Using interpreted CompositeCalls to improve operating system services

F. J. Ballesteros[1]*, Ricardo Jimenez[2], Marta Patiño[2], Fabio Kon[3], Sergio Arevalo[1], and Roy Campbell[3]

[1] *Systems and Communications Group. Rey Juan Carlos University of Madrid. C/ Tulipan E-28933 Mostoles, Madrid (Spain).*
email: {nemo,sarevalo}@gsyc.escet.urjc.es

[2] *Facultad de Informatica. Technical University of Madrid, E-28660 Boadilla del Monte, Madrid (Spain).*
email: {rjimenez,mpatino}@fi.upm.es

[3] *Systems Research Group. Digital Computer Lab. University of Illinois at Urbana-Champaign. 1304 W Springfield Av. Urbana, IL 61801 (USA).*
email: {f-kon,rhc}@cs.uiuc.edu

SUMMARY

A large number of protection domain crossings and context switches is often the cause of bad performance in complex object-oriented systems. We have 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 that are evaluated in the server domain.

We implemented CompositeCalls for both a traditional OS, Linux, and an experimental object-oriented 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 CompositeCalls 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.

KEY WORDS: Operating Systems. Extensibility. CompositeCall. Design Patterns.

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. Nevertheless, many applications spend most of their time within a tight

*Correspondence to: Systems and Communications Group. Carlos III University of Madrid. Avda Universidad, Leganes (Madrid). Spain.
Contract/grant sponsor: Partially supported by Spanish CICYT grant # TIC-98-1032-C03-03; TIC-98-1032-C03-01; Madrid Regional Research Council grant number CAM-07T/0012/1998; grant from CAPES-Brazil, proc.# 1405/95-2; US NSF grant 98-70736.

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 that 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†. 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 function like `pc_copy`, which uses a file server, can be an example of such system usage pattern:

