Building reconfigurable component-based OS with THINK

Juraj Polakovic
FranceTelecom R&D
MAPS/AMS Lab, Grenoble

Ali Erdem Özcan
STMicroelectronics
AST Lab, Grenoble
firstname.name@inrialpes.fr

Jean-Bernard Stefani
SARDES Project, INRIA
INRIA Rhône Alpes

Abstract

Dynamic reconfiguration allows modifying a system during its execution, and can be used to apply patches and updates, to implement adaptive systems, dynamic instrumentation, or to support third-party modules. Dynamic reconfiguration is important in embedded systems, where one does not necessarily have the luxury to stop a running system. While several proposals have been presented in the litterature supporting dynamic reconfiguration in operating system kernels, these proposals in general hardwire a fixed reconfiguration mechanism, which may be far from optimal in certain configurations.

In this paper, we present a software-architecture-based approach to the construction of operating systems, and we show that it allows us 1/ to support different mechanisms for dynamic reconfiguration, and 2/ to select between them at build time, with little or no changes in operating system and application components. Our approach relies on the use of a reflective component model and of its associated architecture description language.

1 Introduction

Dynamic reconfiguration, i.e. the ability to alter a system during its execution, is an important requirement for modern operating systems. As has been argued in [19], dynamic reconfiguration can be used for a number of purposes:

- Applying patches and updates, avoiding a trade-off between availability and correctness.
- Implementing adaptive algorithms, where different algorithms, matching different operating conditions, can be hot-swapped to provide adaptation when conditions change.
- Supporting dynamic monitoring, where instrumentation can be put in place dynamically for the duration of the observation, and removed when no longer necessary.
- Supporting application-specific optimizations, where applications can optimize their performance by replacing a system component by a more specialized one.
- Supporting third-party modules, where system components can be dowloaded from the network and installed at run-time.

The above motivations for dynamic reconfiguration are especially valid in the case of operating systems for embedded devices, which must be adapted to a large variety of hardware and applications, under different resource or energy constraints.

The past fifteen years have seen numerous works dealing with extensibility and reconfigurability in operating system kernels, a number of them proposing mechanisms for dynamic reconfiguration. However, the proposed mechanisms are essentially hardwired in the proposed operating system. In a domain with a large variety of supporting hardware, and application requirements, such as embedded systems (spanning such diverse cases as telecom routers, mobile phones and PDAs, home entertainment appliances, etc), different dynamic reconfiguration mechanisms may be required for a better performance, depending in particular on the multi-threading structure of the target system (applications + operating system). With the current proposals for dynamic reconfiguration in operating systems, which we review in section 2, one cannot select the more efficient dynamic reconfiguration mechanism according to the target system.

In this paper, we show how this can be achieved at design time, with minimal effort, using a component-based approach to operating system construction. Specifically, we show how to support two known forms of dynamic reconfiguration, which we call *thread counting* and *dynamic interception*, and select between them, with little or no changes in operating system and application components, via a high-level architecture description of the target system. As part of the evaluation of our component-based design, we show indeed that, depending on the multi-threading structure of the target system, and on the granularity of the possible reconfigurations, the choice of the appropriate dynamic reconfiguration mechanism can have a substantial impact on performance.

The paper is organized as follows. Section 2 discusses related work. Section 3 discusses the support necessary for dynamic reconfiguration and presents our component-based framework for operating system construction, called THINK. Section 4 presents the implementation of the thread counting and dynamic interception mechanisms in THINK. Section 5 provides a quantitative and qualitative evaluation of our approach. Section 6 concludes the paper.

2 Related work

Dynamic reconfiguration has been heavily explored in research areas ranging from programming languages, down to middleware and operating system kernels. In the following, we restrict our analysis to operating systems.

Commercial operating systems, such as Linux or Windows provide limited support for dynamic reconfiguration, typically limited to certain functionalities, like device drivers. It is possible to load and unload kernel modules, however these ad-hoc mechanisms do not offer full dynamic reconfiguration support for the rest of the system.

Compared to monolithic operating systems, micro-kernels (L4 [13], Chorus [16], Kea [20] or Pebble [9]) are a step further in providing reconfigurability - a user-level server is the unit of reconfiguration. However, a micro-kernel itself is not reconfigurable and if reconfigurability is implemented at the level of user-level servers, the system pays the price of the IPC communication between these servers. An Exokernel approach [6] allows kernel developers to build systems on top of minimum hardware abstraction, however it is also up to the kernel developper to implement a reconfiguration support.

