International Journal of Computer Science and Software Engineering (IJCSSE), Volume 5, Issue 8, August 2016
ISSN (Online): 2409-4285 www.IJCSSE.org Page: 161-174

Functionality Farming in POK/Rodosvisor

Adriano Carvalho[1], Francisco Afons[2], Paulo Cardoso[3], Jorge Cabral[4], Mongkol Ekpanyapong[5], Sergio Montenegro[6] and Adriano Tavares[7]

[1, 3, 4, 7] Embedded Systems Research Group, Universidade do Minho, 4800–058 Guimarães — Portugal

[2] Instituto Politécnico de Coimbra (ESTGOH), 3400–124 Oliveira do Hospital — Portugal

[5] Asian Institute of Technology, Pathumthani 12120 — Thailand

[6] Universität Würzburg, 97074 Würzburg — Germany

{[1]acarvalho,[3]paulo.cardoso,[4]jcabral,adriano.[7]tavares}@dei.uminho.pt, [2]francisco.afonso@estgoh.ipc.pt, [5]mongkol@ait.ac.th, [6]montenegro@informatik.uni-wuerzburg.de

ABSTRACT

Most operating systems today follow a monolithic architecture which is associated with some drawbacks, such as: low reliability, weak security, high certification effort, as well as poor predictability and scalability. The solutions proposed in the literature to address those drawbacks, however, depend on a significant upfront investment, lead to poor performance and, in the end, do not tackle the source of the problem (i.e., the large size of the kernel in most commodity operating systems) and just work around it.

This paper presents functionality farming and FF-AUTO. On one hand, functionality farming consists in time and space partitioning an existing kernel, thus reducing its size and tackling the source of the problem. It depends on a lower upfront investment and it is also a more agile approach. On the other hand, FF-AUTO performs functionality farming semi-automatically in POK/rodosvisor. With FF-AUTO, the engineering effort, and thus, the risk associated with functionality farming is significantly reduced, making it an ideal tool for design space exploration. This paper also demonstrates how functionality farming is able to improve the design and the performance of POK/rodosvisor, and how FF-AUTO enables a significant reduction of the required engineering effort.

Keywords: *Operating Systems, Software Reliability, Software Security, Software Evolution, Automation.*

1. INTRODUCTION

Most operating systems today follow a monolithic architecture [1]–[3]. In a monolithic architecture many of the services provided by the operating system (e.g., device drivers and protocol stacks) are deployed in the same address space as the kernel. For example, in the Linux kernel, version 2.6, 88% of the code is related to protocol and device drivers [4]. This, however, leads to a large kernel, which in turn, is associated with the following drawbacks:

- A large number of bugs and low reliability: a conservative estimate indicates that there are six bugs per 1,000 lines of code, and device drivers have bug rates that are three to seven times higher normal code [1], [5];
- A large attack surface and weak security: since the kernel is part of the trusted computing base of the entire system, if even a small function in the kernel is compromised, then, the entire system is at risk [4], [6];
- A high certification effort: a large kernel is also harder to certify than a small one [3], [7].

Furthermore, most of those services are not explicitly schedulable and "steal" other schedulable entities' execution time, leading to poor predictability and scalability as well.

To reduce the size of the trusted computing base (TCB), and thus, to improve security and reduce the certification effort, some authors propose the use of architectures based on: (1) a virtual machine monitor (or hypervisor) [4], [6], [8], or (2) a microkernel [3], or (3) a combination of the two (i.e., a microkernel with virtualization support) [9], in which the size of the core/root kernel is much smaller than the kernel found in most commodity operating systems. In these architectures, commodity operating systems are pushed onto a virtual machine (on a hypervisor-based architecture) or onto one or more user-level servers (on a microkernel-based architecture), reducing the effects that a compromised commodity operating system can have on the system as a whole. Alongside commodity operating systems, critical services are deployed on other virtual machines (or user-level servers), which depend on a much smaller kernel/TCB than that in a commodity operating system. These architectures, however, depend on an additional level of indirection which leads to poor performance. Moreover, these

architectures depend on the development of a new kernel, and thus, of a significant upfront investment; if the development of the new kernel fails, the cost is huge. In the end, these architectures do not tackle the source of the problem, i.e., the large size of the kernel in commodity operating systems, and just work around it.
This paper presents functionality farming which, instead of a new architecture, consists in time and space partitioning an existing kernel by (1) moving functionality out of the kernel and onto the partition (or application) level, to reduce the size of the kernel, and by (2) replacing the functionality being moved with remote procedure calls to the partition level, to bridge the gap between the kernel and the partition level. This is illustrated in Fig. 1. Through time and space partitioning, it is possible, among other things, to: (1) reduce the size of the kernel/TCB, and thus, to improve its reliability and security, as well as to reduce the certification effort; (2) improve the kernel's predictability and scalability by making functionality, which was previously part of the kernel, explicitly schedulable.

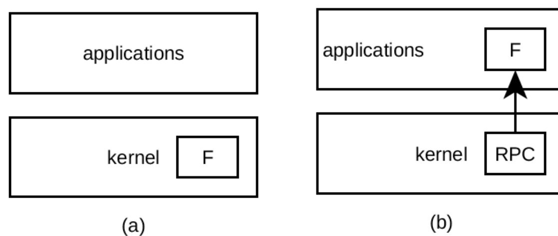

Fig. 1. Illustration of functionality farming: (a) before functionality farming; (b) after functionality farming.

Unlike other works, functionality farming tackles the source of the problem (i.e., the size of the kernel in commodity operating systems). It requires a lower upfront investment, since it enables a progressive reduction of the size of the kernel, instead of an all-or-nothing approach, and thus, it is also a more agile approach since it enables some decisions to be postponed closer to delivery time when information about the system's requirements is more precise. Functionality farming, however, despite the benefits, not only depends on a significant engineering effort, as will be shown later, but its effects are often very hard to predict, meaning that the associated risk is still high.
This paper also presents FF-AUTO, a tool which performs functionality farming semi-automatically in POK/rodosvisor. With FF-AUTO, the engineering effort, and thus, the risk associated with functionality farming is significantly reduced, making it also an ideal tool for design space exploration. To the best of our knowledge, this is the first paper ever to present a methodology for semi-automatically moving functionality out of the kernel and onto partition level.

Lastly, this paper demonstrates how functionality farming is able to improve the design and the performance of POK/rodosvisor, as well as how FF-AUTO enables a significant reduction of the required engineering effort. Currently, we are unable to demonstrate a reduction of the size of the kernel since POK/rodosvisor is already a very small kernel (very close to microkernel).
Even though functionality farming and FF-AUTO, currently, target only POK/rodosvisor, the underlying methodology can be applied to any other operating system.
This paper is organized as follows. In the following section, an overview of related work is given. In section 3, POK/rodosvisor is described in more detail. In section 4, a more detailed description of functionality farming is given. In section 5, FF-AUTO and its underlying methodology are presented. In sections 6 and 7, two use cases are presented, demonstrating that functionality farming is able to improve the design and the performance of POK/rodosvisor, and that FF-AUTO contributes to a significant reduction of the required engineering effort, and thus, of the associated risk. Finally, in section 8, a summary of the major findings and contributions is given, and future work is proposed.