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

Calls to either operating system or remote services 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() {
  otherFile.copy(aFile)
}
```

The difference between the original `pc_copy` and `cc_copy` is that the former uses four domain crossings per loop, i.e. two per call. The latter uses just two domain crossings, no matter what the size of the file is.

Typically, servers often provide just primitive operations—i.e. operations that cannot be built using other operations already provided by the server. Therefore, an 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, or two domain crossings, would be enough to perform the file copy.

The CompositeCall design pattern enables the extension of server interfaces for safe execution of repeated sequences of service calls and *simple* control structures. The pattern is also known as Batching [1]. 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.

Some operating systems, like SPIN [2], include support for code-downloading as a means for extensibility. Such systems have been designed with code downloading in mind, and can be extended by dynamically loading user code into the kernel. Our main contributions are that we have identified the CompositeCall pattern and that we have applied it to systems not designed to support such feature; i.e. to "legacy" systems. In our implementations, very light-weight interpreters process the composite calls. They perform surprisingly well when compared to heavy-weight compilers or interpreters used in systems like SPIN and μChoices [3].

†As the implementor may fall into the temptation of using some internal feature of the service.

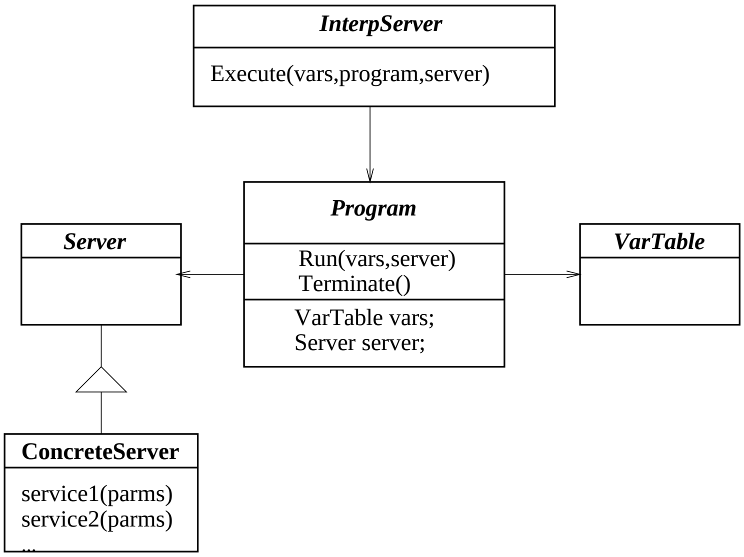

Figure 1. Main participants in the COMPOSITECALL *pattern.*

In the case of bulk data transfer operations, a very large amount of data copying can be avoided by using COMPOSITECALLS. 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 COMPOSITECALLS, 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 I/O, 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 COMPOSITECALLS 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 kinds of operating system environments, Unix and *Off*++ [4, 5]. In *Off*++, we apply the pattern to provide support for disconnected operations, gather/scatter I/O, and heterogeneous resource allocation; services not provided as primitive operations.

THE COMPOSITECALL PATTERN

As shown in Figure 1, the COMPOSITECALL pattern combines a simple control command language, the `Program` class in the figure, with an existing server, `Server` in the figure. This figure and the following ones follow the OMT notation [6] variant used in reference [7].

The goal of COMPOSITECALLS is to enable users to send simple groups of calls or programs to the server. One can avoid sending separate single calls in many cases.

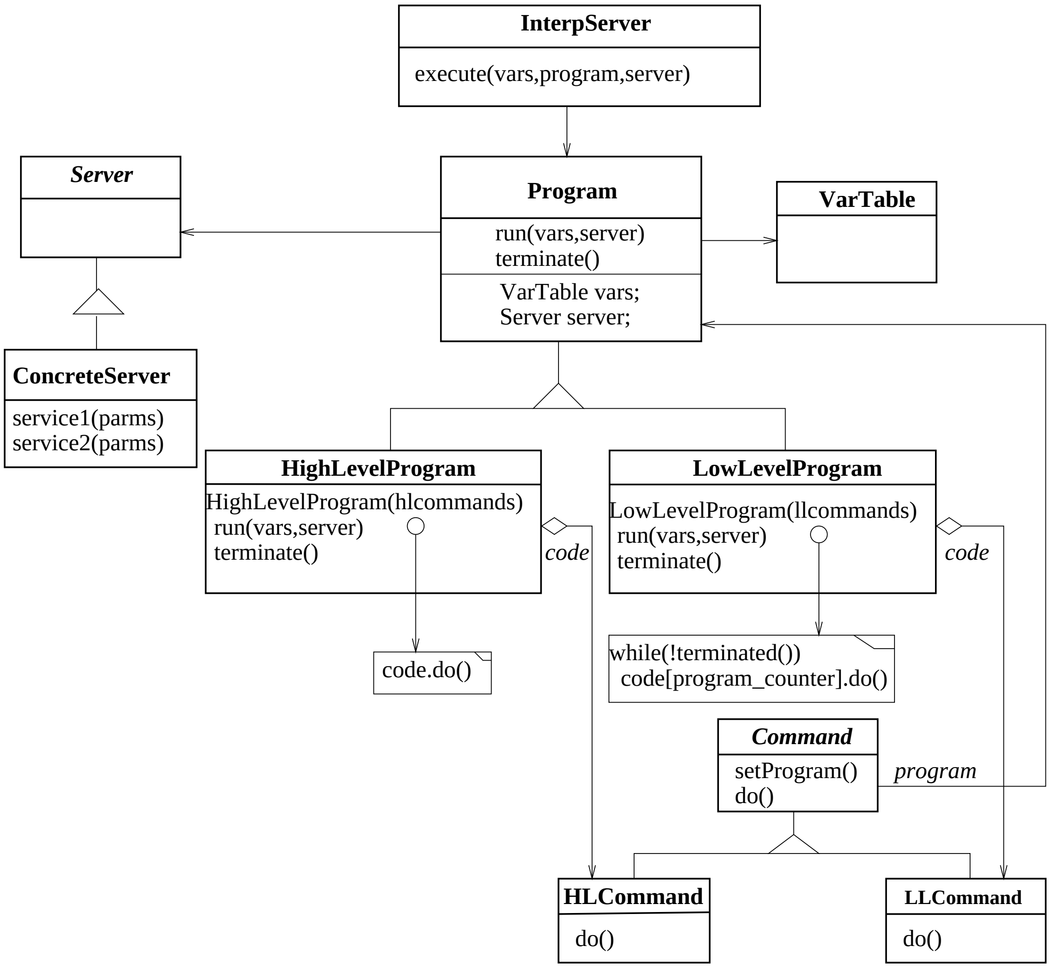

Figure 2. COMPOSITECALLS*: The whole picture.*

In fact, COMPOSITECALLS shifts the programming model from a "protected library" that provides several entry points to an "interpreter" that executes client programs to service requests.

Clients compose primitive calls to build a COMPOSITECALL, also known as a `Program`. Then they send the program 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.

To enable the use of a single `InterpServer` with different servers, we pass a reference to an abstract `Server` to `execute` whenever a program is executed. `ConcreteServer`s 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 `Program`s are made of `Command`s. The set of `Command`s accepted by a `Program` can be divided into:

Prepared using speauth.cls

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

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

Depending upon the chosen control command language, different interpreters can be used. In particular, we design and implemented both a high-level command language, associated with the `HighLevelProgram` class, and a low-level byte-code based language, associated with a `LowLevelProgram` class. The high-level program is interpreted recursively by means of the program syntax tree, and the low-level program is interpreted iteratively by means of a byte-code array. We chose these languages because typical interpreted languages fall into one of these two categories; even compiled languages can be considered to be iteratively interpreted by the hardware processor. Therefore, by contemplating both languages, the pattern shows how to integrate any language the user may choose.

Our goal is to let users write high-level programs – like the one shown in Figure 3 – and compile them to generate low-level programs that 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 can already produce significant performance improvements. Note that, as discussed later, there might be more reasons than just latency to use COMPOSITECALLS.

As shown in Figure 3, constructors for concrete classes representing "control structures" and "server call" commands allow for a convenient syntax. By invoking these constructors, programmers build syntax trees representing program structures. After users build these "high-level" programs, they are serialized and sent to the server, where they are deserialized for interpretation. Alternatively, clients can compile them into "low-level" programs before sending them, 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.

The main method of `Program`, `run`, triggers program execution by calling the `do` method of the proper `Command`. For example, in Figure 3, the cross-domain call `execute` calls `program.run`, in the server domain; afterwards, `program.run` calls `Sequence::do` in the `Sequence` instance.

A single storage area, named `VarTable`, is required to run a program. `execute` receives the storage area as a parameter. Some entries in `VarTable` act as input and/or output arguments for the program, others behave as local temporary variables. We return the storage area back to the user upon program completion.

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, *Multithreaded-Rendez-vous* [8], or any other IPC mechanism.

Related patterns

The `Program` class is an interpreter for programs made of `Commands`, while `Program` instances are the programs themselves. The INTERPRETER pattern [7] is useful to implement the desired command language. In turn, `Commands` are usually COMPOSITES [7], so high-level constructs like loops, conditionals, etc. can be expressed cleanly.

The VISITOR [7] 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 MUTITHREADED-RENDEZVOUS pattern [8].

The ACTIVEOBJECTS [9] 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.

USING COMPOSITECALLS

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

Should CompositeCall be used? Using COMPOSITECALL is worthwhile when it issues enough calls. 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.

COMPOSITECALLs 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 object can be used as an *indirect call* [10], as one can transmit the program to the server by different means.

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

Existing services need no changes to support CompositeCalls. 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. The user gains no access other than that granted by existing system services.

Verifying the program for safety is a very simple operation. The process consists on ensuring that the program includes only valid commands. The simpler the command language, the simpler the program verification. In the extreme case, when the command language is made just of call commands, it suffices to ensure that called entry points exist. We found that the complexity of the interpreter influences CompositeCall performance heavily—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.

One cause of security problems is pointer handling. In this respect, it is not enough that every system call verifies its arguments. We must ensure that, after verifying the interpreted program, the CompositeCall interpreter takes care of any additional pointer dereference performed during interpretation. To make it simple, we chose to avoid pointers within our implementations of CompositeCall, and provide a generic "move" instruction instead. Thus, once the program is verified, there is no security risk regarding pointer handling.

Error handling and recovery. When users call system services directly, they are 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 CompositeCalls shows that users typically build programs assuming that either

- every call succeeds, and no error condition is checked by any command in the program; or
- 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. In this case, call `Command`s perform error checking that abort the execution when an error occurs; the user does not need to insert more commands to check error conditions.

