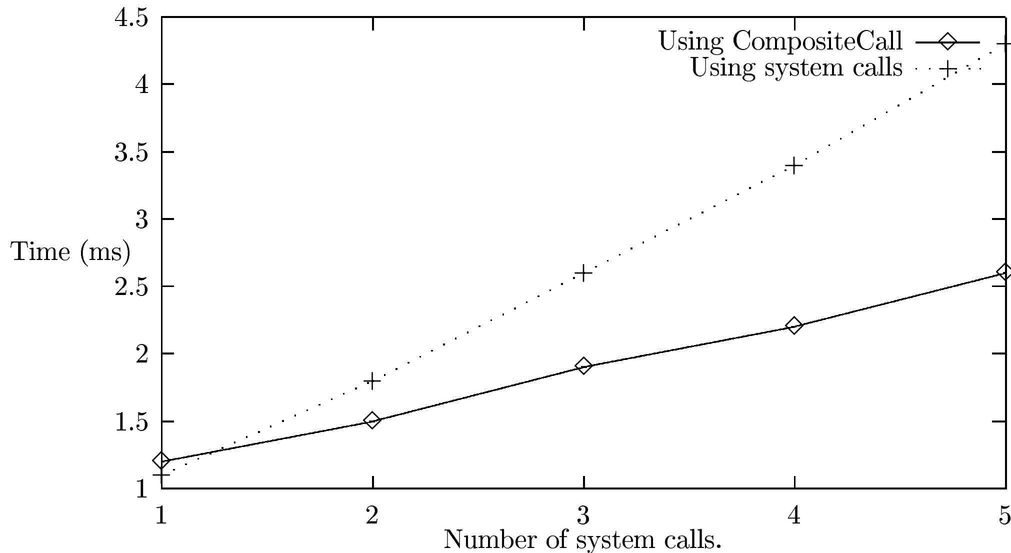

Figure 10: Performance of page frame allocation using *CompositeCall* in *Off++*.

For us, this pattern has been a process where we first learned some "theory" from existing systems, and then, applied what we had learned back to "practice".

5 Related work and other pattern instances

Our implementation of *CompositeCall*s, which entails a `Program` and a variable array is similar to the concept of *closure* [9]. In programming languages like scheme, a closure is a structure containing a lambda expression (equivalent to our `Program`) and an environment, i.e., an association of values with variables (equivalent to our variable array). A given closure represents a lambda expression with some of its free variables substituted by values in the environment. The idea of sending a piece of code and its environment for execution in a different context was applied before in different situations.

Database systems supporting Stored Procedures [5] instantiate *CompositeCall* too. A Stored Procedure can be thought of as a small data access program to be used for retrieving information from the data store.

Systems like SPIN [3], μChoices [12], and VINO [19, 18], use code downloading. They include such mechanism as a means for adaptability and extensibility.

In these systems, downloaded user programs are expected to execute at almost the same speed as native kernel code. A general-purpose language is used for programming system extensions.

On the other hand, the command language in *CompositeCall* is simply a "domain specific" language designed with the objective of composing existing calls. Thus, the language can be much simpler (therefore, safer), and users cannot cause damage to sensitive kernel or server data. Therefore, it is not a surprise, that systems mentioned above, restrict downloading of programs to trusted users, to trusted compilers, or to the intersection of both. We can summarize the difference between those systems and our work on *CompositeCall*s by saying that:

1. Code downloading in those systems may be considered as concrete instances of the *CompositeCall*s pattern where the program can be expressed in Modula 3, Java, or other general-purpose language.

2. The instances of *CompositeCall*s described in this paper—which have been developed by following the design pattern—are simpler and smaller than any comparable system. The implementation of our *CompositeCall* instance for Linux has 382 lines of code, and uses less than 1 Kbyte of memory. Compare that with the complexity and size of the Sun JVM [13].

3. The *CompositeCall* pattern can be applied to systems not designed with *CompositeCall*s in mind, as we demonstrated for both Linux and *Off++*. No change was necessary on those systems. That was not the case of systems like SPIN, which were designed with code downloading in mind. Our approach require neither ad-hoc mechanisms, nor specific compilers, nor any special kernel support to include *CompositeCall*s (apart, of course, of the added code for the *CompositeCall* interpreter).

The idea behind Agent systems [22] is closely related to *CompositeCall*. However, the aim of Agent systems is to build mobile stand-alone programs. In a *CompositeCall*, the program will remain in the server domain domain until termination; it will not move to a different domain. In *CompositeCall* the emphasis is made only on the interface shift (from a single entry point to a *CompositeCall*), and other (unrelated) technologies are left apart.

Nevertheless, some of the machinery needed for implementing Agents [22, 8] can also be considered as another instance of the pattern. Again, it is a program sent to an interpreter with some storage area. The peculiarity is that, in their case, the command language includes a `go` instruction to move the program to a different server.