2. RELATED WORK

Functionality farming is loosely based on the concept of task farming. Task farming consists on the decomposition of computations into identical and independent serial tasks, which are then executed by different processor cores in a multicore processor or in multi-processor systems such as computational grids [10]. Task farming is suited for computations such as a Montecarlo simulations in which the same model is run many times but with different start points (or inputs). A task farm is generically composed by:

- The farmer, responsible for distributing the input to a pool of tasks and for retrieving and merging the output;
- A pool of identical tasks, which perform the actual computation in parallel.

Similarly, functionality farming consists on the decomposition/partitioning of the kernel into partitions which perform the actual computation (not necessarily in parallel). Continuing with this analogy, then, the kernel is the farmer, and the partitions are the pool of identical tasks.
To reduce the size of the trusted computing base (TCB), and thus, to improve security and reduce the certification effort, some authors propose the use of architectures based on: (1) a virtual machine monitor (or

hypervisor) [4], [6], [8], or (2) a microkernel [3], or (3) a combination of the two (i.e., a microkernel with virtualization support) [9], in which the size of the core/root kernel is much smaller than the kernel found in most commodity operating systems. More specifically, [4] and [6] propose an architecture based on a virtual machine monitor (or hypervisor). In particular, [4] uses paravirtualization to achieve an even smaller size of the kernel (and thus of the TCB); to accomplish communication between virtual machines, in [6] the hypervisor fully virtualizes one or more network devices, while in [4] paravirtualization is used to achieve the same result but with a smaller kernel/TCB (at the cost, however, of requiring modification to legacy software). Similarly, [8] proposes an architecture based on a hypervisor to better partition the system across the cores of a multicore processor system. On the other end, [3] proposes a microkernel architecture, and [9] proposes an architecture based on a microkernel with virtualization support. In these architectures, commodity operating systems are pushed onto a virtual machine (on a hypervisor-based architecture), or onto one or more user-level servers (on a microkernel-based architecture), reducing the effects that a compromised commodity operating system can have on the system as a whole. Alongside the operating system, critical services are deployed on other virtual machines (or user-level servers), which depend on a much smaller TCB than that in a commodity operating system. These architectures, however, depend on an additional level of indirection which leads to poor performance. Furthermore, on a hypervisor-based architecture in particular, virtual machines are often coarse-grained and heavyweight leading to high resource usage. On a microkernel architecture, on the other end, there is no compatibility with existing software, and the porting effort can be very significant. An architecture based on a "microkernel with virtualization support" solves the issues with the other two architectures, at the cost, however, of a larger size of the kernel/TCB. Still, these architectures depend on the development of a new kernel, and thus, on a significant upfront investment; if the development of the new kernel fails, the cost is huge. In the end, these architectures do not tackle the source of the problem, i.e., the large size of the kernel in commodity operating systems, and just work around it. Conversely, functionality farming tackles the source of the problem, by enabling a progressive reduction of the size of the kernel/TCB, and requires a lower upfront investment. Moreover, through FF-AUTO, the associated engineering effort is significantly reduced, facilitating even further the application of functionality farming.

Similarly to FF-AUTO, other works propose the improvement of existing software through refactoring. For example: [11]–[14] focus on parallelization of existing sequential programs, [15]–[18] on improving security, [19] on improving modularity, and [20] on enabling reentrancy, among others. Similarly to our work, these works enable a reduction of the engineering effort required to implement and evaluate different design alternatives, and thus, also reduce the associated risk. These works, however, imply modifications to the original source code, and require knowledge of the implementation details. Furthermore, these works are limited to hardware-independent applications. Our work, on the other end, instead of transformations at the level of the source code, relies on link time transformations. The original source code is not modified, and only the knowledge of external control and data dependencies is required. Moreover, our work addresses the requirements of hardware-dependent, kernel-level software, and thus, it is not limited to hardware-independent application.

3. POK/RODOSVISOR

POK/rodosvisor is a separation kernel featuring three partition types, as illustrated in Fig. 2, namely: ARINC 653 partitions [21], privileged partitions, and virtual machines. POK/rodosvisor is the result of the integration of POK [22], a separation kernel featuring ARINC 653 partitions, and Rodosvisor [23]–[25], a previously bare metal hypervisor, featuring full virtualization of the IBM PowerPC 405 [26]. Support for privileged partitions has been added subsequently.

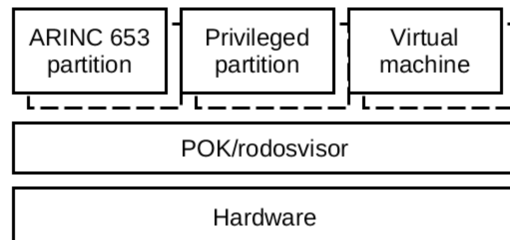

Fig. 2. The architecture of POK/rodosvisor.

Partitions are scheduled according to a static scheduling policy, as defined by ARINC 653 [21]. ARINC 653 partitions enforce time and space partitioning, have access to the APlication EXecutive (APEX), and their threads are scheduled according to a priority-based scheduling scheme, all of which, as defined by ARINC 653. The APEX is a programming interface for partition, thread and time management, inter and intra-partition communication, among other things.

A privileged partition is like an ARINC 653 partition, except that it does not enforce space partitioning. A privileged partition does not enforce a protected address space, it runs in the same address space as the kernel and with the same level of privilege. Because a privileged

partition does not enforce space partitioning, for any given system, only one privileged partition is ever required, and thus, the overhead in terms of the kernel's footprint is not as high as other partition types. A privileged partition also has access to the APEX and, like an ARINC 653 partition, its threads are scheduled according to a priority-based scheduling scheme.

Virtual machines feature full virtualization of the IBM PowerPC 405 [26], and, if desired, paravirtualization. Virtual machines enforce time and space partitioning, and, through full virtualization, also provide compatibility with both application and kernel-level software. Virtual machines support the configuration of: (1) the memory address space (including memory-mapped I/O); (2) a virtual I/O controller for I/O virtualization; and (3) an hypercall controller for paravirtualization. Unlike ARINC 653 and privileged partitions, the guest of a virtual machine is responsible for thread scheduling. Additionally, by default, virtual machines have no access to the APEX; however, an hypercall controller can be used to enable access to a limited subset of the APEX, such as to the inter-partition communication services.

Development of a system based on POK/rodosvisor starts with the creation of an AADL model [27] specifying the desired system configuration. An AADL model enables, among other things, the specification of:

- partitions: their type and their implementation;
- threads: a partition's implementation can be specified in terms of constituent threads;
- scheduler's configuration: this includes the configuration of the partition-level and the thread-level schedulers;
- inter and intra-partition communication configuration.

Having created a model specifying the desired system configuration, this model is compiled, using Ocarina [28], into a POK/rodosvisor configuration, as illustrated in Fig. 3, which implements the system configuration specified in the model. A POK/rodosvisor configuration is composed of: (1) C source and header files containing the implementation and the configuration of a POK/rodosvisor kernel; (2) the implementation and the configuration of the partitions, if required; and (3) Makefiles which implement the configuration's build system. A POK/rodosvisor configuration is sometimes manually modified to address particular concerns which, currently, cannot be specified in the model. Finally, compilation of a POK/rodosvisor configuration produces an executable ready to be downloaded, executed and profiled on the target, as illustrated in Fig. 3.