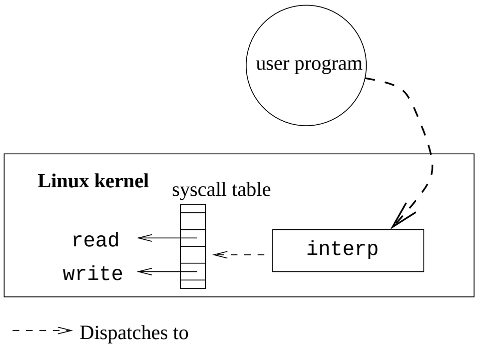

Figure 4. The COMPOSITECALL *instance for Linux system calls*

In the second case, the interpreter ignores error conditions. The user builds the `Program` including some commands that test error conditions in the `Commands` following every call `Command` that might fail.

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. For example, our interpreters include `AbortOnError` and `DoNotAbortOnError` commands.

Side effects may behave differently with clients issuing cross-domain calls and clients using COMPOSITECALLS. With COMPOSITECALLS, server calls are issued *within* the server domain, not from the client domain. For example, 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 fact should be taken into account when implementing an instance of COMPOSITECALLS.

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

In modern and cleanly designed operating systems this should not be a problem. In other cases including some instances of UNIX and Windows that is certainly an issue, as shown later in a section devoted to side effects.

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 skeletons from the network to the server, the side effect may not be triggered as they were when using simple calls. Server implementations assuming that side effects are honored frequently or between any two successive server calls may behave badly with COMPOSITECALLS. We show later how we addressed this problem in our experimental implementations.

APPLYING CompositeCall on Linux

We instantiated the COMPOSITECALL pattern using the Linux kernel as the `Server`. The interpreter was written in C. Initially we considered Java as an alternative—

μChoices uses Java to allow safe code downloading into the kernel. However, we felt that Java was too complex and the Java Virtual Machine too big for the simple purpose of supporting CompositeCall on Linux and decided to try something simpler. As shown later, it turned out that the decision was right: our interpreter has only 358 lines[‡] of C code, including the header file. The interpreter itself has only 235 lines. Nevertheless, it performs better for our purposes than Java, as shown later.

Even though the interpreter was written in C, its implementation matches the design pattern described here. Therefore, all elements found in the pattern, as shown before, can be found in this instance. Figure 4 shows an schematic picture.

An instance of the CompositeCall `InterpServer` was added to the kernel as a new system call named `interp`.

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

The `interp` system call receives the program `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 `INC`, which operate on two entries of the `vars` variable table.
- Comparison and branch commands, which 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 data 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 can be either preset in the variable array when the user calls `interp`, or can be set at program, `prog`, run time by move or arithmetic instructions. Of course, an input value for a system call can come from an output value of a previous call.

Implementation for Linux

We linked the code for the new `interp` system call statically to the Linux kernel, although we could have used a loadable module instead.

As all arguments for existing Linux system calls fit into `long` integers, we 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[§]. Each concrete `LinuxCall` command contains:

- The system call ID number, also implicit in the command type.

[‡]Measured with the `wc` tool on Linux.
[§]That is indeed the way Linux and most of other OSes implement their system calls.

- 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 fields we can dispatch to the proper 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 be the first impression, when looking at the pattern, that additional argument data copying is needed. That is not the case. Note that, in calls accepting user supplied buffers like `read` and `write`, the 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 that is still handled by `read` as if it were called by the user.

Side effects on Linux

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 honored at the end of program execution.

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` 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 that is set by the kernel whenever the processor quantum expires. This *needs-reschedule* flag must be checked after each system call. The interpreter must also check it periodically, even when no system call is 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.

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 and is checked later, when system calls returns. If a signal is sent to a process executing `interp`, it is not 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. The code operates on the process stack and behaves in different ways depending on the caller.

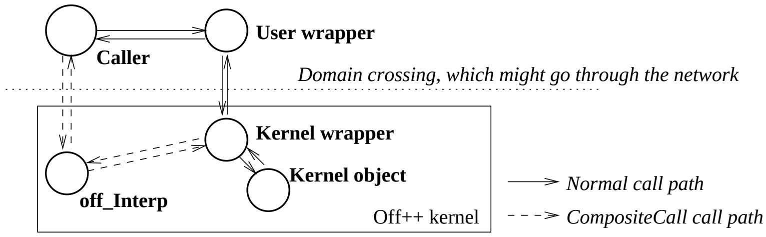

Figure 5. Normal system call path and COMPOSITECALL *call path in Off++.*

Although one could expect 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.

It is possible to provide mechanisms to resume the program from the state where it stopped when the signal was delivered. 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 [7]. The program can then be adjusted and re-downloaded to complete its execution. Alternatively, it could be cached within the kernel to avoid repeated downloading.

Applying CompositeCall on Off++

To experiment with COMPOSITECALL on a system different from Unix, we applied COMPOSITECALL on *Off*++, our research OS.

Off++ [4, 5] is a distributed object oriented μkernel used by the *2K* [11] operating system. In *Off*++, calls to system objects proceed through remote method invocation, or RMI, into the kernel domain. RMI employs user and kernel wrappers, as shown in Figure 5, and it might cross the network as *Off*++ is a distributed μkernel. The user wrapper is a proxy that delivers messages to the kernel domain; the kernel wrapper verifies user arguments and performs access checks.

Note that by "user", we mean any code running in non-privileged mode; i.e. any non-kernel code. Thus, we consider as a kernel user most of the actual OS code, which runs in user-space.

The services *Off*++ provides are mainly allocation and deallocation of distributed physical resources like page frames, address translations, processor slots, etc. Therefore, it is common for users to issue several calls at a time. For example, the user code for virtual memory allocates a page frame, allocates an address translation, and sets up the translation so that it points to the allocated page frame.

We implemented the COMPOSITECALL pattern in *Off*++ using using *C++*. In this case, we developed 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. For example, the kth item starts 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 that 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 calling the `OffCall` `do` method within the kernel, the 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. Output values are incrementally stored into the output message, and returned to the caller.

As it happens with the Linux instance, in-kernel wrappers perform access checks within the kernel.

Implementation for Off++

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 diagram shown for the pattern in figure 1. Thus, there are no further implementation issues to be discussed.

Using CompositeCall on distributed systems

The CompositeCall does not provide new system services. It only exports existing system services in a different way. Therefore, to use CompositeCall on a distributed environment, distributed system services must already exist.

We learned this lesson well when we implemented a version of the Linux CompositeCall for use in a networked Linux environment. Linux does not allow to issue system calls from remote nodes in the network. Therefore, we had to implement a new user-level server on Linux to perform the experiments with distributed CompositeCalls shown later. The new user-level server provides remote access to local system services.

On heterogeneous environments, data exchange requires data conversion. Data is converted to a common "network" format when it is transmitted. Note that on a distributed environment the client and server stubs used to access system services handle data conversion.

To use CompositeCall, the only requirement is that the server must receive the data in the same format no matter whether it is a regular service call, or a call from within a CompositeCall. We see three ways for meeting this requirement:

