

MULTIKERNEL ARCHITECTURE AND VIRTUAL MACHINES

TECHNICAL FIELD

The present disclosure relates to a multikernel architecture and virtual machines.

- 5 Embodiments relate particularly to server architectures adapted for provision of services over network connections, in particular cloud services.

BACKGROUND

- Cloud computing involves the on-demand availability of computer system resources without
10 direct active management by the user – cloud services are services provided to a user using a cloud computing model. Cloud services are typically provided to a user through a cloud server – this will typically involve the provisioning and use of a virtual machine to provide the cloud service to the user. Certain cloud services contain sensitive information, and may need to operate in network environments where there is a risk of compromise. For such cloud
15 services, it is particularly desirable for the cloud services to be provably secure, and for the privacy of a user's data to be guaranteed. It is also desirable for the user space for a cloud server to operate dynamically, rather than to be fully defined statically at compile time. This poses a challenge for cloud services that need to be provably secure.

20 SUMMARY

- In a first aspect, the invention provides a computer system having a multikernel architecture comprising a plurality of operating system microkernels, the computer system comprising: a plurality of processing cores; a memory comprising a plurality of private memory areas and a shared memory area; and a plurality of computing nodes each comprising one or more of the
25 processing cores and further comprising one of the plurality of operating system microkernels and an associated user space for non-kernel operations, wherein only the operating system microkernel runs at a highest privilege level of the processing core, or processing cores, of a

computing node; wherein each of the plurality of computing nodes has an associated private memory area from the plurality of private memory areas for use by its operating system microkernel, and wherein for non-kernel operations each of the plurality of computing nodes accesses the shared memory area.

- 5 In embodiments, the computer system may be adapted to partition system hardware statically into nodes at boot time.

In embodiments, one of said plurality of computing nodes may be a management node for managing user nodes, and one or more other computing nodes may be user nodes for running user payloads. In some embodiments, a plurality of computing nodes may be management
10 nodes. In some embodiments, the computer system may be adapted to provide one or more virtual machines for users. In some cases, no more than one virtual machine is implemented in any user node. In other cases, two or more mutually trusted virtual machines may be implemented in one or more user nodes. One or more virtual machines may be implemented using multiple user nodes. In some embodiments, the computer system may be adapted to
15 operate as a cloud server.

The management node may comprises a manager component and each user node may then comprise an agent, wherein the node management component is adapted to instruct the agent of a user node to create, destroy or modify virtual machines in that user node. In some
20 embodiments, communication between user nodes is not privileged – in the sense that it is not at the level of privilege, between privileged components, but communication between the manager component and the one or more agents is privileged, in the sense that it is communication between privileged components. Direct communication between nodes may then be limited to direct communication between the user spaces of those nodes.

25 In embodiments, each user node may have a virtual machine manager component for managing a virtual machine in that user node by interaction with the manager component. In certain cases, a virtual machine manager is provided for each virtual machine in a user node. Each user node may comprise a console driver and the management node may comprise a
30 console relay for communication between the management node and virtual machine

managers. Each virtual machine manager component may be statically defined at boot time for subsequent configuration.

- In some embodiments, the computer system further comprises one or more network devices, wherein the management node then comprises a physical function device driver for each of the one or more network devices for physical interaction with the associated network device; and wherein one or more of the user nodes then comprises a virtual function device driver for a virtual function network device corresponding to one of the one or more network devices, wherein the virtual function device driver provides a virtual network device function in said user node. Such a physical function device driver may be provided by an SR-IOV mechanism. In some cases, the computer system may further comprise one or more network devices, wherein one or more of the network devices are storage devices, and data may then be encrypted for communication between the virtual machines and the storage devices.
- 15 In embodiments, the computer system further comprises a remote administration interface comprising a dedicated network link. In embodiments, each microkernel may have proven implementation correctness. In embodiments, each microkernel may be a third-generation microkernel, such as a seL4 microkernel.
- 20 In a second aspect, the invention provides a method of operating a computer system comprising a plurality of processing cores and a memory to implement a multikernel architecture, the method comprising: defining a management computing node having one or more processing cores, booting the management computing node and statically partitioning processing cores into a plurality of further computing nodes, each computing node comprising one or more processing cores and having an operating system microkernel and an associated user space, wherein only the operating system microkernel runs at a highest privilege level of the processing core, or processing cores, of a computing node; and booting and provisioning the further computing nodes as user nodes.
- 25 30 In embodiments, this may further comprise establishing the memory as a plurality of private memory areas and a shared memory area, wherein each of the plurality of computing nodes may then be provided with an associated private memory area from the plurality of private memory areas for use by its operating system microkernel, and wherein some or all of the

plurality of computing nodes may then access the shared memory area for non-kernel operations. Such user nodes may be adapted to provide one or more virtual machines for users. In some cases, no more than one virtual machine may be implemented in any user node. In other cases, two or more virtual machines may be implemented in one or more user
5 nodes. One or more virtual machines may be implemented using multiple user nodes.

In embodiments, the computer system may be adapted to operate as a cloud server.

In embodiments, the management node may comprise a manager component and each user
10 node may comprise an agent, wherein the method may then further comprise the node management component instructing the agent of a user node to create, destroy or modify virtual machines in that user node.

Each user node may be provided with a virtual machine manager component for managing
15 virtual machines in that user node by interaction with the manager component. In embodiments, the method may further comprise statically defining each virtual machine manager component at boot time. In embodiments, the method may further comprise configuring the virtual machine manager component of a user node on creating a virtual machine in that user node.

20

The computer system may further comprise one or more network devices, wherein the management node may comprise a physical function device driver for each of the one or more network devices for physical interaction with the associated network device; and wherein one or more of the user nodes may comprise a virtual function device driver for interaction with a
25 virtual function network device corresponding to one of the one or more network devices, the method may then further comprise the virtual function device driver providing a virtual network device function in said user node. In a further such case, one or more of the network devices may be storage devices, and the method may further comprise encrypting data for communication between the virtual machines and the storage devices.

30

In embodiments, all communication between nodes may be performed by user spaces components of the nodes. Any privileged communication between nodes may be between a

manager node and another node, with privileged communication including messages and commands that would result in a reconfiguration of some components in the node.

5 In a third aspect, there is provided a method of providing dynamic user spaces in a computer system, wherein the computer system is adapted for virtualisation, the method comprising: statically defining a plurality of computing nodes, each said computing node comprising an operating system kernel, wherein one of the plurality of computing nodes is a management node and one or more other computing nodes of the plurality of computing nodes are user nodes; wherein the management node comprises a manager component for management of
10 user nodes, and wherein the or each user node comprises an agent; and wherein a dynamic user space is provided in a user node by the manager component instructing the agent of a user node to provide creation, destruction or modification of virtual machines in that user node.

15 In embodiments, the operating system kernel of a computing node may be a microkernel. In embodiments, in a computing node, only the operating system kernel runs at a highest privilege level.

In embodiments, there may be a plurality of management nodes. In embodiments, the
20 manager component may be adapted to receive management requests from a remote administration interface comprising a dedicated network link. In embodiments, the manager component may determine the allocation of all memory for a virtual machine.

In embodiments, the method may further comprise the creation of a virtual machine in a user
25 node by instruction from the manager component, the instruction comprising identification of memory to be used by the virtual machine and physical devices to be passed-through to the virtual machine. The virtual machine may be created in halting state. The method may further comprise the manager component instructing the agent to start the virtual machine.

30 In embodiments, the user node may comprise a virtual machine manager component, and the method may further comprise configuring the virtual machine manager component for the virtual machine to be created. Such embodiments may further comprise creating the virtual

machine manager component statically on system boot as a shell component for configuration on creation of a virtual machine.

In embodiments, each user node may comprise a console driver and the management node
5 may comprise a console relay for communication between the management node and virtual machine managers.

Communication between user nodes may not be privileged, but communication between the manager component and the one or more agents may be privileged, in the sense of being
10 between privileged components. Direct communication between nodes may be limited to direct communication between the user spaces of those nodes.

In embodiments, the computer system may further comprising one or more network devices, wherein the management node may comprise a physical function device driver for each of the
15 one or more network devices for physical interaction with the associated network device; the method may then further comprise providing at one or more user nodes a virtual function device driver for a virtual function network device corresponding to one of the one or more network devices, wherein the virtual function device driver may then provide a virtual network device function in said user node. The physical function device driver may be provided by an
20 SR-IOV mechanism. One or more of the network devices may be storage devices, and the method may further comprise encrypting data for communication between the virtual machines and the storage devices.

In a fourth aspect, the invention provides a computer system adapted for virtualisation using
25 dynamic user spaces, the computer system comprising: a plurality of computing nodes, each said computing node comprising an operating system kernel, wherein one of the plurality of computing nodes is a management node and one or more other computing nodes of the plurality of computing nodes are user nodes; the management node comprising a manager component for management of user nodes, and wherein the or each user node comprises an
30 agent; and wherein the computer system is adapted to provide a dynamic user space is provided in a user node by the manager component instructing the agent of a user node to provide creation, destruction or modification of virtual machines in that user node.

Each computing node may comprise one or more processing cores. The operating system kernel may be a microkernel. In some embodiments, only the operating system kernel runs at a highest privilege level.

- 5 Communication between user nodes may not be privileged, but communication between the manager component and the one or more agents may be privileged. Direct communication between nodes may be limited to direct communication between user space components of those nodes.

In a fifth aspect, the invention provides a computer system having a multikernel architecture comprising a plurality of operating system kernels, wherein the computer system is adapted for provision and management of virtual machines, the computer system comprising: a plurality of processing cores; a plurality of computing nodes, wherein each computing node is assigned one or more processing cores and wherein each computing node comprises an operating system kernel, and wherein one of the plurality of computing nodes is a management node for management of user nodes and one or more other computing nodes of the plurality of computing nodes are user nodes; wherein the management node comprises a manager component, and wherein the user nodes each comprise a virtual machine manager component for managing virtual machines in that user node by interaction with the manager component.

- 10 20 In embodiments, each virtual machine manager component may be statically defined at boot time for subsequent configuration. The virtual machine manager component may be statically defined as a shell for population on creation of a virtual machine in the user node.

In embodiments, a virtual machine manager may be provided for each virtual machine in a user node. In embodiments, the computing system may comprise a plurality of management nodes. In embodiments, each user node may comprise a console driver and the management node may comprise a console relay for communication between the management node and virtual machine managers.

- 25 30 In embodiments, the management node may comprise a manager component and each user node may comprise an agent, wherein the node management component is adapted to instruct the agent of a user node to create, destroy or modify virtual machines in that user

node. The agent of a user node may then be adapted to configure the virtual machine manager component for that user node on creation of a virtual machine.

In embodiments, the operation system kernels of computing nodes are microkernels. In embodiments, only the microkernels may run at a highest privilege level of the CPU cores. In
5 such a case, the virtual machine manager components may not run at the highest privilege level.