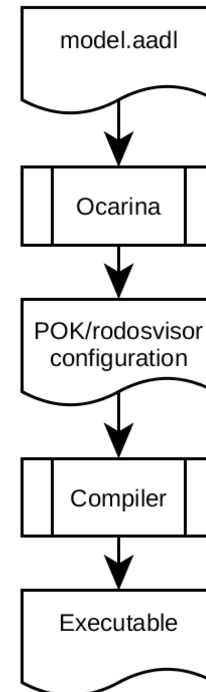

Fig. 3. Development of a system based on POK/rodosvisor.

4. FUNCTIONALITY FARMING

Functionality farming consists in time and space partitioning an existing kernel by (1) moving functionality out of the kernel and onto the partition (or application) level, to reduce the size of the kernel (and thus, of the TCB), and by (2) replacing the functionality being moved with remote procedure calls to the partition level, to bridge the gap between the kernel and the partition level.

At the partition level, memory protection is enforced (space partitioning), and, previously non-schedulable entities, become explicitly schedulable (time partitioning). Through space partitioning, it is possible to reduce the size of the kernel/TCB, and thus, improve its reliability and security, as well as to reduce the certification effort. Additionally, in case of failure only the faulting partition, and not the entire kernel (and thus, not the entire system), needs to be restarted, leading to higher availability as well. On the other end, through time partitioning, it is possible to improve the kernel's predictability and scalability by making functionality, which was previously part of the kernel, explicitly schedulable. At the partition level, moreover, it is easier to distribute the operating system's services across the cores of a multicore processor architecture, leading to improved predictability and scalability on those platforms.

Time and space partitioning an existing kernel, nevertheless, is not an easy task. In some cases, because of a functionality's level of coupling with the kernel or its functional requirements (e.g., compatibility with hardware-dependent, kernel-level software), ensuring that time and space partitioning is possible, and at the same time, fulfilling the functionality's functional requirements may depend on a significant engineering effort. To address this issue, functionality farming relies on various partition types.

The different partition types each provide distinct levels of partitioning (e.g., from only time partitioning to both time and space partitioning), as well as they fulfill different functional requirements (e.g., from compatibility with hardware-dependent, kernel-level software to compatibility with only hardware-independent software). POK/rodosvisor, in particular, features three partition types:

- ARINC 653 partitions [21]: support only hardware-independent software but enforce both time and space partitioning;
- privileged partitions: support all kinds of software (i.e., hardware-dependent and hardware-independent software), enforces time partitioning but not space partitioning;
- and virtual machines: support all kinds of software, and enforces both time and space partitioning.

These different partition types, not only increase the extent to which functionality farming is more easily accomplished, but also enable an even more progressive reduction of the size of the kernel. Thus, these different partition types enable faster design space exploration, and reduce the associated risk. As an example, consider the following. In the beginning, when a functionality is tightly coupled with the kernel, achieving space partitioning may require a significant engineering effort, while achieving time partitioning may not be as hard, then, a partition which provides only time partitioning can be used. At this stage, it cannot be expected that the size of the kernel will be reduced; nevertheless, as will be shown later, time-partitioning-only can reveal interesting design alternatives (as well as it enables bad design alternatives to be ruled out early on). After modifying the functionality to enable space partitioning, it may still depend on compatibility with kernel-level software, then, a partition providing time and space partitioning as well as compatibility with kernel-level software can be used. Lastly, after the functionality is made hardware-independent, then, a lightweight, hardware-independent partition can be used.

Unlike other works, functionality farming tackles the source of the problem (i.e., the size of the kernel in commodity operating systems). It requires a lower upfront investment, since it enables a progressive reduction of the size of the kernel, instead of an all-or-nothing approach, and thus, it is also a more agile approach since it enables some decisions to be postponed closer to delivery time when information about the system's requirements is more precise. Functionality farming, however, still depends on a significant engineering effort, as will be shown later, and its effects are often very hard to predict, meaning that the associated risk is still high. The following section presents FF-AUTO, a tool which performs functionality farming semi-automatically in POK/rodosvisor. With FF-AUTO, the engineering effort, and thus, the risk associated with functionality farming is significantly reduced, making it also an ideal tool for design space exploration.

5. FUNCTIONALITY FARMING AUTOMATED: FF-AUTO

As shown in Fig. 4, FF-AUTO requires as input: (1) an AADL model, specifying the reference configuration, and (2) a functionality farming configuration (FFC) file, which specifies the functions that should be farmed and how, based on the results from a profiler, by manual inspection, etc. As output, FF-AUTO generates a POK/rodosvisor configuration (i.e., a modified configuration) which is the result of applying functionality farming to the reference POK/rodosvisor configuration specified in the input model.

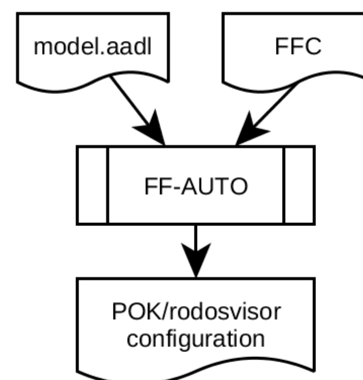

Fig. 4. Inputs and outputs of FF-AUTO.

The modified POK/rodosvisor configuration needs, in some cases, to be manually modified as FF-AUTO is unable to automatically derive all of the necessary parameters, as will be explained later. In most cases, however, only the scheduler's configuration needs to be updated. In the future, we expect to enable the specification of the modified scheduler's configuration in the FFC file. Similarly, for optimization purposes, the output from FF-AUTO may be manually modified.

In the following subsections, the FFC file format and the transformations applied to the reference POK/rodosvisor configuration are described.

5.1 Functionality Farming Configuration File

The FFC file format consists of a list of statements preceded by "%" followed by a list of function prototypes, as shown in Fig. 5. The following statements are required:

System: this statement requires three parameters separated by a semicolon. The first parameter specifies the name of the AADL system implementation containing the AADL processor which is being farmed. In AADL, a processor represents a combination of software and hardware responsible for scheduling threads, enforcing protected address spaces, among other things. In other words, an AADL processor corresponds to the operating system, including the underlying hardware platform. In Ocarina, it corresponds to a POK/rodosvisor kernel. The second parameter specifies the implementation type name of the AADL processor to be farmed. The third parameter specifies the instance name of the AADL processor to be farmed in the AADL system specified as the first parameter. This statement can appear only once. For example, the system statement for the model in Fig. 6 is: "%system example::node.impl; example::pok.impl; cpu." In this example, the instance of "example::pok.impl," named "cpu" in "example::node.impl," is the processor to be farmed.