```
// 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  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 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.*

- Use a high-level language in the COMPOSITECALL, and add code to its `compile` method to translate data to the common format.
- Reuse code from the system stubs, and call them explicitly to translate data to the common format.
- Use a single format everywhere and convert data only when the local architecture is different.

In the first two ways, the client translates the data to a network format, like on regular single calls to the server, and the COMPOSITECALL interpreter performs calls to the server stubs. When the interpreter calls the server stub, it converts the data to the native server format. In the third way, the client builds the data in the server format, and the COMPOSITECALL interpreter calls the server entry point directly.

Apparently, another problem for COMPOSITECALL on distributed environments is how to handle side effects and exceptions. But, this problem is not a real one. Both side-effects and exceptions are handled within the server's node. Therefore, the same techniques shown on a centralized setting can be applied on a distributed environment. Side effects are handled within the interpreter, as in a centralized environment. The interpreter also catches exceptions, as in a centralized environment, but they deserve further explanation.

The COMPOSITECALL interpreter can abort the program when an exception occurs and return its state to the user. The user can resume program execution later, as we said during the discussion of error handling and recovery. However, on a distributed environment the program must be resumed in the same node where it was executing before the exception. The reason is that the program might have acquired resources on that node during its execution.

For example, in our implementation of a remote COMPOSITECALL interpreter for Linux a user can download a program to open, read, write, and close files. Before an exception occurs, a program might have acquired new file descriptors, which should remain open if the program is resumed.

In our centralized version of the interpreter for Linux, one executes and resumes the program by calling the same entry point. The reason is that the program always executes within the client context and any acquired resource is still available when the program is resumed.

In the distributed version for Linux, we can execute every COMPOSITECALL on a different process. When an exception occurs we only need to keep that process alive

Prepared using speauth.cls

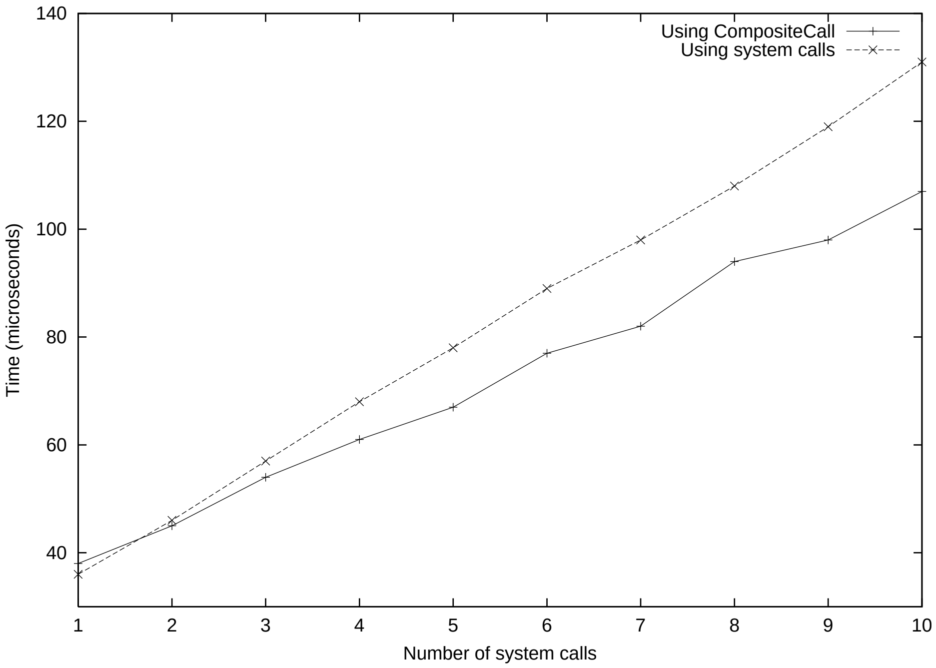

Figure 7. COMPOSITECALL-*based copy vs. traditional copy on Linux.*

until the user either resumes the program or aborts it. If the program is resumed, it executes within its old process so that acquired resources are still there. If the program is aborted we can kill the process.

At the present moment, our distributed implementation for Linux does not allow to resume or abort the program, but the implementation would be straightforward and can be implemented as described in the previous paragraph.

EXPERIMENTAL RESULTS

To experiment with COMPOSITECALLS on Linux, we implemented a copy program both with and without COMPOSITECALLS. The program copies its input to its output. The COMPOSITECALL copy program uses a byte-code interpreter.

This program is a model for many common utilities including `cp`, `tar`, `dump`, and `dd`.