In a sixth aspect, the invention provides a computer system adapted for virtualisation with use of network devices, the computer system comprising: a plurality of processing cores; a plurality of computing nodes, wherein each computing node is assigned one or more processing cores
10 and wherein each computing node comprises an operating system kernel, and wherein one of the plurality of computing nodes is a management node and one or more other computing nodes of the plurality of computing nodes are user nodes; and one or more network devices; wherein the management node comprises a physical function device driver for each of the one or more network devices for physical interaction with the associated network device; and
15 wherein one or more of the user nodes comprises a virtual function device driver for a virtual function network device corresponding to one of the one or more network devices, wherein the virtual function device driver provides a virtual network device function in said user node.

In embodiments, the physical function device driver may be provided by an SR-IOV mechanism. In some cases, one or more of the network devices may be storage devices, and
20 data may be encrypted for communication between the virtual machines and the storage devices.

In embodiments, the operation system kernels of computing nodes may be microkernels. In such cases, in some embodiments only the microkernels may run at a highest privilege level of the CPU cores.

25 In certain embodiments, the computing system comprises a plurality of management nodes.

BRIEF DESCRIPTION OF THE DRAWINGS

One or more embodiments of the disclosure will now be described, by way of example only, with reference to the accompanying drawings, in which:

Figure 1 compares a monolithic and a microkernel architecture;

Figure 2 illustrates a prior art operating system using a multikernel architecture; and

Figure 3 illustrates an operating system according to an embodiment of the invention;

Figure 4 shows the system of Figure 3 after a single VM has been created on Node 1;

5 Figure 5 shows the system of Figure 3 where a VM is created using two nodes;

Figure 6 shows a booting strategy for use in the embodiment of Figure 3;

Figure 7 illustrates a cross-node communication link used in certain embodiments of the invention;

Figure 8 illustrates the management of network devices in the embodiment of Figure 3;

10 Figure 9 illustrates the use of external storage in the embodiment of Figure 3;

Figure 10 illustrates a structure for remote administration in the embodiment of Figure 3;

Figure 11 illustrates use of a PKI for remote administration in connection with the arrangement of Figure 10;

Figure 12 illustrates an operating system according to an alternative embodiment of the
15 invention;

Figure 13 illustrates a hybrid static/dynamic software stack implementable in the server architecture illustrated in Figure 12; and

Figure 14 illustrates management of a virtual machine spanning two nodes.

20 DETAILED DESCRIPTION

General and specific embodiments of the disclosure will be described below with reference to the Figures.

The present inventors have appreciated that to achieve provably secure cloud services and to guarantee user data privacy, a cloud server design based on a multikernel architecture using microkernels will be particularly effective. First of all, microkernels will be described, with particular reference to the seL4 operating system microkernel used in embodiments of the invention. Multikernel architectures using microkernels will then be discussed, together with a hypervisor designed for creation and management of virtual machines on such an architecture. Specific embodiments according to embodiments of the invention will then be described, relating to both architecture implementation options and associated hypervisor design. As will be indicated below, these are particularly suitable for use for a cloud server providing cloud services will then be described.

For reference throughout this specification, the following term definitions will be used – for each term, the first sentence is the working definition, with further sentences providing explanation as necessary:

Kernel - A kernel is the core of a computer system's operating system (OS) – it typically controls everything within that computer system, mitigates conflicts between processes, and facilitates interactions between components. A kernel is low-level system software – executing in a higher privilege mode of the CPU, it controls a computer system's resources, and it enforces security. In most systems, a kernel controls all hardware resources (such as I/O, memory, and cryptographic services) via device drivers, arbitrates conflicts between processes concerning such resources, and optimizes the utilization of common resources (such as CPU and cache usage, file management and network sockets).

Resources - This comprises anything that may be used by or in a computer system, and unless specified otherwise, includes CPU cores, memory regions and devices.

Node - This is a set of resources under the exclusive control of a single OS kernel.

Multikernel - This is an OS architecture that consists of a plurality of kernels operating concurrently. These kernels will have varying degrees of independence from each other in different architectures (or in different arrangements).

- 5 Microkernel - This is an OS architecture comprising a minimal amount of software executing with high CPU privilege level to implement an operating system, typically including address space management, thread management and inter-process communication. A large number of components typically hosted within a kernel (such as device drivers, file systems, and networking) are instead implemented in user space and executing with low CPU privilege level.
- User Space - This comprises a set of OS components running at low CPU privilege level (e.g. x86 ring 3, ARM EL0), as opposed the OS kernel, which is the only component running at high CPU privilege level (e.g. x86 ring 0, ARM EL1/EL2).
- 10 Symmetric Multi-Processing (SMP) - This is an OS architecture where a single OS runs on multiple CPU cores.
- Virtual Machine (VM) - A virtualized computer system, managed by a virtual machine monitor (VMM).
- 15 Virtual Machine Monitor (VMM) - System and process for supporting the management and execution of virtual machines (part of the virtualization stack). The VMM emulates those parts of the hardware that aren't self virtualising – for example, by implementing virtual network interfaces in software. This contrasts with self-virtualising hardware, such as an SR-IOV network interface which presents independent isolated personalities to each VM.
- 20 Hypervisor - System and process for management of kernel-space functionality for a system of virtual machines (part of the virtualization stack). The hypervisor generally executes at an even higher privilege level than an OS kernel. Its main task is configuring those parts of the system that are capable of self virtualisation – for example, by dividing memory between virtual machines via 2-stage translation.
- 25 By extension, the term Hypervisor is also used to describe a complete virtualisation stack.
- Private resources - Resources only accessible to a single node.
- Shared resources – Resources accessible to one or more nodes.

Privileged communication – Communication between privileged components.

Unlike applications, the kernel has exclusive access to a more privileged execution mode of the processor – kernel mode – that gives it direct access to hardware. Applications only ever execute in user mode, and they can only access hardware as permitted by the kernel. The role of the kernel is therefore of particular importance to secure operation. In a conventional operating system such as Windows or Linux, a large and complex monolithic kernel is used. The control of so much functionality through the kernel has some benefits for application programming – it means that the ways in which the application can access computer system resources can be very well defined – but it also has significant disadvantages. Such kernels can be difficult to verify and prone to errors, and the existence of such errors can easily lead to full system compromise. A different approach involves the use of microkernels. A microkernel only implements the minimal fundamental primitives that allow for an operating system to be built upon it, reducing the amount of code that executes at higher privilege to a minimum.

Figure 1 shows how these architectures compare. The conventional kernel architecture has a large kernel 1 sitting over the hardware layer 3 with stacked functionalities (here from device drivers at the lowest part 1a of the stack, to a virtual file system at the top layer 1d), and applications are built over this in an application layer 2, the applications accessing the kernel 1 through syscalls – only the applications are in user mode 7, with the large number of functionalities handled by the kernel operating in kernel mode 8. The microkernel architecture has a much smaller kernel 10 handling virtual memory and inter process communication – all other functionalities, such as not only applications 12 but functionalities that would be in the kernel of a conventional operating system (such as device drivers 13 and servers such as Unix server 14 and file server 15) are built over the microkernel 10, communicating with it using inter process communication, and these functionalities operate in user mode 7 and do not operate in kernel mode 8 or have kernel mode privilege.

Embodiments of the disclosure use the seL4 operating system microkernel – seL4 is available as open source code from GitHub and is supported by the seL4 foundation – seL4 and its development is further described at <https://sel4.systems>. seL4 was developed as a third-generation microkernel to be a basis for highly secure and reliable systems. Third-generation microkernels are characterised by a security-oriented API with resource access controlled

by capabilities, virtualization as a primary concern, novel approaches to kernel resource management, and are intended to be suitable for formal analysis. seL4 is formally proven to have implementation correctness (there exists a continuously maintained machine-checked proof of this), and if correctly configured, can guarantee the following properties:

- 5 • Confidentiality – seL4 will not allow any entity without read access to data to read or infer that data;
- Integrity – seL4 will not allow any entity to modify data unless it has write access to that data; and
- 10 • Availability – seL4 will not allow an entity to prevent another entity's authorised use of resources.

seL4 provides a small set of primitives for securely multiplexing hardware resources – there is as little policy embodied within these primitives as possible. The kernel provides mechanisms that allow operations which require elevated privilege on a given machine to be invoked according to a policy defined by unprivileged components. The kernel's security
15 mechanisms are constructed such that mutually-untrusting components can manage disjoint sets of resources without interference.

Every resource in the system is managed as an object, and objects are named by capabilities representing rights over them. seL4 uses “capabilities” for access control. Capabilities are access tokens that support very fine-grained control over which entity can access a particular
20 resource in a system – capabilities name both the object of the operation and establish the caller's authority to perform the operation. Capabilities support strong security according to the principle of least privilege – a user is given the minimum access or permission needed to perform their job functions. seL4 identifies the unit of authority with the unit of scheduling – the thread. Each thread has a defined scope of authority, and threads will typically initiate
25 kernel operations by invoking their capabilities. This core design principle for secure systems is not in practice achievable with the types of access control that need to be used with conventional operating systems such as Linux or Windows.

seL4 is a particularly suitable choice for use as a microkernel in the context of cloud service provision as it is capability-based and formally verified (and so secure), but also fast. The
30 principle machine resources managed by the kernel are CPU time and RAM. Access to the CPU is abstracted by kernel-visible threads, and RAM as frames. The seL4 kernel is designed

for lightweight execution, with a minimum of kernel state outside explicit kernel objects. The kernel itself uses an atomic, single-stack design. Invocations either fail without modification to user-visible state or run to completion before any user thread is scheduled (there are very limited exceptions to this for certain long-running operations, but even here state is stored in explicit kernel objects). Consequently from user space, kernel execution simply appears as a series of atomic steps, without need to assess partially-completed operations or to store kernel execution context. The kernel uses a single, statically allocated stack that is simply overwritten on each invocation. The architecture relies on a single thread of execution within the kernel itself.

While the seL4 microkernel was initially designed for single core systems, it has subsequently been developed for multiple cores. The support for Symmetric Multi Processing (SMP) systems was introduced later as multicore CPUs became more available. To achieve this where there is a single thread of execution with a kernel, a lock strategy is required. The SMP strategy employed by seL4 is a so-called Big Kernel Lock (BKL), as opposed to finer grain locks used by monolithic kernels such as Linux or BSDs. Finer grain locking mechanisms try to maximise performance by identifying all critical sections, narrowing them down to the shorter code path, and protecting them with dedicated locks such as mutexes and spin locks. A BKL on the other hand tries to minimise the complexity of managing execution concurrency by taking a global lock on every system call entry point and releasing it when the system call exits.