***Worker*:** two parameters separated by a semicolon. The first parameter specifies a worker's identifier, and the second parameter specifies its type. Worker is a synonym of partition; however, we use "worker" to distinguish the partitions in the reference configuration (partitions) and the partitions involved in functionality farming (workers). Worker threads, described below, that should be assigned to this particular worker must use the identifier specified as the first parameter. Currently, there are three types of workers supported, namely: "arinc653" for a worker based on an ARINC 653 partition; "privileged" for a worker based on a privileged partition; and "vm" for a worker based on a virtual machine. This statement can be repeated more than once, as necessary. For example, "%worker worker_1; privileged," specifies a worker based on a privileged partition, identified as "worker_1."

worker_thread: this statement requires two parameters separated by a semicolon. The first parameter specifies an identifier for a worker thread. The second parameter specifies the worker's identifier with which the worker thread is assigned to. Functions that should be assigned to this particular worker thread must use the identifier specified as the first parameter. This statement can be repeated more than once, as necessary. For example, "%worker_thread worker_thread_1; worker_1," specifies a worker thread identified as "worker_thread_1" and assigned to the worker identified as "worker_1."

```
%system example::node.impl; \
        example::pok.impl; cpu
%worker worker_1; privileged
%worker worker_2; vm
%worker_thread worker_thread_1; worker_1
%worker_thread worker_thread_2; worker_2
void pok_port_flushall(); worker_thread_1; \
                          call-only
void pok_cons_write(char*, int);
worker_thread_2
```

Fig. 5. A functionality farming configuration file.

```
package example
public

processor pok
end pok;

processor implementation pok.impl
end pok.impl;

system node
end node;

system implementation node.impl;
subcomponents
    cpu : processor pok.impl;
end node.impl;

end example;
```

Fig. 6. An incomplete AADL model.

The list of statements just described is followed by a list of function prototypes. A function prototype is specified using the syntax of C, followed by a semicolon and the identifier of the worker thread which the function is assigned to. Optionally, a second semicolon followed by the keyword "call-only" can be specified, as shown in Fig. 5. The "call-only" keyword indicates that only the function call, and not its implementation should be farmed. Function call farming and its motivation are explained in the following section.

5.2 Function Call Farming and Complete Farming

Some partition (or worker) types are unable to support specific kinds of functionality, and therefore, the implementation of some functions cannot be moved to them. To address this limitation, function call farming can be performed instead. With function call farming, the function's implementation remains in the kernel, and only the "function call" is moved. With function call farming, it cannot be expected that the size of the kernel will ever be reduced, because the function's implementation is not moved; however, as will be shown later, function call farming alone can still reveal

good design alternatives. And, after finding a good design alternative, the function's implementation can be modified to enable complete farming (i.e., where the function's implementation is actually moved to the partition).

More specifically, ARINC 653 partitions cannot support hardware-dependent functionality; therefore, when farming hardware-dependent functionality onto an ARINC 653 partition, function call farming must be specified. A privileged partition, on the other end, because it runs at the same level as, and with the same level of privilege as the kernel, farming any functionality onto a privileged partition means that the function's implementation will remain in the same address space, and therefore, only function call farming is ever performed. A virtual machine supports both hardware-independent and hardware-dependent functionality, which means it is able to support complete farming of all kinds of functionality; however, with some exceptions, as explained next.

Independently of the partition type, currently, our methodology does not support moving functionality with kernel dependencies (i.e., which requires access to kernel data), and thus, for those kinds of functionality, function call farming needs to be specified. In the future, we expect to use existing code rewriting or binary translation techniques to automatically replace direct accesses to kernel data with other mechanism such as, for example, system calls.

5.3 Workers and Worker Threads

For each worker specified in the FFC file a worker is added to the reference configuration. The type of the worker is as specified in the FFC file. Similarly, worker threads are added to the corresponding workers, as specified in the FFC file. This is illustrated by the transformation between (a) and (b) in Fig. 7.

For functions which require access to I/O and which have been assigned to a virtual-machine-based worker, the output from FF-AUTO needs to be manually modified to update the virtual machine's configuration such that it can have access to the required I/O. In the future, it is expected that a function's I/O requirements could be derived automatically using methods such as those described in [29]. The output from FF-AUTO also needs to be manually modified in order to update the scheduler's configuration to accommodate the new workers and worker threads, as explained earlier. In any case, as will be shown in section 6 and section 7, the required modifications are very small.

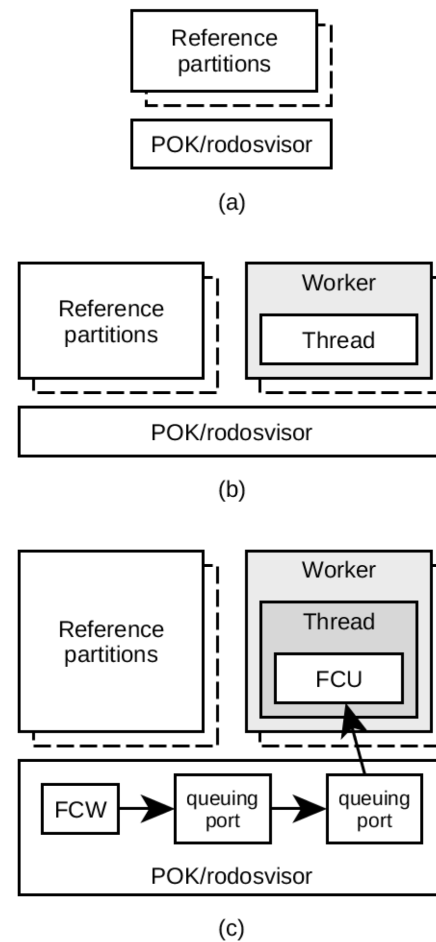

Fig. 7. Illustration of the transformations applied to the reference configuration by FF-AUTO: (a) the reference configuration; (b) workers and worker threads added to the reference configuration; (c) FCW, FCU, and associated communication channels added to the reference configuration.

5.4 Function Call Wrappers and Unwrappers

For each function prototype specified in the FFC file, a function call wrapper (FCW) and a corresponding function call unwrapper (FCU) is generated. Additionally, in the kernel, function calls to the functions specified in the FFC file are replaced with function calls to the corresponding FCW. Finally, for those functions specified for complete farming, the functions' implementation is moved from the kernel onto the specified worker. This is illustrated by the transformation between (b) and (c) in Fig. 7. For those functions specified for function call farming, on the other end, the functions' implementation remains in the kernel.

A FCW, as shown in Fig. 8, serializes function calls through a communication channel, and returns immediately after. A FCU, conversely, as shown in Fig.

9: (1) deserializes function calls from a communication channel, and (2) jumps to the function's implementation. Return values are currently not supported. Serialization and deserialization of function calls relies on a data structure declaration that is generated based on the function's prototype. In POK/rodosvisor, communication channels are queuing communication channels, with one sending port (used by the FCW), and one receiving port (used by the corresponding FCU), as illustrated in Fig. 7(c). For each function prototype specified in the FFC file, currently, a dedicated communication channel is established. In the future, it is expected that some functions will be able to share the same communication channel, forming a function group, and thus, lowering resource usage.

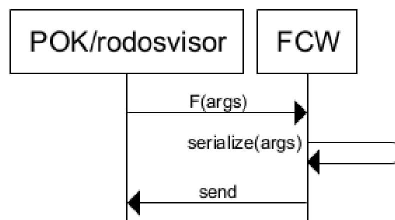

Fig. 8. Sequence diagram of a generic function call wrapper (FCW).

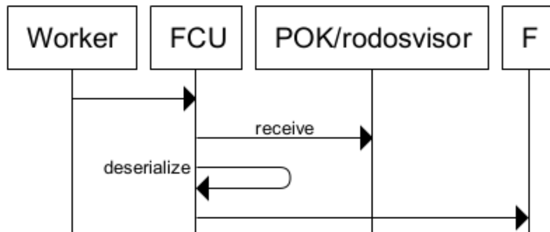

Fig. 9. Sequence diagram of a generic function call unwrapper (FCU).