CompositeCall performance on Linux

We measured performance using a traditional `copy` program and a modified one, `icopy`, using the interpreted low-level 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 outweighted 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, if 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 the overhead due to program copying and leads to the figures¶ shown in Figure 7. Cached CompositeCall-based programs can run faster than their traditional counterparts, even when the CompositeCall issues *only two* system calls.

In our experiment, for non-cached CompositeCall-based programs, 16 μseconds should be added to the execution times shown in 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 CompositeCalls could be even more useful in Linux environments. Nevertheless, even our simplistic interpreter was able to achieve 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 obtained with the CompositeCall mechanism should be even higher. Finally, the experiment shown in this section does not show the case where CompositeCall avoids sending data through the network as in the NFS `copy` example from the introduction section. The next experiment shows the performance implications of avoiding unnecessary data transfers.

CompositeCall performance on a distributed system

The centralized system of the experiment shown above is the worst-case for us. Using CompositeCall pays more when running on a distributed environment. To show it experimentally, we measured a distributed CompositeCall for our copy example and compared it with NFS.

Our experiment measures the time needed to execute the `copy` program to copy a file from a server to a different file within the same server. When using NFS, data goes through the network twice; when using a CompositeCall, data does not leave the server machine.

Even if the destination of the copy is different server, NFS would still perform two data transfers, and the CompositeCall would perform a single data transfer from the data source to the data sink. We did not measure this scenario because we

¶Figures shown correspond to the average of 10,000 experiments on a 100MHz Pentium-based Toshiba 110CS.

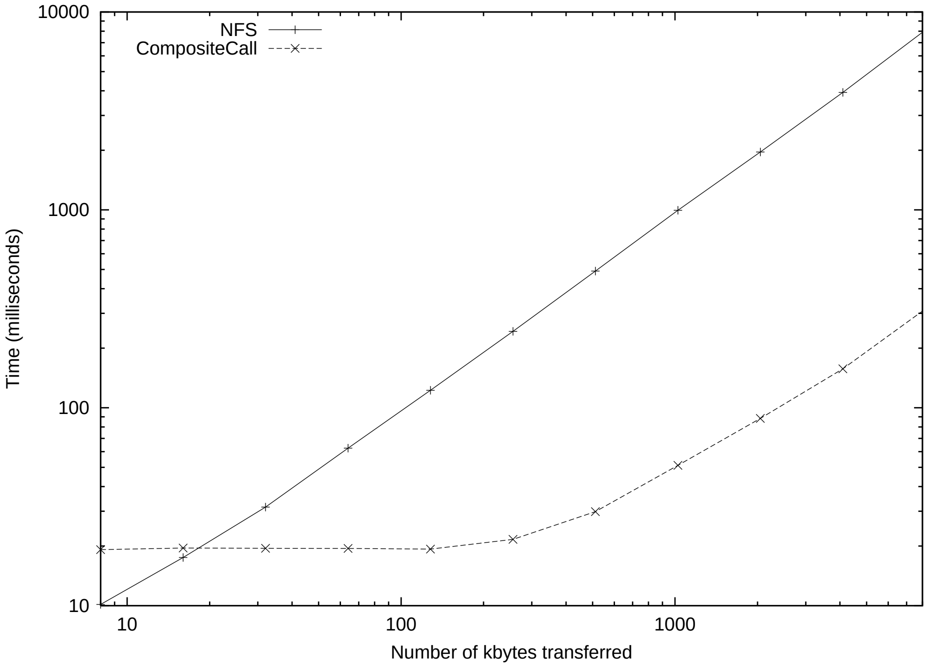

Figure 8. COMPOSITECALL *based copy vs. NFS based copy.*

believe that it suffices to show the previous one—provided that we have described the performance improvement even on a centralized system. Besides, after instrumenting the copy program, `cp`, on our Linux environment we saw that most of the calls during an entire business day are copying programs within the same home directory, i.e., within the same file server.

The experiment was performed on an idle network. We transferred files of different sizes, from 8 Kbytes to 8 Mbytes. For NFS, we used the `cp` utility distributed with Linux. For COMPOSITECALL, we used the program shown in Figure 6 with four extra instructions, two for opening the files, and two for closing them.

Because Unix does not provide remote system calls, we had to implement a user-level server for Linux that accepts a COMPOSITECALL on a socket and executes it within user-level. Although it does not authenticate the remote user, the same scheme used on NFS can be applied. To be fair, authentication was switched off in NFS during the experiment.

We must say that the experiment favors NFS, because the server's kernel replies to the NFS requests promptly. The implementation of the COMPOSITECALL server at user-level, which reads from a socket, and writes the reply back, needs more user-kernel domain crossings and incurs on extra latency due to the socket servicing code in the kernel.

Although we could have used an in-kernel COMPOSITECALL interpreter as shown in the previous section, we did not. In that way, our measures are more pessimistic regarding the speedup gained by using COMPOSITECALL.

Figure 8 shows the results of the experiment. It shows the time in milliseconds as a

```
// C version (SPIN-like)
// s_{read,write} are system
// call entry points.
int interp(int i, int j, int k)
{
        while(s_read(i,j,k))
                s_write(i,j,k);
}
```

```
// Java version
// sc.{read,write} are system
// call entry points.
class Cc {
     public void execute(Sc sc) {
          while (sc.read(i,j,k))
                 sc.write(i,j,k);
     }
}
```

Figure 9. Copy programs for our C version of a SPIN extension and for Java.

function of the data transfer size using a logarithmic scale.

The CompositeCall has about 19 ms of fixed overhead needed to receive the request through a socket and send the reply back. Until a transfer size of about 200K, the time for the CompositeCall increases very slowly with the transfer size going up to 19.5 ms; the increment in time is almost negligible as shown in the figure. Above 200K, time increases linearly. The reason is that data is copied locally and it can be performed really fast until the point in which the server itself gets saturated. The time for NFS increases linearly for any transfer size.