It has been found that for a microkernel such as seL4, the simple BKL mechanism is sufficient to enable scaling a system to 4 and even 8 cores depending on the load case. This renders it suitable for use with embedded systems, but it does not appear suitable for a cloud server application. Modern cloud servers use CPUs with a much larger number of cores, by an order of magnitude, and on such systems the BKL mechanism would not perform well as most CPU cores would be competing for the global lock.

Another difficulty is that the verification framework that is used to verify the seL4 kernel does not currently support execution concurrency - that is, it requires that the system memory is only alterable by one single thread of execution. On an SMP system, the memory can be modified independently by all concurrently running threads of execution (each CPU core), making this configuration unverifiable with the current framework. While the BKL mechanism

of seL4 would seem to satisfy the requirement of non concurrency within the kernel, the code that acquires and releases the lock would not itself be verifiable. Furthermore, the proofs would anyway need to be modified to support multicore systems.

This leaves two types of configuration option for seL4:

- 5 • Single-core configuration that can run the formally verified kernel but can only use one core.
- Multicore configurations that cannot run the formally verified kernel but can reasonably scale to 4 or even 8 cores.

Neither of these configurations would be considered suitable to build a secure and performant
10 cloud server: this requires a configuration that can scale to a much larger number of cores, and that can reuse the formal proofs of the single-core kernel. For effective use of a multicore system, it would be desirable to use not only multicore configurations for a kernel, but also to be able to use multiple kernels for different cores, or sets of cores. This constitutes a multikernel architecture. In a multikernel architecture, there will typically be local handling of
15 common and performance-critical operations without communication or synchronisation between compute units (such as discrete CPU cores or devices such as network interface cards (NICs)). A separate OS instance – for example, a seL4 microkernel – is run on each device. Consistency of state – such as for memory allocation – is then handled by distributed protocols operating outside the kernel.

20 This effective partitioning of an OS between cores does not prevent user-level applications, or virtual machines, from spanning multiple cores. A coordination protocol can be used to provide separate instantiation of an application on different kernels, with the application then executing as on any OS, with its threads coordinated via shared memory and inter-core signalling. Inter-kernel communication at the OS level is typically not used, with
25 communication between user nodes only performed by user space tasks.

Existing multikernel architectures include Barrelfish and the Clustered Multikernel. Barrelfish, shown in Figure 2, is an OS architecture that implements the Multikernel OS architecture proposed by the research group of Timothy Roscoe at ETH Zurich. Barrelfish was designed to solve the scalability issues of common SMP systems by treating the underlying hardware

as a distributed system, where each CPU core runs its own kernel instance and communicate with one another using explicit messages.

As shown in Figure 2, the Barrelfish implementation runs one kernel instance 21 per CPU core 22. Each kernel 21 manages the tasks running on its CPU core 22. User-space tasks can
5 communicate with other user-space tasks on all other kernels 21 to maintain the illusion of a single unified system – the CPU cores 22 are linked by physical interconnect 20 and communicate by asynchronous messaging 23. A complex algorithm runs on each OS node to efficiently maintaining a local state replica 25. The application programming model of such an OS is broadly similar to that of a conventional SMP system such as Linux: applications 24 are
10 created dynamically, threads may migrate from core to core, multi-threaded applications may even have threads running concurrently on different cores.

The Clustered Multikernel (CMK) was proposed by Michael von Tessin in his 2013 thesis using a modified version of the seL4 kernel. In the CMK model the machine is divided into a number of clusters, each comprising one or more CPU cores and a fixed amount of system memory,
15 and an seL4 instance is spawned in each cluster. All inter-node communication happens outside of the seL4 kernels - these are managed directly by user-space tasks using memory shared between multiple nodes and Inter-Processor Interruptions (IPIs). This implementation assumes a static system where the hardware partitioning into clusters, the communication channels, and the user-space applications are all defined at compile time. This implementation
20 is much simpler than that of Barrelfish and its scope is essentially focused on the verification aspects.

The primary function of a cloud server software stack is to enable the dynamic partitioning of a large physical machine into a number of virtual machines (VMs) that can run end-user OS's as payloads. The VMs are assigned resources such as memory regions, exclusive physical
25 devices (pass-through), as well as virtual devices such as virtual network links and virtual storage devices. The basic unit of computing is therefore the VM, which is a much simpler object to manage than a classic user application: its resources and connections are defined at creation time and are fixed for the duration of its life cycle. The cloud server is an appliance that is remotely administered using an Application Programming Interface (API). A cloud
30 operator would typically manage a large number of such cloud servers in a datacenter and administer them as a fleet via a management console.

Such VMs would be managed by a suitable hypervisor – embodiments of the invention relate to provision of an architecture, and a hypervisor, suitable for provision of highly secure cloud services. Such a secure cloud hypervisor may be designed to execute mutually untrusted Virtual Machines (VMs) on top of bare metal cloud-scale hardware. Threats to be addressed
5 include VM escapes, hardware timing channels, and network compromise. Preferably, formal verification methods can be applied to prove implementation correctness as well as desired strong security properties, in particular the strict isolation of VMs.

The seL4 microkernel is particularly suitable for use for this purpose, as it comes with a formal, mathematical, machine-checked proof of implementation correctness. Specifically, seL4 has
10 a specification of what constitutes correct behaviour, a simplified model of the hardware, an implementation in C, and a machine-checkable proof (for a hardware model) that the C implementation corresponds to the specification of correct behaviour. It should be noted that embodiments of the invention are not limited to use of the seL4 microkernel, but that these properties render implementation using seL4 to be particularly suitable.

15 The seL4 microkernel and its ecosystem are historically focused on embedded and cyber-physical devices, which are small systems based on ARM or RISC-V architectures that have a few Central Processing Unit (CPU) cores, a few GBs of memory, and that are statically defined at compile time. The architecture of embodiments of the invention implemented using seL4 extends the applicability and formal guarantees of seL4 to large cloud server machines
20 running multiple virtual machines on behalf of cloud tenants.

Rather than trying to scale the seL4 kernel and its proofs to many cores, a hypervisor according to an embodiment of the invention uses a multikernel system architecture where multiple single-core seL4 instances run concurrently, each one bound to a single dedicated CPU core and using a small amount of private memory. In addition, a large shared pool of
25 memory is made available to client VMs. In this configuration, the individual kernels require no coordination and do not directly interact with one another. Such a multikernel architecture enables scaling of the system to a very large number of cores without compromising the validity of their formal verification proofs. The strict partitioning of system resources also provides protection against side-channel and rowhammer-type attacks.

In relation to this architecture, a distinction is drawn between the term *hypervisor*, used to refer to the complete kernel-mode system required to support VMs, and the term *Virtual Machine Monitor (VMM)*, which is an unprivileged task running on a core and supporting exactly one VM running on the same core. This design enables building a secure system with strong
5 isolation between VMs using untrusted and isolated VMMs. The memory regions backing the VMs are allocated from memory shared between all kernels. Multicore guest VMs are implemented using a distributed VMM where each VM's virtual CPU is handled by a separate VMM task running on a different physical CPU core.

Such a hypervisor may be remotely administered using an API on a dedicated network link. A
10 cloud operator would typically manage a large number of such cloud servers in a datacentre and administer them as a fleet via a management console. A hypervisor according to an embodiment of the invention may for example interface with storage servers using the Internet Small Computer Systems Interface (iSCSI) protocol on dedicated network links. All data sent to the storage servers is, in embodiments, transparently encrypted by the hypervisor. Network
15 access may be provided using dedicated network interfaces that support Virtual Functions (VFs) using Single Root - Input/Output Virtualisation (SR-IOV). The VMM may then implement the network interface VF device drivers and the iSCSI initiator, and expose VirtIO Block, Network, and console devices to the guest.

A primary function of such a hypervisor is to enable the dynamic partitioning of a large physical
20 machine into a number of VMs that can run end-user operating systems as payloads. The VMs are assigned resources such as CPU cores, private memory regions, exclusive physical devices (pass-through), as well as virtual devices such as virtual network links and virtual storage devices. The basic unit of computing is therefore the VM, which is a much simpler object to manage than a classic user application: its resources and interfaces with system
25 components are defined at creation time and do not change until the VM is shut down. A hypervisor according to embodiments of the invention uses multiple VMMs, each one running as an unprivileged user-space task and only managing a single VM's Virtual CPU (vCPU). This design enables building a secure system with strong isolation between VMs using untrusted and isolated VMMs.

30 In embodiments of the invention, a node-based approach can be used to provide a system architecture particularly suitable for cloud server implementation. This requires a suitable

- virtualisation platform, or hypervisor, to be provided. Using such an approach, the system hardware may be partitioned into nodes dynamically or statically at boot time. In the simplest version of this approach, single core instances of the seL4 microkernel are run on each CPU core. Each instance – the kernel and its user-space running on a single CPU core – is for
- 5 specific embodiments here termed a node. More generally, a node may be considered to be a set of resources under the exclusive control of a single OS kernel. Other approaches may be taken to node constructions – nodes may, for example, comprise one or more CPU cores, or specific nodes may be located separately (for example a manager node, as described below in relation to specific embodiments, may be provided on another device such as a PCIe card).
- 10 Nodes are given exclusive access to a certain subset of the system resources (e.g. physical memory and devices), as well as shared access to regions of memory used to implement communication channels between nodes and can share data between software components located in different nodes. The set of resources assigned to a node at creation time may be fixed for the life time of this node: in such an arrangement a node cannot gain access to more
- 15 resources during its life time. In other arrangements, nodes may be able to give or share resources provide that this can be done through kernel capabilities that can be formally verified.

This approach requires minimal change to the seL4 microkernel and it has limited impact on provability. It does require a suitable hypervisor to manage the virtualization process and a

20 virtual machine manager (which may be incorporated within the hypervisor) to manage virtual machines. Broad principles for a suitable hypervisor will now be described, before specific embodiments are discussed.

A suitable hypervisor provides for secure virtualisation built on the guarantees provided by the seL4 microkernel and its verification. This enables isolation between mutually-untrusting virtual

25 machines in a cloud hosting environment. To do this, the hypervisor will run on virtualisation-capable hardware maintained by a hosting provider. The hypervisor enables the system administrator to control the allocation of virtual CPUs to physical CPUs. It provides lowest-level system software, allowing mutually-untrusting VMs be collocated on the same physical hardware with the highest available degree of isolation. To do this, the hypervisor is a “bare-metal” hypervisor, multiplexing VMs on the shared hardware over only the hypervisor

30 implementation (and without a conventional operating system operating at a higher privilege

level). In this way, the guarantees provided by use of seL4 relate directly to the isolation of VMs.