Extensible operating systems, such as SPIN [2], provide a way to extend the kernel without the cost of domain-crossing inherent to the micro-kernel design, however with several limitations. First, an underlying kernel core itself is not reconfigurable. Second, extensions are only limited to some prefefined parts of the kernel. Third, the interactions between extensions and the kernel are expensive - an extension is a handler reacting to an event raised by a kernel module.

Component-based OS, such as OSkit [8] or eCos [5], provide a way to build customized kernels, based on compile-time selection of components to be included into the kernel. These systems also provide an architecture description language (ADL), such as Knit [15], to assist the assembly of the kernel. However, these systems have no support for dynamic reconfiguration.

Reconfigurable operating systems include VINO [17, 18], Synthetix [14], MMLite [10] or more recently K42 [1, 19]. In VINO all reconfigurations are handled as transactions, largely using locking to synchronize the access to kernel modules. Synthetix and MMLite use read-write locks to synchronize accesses to a reconfigurable component (we call this solution thread counting). K42 supports reconfiguration through a mechanism, we call dynamic interceptors, which consist in introducing interceptors at runtime. Common to these systems is the fact that the provided dynamic reconfiguration mechanism is strongly tied to the design of the system.

Our approach can be contrasted with the approaches mentioned above along three axes:

First, we use fine-grained components and in contrast to other systems, including OSKit or eCos, our approach does not require any minimal core set of functionalities.

Second, we use an ADL and the associated compiler that allow us to construct operating systems by composition and to integrate non-functional elements as a reconfiguration mechanism for example. This allows the kernel developer to add a reconfiguration mechanism without impacting the functional part of the system.

Third, the above mentioned systems lack support for interface type verification, generic state transfer, component version management for reconfiguration or more complex reconfigurations involving replacing several components. The combination of the ADL and reflection capabilities of the OS enables us to tackle these issues.

3 Component-based approach for reconfiguration in THINK

3.1 Issues in dynamic reconfiguration

Reconfiguration of a software on-the-fly comprises of different steps. First, the part of the system to be reconfigured (for brevity we call it the *reconfiguration target*) must be clearly identified. Then, before the reconfiguration takes place, the reconfiguration target must reach a safe state. A common notion of safe state is that of quiescent state, i.e. a state in which no activity currently takes place in the reconfiguration target. When this safe state is detected, the state of the reconfiguration target must be captured and transfered to the new component and the change of configuration can now take place. The change of configuration can imply changing some attributes, modifying the connections between modules, as well as altering the architecture of the reconfiguration target. Before resuming the execution after the configuration change, the references to the old component must be redirected to the new one.

We discuss below, how a component-based approach can help to implement these requirements.

Boundaries of reconfiguration First of all, components are well defined encapsulation units that isolate state and code behind well-defined access points, i.e. interfaces.

Thus the unit of reconfiguration is a component. In a hierarchical component model we even have the possibility to tune the granularity of the reconfiguration, i.e. to proceed to coarse-grain or fine-grain changes in the system.

Detection of safe state In this paper, the notion of safe state that we consider is that of quiescent state. This notion of quiescent state is well adapted to multi-threaded operating system kernels. A component has reached a quiescent state in multi-threaded systems when no threads are executing inside a component.

State transfer Solutions for state transfer may be more or less complex. We consider that the state to transfer is the private data encapsulated by the old component. In a basic case, one can only copy this data to the new component. Components may also provide interfaces to give access to such kind of operations (e.g. *getter* and *setter* operations). This may not be sufficient if the internal data representations of the old and new components differ. In this case, state transfer must comprise a translation between old and new components' data representations.

Reference redirection A component-based approach implies programming against interfaces rather than implementations. In general, inter-component references are formalized as bindings (also called connectors) between a client interface and a server interface. This provides a solution to the localization of the references that should be updated.

In the rest of this section, we present our approach, the THINK framework which comprises a component model, an architecture description language with its compiler to C, and a library of operating system components.

3.2 Component model

We based our work on Fractal [3, 4], a hierarchical and reflective component model intended to implement, deploy and manage a wide range of software systems including operating systems and middleware.