The overhead of the user-level interpreter does not pay for transfers below 16K. The reason is that our NFS transfers data in blocks of 16K; when the transfer is of only one block, the extra time required by our user-level implementation outweights the time saved within the data transfer. For just one byte, our server needs 19 ms to receive, process, and reply to a CompositeCall request. A kernel-level implementation could perform better. Figure 8 shows that for 8K, CompositeCall takes 189% of the time taken by NFS. However, for 32 K, CompositeCall takes 61.9% of the time; for 256K, it takes 9%; and for 8M it takes just 4%. The savings in time due to avoiding unnecessary data transfer can be impressive.

CompositeCall compared with other approaches

We measured our CompositeCall and compared it with the JDK Java Virtual Machine, or JVM, with the Kaffe JVM with Just-In-Time compilation, or JIT, and with the SPIN μKernel approach. Note that by comparing it with Java, we also compare it with μChoices, which uses Java as it extension mechanism.

To perform the comparison, we took the copy program of Figure 6, used as our running example, and implemented two new copy programs, one for Java, and another one for SPIN. Measures do not include the time needed to compile and download the program in the kernel. Besides, we replaced the real system calls read and write with null calls. In this way, we can compare the run-time performance of the different extensions independently of the different compilation and downloading techniques.

Figure 9 shows the Java and the SPIN versions. The `i`, `j`, and `k` arguments are passed to `read` and `write` to measure the overhead of performing a system call with three arguments. That is to say that the program in its three versions, CompositeCall, Java, and SPIN is exactly the same.

It is important to note that the SPIN version is not a real SPIN program. In SPIN,

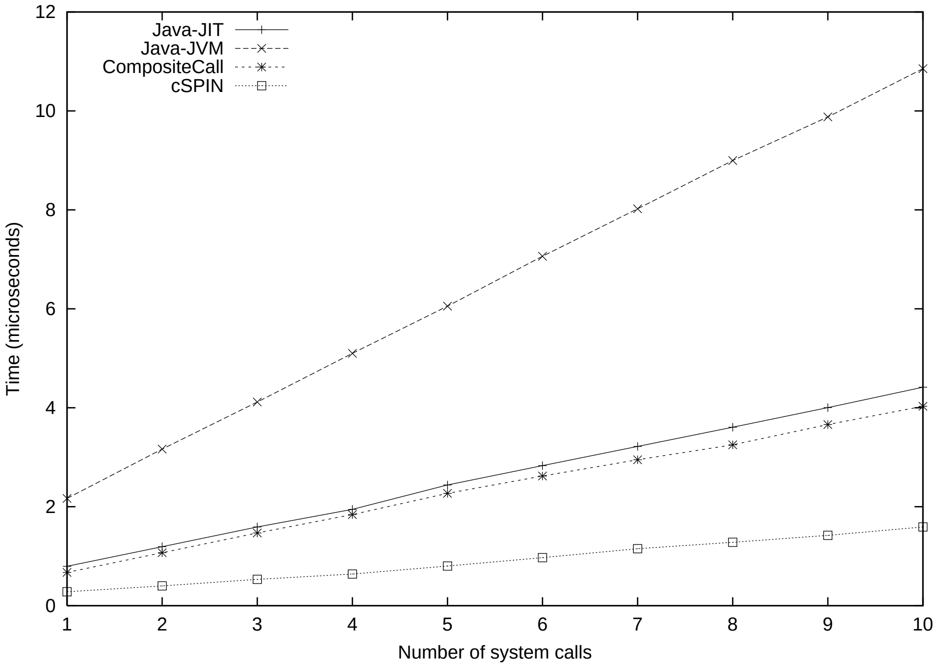

Figure 10. Performance in μseconds of the copy program with COMPOSITECALL, *interpreted Java, Java compiled with JIT, and the C version of SPIN.*

extensions and the kernel itself are written in Modula 3. The reason is that it is crucial in the SPIN design that the compiler can do type checks between the extensions and the existing system modules.

However, Linux is written in C and we do not have Modula 3 interfaces for Linux. Taking into account that the SPIN extension runs on the hardware because it is compiled code, and there is no interpreter, we measured a compiled C program instead of a Modula 3 extension. It is clear that the C program measured would not run slower than the equivalent Modula 3 SPIN extension. Thus, we are measuring a lower bound for the execution of a SPIN extension, which is the worst case for us.

Figure 10 shows the execution time in μseconds for these three different variants of the copy program. The Java program has been measured both with and without JIT compilation. All the experiments are measured on a Pentium Celeron at 366MHz with 128Kb cache installed.

As it could be expected, the native code—i.e. our SPIN-like extension—is the fastest, because the native processor executes it. Our interpreter performs better than Java code. Moreover, compare the size of our interpreter consisting of 1.5 Kbytes of code and data, and 235 lines of source C code with the size of a typical Java Virtual Machine which means 1.1 Mbytes of code and data, and several MBytes of source code.

Besides, our interpreter outperforms the Java version even when the Java bytecode is compiled to native code with JIT compilation.

For most occasions, our interpreter is both simple and fast enough. Note that simpler also means more secure: the larger the interpreter, the more likely are security flaws and bugs in the implementation.

Compiling extensions

For high-level extensions, we can compile the code into a low-level language, byte-code in our implementation. The execution time for the high-level extension would be the same of a low-level one once it has been compiled. Of course, the compilation time must be accounted as overhead when compilation is done on-line.

We implemented a `compile` method for the copy program shown in Figure 3. Its execution takes 3.47μseconds on the platform used for the experiment. For the versions of the same program written in Java and C that we used to compare CompositeCall to other approaches, shown in Figure 9, the Java compiler `javac` took 0.8 seconds to compile and our C compiler took 0.3 seconds. Thus, a CompositeCall can be compiled on-line with less overhead than an extension for μChoices, i.e. Java, or SPIN. Note that we measured a C compiler for SPIN because we have been using C instead of Modula 3 for the reasons mentioned above. Nevertheless, a Modula 3 compiler does far more checks than the C compiler we have used, so we are measuring a lower-bound for the extension compilation time in SPIN.