The following choices are made in the embodiments described in detail below. There is a system of distributed virtual machine monitors (VMMs), with a single VM instantiated per VMM
5 (but possibly multiple VMMs per VM – one per virtual CPU core). If these are treated as a single subject for security purposes, it mitigates the risk of a VMM being compromised by an untrusted VM. Memory is divided into system memory – which is partitioned between nodes, and which may contain seL4 objects – and shared memory, which is accessible from every node, but which can only ever be used as mappable frames. Both types of memory are
10 represented by seL4 capabilities. System memory capabilities exist only on the node to which they belong – memory allocation is described in more detail below in relation to specific embodiments. System memory is used to create all kernel objects necessary for the operation of the system, while shared memory is provided for other uses.

For using physical resources, single-root input/output virtualization (SR-IOV) may be used to
15 allow a physical PCIe device to present itself multiple times allowing isolation between VMs. Interaction with external storage servers may be by the iSCSI protocol, with the contents exposed to VMs as VirtIO devices. The hypervisor may handle encryption and decryption of data hosted on the external storage server – which should be considered untrusted – on behalf of the VM (which need not itself implement encryption or the iSCSI protocol). Remote
20 administration may be provided through an API, as will be described below for embodiments.

In this way, the hypervisor can provide the following functionality:

- A *front-end network* provides that general network access to the guest VMs. The hypervisor makes use of NICs that support virtual functions via SR-IOV. Guest VMs gain network access via virtual function network devices mapped to dedicated physical
25 network ports.
- A *back-end network* that provides access to the storage servers. The hypervisor implements the iSCSI protocol to communicate with external storage servers on dedicated physical network ports.
- A *management network* on a dedicated physical network port. The hypervisor is
30 controlled via an API accessed over an IP network. Authentication and authorization

are delegated to a Public Key Infrastructure (PKI) managed by the hosting provider. The Atoll hypervisor implements x509 certificates with specific extensions. The server's Baseboard Management Controller (BMC) may be exposed on a dedicated NIC or share the management network NIC port.

- 5 As will be described below, in embodiment one node is used for system management, whereas other nodes function as user nodes with user spaces that run user payloads. While the discussion below treats there as being one management node, in other embodiments there may be multiple management nodes – for example, a physical hardware estate may be split between different management nodes, or different management nodes may be used for
- 10 different purposes. Such management nodes would either need to use different sets of resources or to be provided with mechanisms to determine which management node had responsibility for particular resources. It is also possible that the system of management node (or nodes) and user nodes as described only describes part of the system – for example, additional nodes could be provided running alternative operating systems (such as single-core
- 15 Linux) or could be hardware nodes – while such nodes will not have the properties described in the embodiment below, they could be integrated into a system which did operate as described below.

- Embodiments of an architecture and associated processes and methods of operation will now be described. These use a hypervisor of the type described above to implement VMs in a
- 20 virtualisation platform using a multikernel architecture based on seL4 microkernels. Such a system may be particularly appropriate for implementing relatively long running VMs running network services such as e-mail servers, DNS servers, file servers, application servers or web servers.

- A specific embodiment of such an architecture and associated components and processes is
- 25 illustrated in Figures 3 to 11 and 14. In this approach, the server's hardware is partitioned statically into nodes at boot time and this partitioning is fixed throughout the execution of the software. Services such as networking and storage are provided through appropriate hardware functionality, such the use of PCIe SR-IOV virtual functions. As will be described below, dynamicity is provided by agents.

In this approach, each node is started at boot time and remains active until system shutdown. Here, there are only two types of node: a root node (Node 0) and all other nodes, which are used for user payloads.

Node 0 runs on the boot processor core 30. It is a special node that performs system management duties. It hosts the remote API implementation 305 that is bound to a dedicated network interface and receives administration commands from the system administrator. It also hosts the privileged manager component 302 that communicates with agents 312, 322, 332 running on the other nodes 31, 32, 33 to execute the system administrator demands. Node 0 does not execute customer payloads. Node 0 also comprises one or more physical function network drivers 304 for driving network devices and a console relay function 303 for out of band access.

All other nodes 31, 32, 33 run on the remaining processor cores and are dedicated to running user payloads: VMs and their supporting tasks. Each regular node hosts a privileged agent task 312, 322, 332 that maintains communication with Node 0 (here over ring buffer 38) and executes configuration demands, e.g. creating and tearing down VMs. Each node 31, 32, 33 also hosts a trusted console driver task 313, 323, 333 that provides out-of-band access to the VM (here using ring buffer 37). The agent and console driver components are statically defined while other components such as the VMM 315, 325 and 335 and supporting components are created - or recycled - dynamically inside the node.

Each node is provided with a small and exclusive amount of memory – this is from private memory region 401 of the system memory 40. For simplicity and convenience, this memory area is fixed in size and statically defined when assembling the system image, so nodes may not grow or shrink during their life time – in other types of embodiment it may be possible to transfer resources dynamically from one node to another. This memory is used to allocate the seL4 kernel itself (e.g. its text and data segments) as well as all the kernel objects that will ever be created by the node during its life time. This memory is guaranteed to be private and exclusive to the node: by construction no capabilities pointing to it exists anywhere else on the system but on the single node that owns it.

Each regular node is also provided with capabilities to small regions of memory marked as device memory 402 that are exclusively used to implement the communication mechanism between the node and Node 0. These memory region are fixed in size, statically created, and

governed by capabilities that are shared by both the manager and console relay components on Node 0 and the local agent and console components on the node, effectively making them shared memory. The communication mechanism is bi-directional and point-to-point: agents only communicate with the manager on Node 0, they specifically do not communicate with one another. The communication is fully managed by user-space tasks without any direct involvement from the kernels.

The remaining memory (excepting memory mapped to specific devices 404), that is, memory that has not been allocated as private memory to a node and that also has not been allocated for the fixed communication buffers between all nodes and Node 0, is made available to all nodes in the form of device memory 403. Memory marked as device memory to the seL4 microkernel can only ever be mapped as frames into an address space: it can never be used to create kernel objects. To enforce that memory mapping attributes and sizes are consistent between all cores, the memory is broken down into frame capabilities that cannot be further split on any node. This large memory region is reserved to dynamically allocate memory for the guest VMs and for any additional buffer required for inter-node communication and to support emulated devices. The allocation is performed by the manager component on Node 0 and executed by the agent components on the appropriated nodes. The manager does not possess capabilities to these memory regions, it only makes policy decisions that the agent components execute by invoking capabilities within their node. Both the manager and the agents are responsible for the correct allocation and mapping of pages and are thus highly trusted.

Figure 3 shows the memory allocation and node breakdown of a small exemplary system with four cores immediately after boot. Figure 3 represents the ground state reached at boot time, where all components are statically defined, all resources statically allocated, and all communication mechanisms fixed. Being fully static, the ground state system is built using a version of the Microkit framework (described at <https://docs.sel4.systems/projects/microkit/>) modified to build multikernel-based systems.

Nodes are allocated in a fixed region of physical memory known as private memory. As noted, Node 0 is the special system administration node that hosts the remote management API and the manager component that centralises system management. Nodes 1 through 3 are started at boot and initially only host the static components such as the agents that will be responsible

for populating the nodes. The manager communicates with the agents using ring buffers located in memory shared by the communicating nodes. All remaining physical memory, regular and device memory, is initially allocated to Node 0. In certain embodiments, Node 0 may be implemented differently from other nodes, given its different function. For example,
5 for greater isolation, some or all of the Node 0 functionality could be moved to a dedicated PCIe device – this could be an external device for running system management tasks.

In this architecture, it is assumed that at most one VMM task will be running in each node: each virtual CPU (vCPU) is exclusively mapped in this embodiment to a single physical CPU core (though it should be noted that a VM may be associated with multiple CPU cores). This
10 design increases the isolation between guest VMs since they are guaranteed by construction to never share a CPU. It also makes the total number of VMM components known at compile time, which enables their allocation at image creation time. Note that the VMM components initially created in the ground state are really empty shells: they are not started and their memory regions are not populated.

15 VMs are created and torn down on request from system administrators via the Application Programming Interface (API) component located on Node 0 and accessible via a dedicated network interface. All system administration is orchestrated by the manager component on Node 0 and executed by the agents on the selected target nodes.

The manager component running on Node 0 maintains a consistent view of the entire system
20 spanning multiple nodes and remains at all times the custodian of all device memory in the system. It keeps track of all resource allocations, and can grant and revoke access to these resources to the VMMs running on all nodes via the locally running agents. The agent component act as local deputies and execute the requests coming from the manager - they do not take part in policy decisions.

25 When a system administrator requests the creation of a VM, the manager component performs the following actions:

- Selects one or more target nodes to host the newly created VM. When a multicore guest VM is requested, the manager selects one node per virtual CPU core where each VMM task emulates a single virtual CPU core. The allocation of CPU cores to
30 host a multicore guest VM should follow the underlying NUMA (non-uniform memory

access) topology of the hardware such that all selected cores are preferably allocated near each other.

5 • Allocates a region of memory from the large pool of available memory marked as device memory to use as guest memory for the VM. This memory is also used to store the VM state data that needs to be shared between the collaborating VMMs, in case of multi-core VMs, and any additional buffer required for their communication. The allocation of guest memory should follow the underlying NUMA topology of the hardware to minimise access time from the selected CPU cores.

10 • Allocates PCIe devices to be passed-through to the VM. In embodiments of this type, there is reliance on network devices that support virtual functions and SR-IOV. The physical function drivers are trusted components running on Node 0, while the virtual function drivers are untrusted and implemented in the VMMs. and the virtual devices are directly allocated to the VMM and passed-through to the guest VM. The VMM exposes VirtIO devices to the guest VM.

15 • Prepares requests to the agent components in the selected nodes to populate and configure the local VMM and its supporting tasks. The request includes references to all physical resources allocated to the VM to be created. No capability is transferred between nodes: all nodes already have capabilities to all shared memory regions, and the manager and agent components are together responsible for their correct
20 allocation and mappings.

Upon reception of the VM creation request from the manager, the agent performs the following actions:

25 • Maps the region of available memory marked as device memory into the address space of the VMM. This region may be primarily used to create the guest address space.

 • Maps memory related to virtual function devices to the address space of the VMM. These regions will be used by the VMM to emulate the VirtIO network and to block devices to the guest VM.

- Loads the VMM binary code into the address space of the VMM. Populate a region of the data segment with VM metadata received from the manager, including the addresses and sizes of all memory regions mapped into its address space (guest memory, devices, etc.), references to the I/O interrupt notification objects, etc.
- 5 • Starts all VMM threads, and notify the manager of the successful execution of the request.

Figure 4 shows the system of Figure 3 where a dual-core VM is requested and dynamically created on Nodes 1 and 2. Following the approach described above, the manager component allocates memory for the VM from the large pool of unused memory marked as device
 10 memory. Physical network devices are divided into virtual functions using SR-IOV. The manager sends the VM creation request to the agents on Nodes 1 and 2, along with references to all resources allocated to the VM. The agents maps these regions into the address space of the VMM. The VMM eventually maps these regions into the address space of the VMM component, populates their contents, and then starts the VMM component 315a, 325a in fully
 15 operational form.