The process of checking for and receiving new function calls depends on the type of the worker. For workers based on ARINC 653 or privileged partitions, function calls are received using the communication services provided by the ARINC 653 APEX to which they have access by default. For workers based on virtual machines, which by default do not have access to the APEX, a dedicated hypercall controller is generated which, indirectly, enables a virtual machine to access the communication services provided by the APEX, and thus, receive function calls.

Similarly, the process of jumping to a function's implementation depends on whether or not its implementation has been moved from the kernel onto the specified worker, and it depends on the specified worker's type. For those functions whose implementation is moved from the kernel onto the specified worker (i.e., complete farming), jumping to its implementation is performed directly. On the other end, for those functions whose implementation is not moved from the kernel to the specified worker (i.e., function call farming), jumping to its implementation, which remains in the kernel, depends on the specified worker's type. If the worker is an ARINC 653 partition, then, support for a dedicated system call is added to the kernel so that the worker can request the kernel to jump to the function's implementation. If, however, the worker is a privileged partition, then: (1) preemption is disabled, (2) a direct jump to the function's implementation is performed, and (3) when the function's implementation returns, preemption is enabled again. Preemption is disabled in order to prevent the kernel from becoming in a corrupt state. Furthermore, a privileged partition is part of the kernel and, therefore, it can jump to the function's implementation directly. Lastly, if the worker is a virtual machine, then, an hypercall is used to request the virtual machine's hypercall controller to jump to the function's implementation. A hypercall controller is part of the hypervisor, which is a part of the kernel, and thus, has access to the function's implementation. A dedicated hypercall controller is automatically generated whenever necessary.

6. USE CASE: SERIAL PORT DEVICE DRIVER

In this section, first, a POK/rodosvisor configuration (i.e., the reference configuration) is described and it is demonstrated that it reveals a limitation in the design of POK/rodosvisor, more specifically, in the design of the serial port device driver. Second, it is described how functionality farming has been applied to the reference configuration and how it is expected to address the limitation identified earlier. Third and last, it is demonstrated that functionality farming addresses that limitation by comparing the performance before and after functionality farming. Even though functionality farming is used to address a limitation in the design of POK/rodosvisor, we do not claim it is the only or the best way to do it; our goal is only to demonstrate a possible use case for functionality farming.

The reference architecture is illustrated in Fig. 10. It consists of a POK/rodosvisor-based system with one ARINC 653 partition composed by one thread (the writer). As illustrated in Fig. 11, for each partition window, the writer sends data to the serial port, through an ARINC 653 virtual queuing port which, in turn, forwards all data to the serial port. After sending the data, the writer reports the time (as number of CPU clock cycles) required to so, and then goes idle until the next partition window.

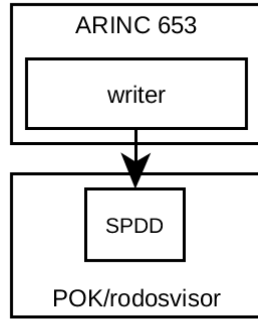

Fig. 10. The reference architecture used for the serial port device driver's use case.

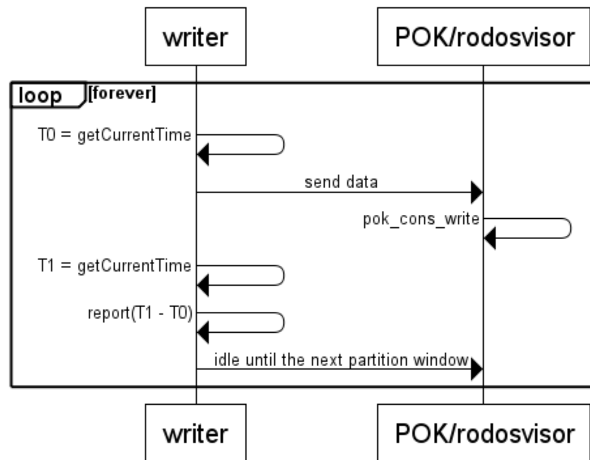

Fig. 11. A sequence diagram illustrating the process perform by the writer in the serial port device driver's use case.

From the reference configuration, the average number of CPU clock cycles per byte required to send data to the serial port for various sizes of data have been measured. For each data size, the configuration ran until 100 samples have been collected; the average number of CPU clock cycles was obtained by averaging all the samples.

Fig. 12 shows, for the reference configuration, the average number of CPU clock cycles per byte required to send data to the serial port for different sizes of data. It can be seen that, the larger the data, the higher the number of CPU clock cycles per byte, indicating that sending data to the serial port does not scale well.

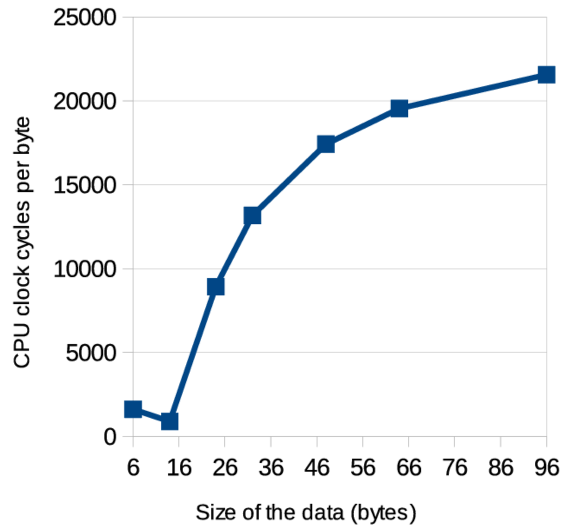

Fig. 12. The average number of CPU clock cycles per byte required to send data to the serial port for different sizes of data for the reference configuration.

Further investigation revealed that the lack of scalability identified above could be traced back to a function in POK/rodosvisor named “pok_cons_write,” illustrated in Fig. 11. In POK/rodosvisor, “pok_cons_write” is the function which interacts directly with, and sends data to the serial port. Once the serial port's transmit buffer (i.e., a 16-byte buffer) becomes full, “pok_cons_write” busy-waits until the buffer becomes available before sending more data. This means that, when the size of the data is larger than the serial port's transmit buffer, “pok_cons_write” needs to busy-wait constantly until all data is sent. Taking into account that the rate at which the serial port dispatches data into the transmission line is very slow compared to the CPU, then, as illustrated in Fig. 12, when the size of the data is larger than the serial port's transmit buffer, the impact on the number of CPU clock cycles required to send the data is significant.

Seen this, the application of functionality farming has been considered. More specifically, to farm “pok_cons_write” into a dedicated worker. In this way, “pok_cons_write,” instead of interacting directly with, and sending data to the serial port, sends data to a sending queuing port, which can feature a much larger buffer and is also much faster than the serial port. The worker, on the other end, reads data from a receiving queuing port and, only then, sends it to the serial port. Because the worker is assigned with a dedicated execution time in the major frame, the lack of scalability when sending data to the serial port does not affect the rest of system.

Functionality farming has been applied using FF-AUTO and three FFC files, which specify that “pok_cons_write” shall be farmed into one worker with one worker thread.