A Fractal component is both a design and runtime entity that constitutes a unit of encapsulation, composition and configuration. Components provide *server interfaces* which are the access points to the services that they implement. They express their service requirements via *client interfaces*. Fractal distinguishes two kinds of components. *Primitive components* are implemented in a host programming language (e.g. C, Java) and can be seen as black boxes providing and requiring services through their interfaces. *Composite components* correspond to a composition of other components (called subcomponents), either primitives or composites. The existence of composite components makes the Fractal Component Model a hierarchical component model.

Components in the Fractal Component Model interact via client/server *bindings*. A binding constitutes a communication path between components and can implement arbitrary forms of communication (e.g asynchronous requests, synchronous requests/replies, multicasting and so forth). The simplest form of binding is a language reference (e.g. a method invocation).

A Fractal component logically comprises two different parts. The internal part, that we call *content* implements the functional interfaces of the component. The content is encapsulated by a *membrane* which can implement control over its behavior. In addition to the functional interfaces of a component, the *membrane* can provide an arbitrary set of control interfaces.

The presence of control interfaces makes Fractal a reflective component model where control interfaces provide the means to observe and manipulate the internal structure of a component. An important point is that the Fractal Component Model does not mandate a fixed and predefined meta-object protocol (i.e. a set of control interfaces). Instead, a programmer can define and implement his own set of control interfaces.

Fractal defines some standard control interfaces in order to manipulate a component's interfaces, its subcomponents, its client bindings, attributes, or its life-cycle (respectively called ComponentIdentity, ContentController, BindingController, LifeCycleController).

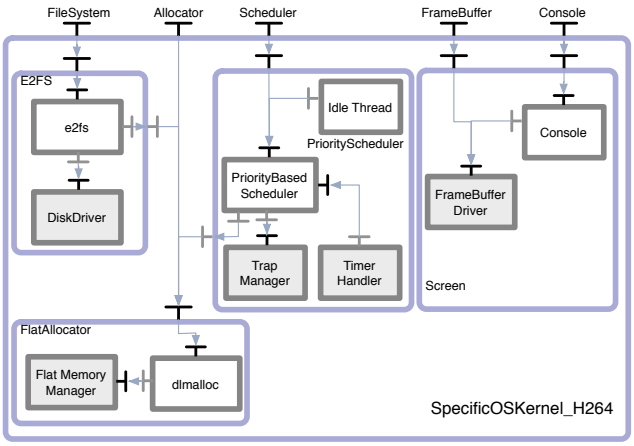
3.3 Architecture Description Language

The Fractal model is equipped with an architecture description language (ADL). The Fractal ADL is a high-level declarative language in which programmers express software configurations in terms of interfaces, attributes, component compositions and bindings. An example of an ADL description is given in the figure 1.

The major contribution of this paper is the ability of adding selective reconfiguration features into a kernel using ADL descriptions. This is done by specifying a specific *membrane* for reconfigurable components which implements the control interfaces that are needed for reconfiguration. This way, the reconfiguration aspect of a component is specified orthogonally to its *content* (functional code) and its architecture. The ADL compiler generates these specific membranes.

3.4 Building OS kernels with THINK

THINK is a general framework for building component-based systems and especially operating system kernels. It is made of the combination of the Fractal component model,




```
composite SpecificOSKernel_H264 {
   provides FileSystem as fs
   provides Allocator as allocator
   provides Scheduler as scheduler
   provides FrameBuffer as fb
   provides Console as console

   contains fs = E2FS
   contains alloc = FlatAllocator
   contains sched = PriorityScheduler
   contains screen = Screen

   binds this.fs tp fs.fs
   binds this.allocator to alloc.allocator
   binds this.scheduler to sched.scheduler
   binds this.fb to screen.fb
   binds this.console to screen.console
   binds scheduler.allocator to alloc.allocator
   binds fs.allocator to alloc.allocator

   membrane Simple
}
```

Figure 1: A simplified view of a specific OS kernel built for the H.264 decoder and the associated ADL description. White boxes represent Kortex's generic components while grey boxes represent the platform specific ones.

its ADL with the associated ADL-to-C compiler and its component library that comprises various standard operating system functions and architecture-dependent components. The initial version of THINK [7] was based on a flat component model. Our current version differs from the original one in that it is entirely based on the hierarchical Fractal Component Model.