The ground state (as shown in Figure 3) is therefore a purely static system, with dynamic behaviour exhibited solely through the agents. The following aspects are static and configured at image creation time:

- 20 • The multikernel configuration breaking down a large system into a number of nodes, in particular the number of nodes and the amount of memory to allocate to each node to use as private memory.
- The communication channels between the manager and the agents, the number of channels, the memory regions allocated as shared memory, and the Inter-Processor Interrupt (IPI) objects used for signalling.
- 25 • All components running on Node 0.
- The root component, the console and the agent component running on all regular nodes.
- The VMM shell component created on all regular nodes.

Figure 5 illustrates this process. The system administrator 51 requests 505 creation of the VM, and the manager 52 component at Node 0 allocates 510 global resources accordingly, in this case two nodes as well as memory and devices. The manager component then “creates” 515 the Node 1 VMM (as established above, this involves activating an already statically created “shell” VMM) using the agent 53 at Node 1. The Node 1 agent 53 configures 520 the Node 1 VMM task and initialises 525 the Node 1 VMM 54, this being communicated back to the Node 1 agent 53 and subsequently to the manager 52. A similar process takes place for Node 2: the manager 52 “creates” 530 the Node 2 VMM at the Node 2 agent 56, configures 535 it and initialises 540 the Node 2 VMM 57., this being again reported back to the manager 52. The VM hosting the guest can then be started by the manager 52 – this starts with an instruction 545 to the Node 1 agent 53, which similarly instructs 550 the Node 1 VMM 54, which thereby starts 555 the Node 1 Guest 55. Its first action is to boot the second CPU – that is, the CPU of the second node assigned to the VM, Node 2. The Node 1 Guest 55 passes this boot instruction 560 to the Node 1 VMM 54, which instructs 565 the Node 2 VMM 57 to start the Node 2 Guest 58, which it subsequently does 570. When the process of starting the guest is under way, the manager 52 at Node 0 is informed and informs the system administrator.

Figure 14 illustrates such an arrangement, in which by allocating the virtual memory for a guest VM in a memory region that is shared between nodes 147, 148, a set of VMMs 141, 142 running in different nodes are enabled to manage the same guest VM instance collaboratively, where each VMM emulates one single vCPU of the guest VM and each node has its own microkernel 140. The VMMs 141, 142 communicate between each other using dedicated areas of shared memory 143 as well as IPIs 144 for synchronisation. Figure 14 illustrates a split VMM exposing one VM with two vCPUs, each vCPU being virtualised by a VMM task running in a different multikernel node. Node A hosts the VMM task that emulates the boot processor of the guest VM (CPU #0). Node B hosts the VMM task that emulates the first application processor (AP) of the guest VM (CPU #1). Both nodes execute additional helper tasks 145, 146 that provide services to their local VMM such as backends for emulated devices. The guest OS 149 runs on system memory that is shared between the two nodes. The two VMM components use Inter-Process Interrupts (IPIs) to communicate with each other and stay synchronised.

At initialisation time the VMM that emulates the boot processor assumes the status of primary VMM. It configures the guest memory layout, loads the guest OS image and bootstrap code, and waits for all other VMMs to check in. The primary VMM then communicates the specifics of the guest VM setup, in particular the memory layouts and mappings, with the secondary
5 VMMs such that they can build an identical configuration for the same VM. Once all VMMs are configured and ready, the primary VMM begins emulating the guest boot vCPU by executing the guest OS boot code, while all other VMMs start their vCPU in initially halted state. The VMMs together implement the Symmetric Multi Processing (SMP) specifications and let the guest OS request each of the vCPUs to be put online.

- 10 It can therefore be seen that such a distributed VMM can enable relatively straightforward emulation of an SMP system for general execution. However, access to passed through and emulated devices does require additional attention.

Passed-through devices are implemented by directly mapping the physical device memory into the guest VM's address space, thus requiring very little work by the VMM once the memory
15 mapping is configured. In a distributed VMM configuration, the physical memory regions of passed-through devices should be exposed to all multikernel nodes that host a VMM component of the guest VM, and this memory region should be mapped at the same guest physical address in each VMM.

Emulated devices are implemented by special calls (hypercalls, page faults) that trap into the
20 VMM when accessed, such that it can emulate the behaviour of the specific device and return control to the guest. In a distributed VMM configuration, the hypercalls should be configured identically on all VMM that participate in the emulation of a given VM. All VMMs should be able to respond to the hypercalls and to emulate the device. When synchronisation between VMMs is required to properly emulate a device, the VMMs can use dedicated communication
25 channels, built on shared memory and IPIs.

Different elements of this architecture and associated processes will now be described in more detail, starting with the boot process. This architecture operates differently on boot to conventional sel4 systems, which follow a three phase process (hardware discovery and configuration, kernel initialisation, and root task creation). This different boot behaviour is
30 illustrated with respect to Figure 6. The first phase 610 of the boot process remains largely as in existing sel4 systems, and runs from the boot processor core while all other processor

- cores are powered down. A new boot phase 615 is now inserted for this type of embodiment to let the kernel allocate physical memory regions to each node, to populate these regions with a copy of itself, with the node's root task, and with a structure in memory describing the subset of the system resources available to the node, and to start all remaining processor
- 5 cores. The second and third phases of the boot process (kernel state initialisation 620, root task creation 630) then proceed largely as for conventional seL4 systems on each processor core, the main difference compared to conventional seL4 arrangements being that they all start in parallel and do not share state. Generally all seL4 instances are oblivious of the existence of each other, all cross-node communications being managed by user-space tasks.
- 10 The specifics of this new phase that allocates the multikernel nodes are statically defined at compile time: the number of nodes to allocate, the size of their private memory regions, the list of devices to expose to each node, etc. This multikernel configuration is therefore purely static.

As can be seen from Figure 6, unlike in a more conventional symmetric multiprocessing (SMP)

15 configuration, the nodes do not share the kernel state, they run their own root task with capabilities for a different set of resources.

The root tasks that execute on each node are statically built and aware at compile time of the specifics of the node on which they will run. They receive capabilities to the following resources:

- 20
- Memory regions exclusively allocated to the node, as regular Untyped objects, excepted of course the regions used by the node kernel for its text segment and internal state. No other node has capabilities to these regions. All future kernel objects to be created in this node will be exclusively retyped from these Untyped objects.
- 25
- Memory regions shared between all nodes, marked as device Untyped. All nodes are given capabilities to these regions, making them effectively shared. The manager and agent components work together to guarantee a consistent and secure usage of these regions across nodes. These regions, being marked as device memory, can never be used to create kernel objects, they can only be used to create frames that are mapped into address spaces.

- Device Memory Mapped Input/Output (MMIO) memory, marked as device Untyped. The multikernel is statically configured to present a subset of the devices present on the system to all nodes. Generally only devices that will be made available to VMs need to be allocated to all nodes such that they can be mapped into the address space of the VMM, for emulated devices, and into the address space of the guest VM, for passed-through devices.

In this type of arrangement, the user-space of each node is therefore built statically at compile time. Here, the user-space of a node is a static/dynamic hybrid, with the agent task providing dynamicity.

- 10 The communication mechanisms that link the manager component in Node 0 and the agent components in all other nodes are statically defined at compile time. The root task of each node is responsible for creating all statically defined kernel objects in the node, including the kernel objects allocated to the communication mechanisms. These are generally device Untyped memory used as shared memory buffers and IPIs for synchronisation.
- 15 All communications between the nodes are strictly implemented outside of the kernels by user-space tasks, as is discussed further below: the kernels do not take part in said communication mechanisms – this is in contrast to a conventional SMP software architecture, where IPIs are used internally by the kernel to synchronise execution across multiple cores.

Running independent kernels on multiple cores of the same CPU provides some challenges for memory management – the following features are implemented in appropriate embodiments:

- Regions of physical memory that are shared between multiple cores should use the same page size on all cores when mapped as virtual memory. These memory regions are used as guest-physical memory and to create communication channels between nodes. One solution is to exclusively map these regions using 2MB pages - these are supported by many 64-bit architectures and provide a good trade-off between memory allocation granularity and memory management overheads. Regions of physical memory that are private to a node are not a concern because by construction they will not be mapped as virtual memory on any other core.

- The address space identifiers – Address Space Identifier (ASID) and Virtual Machine Identifier (VMID) on ARM, Process-Context Identifier (PCID) on x86 – should be globally unique on all cores of the same CPU. The identifiers are determined independently by each node kernel and seL4 does not expose mechanisms that would allow the definition of policies on each kernel instance. The user-space should then allocate the identifiers and assign them to VSpace objects, making tasks unschedulable until their VSpace is assigned an address space identifier. The manager component would centralise the allocation of address space identifiers and guarantee their global uniqueness. It is assumed that the total number of tasks running globally does not exceed the number of address space identifiers, such that address space identifier recycling is only required at VM creation and tear down.
- For a x86 implementation, the per-core TLBs entries should be manually invalidated by the system software whenever mappings change, which is typically implemented using IPIs. Here, multicore VMs are implemented by multiple VMMs running on different nodes and sharing memory regions. The mapping will not change for the duration of the VM's life. When VMs are torn down, the system should carefully invalidate the corresponding entries in the TLBs of all participating nodes. Instead of using IPIs between the selected nodes, which would require the node's awareness of each other, the synchronisation is orchestrated by the manager component running on Node 0 and coordinated by the agents on each participating node.

Special consideration needs to be given to the I/O Memory Management Unit, as this is a critical resource that is shared between all cores – consequently something other than simply strict resource partitioning, used otherwise as the key way to prevent undesired concurrency, is required.

The System Memory Management Unit (SMMU) present on modern ARM devices features multiple contexts, each with their own dedicated root table and the PCIe Requester ID (RID) eventually used as context descriptor. Virtual Function (VF) devices derive their RID from the RID of their Physical Function (PF) device – this simply follows the SRIOV specification. Each VF device can therefore be assigned its own I/O MMU root page table which is not shared

with any other device. A strict partitioning of the set of virtual functions to kernels then guarantees non-concurrent access to the I/O MMU configuration.

As previously indicated, all communications between the nodes are implemented outside of the kernels by user-space tasks. Cross-node communication is required for the manager↔agent communication channels, and as a backend to the VirtIO console devices of the running VMs. The communication mechanism adopted uses pages from the memory regions marked as device memory and shared between the nodes. These pages are mapped in the address space of the two components that act as communication endpoints in their respective nodes, making them effectively shared memory. Synchronisation between the two endpoints relies on regularly polling the shared memory for new data at a fixed rate. IPIs may in principle be used in addition to polling – in certain architectures they may be more efficient.

It is assumed that the two endpoints participating in a cross-node communication link are mutually trusted: their cooperation is required for the correct implementation of the mechanism.