What we have said also applies to systems borrowing techniques from the field of mobile Agents, like NetPebbles [14]. Active networking frameworks [21] also instantiate *CompositeCall*, their programs or *capsules* can be considered to be calls to the involved network elements.

Systems supporting disconnected operation also instantiate *CompositeCall*. Examples could be distributed systems like Coda [17] and Bayou [4], which defer changes while the system is disconnected. Pending changes are aggregated and processed by the servers when the system is reconnected. As dictated by the *CompositeCall* pattern, primitive calls (a single change or update) are composed and processed later. Most notably, Bayou [4] operations are actually programs which can detect and resolve conflicts.

Finally, lessons learned in the design of domain specific languages for other applications like user interface specification, software development process support, and text processing [20] can be applied to design adequate languages for concrete *CompositeCall* instances.

6 Conclusions and future work

We identified the *CompositeCall* pattern and discussed how it is instantiated in several existing systems. We developed two new instances of the pattern on a traditional, monolithic kernel and on a object-oriented research μkernel. No change was needed to those systems, which were not designed with *CompositeCall*s in mind.

The experimental results show that, although the *CompositeCall* mechanism can provide great performance improvements, its use must be carefully analyzed. In some cases, the overhead it imposes may be larger than the performance gain it provides. We plan to perform further experiments on distributed services in which we expect to obtain very significant speedups.

As future work, we plan to implement an optimized interpreter in assembler, so smaller Linux and *Off++* programs could benefit from *CompositeCall*s. We also plan to develop applications using *CompositeCall*s as the main abstraction for client/server interaction.

Acknowledgments

We are grateful to Gorka Guardiola Muzquiz for his help in the implementation of the Linux *CompositeCall* mechanism.

References

[1] Franscisco J. Ballesteros, Fabio Kon, and Roy H. Campbell. A Detailed Description of Off++, a Distributed Adaptable Microkernel. Technical Report UIUCDCS-R-97-2035, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1997.

[2] Carlos Baquero. Indirect Calls: Remote invocations on loosely coupled systems. http://gsd.di.uminho.pt/People-/cbm/public/ps/icalls.ps, 1996.

[3] 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. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. ACM, December 1995.

[4] W. K. Edwards, E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry, and M.M. Theimer. Designing and Implementing Asynchronous Collaborative Applications with Bayou. In *Proceedings of the Tenth ACM Symposium on User Interface Software and Technology (UIST)*, Banff, Alberta, Canada, October 1997.

[5] A. Eisenberg. New standard for stored procedures in SQL. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(4):81–??, December 1996.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Object-Oriented Software*. Addison-Wesley, 1995.

[7] R. Jiménez-Peris, M. Pati no Martínez, and S. Arévalo. Multithreaded Rendezvous: A Design Pattern for Distributed Rendezvous. In *ACM Symposium on Applied Computing*. ACM Press, Feb. 1999. To appear.

[8] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, Wa (USA), May 1995. IEEE.

[9] Samuel N. Kamin. *Programming Languages*. Addison-Wesley Publishing Company, 1990.

[10] Fabio Kon, Ashish Singhai, Roy H. Campbell, Dulcineia Carvalho, Robert Moore, and Francisco Ballesteros. 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments. In *Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.

[11] R. Greg Lavender and Douglas C. Schmidt. Active object – an object behavioral pattern for concurrent programming. In *Proceedings of the Second Pattern Languages of Programs conference (PLoP).*, Monticello, Illinois, September 1995.

[12] Y. Li, S. M. Tan, M. Sefika, R. H. Campbell, and W. S. Liao. Dynamic Customization in the μChoices Operating System. In *Proceedings of Reflection'96*, San Francisco, April 1996. Reflection'96.

[13] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Weley, 1996. Java Series.

[14] Ajay Mohindra, Apratim Purakayastha, Deborra Zukowski, , and Murthy Devarakonda. Programming Network Components Using NetPebbles: An Early Report. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, Santa Fe, New Mexico, April 1998. USENIX.

[15] *Off*++ web site. `http://www.gsyc.inf.uc3m.es/off`.

[16] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[17] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. Technical Report CMU-CS-89-165, Department of Computer Science at Carnegie Mellon University, November 1989.

[18] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, October 1996. USENIX Assoc.

[19] Christopher Small and Margo Seltezer. Vino: An integrated platform for operating system and database research. Technical report, Hardvard Computer Science Laboratory, Hardvard University, Cambridge, MA 02138, 1994.

[20] Diomidis Spinellis and V. Guruprasad. Lightweight Languages as Software Engineering Tools. In *;login: DSL'97 Conference Summaries*, volume 23, Santa Barbara, CA, February 1998.

[21] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1), January 1997.

[22] Jim White. Mobile Agents. General Magic Corporation, 1996.