The Fractal ADL compiler translates the ADL code to ANSI C and compiles and links the generated code with component implementations (also written in C) using a standard C compiler and linker. The generated C code obeys to a binary format for components detailed in [7]. Figure 1 shows a THINK-based kernel example.

4 Implementation of dynamic reconfiguration in THINK-based OS

We implemented two mechanisms for achieving quiescent state - the first based on thread counting and the second is based on dynamic interceptors. These mechanisms are provided as two different membrane implementations in the ADL, namely the `Reconfig_ThreadCounting` and `Reconfig_DynamicInterceptors` membranes. The kernel designer chooses the appropriate membrane for a given component and the ADL compiler generates an appropriate implementation of the membrane.

Requesting reconfiguration is done via the `ReconfigurationController` control interface implemented by both membranes. This interface defines three methods: add, remove and replace. The implementations of these methods reuse Fractal control interfaces of the membrane and of the components concerned by the reconfiguration. For example, the following pseudo code shows a simple implementation of the `replace` operation.

```
/*  CC:  ContentController    */
/*  BC:  BindingController    */
/*  LCC: LifeCycleController  */
ReconfigurationController->replace(old, new, ...) {
  target::CC->add(new);

  /* Quiescent state request */
  quiescence_request(); //method specific
  //quiescence when resumed

  /* stop the component: ex. driver shutdown */
  old::LCC->stop();

  /* State Transfer */
  state = old::StateTransfer->getState();
  ... //state transformation
  new::StateTransfer->setState(state);

  target::CC->remove(old);
  new::LCC->start();

  /* reference redirection */
  rebind_all_client_components();
}
```

The following subsections detail the two reconfiguration mechanisms we have implemented in the THINK framework. This work applies to multi-threaded kernels, thus we assume the existence of a scheduler and semaphore components, manifested by the presence of scheduler and semaphore client interfaces in figures 2 and 3.

4.1 Thread counting

Thread counting uses interceptors to track thread accesses to a component since the start of the system. On every invocation of a component's interface, the interceptor increments the access counter. The counter is decremented on every method return. Quiescent state is achieved, when the access counter is zero - no thread is executing in the

component at that moment. This mechanism is described as *mutation* in MMLite [10].

In the THINK framework, in order to count thread accesses on components we put in place interceptors for every interface of the component. These interceptors have two functions: they count accesses and, depending on the component's state, block or forward calls to the component. Thread counting interceptors use two semaphore components. One is used as a mutex to protect the counter from concurrent access. This could be done by only disabling interrupts when manipulating the counter, however it is an architecture-dependent optimization of the generic solution. The second semaphore is used only to block a reconfiguration request when waiting for quiescent state.

The figure 2 shows the architecture of a reconfigurable variant of the `screen` component identified on figure 1. As such, this component is reconfigurable using thread-counting - I_1 and I_2 interceptors are placed in front of console and framebuffer components' interfaces. From the point of view of the kernel designer (ADL specification), the screen component has two subcomponents and a membrane providing dynamic reconfiguration using the thread-counting mechanism. The ADL compiler generates the necessary membrane components and their bindings (implementations of ReconfigurationController, ContentController, bindings to semaphore components...).

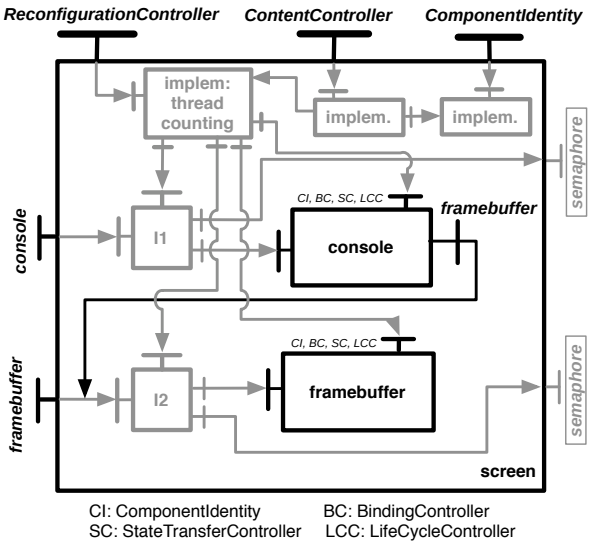


Figure 2: Thread counting interceptors: A fully reconfigurable composite component with interceptors. Parts actually written by the developer are in bold components generated by the ADL compiler are in grey.