One optional communication system, used in certain embodiments, is illustrated in Figure 7. Here, the pages are organised as a ring with the current read and write indexes stored in separate pages and configured with write-through caching. This configuration enables a lockless implementation relying only on memory barriers and atomic operations. While the mechanism would function as described with the two endpoints regularly polling the Read and Write indexes for changes, a notification mechanism would be more efficient. The traditional mechanism used to signal remote CPU cores is the Inter-Processor Interrupt (IPI). These interrupts are typically used by SMP kernels to synchronise local processor states with the global system state, for instance to update the Memory Management Unit (MMU) configuration. To do this, the conventional sel4 kernel is modified to provide mechanism to send and receive IPIs from a user-space component. These IPIs are used as notifications between the two communicating endpoints in their respective nodes to indicate changes to the Read and Write indexes.

Figure 7 provides an example of a cross-node communication link. The 8 shared data pages are organised in a ring. The Read and Write page indexes are stored in separate pages. The Sender component compares the values of the Read and Write page indexes to determine the number of pages that can be written before overwriting unread pages (2 in this example),

writes the data in the pages starting at the Write index, and eventually updates the Write index to point at the next page. The Receiver compares the values of the Read and Write page indexes to determine the number of pages that remains to be read (5 in this example), reads the data from the pages starting at the Read index, and eventually updates the Read index to point at the next page. IPIs are used as signalling mechanism to indicate a change of the Read or Write position indexes.

The nature of the manager component and agent components and their interaction will now be described in more detail. The manager component running in Node 0 and the agent components running on every other node are together responsible for the dynamic behaviour of the otherwise static system provided as the ground state. These components are themselves part of the ground state and defined statically. The manager does not hold capabilities to the memory regions to be used as guest physical memory - these capabilities do not exist at all in Node 0. They do exist on all other nodes, and are held by the agent components. Responsibilities are distributed between manager and agent in the following way:

- The manager component receive system management requests from the API component running in Node 0. The manager is the custodian of all shared resources in the system. Its prime function is to allocate the resources (memory and passthrough devices) necessary to create new VMs and to assign them to the selected nodes, and to reclaim resources from VMs that have been terminated. The manager also relays system and VM status information to the API for monitoring.
 - The agent components receives commands from the manager and execute them on their local node. The agents are the custodian of all resources private to their node. The agents hand over the capabilities to the resources allocated by the manager to their local VMM task on VM creation, populates their memory regions (code and data), and starts them. By construction, the shared resources are marked as memory frames to be mapped into the address spaces of the VMM: specifically no kernel object can be created from these resources. The agents also monitor the state of the VMs that run on the same node and reports status information to the manager.
- The manager component is essentially a resource allocator and system manager. It does not make use of the capabilities to the resources shared in the system - in fact, it does not need

to own capabilities to the resources that it manages. It only needs to know the address and size of the memory regions that it can allocate to shared objects. The agents alone use the capabilities to these regions to create frames and map them into the address space of the appropriate components (typically the VMMs).

- 5 The manager component enforces the allocation of all VM memory – both the guest physical memory and the memory used by the VMM itself – such that DRAM subarrays are never shared between VMs as a mitigation against vulnerabilities in DRAM hardware (e.g. Rowhammer). The specific layout of DRAM modules is detected at configuration time and used to partition the available memory – the DRAM subarray size then becomes the
- 10 granularity of memory allocation.

The manager communicates directly with each of the agents using communication mechanisms as described above, configured as a bi-directional link. agents do not communicate with each other. The communication protocol comprises the following messages:

- 15
- CREATE: Sent by the manager to an agent to request the creation of a VM on its node. The message includes all the information required to create the VM, including the address and size of the physical memory pages to use as guest physical memory, the list of physical devices to pass-through to the VM, the address and size of memory regions to use additional communication channels (e.g. between VMMs managing the
- 20 vCPUs of the same VM), the vCPU number, etc. Upon reception of this message the agent reconfigures the shell VMM task and reports back to the manager. On failure, any shared resources allocated to the VM are returned to the manager and marked as unallocated.
- DESTROY: Sent by the manager to an agent to request the destruction of a VM on its
- 25 node. Upon reception of this message, the agent revokes all the additional capabilities that were transferred to VMM, and then resets the VMM task to its initial ("shell") state. Note that no untyped memory is ever handed over to the threads supporting the execution of the VM to avoid a potential situation where a thread would be able to create more threads and avoid its own destruction. On success, all shared resources
- 30 allocated to the VM are returned to the manager and marked as unallocated.

- START: Sent by the manager to an agent to request the start of a VM on its node. VMs are created in halted state to wait for the successful creation of all VMMs on all participating nodes before being started. Note that only the VMM implementing the virtual boot processor core needs to be explicitly started. The VMMs implementing the other virtual processor cores will be independently started by the guest OS running in the VM.
- RESULT: Sent by an agent to the manager as a response to the CREATE, DESTROY, and START messages. The message includes a status code that indicates whether the request was successfully executed or not, and in this case an indication of the error is included in the message.
- STATUS: Sent by an agent to the manager to report on the status of the node the agent runs from. The message includes the amount of private resources available in the node and the status of the VMs that run in the node.

If correctly implemented, this construction of the manager and the agents can provide the security properties of integrity and confidentiality to the VMs. A misbehaving manager or agent could otherwise map the virtual memory of one VM into the address space of another VM and breach their isolation.

Different aspects of device management will now be considered with reference to Figures 8 and 9. In this embodiment, network devices are used that support the Single Root - Input/Output Virtualisation (SRIOV) mechanism to divide a physical network device into multiple virtual network devices that are then directly allocated to VMs. Network data is transferred between the physical device and the VMs with very little intervention from the VMMs, and thus with low virtualisation cost and latencies. The complete implementation is split into different components as follows:

- A Physical Function (PF) device driver that runs in the administrative Node 0 and is responsible for managing the global configuration of the network adapter. This driver is only involved in control.
- A Virtual Function (VF) device driver that runs in the VMM component and manages the configuration of the VF device. The VMM exposes a standard VirtIO network interface to the guest VM, abstracting away the details of the physical network adapter.

This is as shown in Figure 8: the PF device driver 800 runs in the administrative node 80, Node 0, while the VF device driver 810 runs in the VMM component 81 of Node N. The PF and VF drivers communicate using channels provided by the physical network device 82. The VMM provides a standard VirtIO interface 83 to the guest VM 84 (the guest VM therefore including a VirtIO Network Driver 840).

Such division of responsibilities requires communication channels between the PF and VF device drivers. The SRIOV standard does not define this mechanism, which could be implemented by the hardware device or left to the host software – use of a hardware mailbox system (as is implemented in Intel 800 series network devices) is one effective communication route that would reduce latency – these can use Message Signaled Interrupts (MSIs) to signal the VFs of network events, removing the need to forward signals from the PF driver to the VF driver.

In embodiments, an external iSCSI server is used to provide permanent storage to the VMs. Decoupling storage from compute has advantages in a hosting environment: the VMs are not limited to the amount of storage present on their hosts; the cloud operator does not need to populate all compute nodes with large storage devices that may be underused; and data redundancy and backups are easier managed from a few storage servers rather than distributed over the entire compute fleet, and a few consolidated storage servers are more economical.

Figure 9 shows how this can be implemented. Interaction between the administrative node 90 (and its PF network driver 900) and the Node N VMM 91 (and its VF network driver 910) through physical network device 92 is as for the networking case. The virtual storage driver of 920 the VMM 91 interfaces with the external storage server via a dedicated network link exposed as a VF. The virtual storage driver implements the iSCSI protocol. External storage servers are considered untrusted, and so all data must be strongly encrypted. The virtual storage driver 920 may then implement transparent encryption and decryption such that the external storage server only sees encrypted data, while the VM 94 is exposed to the clear text. The virtual storage driver 920 exposes a standard VirtIO block device 93 to the guest VM 94, which comprises a suitable block device driver 940.

Boot media such as ISO files are also exposed to the guest VM using VirtIO block devices backed by external iSCSI storage. However, boot media are exposed read-only.

Initial VM management and configuration as well as system recovery often require an out-of-band mechanism to reach the guest OS. As shown in Figures 3 and 4, the VMM exposes a VirtIO console device to the guest to this effect, exposing the guest kernel output during boot and a console device that can be bound to a command line session.

- 5 The VMM copies the console data stream to a console backend component that runs in Node 0 and gathers the data streams of all console devices in the system. The data streams are accessed via the remote API. It is expected that the cloud operator managing the fleet would use the remote API of the hypervisors to expose the VM consoles to their respective customers.
- 10 Remote administration in this architecture will now be discussed. Preferably, the sole mechanism for administration and monitoring of the software is an HTTPS – based API, enabling ready integration into a centrally managed virtualisation fleet. A dedicated network interface (typically a network adapter on the server motherboard) should be used to carry all system administration operations.
- 15 The remote administration component may be implemented as a VM running on Node 0 and managed by a statically configured VMM. The dedicated network adapter is passed-through to the VM so that a native seL4 device driver is not required. The VM may run, for example, an embedded Linux-based guest OS built with lightweight components (e.g. buildroot, musl libc, busybox etc.). It includes drivers for common network interfaces, a network IP stack
- 20 along with network configuration and management utilities (dhcp, ntp etc.), a lightweight web server, and the remote administration application.

The administration application implements the API endpoint, but it does not directly implement, or authenticate, the requests. Instead, these are forwarded to, and executed by, the trusted manager component also running on Node 0. The administration VM communicates with the

25 manager component using statically allocated shared memory regions and seL4 notifications. The VMM exposes the manager communication channel to the guest OS as a VirtIO Socket device.

The remote administration component includes a complete Linux-based VM – this cannot therefore be considered a trusted component. Instead, it is designed to be a transparent

30 interface between the untrusted administration network and the trusted manager component.

All requests sent to the hypervisor are cryptographically signed, and validated by the manager component. Likewise, all responses from the manager component are cryptographically signed and returned to the external network. Should the administration component (including its VMM) be compromised, an attacker would therefore not be able to impersonate a system administrator (or send forged requests to the manager, or forged responses to the legitimate system administrator) – at most DoS type interference would be achieved.

This arrangement is shown in Figure 10. The remote administration component runs in an (untrusted) Linux VM 1001 and forwards responses between an external system administrator and the manager component 101, which is trusted. The manager validates message signatures before accepting requests using a validation and signing module 1011, using its own private key. Cryptography operations are performed by a hardware TPM module 1012.

As public key certificates are used to authenticate and so authorise all hypervisor operations, the security of the Public Key Infrastructure (PKI) used is critical, and it needs to be highly trusted. The PKI is typically an external PKI controlled by the cloud operator.