The difference in performance between compilation of a CompositeCall and compilation of extensions in SPIN or μChoices can be explained by considering that the language in CompositeCall can be much simpler, and uses only local memory within the user program. This is yet another example where a domain-specific language like our CompositeCall can be better than a general purpose language like Java or C.

In the measurements shown in this article, we use a byte-code interpreter and not a high-level interpreter. The byte-code programs used for CompositeCall were written by hand, with no help from a high-level program compiler. To compare the different approaches using high-level programs and on-line compilation, the compilation times shown above should been added to the times shown in the figures of the article.

Even when the compilation time is significant compared to the execution time, it can be avoided by either compiling off-line or reusing the same compiled program several times. A `tar` utility, for example, can compile a CompositeCall to copy a single file, and reuse it for every file it copies.

Taking into account that the time to compile a CompositeCall is rather small, it still pays to use on-line compilation in many cases. For example, in the `Program` used above to compare CompositeCall to NFS, it makes no difference to add 3.47 μseconds of compilation time to the 19000 μseconds of fixed overhead due to socket communication with the user-level interpreter.

CompositeCall performance on Off++

We measured performance on *Off*++ by implementing several services with both primitive kernel calls and with CompositeCalls. The chosen services were a user-level page fault handler and a page frame allocator. In our experiments, we did not choose the same example program used on Linux, because our μkernel does not include a file system, and the most common operations are physical resource allocation and exception handling.

We describe, here, just the page fault handler shown in Figure 11. For the sake of simplicity, we omitted some few additional parameters and declarations. The page fault handler allocates a page frame and installs a translation to it. This routine can

```
// Handling a page fault on Off++ using primitive kernel 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 11. 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 in off_Interp::execute() calls.
```

Figure 12. Handling page faults with COMPOSITECALL *in Off++*

be transformed into another call to `off_Interp.execute`, passing to it the program shown in Figure 12, which performs the same task done by the one in Figure 11. Figure 12 also shows one way to build the program with COMPOSITECALLs. 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 I 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 kernel calls increases. Results can be seen in Figure 13. The large amount of time spent on executing a kernel call is due to the expensive set of debugging checks performed by the version of the kernel we employed. We did not remove these checks to obtain performance numbers for COMPOSITECALL on a kernel with expensive system calls.

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

Test	$\frac{time}{\text{time of single call test}}$
Using single calls	1
Using CompositeCall	0.68

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 causing most of the problems in the Linux implementation. All interactions with preemption and signal delivering appeared because the Linux 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-preemptiveness of the Linux kernel, apart from degrading performance on multiprocessor systems, makes it infeasible to write system calls that can compute for an indeterminate amount of time. A workaround is to call the scheduler from the CompositeCall interpreter.

Instead of delivering signals asynchronously with respect to the signalled process execution, the kernel is supposed to check for posted signals at particular places within the signalled process context; in particular, on return from a system call. This implementation of signal delivering, which is not really asynchronous, makes infeasible to write system calls that can compute for an indeterminate amount of time. A workaround is to check for pending signals from the CompositeCall interpreter.

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 CompositeCalls 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 to 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 ActiveObjects pattern [9] 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 that 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 Composite-Calls, 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, enters a loop sending a given buffer, and then closes the connection. 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 to keep separated. 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 that indeed could be implemented by using CompositeCalls. Even though we have experience in the field, we never imagined that a single piece of code could replace separate functions like gather/scatter I/O 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 software systems.

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

Simplicity matters. Although it is well-known, we would like to emphasize that simplicity is an important issue. It is the simplicity of our CompositeCall in Linux that allowed us to outperform the JIT version of Java. Although it does not have objects, method calls, threads, etc., our interpreter still allows many extensions. It provides a general purpose "abstract machine", albeit a quite simple one.

RELATED WORK AND OTHER PATTERN INSTANCES

Our implementation of CompositeCalls, which entails a `Program` and a variable array, is similar to the concept of *closure* [12]. In programming languages like *Scheme*, a closure is a structure containing a lambda expression equivalent to our `Program` and an environment 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 [13] utilize CompositeCall. A Stored Procedure can be thought of as a small data access program to be used for retrieving information from the data store.

Operating systems like SPIN [2], μChoices [3], and VINO [14, 15] use code downloading. They include that 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. They use a general-purpose language for programming system extensions. Conversely, 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 and therefore safer. Users cannot cause damage to sensitive kernel or server data. Therefore, it is

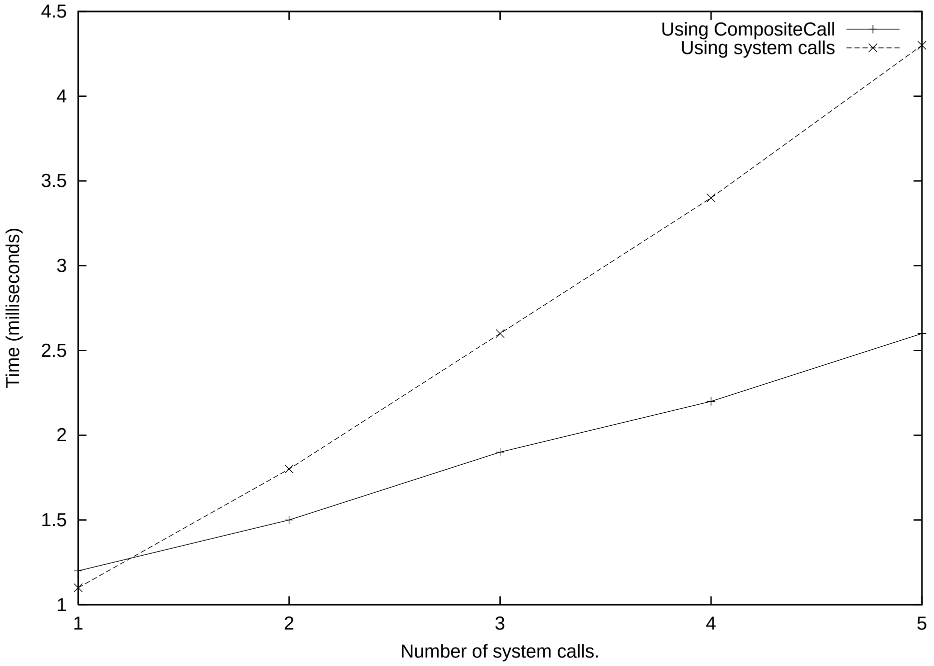

Figure 13. Performance of page frame allocation using CompositeCall *in Off++.*

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 CompositeCalls by observing that:

- Code downloading in these systems may be considered as concrete instances of the CompositeCalls pattern where the program can be expressed in Modula 3, Java, or other general-purpose language.
- The instances of CompositeCalls described in this article—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 358 lines of code, and uses less than 2 Kbytes of memory. Compare that with the complexity and size of the Sun JVM [16].
- The CompositeCall pattern can be applied to systems not designed with CompositeCalls in mind, as we demonstrated for both Linux and *Off++*. No change was necessary to these systems. That was not the case of systems like SPIN, which were designed with code downloading in mind. Our approach requires neither ad-hoc mechanisms, nor specific compilers, nor any special kernel support to include CompositeCalls. Apart, of course, of the added code for the CompositeCall interpreter.

The idea behind Agent systems [17] is closely related to CompositeCall. However, the aim of Agent systems is to build mobile stand-alone programs. In a CompositeCall, the program remains in the server domain until termination; it does not move to a different domain. We put the emphasis only on the interface shift from a single entry point to a CompositeCall; we leave apart other unrelated technologies.

Nevertheless, some of the machinery needed for implementing Agents [18, 19] 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. This also applies to systems borrowing techniques from the field of mobile Agents, like NetPebbles [20]. 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 [22] and Bayou [23] that defer changes while the system is disconnected. Servers aggregate pending changes that are processed when the system is reconnected. As dictated by the CompositeCall pattern, primitive calls like a single change or update are composed and processed later. Most notably, Bayou [23] operations are actually programs that can detect and resolve conflicts.

Finally, lessons learned in the design of domain-specific languages for applications like user interface specification, software development process support, and text processing [24] can be applied to design adequate languages for concrete CompositeCall instances.

CONCLUSIONS AND FUTURE WORK

We have identified the CompositeCall pattern and discussed how it is instantiated in several existing systems. We have developed two new instances of the pattern on a traditional, monolithic kernel and on an object-oriented, research μkernel. No change was needed in these systems, even though they were not designed with CompositeCalls in mind.

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 where 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 CompositeCalls. We also plan to develop applications using CompositeCalls as the main abstraction for client/server interaction.

ACKNOWLEDGEMENTS

We are grateful to Gorka Guardiola Muzquiz for his help in the implementation of the Linux CompositeCall mechanism. We are also grateful to the anonymous reviewers for their useful comments, which helped to improve the quality of this article greatly.

REFERENCES

1. Marta Patiño, Francisco Ballesteros, Ricardo Jimenez, Sergio Arevalo, Fabio Kon, and Roy Campbell, 'Batching: A Design Pattern For Flexible And Efficient Client-Server Interaction', *Proceedings of the Conf. on Pattern Languages of Programs (PLoP'99)*, Monticello, IL, USA, August 1999, pp. 7:1–18.
2. 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.', *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles.* ACM, December 1995.
3. Y. Li, S. M. Tan, M. Sefika, R. H. Campbell, and W. S. Liao, 'Dynamic Customization in the μChoices Operating System.', *Proceedings of Reflection'96*, San Francisco, April 1996. Reflection'96.
4. 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.
5. *Off++* web site. `http://gsyc.escet.urjc.es/off`.
6. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns. Elements of Object-Oriented Software*, Addison-Wesley, 1995.
8. R. Jiménez-Peris, M. Patiño Martínez, and S. Arévalo, 'Multithreaded Rendezvous: A Design Pattern for Distributed Rendezvous', *In Proceedings of the ACM Symposium on Applied Computing, SAC'99*, San Antonio (USA), Feb. 1999. ACM Press.
9. R. Greg Lavender and Douglas C. Schmidt, 'Active Object – An Object Behavioral Pattern for Concurrent Programming', *Proceedings of the Second Pattern Languages of Programs conference (PLoP)*, Monticello, Illinois, September 1995.
10. Carlos Baquero. Indirect Calls: Remote invocations on loosely coupled systems. http://gsd.di.uminho.pt/People/cbm/public/ps/icalls.ps, 1996.
11. 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', *Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.
12. Samuel N. Kamin, *Programming Languages*, Addison-Wesley Publishing Company, 1990.
13. A. Eisenberg, 'New standard for stored procedures in SQL', *SIGMOD Record (ACM Special Interest Group on Management of Data)*, **25**(4), (1996).
14. 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.
15. Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith, 'Dealing with disaster: Surviving misbehaved kernel extensions', *Proc. of the Second Symposium on Operating Systems Design and Implementation*, Seattle, WA, October 1996, pp. 213–227. USENIX Assoc.
16. Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Addison-Weley, 1996. Java Series.
17. Joseph Kiniry and Daniel Zimmerman, 'A Hands-On Look at Java Mobile Agents', *IEEE Internet Computing*, **1**(4) (1997).
18. Jim White. Mobile Agents. General Magic Corporation, 1996.
19. Dag Johansen, Robbert van Renesse, and Fred B. Schneider, 'Operating System Support for Mobile Agents', *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, Wa (USA), May 1995. IEEE.
20. Ajay Mohindra, Apratim Purakayastha, Deborra Zukowski, , and Murthy Devarakonda, 'Programming Network Components Using NetPebbles: An Early Report', *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, Santa Fe, New Mexico, April 1998. USENIX.
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) (1997).
22. 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.

23. 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', *Proceedings of the Tenth ACM Symposium on User Interface Software and Technology (UIST)*, Banff, Alberta, Canada, October 1997.
24. Diomidis Spinellis and V. Guruprasad, 'Lightweight Languages as Software Engineering Tools', *;login: DSL'97 Conference Summaries*, volume 23, Santa Barbara, CA, February 1998.