In this approach, the level of reconfigurability of a kernel, understood as composite component, is fixed at design-time i.e. when writing the kernel architecture description. One can build a fully reconfigurable kernel turning each and every component in the kernel configuration into a reconfigurable component, but this obvisouly implies memory and performance overheads due to the presence of interceptors at every component interface in the system. Some optimizations are possible, for instance in a case where a subcomponent interface is directly exported by a parent composite but we did not implement them.

4.2 Dynamic interceptors

As a second method for detecting quiescent state, we have implemeneted the dynamic interception mechanism. This work is directly inspired by the K42 project ([19, 1]). We have implemented all mechanisms described in K42 - mediators (dynamic interceptors in THINK), the thread generation mechanism algorithm and finally the three phases hot-swapping algorithm. The implementation of this reconfiguration mechanism is provided as the `Reconfig_DynamicInterceptors` membrane.

When a reconfiguration is requested (via the `ReconfigurationController` interface) the interfaces of the component to be reconfigured are wrapped with interceptors that will help to detect that a quiescent state has been reached, so the component can be removed safely. However as interceptors are introduced at run-time, they do not have any knowledge of the component execution history - if any thread is executing in the component. In order to guarantee quiescent state, the system makes two assumptions about the threaded programming model *i)* system threads are short-lived and *ii)* system threads are non-blocking. Thanks to these assumptions, threads can be divided in generations and one can build tracking mechanisms which can advise when a particular generation has terminated. We refer the interested reader to the K42 publication [19] for detailed discussion of the generation tracking mechanism and quiescent state mechanism built on top of it.

The generation tracking mechanism is implemented as a THINK-component and is used by all interceptors. It plays two roles: it keeps track of the thread generations and it handles generation swap requests. In THINK-based OS, thread components are registered for execution with a scheduler component. We modified the scheduler component to call the generation component when creating or destroying a thread. The generation tracking component requires a semaphore that is used to block requests until a generation swap occurs.

A dynamic interceptor tracks and forwards all incoming calls. When the current thread generation has terminated, it can guarantee that it is tracking all the threads accessing the component, a sufficient condition to determine quiescent state. When a quiescent state is reached, dynamic

interceptors block incoming threads on a semaphore. Dynamic interceptors are instanciated by the implementation of the `ReconfigurationController` control interface, using an interceptor factory which is automatically generated from the ADL description.

The figure 3 shows a simplified configuration of a reconfigurable `screen` component with dynamic interceptors. Note, eventhough the architecture of membranes used in thread counting and dynamic interception are very different, the ADL descriptions for a reconfigurable component are similar - they differ only in the definition of the used membrane.

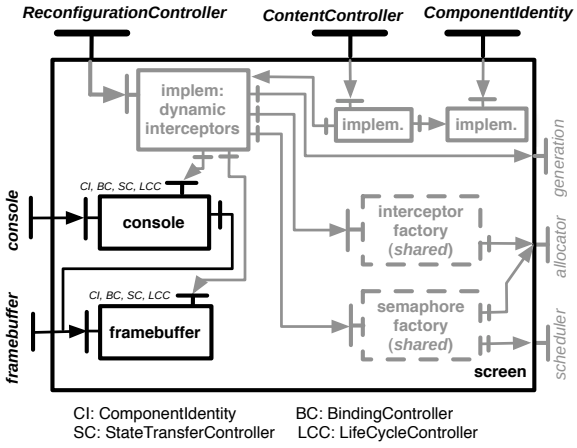


Figure 3: A reconfigurable kernel using dynamic interceptors and the corresponding ADL description, in grey components generated by the ADL compiler.

The main drawback of thread counting was the inflexibility due to the fact that the reconfiguration capabilities were chosen at build-time, moreover, the run-time overhead caused by thread counting interceptors was constant during the whole life-cycle of the system. With dynamic interceptors, interceptors are deployed dynamically, once the reconfiguration is requested, until the completion of the reconfiguration. Thus, there is no overhead during normal kernel run and all components are potentially reconfigurable.

5 Evaluation

This section is devoted to the evaluation of our proposition in terms of flexibility and performance. We used a *componentized* version of a reference H.264 video decoder as application benchmark.