Each party interacting with a hypervisor, and each hypervisor, is identified by an x509 public key certificate. All certificates are signed by an intermediary CA (Certification Authority), and eventually signed by a root CA – typically stored in an air-gapped vault. Requests must be accompanied by a valid x509 certificate authenticating the end-user issuing the request, along with an attribute certificate authorising the request. The attribute certificate supports fine-grained access control filters (e.g. system administration, access to read-only monitoring data, access to system logs, operations on a specific VM such as reboots or console access, etc.). The management of certificates and of attribute certificates is left to the cloud operator.

The hypervisor private key is stored in a locally attached TPM that also performs all cryptography operations. The root CA certificate is also stored in the TPM, and it is used to authenticate the incoming requests. The responses are signed by the hypervisor's key stored in the TPM, and can be remotely authenticated. When provisioning a hypervisor, a private key is generated by the TPM, and the public key is then extracted and certified by the cloud operator's PKI.

The use of public key certificates avoids the need for maintaining databases of users, credentials, roles, etc., and allows user management, authentication and authorization to be managed externally by the cloud operator via its PKI, maximising flexibility.

This approach is illustrated in Figure 11. A PKI 110 managed by the cloud operator signs the
5 certificates of all parties – customers, administrators, hypervisors, etc. – using a root CA 1101 and derived purpose-based CAs 1102. The public key of the root CA 1101 is stored in the TPM of each hypervisor 111 and used to authenticate requests.

Further functionality can be built on this structure. For example, there is a key trade-off
10 between the security (isolation) of VMs and the utilisation of physical resource. VMs co-hosted on the same physical CPU will share access to CPU cycles and resources such as caches, creating performance and security issues. Some applications however may not justify this hard separation of resources and could be aggregated onto a small set of dedicated cores to improve the utilisation of physical resources. These could be implemented as trust domains:
15 a set of VMs that are sufficiently mutually trusted that may share one or more CPUs.

A trust domain could be implemented as an additional constraint affecting the allocation of physical CPUs to virtual CPU. Rather than a single VMM task per node, a fixed number of VMMs components could be present in each node, sharing access to the node's CPU. Priorities between the VMMs could be defined using the seL4's Mixed-Criticality Systems
20 (MCS) extensions designed to support time partitioning.

Mechanisms for inter-VM networking may also be provided. There are many use-cases for VMs connected to private networks. Common examples include backend networks hosting databases for web applications, servers behind Virtual Private Networks (VPNs) or access control services, all the way to private clouds. A hypervisor of the type describe above would
25 not provide mechanisms such as point-to-point VirtIO devices to support direct inter-VM networking or virtual network switch devices. These would be problematic for performance, and private networks are likely to span beyond the host and require interconnecting VMs hosted on different physical servers. Instead, inter-VM private networking may be implemented as a layer above the regular uplink network. The VMM will create virtual switch
30 ports using the VXLAN protocol and expose them as VirtIO devices to the guests. Additional

privacy can be achieved for instance by wrapping the VXLAN traffic in IEEE 802.1AE (also known as MACsec) for encryption. Network parameters, including encryption keys, can be provided via the API.

- 5 Different architectural approaches can also be taken within this general structure. In contrast to other multikernel implementations, the VM server of this type of embodiment does not attempt to create and start all nodes at boot time. Initially only one single node is created and runs on the boot CPU: the system management node. All other CPUs are left in halted state. The system management node is given exclusive access to a dedicated network interface
- 10 onto which it exposes the management API. It is also given access to all resources present in the system. The capabilities to these resources are allocated to a privileged multikernel management component that is responsible for the creation and destruction of all other nodes in the system, including the allocation of physical resources with exclusive or shared access. Specific IPI are used to start and stop application CPUs.
- 15 A full system generally consists in a number of service nodes, a number of user nodes, and a number of internode communication links. Service nodes are managed by the cloud operator and provide fundamental services to user nodes. Example of service nodes include network nodes that multiplex the access to physical network devices, and storage nodes that expose block devices to user nodes. User nodes consist of a custom built Virtual Machine Monitor
- 20 (VMM) task that runs guest OS's. It should be noted that there is a maximum number of nodes in a given system, limited by the number of CPU cores. Depending on the hardware and use-case one might prefer bundling multiple services and VMs together in one node.

Figure 12 illustrates such an architecture. In this type of embodiment, Node 0 runs the system administration components and is the only node started at boot time. Node 1 and Node 2 are

25 service nodes that provide respectively network and storage access to other nodes. Node 3 and Node 4 run end user payload OS's. Communication between nodes is implemented using ring buffers in shared memory and IPIs as signalling mechanisms, with the logic implemented by the cooperating tasks labelled Link Interface. The vertical bar on the left represents the full physical address space of the system and the allocation of various memory areas to nodes

30 and ring buffers.

The user space of all nodes, including the system management node, may be built using the sel4 Microkit (formerly the sel4CP framework), which is described at <https://docs.sel4.systems/projects/microkit/>. A component of the system management node is responsible for assembling sel4CP images at node creation time using locally stored ELF files and user-provided guest OS images and runtime configuration requests. Nodes can communicate with one another using ring buffers stored in shared memory and IPIs for signalling. The ring buffers are managed cooperatively by trusted user-space tasks running on all nodes that require such communication channels. No involvement of the sel4 microkernel is necessary to implement the communication mechanisms, beyond the modelling of IPIs.

A VM server built using this architecture can also scale efficiently to the large number of cores present on modern cloud servers thanks to its multikernel architecture. Here, the system can be considered to be a hybrid static/dynamic system. The software stacks running on each node are statically defined and built using this framework as one would build static sel4 embedded systems - the user-space stacks running in each node are statically defined and their critical components should be sufficiently small to be amenable to formal verification.. The nodes however are dynamically managed, they are created and destroyed at runtime following the requests of system administrators and end-users.

In a cloud server architecture, it is desirable for the user space to be dynamic, rather than static. The sel4 microkernel itself does not require the user space to be statically defined - sel4 is designed to be as policy-free as possible and imposes very few restrictions on the user space. Being a capability-based microkernel however makes implementing a fully dynamic user-space a challenging endeavour due to the number of capabilities to manage. Typical sel4 systems are static and fully defined at compile time, and therefore relatively easy to develop, to reason about, and (very importantly) to eventually verify – the approach indicated in the earlier embodiment does use a fully defined static system, with agents used to provide dynamicity.

Flexibility – and a dynamic user space – can be achieved in an alternative manner using a hybrid static/dynamic approach as shown in Figure 13. In this approach:

- The images are assembled following the conventional approach of building static systems where the system state is described in an XML file that list of Memory Regions (MRs) and Protection Domains (PDs).
- One special PD of this system, called the Node Manager, introduces dynamic behaviour by being assigned additional MRs pointing to memory regions private to the node as well as memory regions shared between multiple nodes. Using the capabilities to these memory regions, the Node Manager can create children PDs from MRs pointing to private node memory, and it can map MRs pointing to region to shared memory to implement cross-node connection mechanisms, at runtime, and without affecting the static components of the user-space stack.

A memory region shared between the administrative node 0 and the Node Manager PD of each other node enables a simple communication channel such that all nodes are centrally controlled from the administration node.

An example of such implementation is presented in Figure 13. The PDs on at the bottom side of the figure are statically created and are initially the only PDs running in this node. The Node Manager PD in this example has created a Link Interface to communicate with other nodes, as well as two VMM PDs that run guest OS rented out to customers. This mechanism enables the contents of nodes to be ultimately managed from an external cloud management console and using the server's API.

20

While different architectures are shown here, the skilled person will appreciate that various features described here in relation to the first type of embodiment can be used with the second type of embodiment, and vice versa. Some such features are described above. In addition, it should be noted that for either type of embodiment, the following may apply: user-spaces may be developed to be dynamically built at run time or statically built, and user-spaces may be static when running or a static/dynamic hybrid; and in different arrangements VMs may or may not be dynamically added to or removed from running nodes. The choice of fully static as opposed to static/dynamic user-space stacks may be considered a design trade-off, with either choice being more suitable for some applications. Another possibility is for some nodes

to be of one type and some another – for example, some nodes may be statically defined and others dynamically created.

A further variation is the provision of various parts of the architecture in hardware rather than software. The manager component on Node 0 (in fact, all of Node 0), the agent components
5 on each node, and (where used) any service node may be suitable for such an implementation.

While embodiments here use the seL4 microkernel, it will be appreciated by the skilled person that the same design principles may be used for multikernel architectures using different microkernels with similar properties to seL4.

10 Further modifications may be made to the above examples without departing from the scope of the present disclosure.

CLAIMS

1. A computer system having a multikernel architecture comprising a plurality of operating system microkernels, the computer system comprising:

a plurality of processing cores;

5 a memory comprising a plurality of private memory areas and a shared memory area;
and

a plurality of computing nodes each comprising one or more of the processing cores
and further comprising one of the plurality of operating system microkernels and an associated
user space for non-kernel operations, wherein only the operating system microkernel runs at
10 a highest privilege level of the processing core, or processing cores, of a computing node;

wherein each of the plurality of computing nodes has an associated private memory
area from the plurality of private memory areas for use by its operating system microkernel,
and wherein for non-kernel operations each of the plurality of computing nodes accesses the
shared memory area.

15

2. The computer system of claim 1, wherein the computer system is adapted to partition
system hardware statically into nodes at boot time.

3. The computer system of claim 1 or claim 2, wherein one of said plurality of computing
20 nodes is a management node for managing user nodes, and wherein one or more other
computing nodes are user nodes for running user payloads.

4. The computer system of claim 3, wherein a plurality of computing nodes are
management nodes.

25

5. The computer system of claim 3 or claim 4, wherein the computer system is adapted
to provide one or more virtual machines for users.

6. The computer system of claim 5, wherein no more than one virtual machine is
30 implemented in any user node.

7. The computer system of claim 5, wherein two or more mutually trusted virtual machines are implemented in one or more user nodes.
- 5 8. The computer system of any of claims 5 to 7, wherein one or more virtual machines are implemented using multiple user nodes.
9. The computer system of any of claims 5 to 8, wherein the computer system is adapted to operate as a cloud server.
- 10 10. The computer system of any of claims 5 to 9, wherein the management node comprises a manager component and each user node comprises an agent, wherein the node management component is adapted to instruct the agent of a user node to create, destroy or modify virtual machines in that user node.
- 15 11. The computer system of any of claims 5 to 9, wherein communication between user nodes is not privileged, but communication between the manager component and the one or more agents is privileged.
- 20 12. The computer system of claim 11, wherein direct communication between nodes is limited to direct communication between the user spaces of those nodes.
13. The computer system of any of claims 3 to 8, wherein each user node has a virtual machine manager component for managing a virtual machine in that user node by interaction
25 with the manager component.
14. The computer system of claim 13 where dependent on claim 6, wherein a virtual machine manager is provided for each virtual machine in a user node.
- 30 15. The computer system of claim 13 or claim 14, wherein each user node comprises a console driver and the management node comprises a console relay for communication between the management node and virtual machine managers.