One of the FFC files is shown in Fig. 13 and it specifies a virtual-machine-based worker (configuration VM); the resulting architecture is shown in Fig. 14. The other two FFC files specified an ARINC-653-partition-based worker (configuration AP) and a privileged-partition-based worker (configuration PP). For configuration VM, complete farming has been specified. For configuration AP, on the other end, function call farming has been specified; “pok_cons_write” is hardware-dependent and, as explained in section 5, only function call farming is supported. For a privileged partition, configuration PP, only function call farming is supported. The output from FF-AUTO (i.e., a modified POK/rodosvisor configuration), for all configurations has been manually modified in order to accommodate the worker and its worker thread in the scheduler's configuration. For configuration VM, additionally, the virtual-machine-based worker's configuration has been manually modified in order to enable access to the serial port hardware, as required by “pok_cons_write.”

```
%system test::node.impl; test::ppc.impl; cpu
%worker worker_1; vm
%worker_thread worker_thread_1; worker_1
void pok_cons_write(char*, int);
worker_thread_1
```

Fig. 13. FFC file for farming “pok_cons_write” on a worker based on a virtual machine.

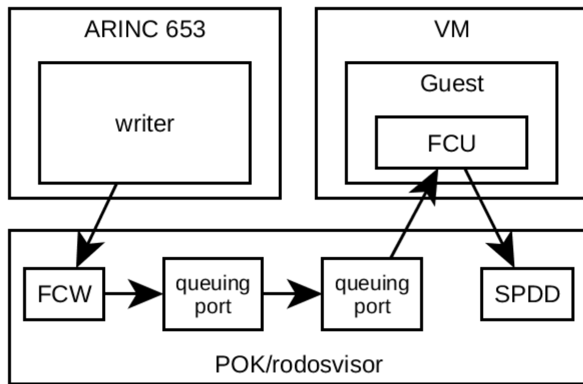

Fig. 14. The resulting architecture for configuration VM on the serial port device driver's use case.

Similarly to the reference configuration, from all the modified configurations described above, the average number of CPU clock cycles per byte required to send data to the serial port for various sizes of data has been measured. The results are presented in Fig. 15. It can be seen that, after functionality farming, for data sizes larger than the serial port's transmit buffer, as the size of the data increases, the cost per byte decreases, indicating that scalability is good. Furthermore, it can be seen that the different configurations display very approximate results.

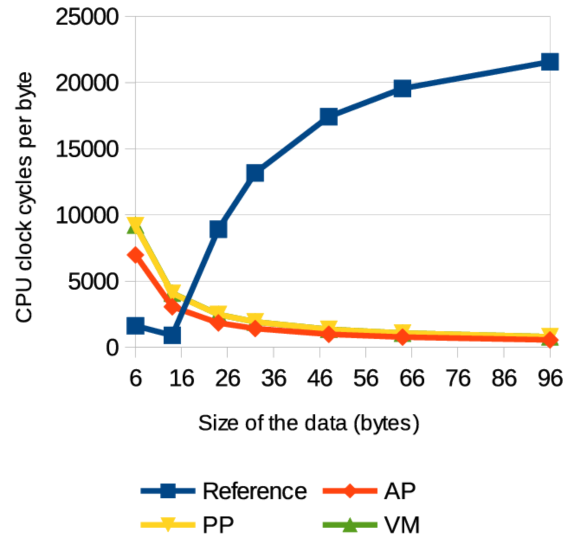

Fig. 15. Comparison of the average number of CPU clock cycles per byte required to send data to the serial port for different sizes of data between the reference configuration and after functionality farming. “VM” is barely seen because it is overlapped by “PP.”

In Fig. 16, the kernel's footprint for the reference and all the modified configurations is presented. It can be seen that the footprint of all the modified configurations is higher than the reference configuration's. This was expected for configurations AP and PP, since the function's implementation is not moved out of the kernel; the added footprint is due to a larger size of the code, and a larger size of the stacks because of the additional partition/worker. For configuration VM, where the function's implementation is moved out of the kernel, the added footprint is much higher than the size of the function's implementation, and thus, overall the footprint is higher than the reference configuration's; the added footprint is mostly due to the hypervisor (code, read-only, and read/write data), and due to a larger size of the stacks.

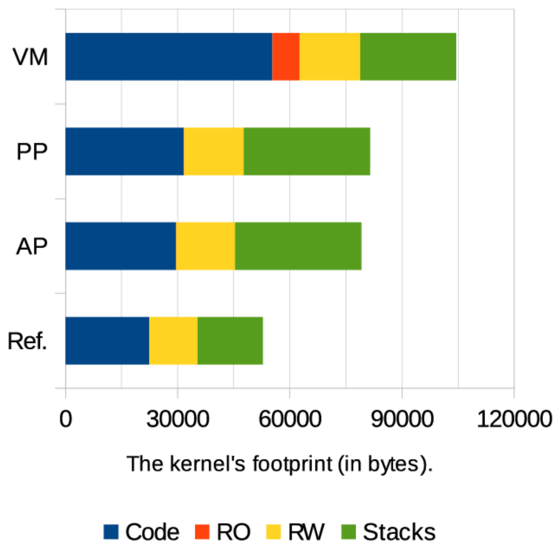

Fig. 16. The kernel's footprint for the reference and all the modified configurations in the serial port device driver's use case, in terms of the size of the code, read-only (RO) and read-write (RW) data, as well as the size of the stacks.

Lastly, in terms of the engineering effort, the reference configuration (i.e., the output from Ocarina) consists of 393 source lines of code (SLOC). As explained in section 3, a POK/rodosvisor configuration is composed by the implementation and configuration of a POK/rodosvisor kernel, the implementation and configuration of the partitions/workers, as well as the implementation of the configuration's build system. The three modified configurations (i.e., the output from FF-AUTO consist of at least 369 new/modified SLOC when compared with the reference configuration. Using FF-AUTO, 4 SLOC were required for the FFC file and, in the worst case, an additional 14 new/modified SLOC were also required. Knowing that, if functionality farming was performed manually, 369 new/modified SLOC would be required, and that, using FF-AUTO, only 18 SLOC were required, then, FF-AUTO enabled a reduction of engineering effort by more than 20 times.

7. USE CASE: INTER-PARTITION COMMUNICATION SUBSYSTEM

In this section, similarly to the previous section, first, a POK/rodosvisor configuration (i.e., the reference configuration) is described and it is demonstrated that it reveals a limitation in the design of POK/rodosvisor's inter-partition communication subsystem. Second, it is described how functionality farming has been applied to the reference configuration, and how it is expected to address the limitation identified earlier. Third and last, it is demonstrated that functionality farming addresses that limitation by comparing the performance before and after functionality farming. Even though functionality farming is used to address a limitation in the design of POK/rodosvisor, we do not intend to claim that it is the only or the best way to do it. Our goal is only to demonstrate another use case for functionality farming.

The reference configuration is illustrated in Fig. 17. It consists of a POK/rodosvisor-based system with two ARINC 653 partitions which communicate with each other through a single queuing communication channel. For each major frame, the “sender” sends data by writing to a sending queuing port, while the “receiver” receives data by reading a corresponding receiving queuing port. The size of the data is never larger than the size of the sending and receiving ports' buffer size.