5.1 Qualitative assessment

In order to demonstrate the flexiblity of the proposed approach consider the following ADL description of the `screen` component as it can be found in the `SpecificOSKernel_H264` composite component in the figure 1.

```
composite screen {
  contains fb = video.lib.framebuffer
  contains console = video.lib.console

  binds console.framebuffer to fb.framebuffer
  binds this.console to console.console
  binds this.framebuffer to fb.framebuffer

  membrane Simple
}
```

As such, the shown composite is not reconfigurable (`Simple` membrane).

To add reconfiguration capabilities using thread-counting the kernel designer simply overloads the `membrane` parameter of the component. For instance, when thread-counting is required the overloaded membrane parameter simply takes the following form:

```
composite screen_reconfig extends screen {
  membrane Reconfig_ThreadCounting
}
```

A version of the `screen` component using the dynamic interceptors reconfiguration mechanism is obtained in a similar way. Both reconfigurable versions of the `screen` component have the same functional subcomponents (`console` and `fb`), they differ in the membrane generated by the ADL compiler. These membrane architectures correpond respectively to figures 2 and 3.

These examples illustrate the fact that with our approach, switching reconfiguration mechanisms at build-time is fairly easy, it only involves a change in architecture descriptions.

5.2 Performance evaluation

In the following we compare performance overheads of both implemented approaches for reconfiguring a THINK-based component kernel. We also evaluated the memory overheads of both mechanism that are order of 1% when using thread counting, and 10% when using dynamic interceptors (overhead due to the fact that we do not use generic interceptors and the ADL compiler generates an interceptor factory for every interface type.)

Micro-benchmarks

Both of the implemented mechanisms for dynamic reconfiguration make a heavy use of interceptors. Table 1 summarizes results obtained by measuring only the cost of different interceptors and compares it to the cost of a simple

function call in C and to a basic THINK component call. These measures were obtained on an ARM XScale processor (Intel PXA255 on an iPAQ h2200) running at 400Mhz, with a 100Mhz memory bus.

call type	time (μs)
C function call	$0,03$
THINK method call	$0,06$
THINK method call via thread counting interceptor	$0,6\ (0,27)$
THINK method call via dynamic interceptor	$0,4\ (0,27)$

Table 1: Overheads of different interceptor types, compared to a simple THINK method call. Results in brackets show results obtained with agressively hand-optimized interceptors.

A THINK component is called via a function pointer involving two memory loads with an indirection. A component's method has at least one argument, the component's data pointer, involving another memory load with indirection. This explains the ratio of two observed between a component call and a plain C function call.

Interceptors used for thread counting protect their access counter by a semaphore component. In total six component calls are performed when a call traverses a static interceptor (two calls to the semaphore component that calls itself twice the interrupts disabling component). The return of a component call is handled in the same way. This is a generic architecture and it is clear that it can be subject of aggressive optimization. We hand-optimized the interceptors (result in brackets) by directly disabling interrupts in the interceptor to protect the access counter. However such a hand-optimization is architecture-dependent.

The computation performed in dynamic interceptors is more complex, involving tracking the identity of the caller (getting the reference from the scheduler component and saving it). However dynamic interceptors perform a computation only when the component is called, not on the return.

Note that thread counting interceptors are used during the whole lifecycle of a system, whereas dynamic interceptors are only deployed only during reconfiguration, thus paying no overhead during normal kernel run.

Performance overheads

A previously *componentized* version of the reference H.264 video decoder [11] was used as an application benchmark [12]. This application was co-located with the dedicated kernel (`SpecificOSKernel_H264`, see figure 1). The coarse-grained operating system components which are the allocator, the screen, the filesystem and the scheduler components were specified as being reconfigurable i.e. implementing a membrane providing dynamic reconfiguration. In each run the application was decoding 200 video frames from a H.264 video stream. We were reconfiguring the frame-buffer subcomponent of the screen composite component. Overheads of both implemented reconfiguration mechanisms were evaluated on two variations of the application, the first using "classic" threads, with no restriction, and the second using threads conforming to the requirement to use dynamic interceptors - non blocking and short-lived.