16. The computer system of any of claims 13 to 15, wherein each virtual machine manager component is statically defined at boot time for subsequent configuration.
17. The computer system of any of claims 3 to 16, further comprising one or more network
5 devices, wherein the management node comprises a physical function device driver for each of the one or more network devices for physical interaction with the associated network device; and wherein one or more of the user nodes comprises a virtual function device driver for a virtual function network device corresponding to one of the one or more network devices, wherein the virtual function device driver provides a virtual network device function in said user
10 node.
18. The computer system as claimed in claim 17, wherein the physical function device driver is provided by an SR-IOV mechanism.
- 15 19. The computer system of claim 17 or claim 18, further comprising one or more network devices, wherein one or more of the network devices are storage devices, and wherein data is encrypted for communication between the virtual machines and the storage devices.
20. The computer system of any of claims 3 to 19, wherein the computer system further
20 comprises a remote administration interface comprising a dedicated network link.
21. The computer system of any preceding claim, wherein each microkernel has proven implementation correctness.
- 25 22. The computer system of any preceding claim, wherein each microkernel is a third-generation microkernel.
23. The computer system of claim 22, wherein each microkernel is a seL4 microkernel.
- 30 24. A method of operating a computer system comprising a plurality of processing cores and a memory to implement a multikernel architecture, the method comprising:
defining a management computing node having one or more processing cores, booting the management computing node and statically partitioning processing cores into a plurality

of further computing nodes, each computing node comprising one or more processing cores and having an operating system microkernel and an associated user space, wherein only the operating system microkernel runs at a highest privilege level of the processing core, or processing cores, of a computing node;

5 and booting and provisioning the further computing nodes as user nodes.

25. The method of claim 24, further comprising establishing the memory as a plurality of private memory areas and a shared memory area, wherein each of the plurality of computing nodes is provided with an associated private memory area from the plurality of private memory
10 areas for use by its operating system microkernel, and wherein some or all of the plurality of computing nodes access the shared memory area for non-kernel operations.

26. The method of claim 24 or claim 25, wherein the user nodes are adapted to provide one or more virtual machines for users.

15

27. The method of claim 26, wherein no more than one virtual machine is implemented in any user node.

28. The method of claim 26, wherein two or more mutually trusted virtual machines are
20 implemented in one or more user nodes.

29. The method of any of claim 26 to 28, wherein one or more virtual machines are implemented using multiple user nodes.

25 30. The method of any of claims 26 to 29, wherein the computer system is adapted to operate as a cloud server.

31. The method of any of claims 26 to 30, wherein the management node comprises a manager component and each user node comprises an agent, wherein the method further
30 comprises the node management component instructing the agent of a user node to create, destroy or modify virtual machines in that user node.

32. The method of any of claims 26 to 31, wherein each user node is provided with a virtual machine manager component for managing virtual machines in that user node by interaction with the manager component.

5

33. The method of claim 32, further comprising statically defining each virtual machine manager component at boot time.

34. The method of claim 33, further comprising configuring the virtual machine manager component of a user node on creating a virtual machine in that user node.

10

35. The method of any of claims 26 to 34, the computer system further comprising one or more network devices, wherein the management node comprises a physical function device driver for each of the one or more network devices for physical interaction with the associated network device; and wherein one or more of the user nodes comprises a virtual function device driver for interaction with a virtual function network device corresponding to one of the one or more network devices, the method further comprising the virtual function device driver providing a virtual network device function in said user node.

15

36. The method of claim 35, the computer system further comprising one or more network devices, wherein one or more of the network devices are storage devices, the method further comprising encrypting data for communication between the virtual machines and the storage devices.

20

37. The method of any of claims 24 to 36, wherein all communication between user nodes is between user spaces of user nodes.

25

38. The method of any of claims 24 to 37, where any privileged communication between nodes is between a manager node and another node.

30

39. A method of providing a dynamic user spaces in a computer system, wherein the computer system is adapted for virtualisation, the method comprising:

statically defining a plurality of computing nodes, each said computing node comprising an operating system kernel, wherein one of the plurality of computing nodes is a management node and one or more other computing nodes of the plurality of computing nodes are user nodes;

- 5 wherein the management node comprises a manager component for management of user nodes, and wherein the or each user node comprises an agent; and wherein a dynamic user space is provided in a user node by the manager component instructing the agent of a user node to provide creation, destruction or modification of virtual machines in that user node.

10

40. The method of claim 39, wherein the operating system kernel of a computing node is a microkernel.

41. The method of claim 39 or claim 40, wherein in a computing node, only the operating
15 system kernel runs at a highest privilege level.

42. The method of any of claims 39 to 41, wherein there are a plurality of management nodes.

- 20 43. The method of any of claims 39 to 42, wherein the manager component is adapted to receive management requests from a remote administration interface comprising a dedicated network link.

44. The method of any of claims 39 to 43, wherein the manager component determines
25 the allocation of all memory for a virtual machine.

45. The method of any of claims 39 to 44, further comprising the creation of a virtual machine in a user node by instruction from the manager component, the instruction comprising identification of memory to be used by the virtual machine and physical devices
30 to be passed-through to the virtual machine.

46. The method of claim 45, wherein the virtual machine is created in halting state.

47. The method of claim 46, further comprising the manager component instructing the agent to start the virtual machine.

48. The method of any of claims 45 to 47, wherein the user node comprises a virtual machine manager component, and the method further comprises configuring the virtual machine manager component for the virtual machine to be created.

49. The method of claim 48, further comprising creating the virtual machine manager component statically on system boot as a shell component for configuration on creation of a virtual machine.

50. The method of claim 48 or claim 49, wherein each user node comprises a console driver and the management node comprises a console relay for communication between the management node and virtual machine managers.

51. The method of any of claims 39 to 50, wherein communication between user nodes is not privileged, but communication between the manager component and the one or more agents is privileged.

52. The method of any of claims 39 to 51, wherein direct communication between nodes is limited to direct communication between the user spaces of those nodes.

53. The method of any of claims 39 to 52, the computer system further comprising one or more network devices, wherein the management node comprises a physical function device driver for each of the one or more network devices for physical interaction with the associated network device; the method further comprising providing at one or more user nodes a virtual function device driver for a virtual function network device corresponding to one of the one or more network devices, wherein the virtual function device driver provides a virtual network device function in said user node.

54. The method of claim 53, wherein the physical function device driver is provided by an SR-IOV mechanism.

55. The method of claim 53 or claim 54, the computer system further comprising one or more network devices, wherein one or more of the network devices are storage devices, and wherein the method further comprises encrypting data for communication between the virtual machines and the storage devices.

5

56. A computer system adapted for virtualisation using dynamic user spaces, the computer system comprising:
a plurality of computing nodes, each said computing node comprising an operating system kernel, wherein one of the plurality of computing nodes is a management node and one or
10 more other computing nodes of the plurality of computing nodes are user nodes;
the management node comprising a manager component for management of user nodes, and wherein the or each user node comprises an agent; and
wherein the computer system is adapted to provide a dynamic user space is provided in a user node by the manager component instructing the agent of a user node to provide
15 creation, destruction or modification of virtual machines in that user node.

57. The computer system of claim 56, wherein each computing node comprises one or more processing cores.

20 58. The computer system of claim 56 or claim 57, wherein the operating system kernel is a microkernel.

59. The computer system of any of claims 56 to 58, wherein in a user node, only the operating system kernel runs at a highest privilege level.

25

60. The computer system of any of claims 56 to 59, wherein communication between user nodes is not privileged, but communication between the manager component and the one or more agents is privileged.

30 61. The computer system of any of claims 56 to 60, wherein direct communication between nodes is limited to direct communication between the user spaces of those nodes.

62. A computer system having a multikernel architecture comprising a plurality of operating system kernels, wherein the computer system is adapted for provision and management of virtual machines, the computer system comprising:
- a plurality of processing cores;
 - 5 a plurality of computing nodes, wherein each computing node is assigned one or more processing cores and wherein each computing node comprises an operating system kernel, and wherein one of the plurality of computing nodes is a management node for management of user nodes and one or more other computing nodes of the plurality of computing nodes are user nodes;
 - 10 wherein the management node comprises a manager component, and wherein the user nodes each comprise a virtual machine manager component for managing virtual machines in that user node by interaction with the manager component.
63. The computer system of claim 62, wherein each virtual machine manager component
- 15 is statically defined at boot time for subsequent configuration.
64. The computer system of claim 63, wherein the virtual machine manager component is statically defined as a shell for population on creation of a virtual machine in the user node.
- 20 65. The computer system of any of claims 62 to 64, wherein a virtual machine manager is provided for each virtual machine in a user node.
66. The computer system of any of claims 62 to 65, wherein the computing system comprises a plurality of management nodes.
- 25 67. The computer system of any of claims 62 to 66, wherein each user node comprises a console driver and the management node comprises a console relay for communication between the management node and virtual machine managers.
- 30 68. The computer system of any of claims 62 to 67, wherein the management node comprises a manager component and each user node comprises an agent, wherein the node management component is adapted to instruct the agent of a user node to create, destroy or modify virtual machines in that user node.

69. The computer system of claim 68 where dependent on claim 64, wherein the agent of a user node is adapted to configure the virtual machine manager component for that user node on creation of a virtual machine.

5 70. The computer system of any of claims 62 to 69, wherein the operation system kernels of computing nodes are microkernels.

71. The computer system of claim 70, wherein only the microkernels run at a highest privilege level of the CPU cores.

10

72. The computer system of claim 71, wherein the virtual machine manager components do not run at the highest privilege level.

73. A computer system adapted for virtualisation with use of network devices, the computer system comprising:

15

a plurality of processing cores;

a plurality of computing nodes, wherein each computing node is assigned one or more processing cores and wherein each computing node comprises an operating system kernel, and wherein one of the plurality of computing nodes is a management node and one or more other computing nodes of the plurality of computing nodes are user nodes; and

20

one or more network devices;

wherein the management node comprises a physical function device driver for each of the one or more network devices for physical interaction with the associated network device; and wherein one or more of the user nodes comprises a virtual function device driver for a virtual function network device corresponding to one of the one or more network devices, wherein the virtual function device driver provides a virtual network device function in said user node.

25

74. The computer system as claimed in claim 73, wherein the physical function device driver is provided by an SR-IOV mechanism.

30

75. The computer system of claim 73 or claim 74, further comprising one or more network devices, wherein one or more of the network devices are storage devices, and wherein data is encrypted for communication between the virtual machines and the storage devices.

76. The computer system of any of claims 73 to 75, wherein the operation system kernels of computing nodes are microkernels.

5 77. The computer system of claim 76, wherein only the microkernels run at a highest privilege level of the CPU cores.

78. The computer system of any of claims 73 to 77, wherein the computing system comprises a plurality of management nodes.

10