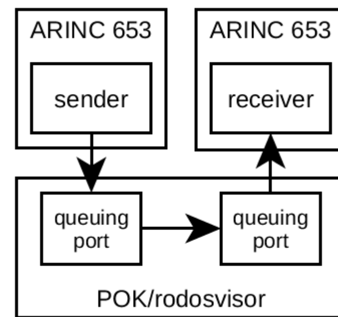

Fig. 17. The reference architecture used in the inter-partition communication subsystem's use case.

From the configuration just described, the scheduling jitter has been measured for different sizes of the data that are transmitted between “sender” and “receiver.” To measure the scheduling jitter, the output of the partitions' context switch times was enabled and the configuration ran until 300 samples were collected. To obtain the scheduling jitter, the expected context switch times were subtracted from the measured context switch times. The average scheduling jitter was obtained by averaging the results from all the samples.
Fig. 18 shows the scheduling jitter for the “receiver” and the “sender,” for different sizes of transmitted data. It can be seen that, the larger the size of the data, the larger the scheduling jitter of the “receiver.” The scheduling jitter of the “sender,” on the other end, is independent of the size of the data.

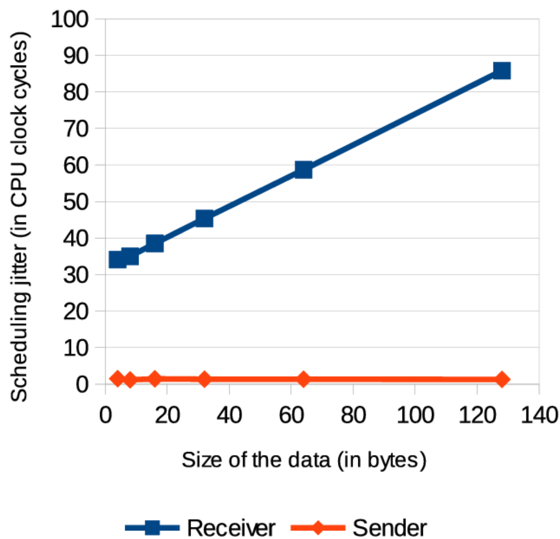

Fig. 18. The average scheduling jitter for the "receiver" and the "sender."

Further investigation revealed that the issue identified earlier could be traced back to a function in POK/rodosvisor named "pok_port_flushall." This function moves (i.e., flushes) the data in all sending ports into the corresponding receiving ports; the higher the size of transmitted data, the higher its execution time. It is called at the beginning of a major frame, and thus, at the beginning of the first partition window in the major frame (i.e., the partition window of the "receiver"). Its execution time, therefore, overwrites the first partition window in the major frame, which as demonstrated leads to high scheduling jitter for the "receiver."

To solve this problem, the application of functionality farming has been considered. More specifically, to farm "pok_port_flushall" into a dedicated worker, and allocate it into a predefined slot in the major frame, such that it does not overwrite other partitions' execution time. Functionality farming has been applied using FF-AUTO and three FFC files, which specify that "pok_port_flushall" shall be farmed into one worker with a single worker thread. One FFC file, shown in Fig. 19, specifies a worker based on an ARINC 653 partition (configuration AP). The other two FFC files specified a privileged-partition-based worker (configuration PP) and a virtual-machine-based worker (configuration VM). Since "pok_port_flushall" has shared dependencies with the kernel, as explained in section 5, function call farming has been specified for all configurations. For all configurations, the modified POK/rodosvisor configuration (i.e., the output from FF-AUTO) has been manually modified in order to accommodate the worker and its worker thread in the scheduler's configuration. Furthermore, because "pok_port_flushall" is part of the implementation of the inter-partition communication subsystem, on which functionality farming depends on, the output from FF-AUTO has also been manually modified such that, instead of a queuing communication channel, counting semaphores are used to serialize calls to "pok_port_flushall," requiring 15 new/modified SLOC. Using a counting semaphore to serialize calls is possible because "pok_port_flushall" accepts no parameters.

```
%system test::node.impl; test::ppc.impl; cpu
%worker worker_1; arinc653
%worker_thread worker_thread_1; worker_1
void pok_port_flushall(); worker_thread_1; \
                                 call-only
```

Fig. 19. FFC file for farming "pok_port_flushall" on a worker based on an ARINC 653 partition.

Similarly to the reference configuration, for all the modified configurations described above, the scheduling jitter has been measured for different sizes of transmitted data. The results are shown in Fig. 20. It can be seen that, after functionality farming, the scheduling jitter of the first partition in the major frame is equivalent to the scheduling jitter of other partitions in the major frame, and it is independent of the size of the data that is transmitted. These results demonstrate that functionality farming addressed the limitation identified earlier.

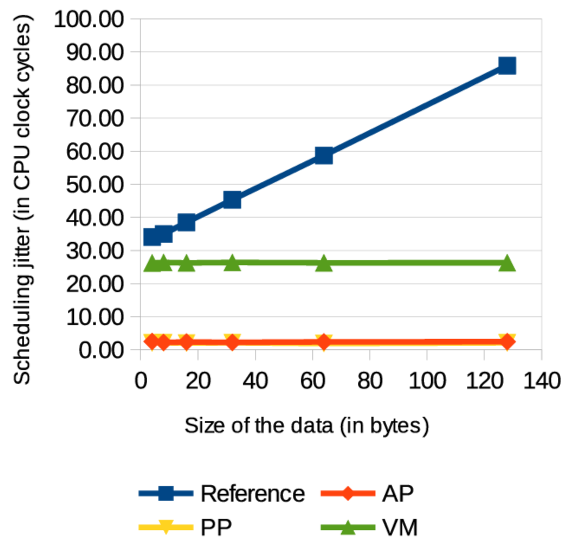

Fig. 20. The average scheduling jitter for the first partition in the major frame on the reference and all the modified configurations. "PP" is barely seen as it is overlapped by "AP."

In Fig. 21, the kernel's footprint for the reference and modified configurations is presented, in terms of the size of the code, read-only and read-write data, as well as the

size of the stacks. It can be seen that, as expected, the footprint for all the modified configurations is higher than the reference configuration's footprint, since only function call farming has been performed. AP's and PP's larger footprint is mostly due to a larger size of the stacks because of an additional partition/worker. VM's large footprint, on the other end, is due to the size of hypervisor (code, read-only, and read-write data) as well as due to a slightly larger size of the stacks.

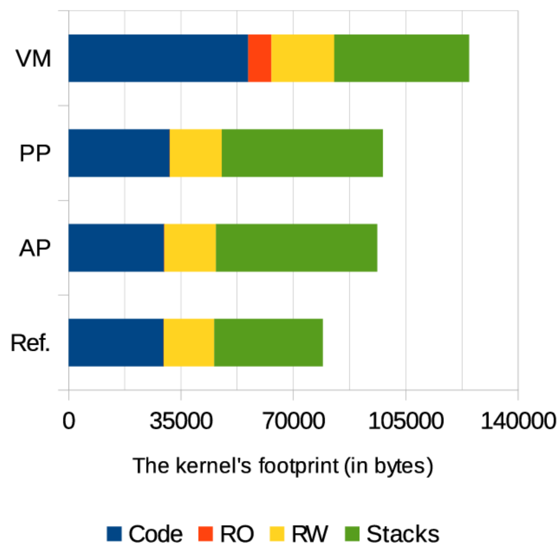

Fig. 21. The kernel's footprint for the reference and all the modified configurations in the inter-partition communication subsystem's use case, in terms of the size of the code, read-only (RO) and read-write (RW) data, as well as the size of the stacks.