Table 2 shows the performance results obtained. Note, the reconfiguration mechanism using dynamic interceptors is applicable only when using short-lived and non-blocking threads. In this case, thread-counting is also applicable. We measured execution times with and without reconfiguration of the frame-buffer subcomponent of the screen component.

	no reconfig.	**one reconfig.**
"classic" threads:		
non reconfigurable	$4,74s$	-
thread counting	$4,95s\ (4,4\%)$	$4.99s\ (5,2\%)$
short-lived and non-blocking threads:		
non reconfigurable	$7,14s$	-
dynamic interceptors	$7,14s$	$7,42\ (3,9\%)$
thread counting	$7,24s\ (1,4\%)$	$7.32s\ (2,5\%)$

Table 2: Duration of decoding 200 video frames with different reconfigurable kernels compared to a non-reconfigurable version of the same kernel architecture.

As expected, there is no overhead during normal run when using dynamic interceptors for dynamic reconfiguration, whereas with thread counting we observed a slight overhead. However with one reconfiguration performed, dynamic interceptors present a more important overhead than thread counting, due to an increased reconfiguration complexity.

The overheads observed depend on the combined ratio between *i)* the complexity of the interceptor used for reconfiguration, *ii)* the complexity of the frame-buffer (simple in this case) and *iii)* the access pattern of the component (and consequently the interceptor). For example in a video application, the frame-buffer component (as opposed to the scheduler for example) is massively accessed, resulting in a poorer performance when counting thread accesses. Also note that the observed measures include the complexity of the new frame-buffer.

An important observation can be made when comparing results of reconfiguration overheads of the two concurrency models. Using thread counting for dynamic reconfiguration exhibits better performance results for this system than using dynamic interception. We conclude that the

choice of a reconfiguration mechanism for a dedicated kernel is not simply a matter of relative performance during nominal behavior, but depends also crucialy on the multi-threading structure of the target system and the chosen degree of dynamic reconfigurability.

As the operational conditions of a kernel may also evolve over time, it may be interesting to switch between dynamic reconfiguration mechanisms. However we did not yet consider how to implement this switch during execution.

6 Conclusion

Dynamic reconfiguration mechanisms provide a way of modifying a deployed operating system without service interruption. We have described two implementations of such mechanisms in the THINK framework providing a full support for reconfiguration in THINK-based OS. With our approach, functional implementation of a kernel can be mostly independent of the choice of reconfiguration mechanism. This is in contrast to previous proposals, such as K42 or MMLite, which tied their reconfiguration mechanisms to the kernel design.

Our application benchmarks showed that there is no universal mechanism for reconfiguration, in particular the choice of the most efficient reconfiguration mechanism depends both on the multi-threading structure of the application and on the granularity of the reconfigurability i.e. which components are targeted to be reconfigurable in the chosen architecture. This comforts our approach for providing a flexible way to choose between different reconfiguration mechanism via architecture descriptions.

THINK is freely available at `http://think.objectweb.org`.

References

[1] A. Baumann, G. Heiser, J. Appavoo, D. DaSilva, O. Krieger, R. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in java. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE)*, 2004.

[4] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal Component Model. http://fractal.objectweb.org.

[5] eCos. http://sources.redhat.com/ecos.

[6] D. Engler, M. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[7] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think: a software framework for component-based operating system kernels. In *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

[8] B. Ford, J. Lepreau, S. Clawson, K. V. Maren, B. Robinson, and J. Turner. The Flux OS Toolkit: Reusable Components for OS Implementation. In *HOTOS '97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[9] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. In *Proceedings of the USENIX Annual Technical Conference*, 1999.

[10] J. Helander and A. Forin. MMLite: a highly componentized system architecture. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, 1998.

[11] JVT software page. http://bs.hhi.de/suehring/tml.

[12] O. Layaida, A. E. Özcan, and J.-B. Stefani. A component-based approach for MPSoC SW design: Experience with OS customization for H.264 decoding. In *3rd Workshop on Embedded Systems for Real-Time Multimedia*, 2005.

[13] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[14] C. Pu, T. Autrey, A. P. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.

[15] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component Composition for Systems Software. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[16] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS distributed operating system. *In Computing Systems*, Vol. 1(4), 1988.

[17] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[18] C. Small and M. Seltzer. Structuring the kernel as a toolkit of extensible, reusable components. In *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems*, 1995.

[19] C. Soules, J. Appavoo, K. Hui, R. Wisniewski, D. D. Silva, G. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference*, 2003.

[20] A. Veitch and N. Hutchinson. Dynamic Service Reconfiguration and Migration in the Kea Kernel. In *Proceedings of the International Conference on Configurable Distributed Systems (ICCDS)*, 1998.