Finally, in terms of the engineering effort, the reference configuration consists of 761 source lines of code (SLOC). The three modified configurations, on the other end, consist of at least 279 new/modified SLOC when compared with the reference configuration. Using FF-AUTO, 4 SLOC were required for the FFC file and, in the worst case, an additional 19 new/modified SLOC were also required. Knowing that, if functionality farming was performed manually, 279 new/modified SLOC would be required, and that, using FF-AUTO, only 23 SLOC were required, then, FF-AUTO enabled a reduction of engineering effort by more than 10 times.

8. CONCLUSION

This paper presented functionality farming and FF-AUTO. The former, functionality farming, consists in time and space partitioning an existing kernel, thus reducing its size and tackling the source of the problem with most operating systems today (i.e., the large size of the kernel). It has been explained that it depends on a lower upfront investment and it is also a more agile approach. The latter, FF-AUTO, performs functionality farming semi-automatically in POK/rodosvisor. With FF-AUTO, the engineering effort, and thus, the risk associated with functionality farming is significantly reduced, making it an ideal tool for design space exploration. This paper also presented two use cases which demonstrate how functionality farming is able to improve the design of POK/rodosvisor, and how FF-AUTO enables a significant reduction of the engineering effort required, and thus, of the risk associated, making it an ideal tool for design space exploration.

Even though FF-AUTO, currently, only supports POK/rodosvisor, the underlying methodology can be applied to any other operating system.

As future work, we propose: (1) to extend the granularity that is currently supported, which limits the extent to which functionality farming is possible; (2) to address the limitations regarding functions with shared dependencies with the kernel; and at last, (3) to reduce the need to manually modify the POK/rodosvisor configuration output by FF-AUTO.

ACKNOWLEDGMENTS

This work was supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT (Fundação para a Ciência e Tecnologia) within the Project Scope: UID/CEC/00319/2013. The work of A. Carvalho was supported by FCT (grant SFRH/BD/81640/2011).

REFERENCES

[1] A. S. Tanenbaum, J. N. Herder, and H. Bos, “Can we make operating systems reliable and secure?,” Computer, vol. 39, no. 5, pp. 44–51, 2006.
[2] F. Armand and M. Gien, “A practical look at micro-kernels and virtual machine monitors,” in Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE, 2009, pp. 1–7.
[3] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters, “Towards trustworthy computing systems: taking microkernels to the next level,” SIGOPS Oper. Syst. Rev., vol. 41, no. 4, pp. 3–11, Jul. 2007.
[4] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro, “Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors,” in Proceedings of the 11th workshop on ACM SIGOPS European workshop, 2004, p. 22.
[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An Empirical Study of Operating Systems Errors,” in Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, New York, NY, USA, 2001, pp. 73–88.
[6] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, “Terra: A virtual machine-based platform for

trusted computing," in ACM SIGOPS Operating Systems Review, 2003, vol. 37, pp. 193–206.
[7] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal Verification of an OS Kernel," in Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, New York, NY, USA, 2009, pp. 207–220.
[8] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz, "Tessellation: Space-time partitioning in a manycore client OS," in Proceedings of the First USENIX conference on Hot topics in parallelism, 2009, pp. 10–10.
[9] G. Heiser and B. Leslie, "The OKL4 microvisor: convergence point of microkernels and hypervisors," in Proceedings of the first ACM asia-pacific workshop on Workshop on systems, New York, NY, USA, 2010, pp. 19–24.
[10] H. Casanova, M. Kim, J. S. Plank, and J. J. Dongarra, "Adaptive scheduling for task farming with grid middleware," International Journal of High Performance Computing Applications, vol. 13, no. 3, pp. 231–240, 1999.
[11] C. Brown, V. Janjic, K. Hammond, H. Schoner, K. Idrees, and C. W. Glass, "Agricultural reform: more efficient farming using advanced parallel refactoring tools," in Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on, 2014, pp. 36–43.
[12] K. Molitorisz, "Pattern-based refactoring process of sequential source code," in Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, 2013, pp. 357–360.
[13] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. González-Vélez, P. Kilpatrick, R. Keller, M. Rossbory, and G. Shainer, "The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems," in Formal Methods for Components and Objects, 2013, pp. 218–236.
[14] D. Dig, "A refactoring approach to parallelism," Software, IEEE, vol. 28, no. 1, pp. 17–22, 2011.
[15] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting Applications into Reduced-Privilege Compartments.," in NSDI, 2008, vol. 8, pp. 309–322.
[16] S. F. Smith and M. Thober, "Refactoring programs to secure information flows," in Proceedings of the 2006 workshop on Programming languages and analysis for security, 2006, pp. 75–84.
[17] V. Ganapathy, T. Jaeger, and S. Jha, "Retrofitting legacy code for authorization policy enforcement," in Security and Privacy, 2006 IEEE Symposium on, 2006, p. 15–pp.
[18] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in USENIX Security Symposium, 2004, pp. 57–72.
[19] L. Yu and S. Ramaswamy, "Improving Modularity by Refactoring Code Clones: A Feasibility Study on Linux," SIGSOFT Softw. Eng. Notes, vol. 33, no. 2, pp. 9:1–9:5, Mar. 2008.
[20] J. Wloka, M. Sridharan, and F. Tip, "Refactoring for reentrancy," in Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2009, pp. 173–182.
[21] "653P3A AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE,PART 3A, CONFORMITY TEST SPECIFICATION FOR ARINC 653 REQUIRED SERVICES." [Online]. Available: http://store.aviation-ia.com/cf/store/catalog_detail.cfm?item_id=2189
[22] "POK homepage: home." [Online]. Available: http://pok.safety-critical.net/
[23] A. Tavares, A. Carvalho, P. Rodrigues, P. Garcia, T. Gomes, J. Cabral, P. Cardoso, S. Montenegro, and M. Ekpanyapong, "A customizable and ARINC 653 quasi-compliant hypervisor," in 2012 IEEE International Conference on Industrial Technology (ICIT), 2012, pp. 140–147.
[24] A. Carvalho, F. Afonso, P. Cardoso, J. Cabral, M. Ekpanyapong, S. Montenegro, and A. Tavares, "Cache full-virtualization for the PowerPC 405-S," presented at the 11th IEEE International Conference on Industrial Informatics, Bochum, Germany, 2013.
[25] A. Tavares, A. Didimo, S. Montenegro, T. Gomes, J. Cabral, P. Cardoso, and M. Ekpanyapong, "RodosVisor - an Object-Oriented and Customizable Hypervisor: The CPU Virtualization," presented at the Conference on Embedded Systems, Computational Intelligence and Telematics in Control (CESCIT), University of Würzburg, Germany, 2012, pp. 200–205.
[26] IBM, PowerPC 405-S Embedded Processor Core User's Manual, 1.2 ed. 2010.
[27] SAE International, AS5506B: Architecture Analysis & Design Language (AADL). 2012.
[28] "OpenAADL/ocarina," GitHub. [Online]. Available: https://github.com/OpenAADL/ocarina
[29] V. Chipounov and G. Candea, "Reverse Engineering of Binary Device Drivers with RevNIC," in Proceedings of the 5th European Conference on Computer Systems, New York, NY, USA, 2010, pp. 167–180.